Fast Packet Filtering Using N-ary Decision Diagrams

Adi Attar and Scott Hazelhurst
Programme for Highly Dependable Systems, School of Computer Science,
University of the Witwatersrand, Private Bag 3, Wits, 2050
{*adi,scott*}*@cs.wits.ac.za, Tel: (011) 717-6189, Fax: (011) 717-6199*

Abstract

Packet filters are security devices that connect multiple packet-based networks and provide access control between them. The security policy of a packet filter is specified according to a set of rules which describes what packet types should be allowed from one network to another. However, the improved network security that packet filters provide comes with a cost. The rules of a packet filter are commonly evaluated sequentially, and so long rule sets result in significantly increased look-up latency. Also long rule sets typically occur at high-bandwidth network interfaces such as on border routers, where fast packet processing is essential.

This paper presents a novel technique for representing the rule sets of packet filters, founded on the concept that a rule set can be expressed as a single Boolean function. When these functions are represented as decision diagrams, this rule set representation provides a constant upper bound on packet filtering latency, independent of the rule set length. Furthermore, by increasing the degree of the decision diagram, faster look-up times can be achieved, at the expense of memory. Empirical research examines this space-time trade-off to provide a packet filtering technique that is both fast and reasonable in memory usage.

1 Introduction

Packet filtering is a common and popular approach to enhancing network security due to its simplicity and efficiency [7]. Packet filters are typically deployed between multiple IP networks and provide network security by inspecting the network packets passing through them and deciding whether they should be discarded or allowed to continue to their destination.

These decisions are made according to a set of rules, where each rule specifies a *selection criterion* and an *action*. The selection criterion describes the condition under which a packet matches the rule (common criterion fields include the packet's protocol type, source and destination addresses, as well as source and destination ports if applicable). The action determines whether a matching packet should be accepted or dropped. The complete set of rules is known as an access list (or access control list). The first rule matching a packet determines the action to be taken.

1.1 Traditional Packet Filtering

Due to the semantics of access lists described above, traditional packet filters perform look-up by evaluating the rules in an access list sequentially until a matching rule is found. In general, no context is kept, so this look-up process must be repeated for every packet [7]. Thus the latency incurred by this sequential look-up is proportional to the length of the access list. While this is not a problem for short access lists, long access lists – lists with several hundred rules are common in border routers – can cause significant system degradation. Also, since the order that the rules appear in the list is significant, a network administrator cannot necessarily place all commonly-occurring rules at the top of the list.

1.2 Alternative Approaches

Most of the related work on packet filtering has been concerned with packet classification, which is the problem of finding the least cost rule that matches a packet. Packet filtering is a specific case of classification where the cost of the rule is its position in the access list.

Several suggested techniques take advantage of the structure typically found in access lists to narrow the search space, thereby improving performance. RFC [3], cross-producting [9] and the tuple space search [8] are examples of this. The problem with such approaches in general is that they cannot guarantee good worst case performance, either in terms of look-up time or in terms of memory.

Control flow graphs (CFGs) [6] are acyclic graphs whose nodes represent Boolean predicates (such as *proto = ICMP*) and edges represent control transfers. CFGs have been used successfully in packet classification applications but no theoretical results have been published in terms of their time or space complexity.

vides, but more importantly due to bit-parallelism that allows considerable amounts of computation to be executed in parallel. The approach presented in this paper is software-based, although some work has been done on FPGA implementations.

In general, packet classification algorithms trade off space against time. Traditional sequential look-up has linear time complexity with respect to the number of rules, but is extremely efficient in space requirements. On the other extreme, by precomputing the matching rule for all 2^S possible inputs in a table, constant look-up times are possible, but the memory usage is exponential. Thus the real challenge lies in finding a solution that offers a good space-time trade-off.

1.3 Research Contribution

This paper presents an alternative approach to access list representation that provides fast look-up with reasonable memory requirements. Some features of this approach include:

- Look-up times independent of the access list length.
- A fixed upper bound on look-up times for a given access list, and access list format.
- The ability to reduce average and worst case look-up times at the expense of memory. (This provides the ability to optimise for speed depending on available memory on a per-system basis.)

The key insight of our data structure is that an access list is essentially a Boolean condition, which describes whether packets should be accepted or not. A representation of the access list as a Boolean expression is independent of the original ordering of rules and comes with convenient ways of representing and manipulating access lists. In particular, standard data structures for representing Boolean expressions provide compact and computationally efficient means of manipulating the access list. We propose a data structure called an N-ary decision diagram (NDD), a generalisation of the well-known binary decision diagram. An NDD is a directed, acyclic graph; the nodes represent the variables in the expression (the bits in the header we are using for filtering) and the edges the decisions. By varying the degree of the nodes in the structure, different space-time trade-offs can be achieved. In general, this approach offers a good trade-off and is capable of fast look-up with modest space requirements.

1.4 Structure of the Paper

The remainder of the paper proceeds as follows. Section 2 introduces NDDs, after which Section 3 explains how they can be used to represent access lists is then empirically compared to traditional sequential filtering in Section 4, and this is followed by a discussion of the results in Section 5. Finally, conclusions and some ideas for future work are presented in Section 6.

2 N-ary Decision Diagrams

N-ary decision diagrams (as used in this research) are directed, acyclic graphs with a unique root and two terminal nodes. Each non-terminal node represents $N = \log_2 B$ Boolean variables, and is of degree B (each edge is labelled by the B possible values the N variables can take). The terminal nodes of the graph are the Boolean constants 0 and 1. In this research N is also termed the *squashing factor* (since the greater the value of N, the more 'squashed' the paths from the root to a terminal).

In the special case where $B = 2$ (and $N = 1$), NDDs are reduced to binary decision diagrams (BDDs) – well-known for their ability to represent Boolean functions compactly and efficiently [2].

This research adds the requirements that the NDDs must be *reduced* – contain no redundancy in the form of duplicate nodes and redundant tests – and *ordered* – have the variables appear in the same order on any path from root to leaf. Thus the variables of an NDD obey a partial ordering. To illustrate, Figure 1 gives two NDDs for the Boolean function $(u \vee v) \wedge (w \vee x) \wedge (y \vee z)$. The first depicts the graph for $N = 1$ with the ordering $u \prec v \prec w \prec x \prec y \prec z$ and the second has $N = 2$ with the ordering $\{u, v\} \prec \{w, x\} \prec \{y, z\}$.

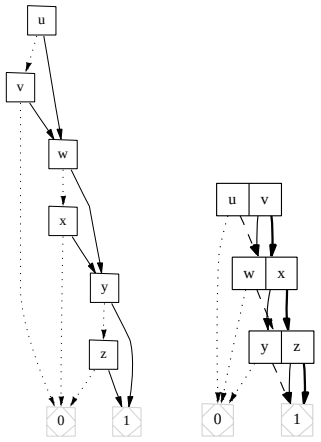

Figure 1. Two NDDs for the same function with $N = 1$ and $N = 2$.

An NDD node has a branching factor of $B = 2^N$, meaning that NDD nodes grow exponentially in size with the number of variables at the nodes. This is somewhat compensated for by the fact that as N increases, the number of nodes decreases, but never-

Therefore, for this technique to be usable, it is important that the original BDD created is fairly compact. In general, a BDD can be quite sensitive to its variable ordering and this can mean the difference between a BDD that is quadratic and one that is exponential in size for a given Boolean function [2]. This is not a problem in our application, as initial experimentation determined a number of simple variable orderings that display good, robust behaviour over a range of synthetic and real access lists. It is one of the contributions of this research that both the BDD and NDD representations of real access lists are well-behaved.

3 NDD-Based Packet Filtering

The strength of NDD-based packet filtering is due to its flexible and compact access list representation. The Boolean expression representation of an access list is very compact, and has efficient algorithms for manipulation.

The approach we propose begins by converting the access list into its corresponding Boolean expression. It then converts the expression into an NDD, and finally uses the NDD to perform the look-up for packet filtering. These steps are discussed in turn in the following sections, after which theoretical bounds on the look-up time are presented.

3.1 Converting an Access List into a Boolean Expression

A Boolean expression representation of an access list is a Boolean expression that describes what packets that are accepted by the access list look like. This expression preserves the ordering semantics of the access list. Each bit in the packet header that is relevant for filtering is represented by one variable in the Boolean expression, and the two filtering actions reject and accept correspond to the Boolean constants 0 and 1 respectively. Thus, packets are accepted by the access list if and only if they satisfy the Boolean expression. The algorithm used to convert an access list into a Boolean expression was proposed by Hazelhurst *et. al.* [4] in the context of analysing access lists. In this paper we generalise by using a more sophisticated data structure, applying the technique to fast look-up. Due to space constraints, the algorithms for performing the conversion have been omitted.

3.2 Conversion into an NDD

Given a Boolean expression, an NDD for the expression can by built by first constructing a BDD using a standard BDD package, and then 'squashing' the BDD to the desired factor. We have also explored constructing the NDD directly, but there seems little advantage.

An NDD is constructed from a BDD as follows. First a root node is created representing the first N variables in the original BDD's variable ordering. For each possible assignment of values to the N variables, B NDD children nodes are created corresponding to the BDD nodes reached by following the appropriate edges in the BDD given by the values of the variables. Of course, some of these children nodes may be the same. This process is repeated until all paths in the original BDD have been processed.

An Example

This example serves to illustrate what information access lists typically contain, as well as what NDD representations of access lists look like. Figure 2 depicts an example access list using the Cisco access list format. (Although the syntax for access list specification differs between packet filtering implementations, their semantics and functionality tend to remain similar.) This access list shows some sample rules that could be applied to inbound traffic at the external interface of the 146.141.0.0/16 network. It allows all inbound mail connections and all ICMP traffic that does not claim to originate from the internal network (the first rule is a simple check for spoofed packets). An NDD corresponding to the Boolean representation of the access list is shown in Figure 3.

```
deny   ip   146.141.0.0/16 146.141.0.0/16
permit tcp  any gt 1023 146.141.0.0/16 eq 25
permit icmp any 146.141.0.0/16
deny   ip   any any
```

Figure 2. A simple access list.

3.3 Performing Look-Up on an NDD

Assume that the squashing factor of the NDD is N, and hence the branching factor is $B = 2^N$. Once an NDD representing an access list has been constructed, performing look-up for a given packet is a simple matter of testing whether the interpretation of the variables given by that packet satisfies the NDD, as follows. Starting at the root of the NDD, the algorithm checks the values of the variables corresponding to that node by inspecting the values of the bits in the given packet. These N values gives a number j in the range $0 \ldots B-1$. The j-th edge is followed to the next node. This process is repeated until one of the terminal nodes is reached, at which point a decision can be made – accept if the terminal node reached is labelled 1, and reject if it is labelled 0.

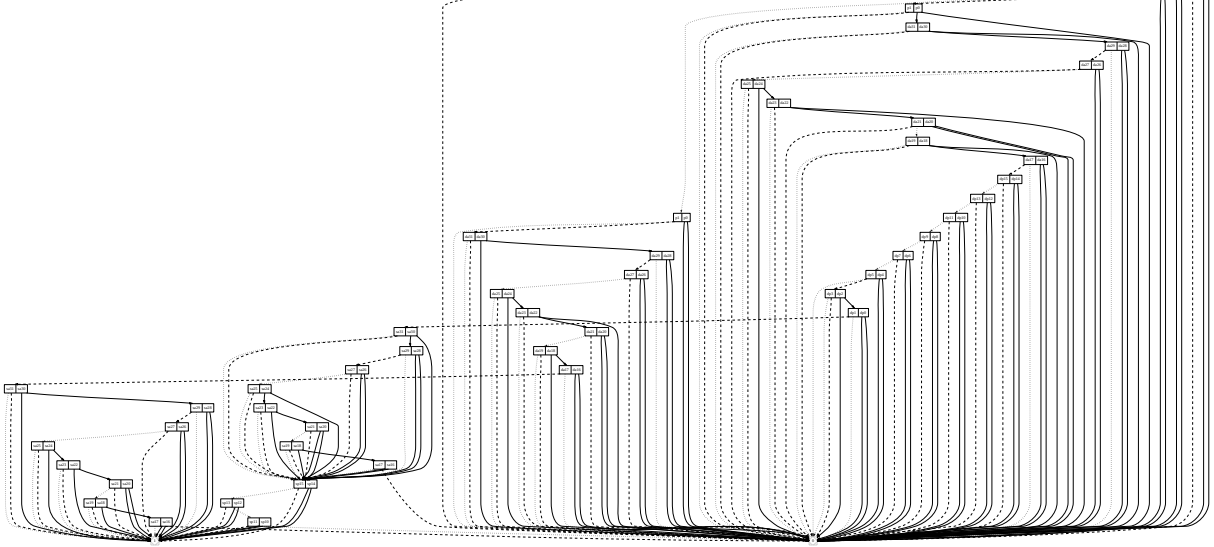

Figure 3. An NDD for the access list in Figure 2 with $N = 2$.

3.4 Theoretical Bounds on Look-Up Performance

The key parameter of NDD-based filtering is the squashing factor, N. Increasing the number of variables at each node results in faster look-up times with an increase in memory usage. This is due to the following factors:

- *A variable is inspected at most once during look-up.* This is due to the fact that the variables in the NDD are ordered, and so cannot appear more than once on any path from root to terminal node.
- *Retrieving the value of a single bit in memory requires the same amount of time as retrieving the value of multiple bits in the same word in memory.* Retrieving the value of a single bit in memory is an expensive task requiring masking and bitwise operations. Retrieving the values of multiple bits can be performed using the same steps with a different mask.
- *The number of variables (T) in the NDD is fixed for a given access list.* Since each variable in the Boolean expression (and hence NDD) corresponds to a bit of filtering interest in the packet header, the number of variables cannot exceed the number of bits in the packet header used for filtering. Furthermore, T is independent of the access list length.
- If the squashing factor is N and there are T bits being used for filtering, then the longest path in the graph is (T/N).

To perform look-up for each incoming packet, the NDD is traversed from the root to one of the terminal nodes. Assuming that each iteration of the look-up algorithm takes a constant amount of time irrespective of the squashing factor, the time taken to perform look-up is proportional to the length of the path traversed by the look-up algorithm.

3.4.1 Worst Case Look-Up Performance

The worst case occurs when the look-up algorithm is forced down the longest path in the NDD. Thus, for a given access list, the worst case look-up time is bounded above by a constant value proportional to (T/N) and is independent of the access list length. Also, the worst case look-up time is halved each time the squashing factor is doubled.

3.4.2 Average Case Look-Up Performance

The average case look-up is important in that it gives a more accurate measure of expected performance. Unfortunately, average case analysis of NDD-based packet filtering is difficult to perform accurately because, in practice, it depends on many external factors, some of which may be impossible to measure. Therefore, the analysis presented in this paper is performed empirically with respect to the actual traffic flow that the corresponding access lists have encountered.

4 Experimental Results

This section presents an empirical evaluation of NDD-based filtering. Memory usage is dealt with first in Section 4.1, after which Section 4.2 evaluates look-up performance by comparing NDD-based look-up to sequential look-up. An NDD-based packet filter has

were conducted on a 1 GHz processor with 512 MB of memory.

4.1 Memory Usage

Memory usage was evaluated in two ways. Firstly, worst case behaviour was elicited by generating "random" access lists. Reducing the structure in the access list makes it less likely that common subexpressions can be shared in the NDD and increases the node count. Secondly, NDDs were also created from real access lists obtained from university departments and an Internet service provider, in order to determine what memory requirements to expect in practice. Since access lists are generally well-structured one can expect the memory requirements of real access lists to be significantly less than those of random lists.

Access lists were generated randomly by generating random, valid values for the fields source address, destination address, protocol, source port, destination port and filtering action, in increasing lengths of 10 rules.

Figure 4 shows the NDD sizes in KBytes, for squashing factors 1, 2, 4 and 8. While the number of nodes actually decreases as the squashing factor increases, the size of an NDD node in the current implementation is $8 + (4B) = 8 + 2^{N+2}$ bytes and the overall memory requirements increase. The exception is a squashing factor of 2 where the overall memory usage decreases relative to a factor of 1. (Factor 16 NDDs were also created for lists up to 160 rules. An NDD for a random list of 160 rules took approximately a minute to generate, so the process was discontinued.)

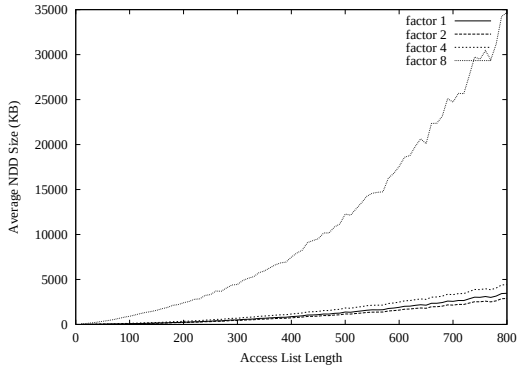

Figure 4. Actual sizes of NDD representations of random access lists of increasing lengths.

Plotting the logarithm of each dataset in Figure 4 produces logarithmic graphs which establishes the growth as polynomial for all squashing factors, rather than exponential. Thus the size of NDD representations of access lists grows polynomially in the worst

Table 1 shows the sizes of NDDs created from real access lists in relation to the list lengths. The last row of the table gives the NDD sizes corresponding to the random access lists of length 160, for comparison. The reduction in memory usage due to the structure in real access lists is evident. Nevertheless, this approach handles unstructured access lists extremely well – a characteristic that many alternative approaches lack.

Length	Factor 1	Factor 2	Factor 4	Factor 8	Factor 16
15	2112	1584	2304	14448	2360400
21	6416	4800	7056	45408	5506224
24	5424	4032	5616	35088	5506224
28	9472	7128	10656	73272	9176352
50	7232	5472	8136	55728	6292680
81	17664	13296	19440	132096	17303064
139	23744	17712	26064	179568	23070408
160	38848	29112	42696	275544	37750920
160	*170175*	*152398*	*246473*	*1787465*	*72306250*

Table 1. Actual sizes in bytes of NDD representations of access lists of various lengths.

4.2 Look-Up Performance

To give a comparison of real performance, the longest real access list (of 160 rules) was used to create NDDs with squashing factors 1, 2, 4, 8, and 16. Then, using packet traces collected from the access list's original inbound network (totalling 30000 packets), the time taken to perform look-up on each packet was recorded. These were compared to the times taken to perform look-up sequentially, on a packet filter specifically implemented for this purpose, also using the *iptables* framework.

The cumulative frequency distributions of look-up times for all the NDDs and the sequential look-up are presented in Figure 5. Furthermore, summary information including the average look-up time, 75th and 95th percentiles, and average number of loop iterations for each filter is given in Table 2.

The reduction in look-up times with increasing squashing factor is clearly visible. The averages in Table 2 indicate improvement factors of approximately 1.5 when the squashing factor is doubled. This is consistent with the theory which states that the longest path length in an NDD is halved when the squashing factor is doubled. This has the effect of reducing the worst case look-up time by a factor of two, and reducing the average look-up time by somewhat less than that since more bits may be inspected than necessary.

The discrepancy in the improvement factor between the factor 8 NDD and the factor 16 NDD is most likely due to caching behaviour. NDDs with squashing factors less than or equal to 8 are reasonably sized and it is extremely likely that a large percentage of these structures remain in the cache for future look-up. When squashed by a factor of 16, the resulting NDD is much

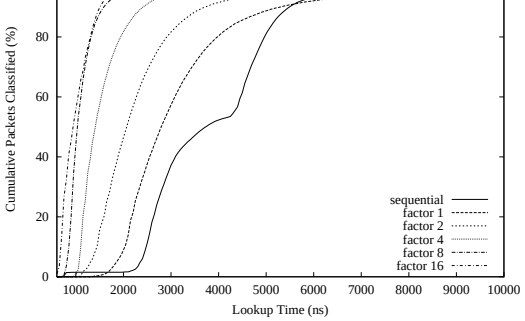

Figure 5. Cumulative frequency distribution of look-up times on packets generated from traces.

Filter	Mean (ns)	75th %	95th %	Iters
Sequential	4132.42	4869	6583	N/A
Factor 1	4011.98	3723	8283	55.93
Factor 2	2622.44	2751	5237	28.77
Factor 4	1754.90	1852	3149	14.86
Factor 8	1251.25	1298	2099	8.04
Factor 16	1173.03	1273	1881	5.29

Table 2. Summary information of look-up times on packets generated from traces.

larger and this reduces the ability of the majority of the graph to remain cached for subsequent use.

5 Discussion

In the worst case, look-up latency on an NDD is proportional to the number of bits filtered on, scaled appropriately by the squashing factor. The worst case look-up has time complexity $O(T/N)$, where T is the number of bits being filtered on. This places an upper bound on all look-up times and also results in fairly constant look-up times on average, independent of the access list length.

The space complexity of NDDs is polynomial in the list length in the worst case, but tends to perform much better on real access lists since the common subexpressions in the rules result in the sharing of nodes in the NDD. Overall, results suggest that NDD representations of access lists are completely viable for NDDs with squashing factor $N \leq 8$ for typical access list lengths.

6 Conclusions and Future Work

This paper has examined the use of NDDs for the representation of access lists for the purposes of fast look-up, and has presented some empirical and theoretical results. NDD-based packet filtering has given access list, no dependence on rule ordering and very little dependence on rule structure.

The independence of rule ordering allows access lists to be optimised for correctness rather than speed, without sacrificing performance – a current problem with traditional packet filters. Furthermore, NDD-based packet filtering offers a good space-time trade-off even when the access lists are poorly structured.

Many alternative approaches can offer good performance, but most cannot offer performance guarantees. NDD-based filtering behaves well in the worst case, and in the average case offers fast look-up with very reasonable memory requirements.

There are some interesting directions for future work. Perhaps the most interesting is to extend NDDs to support packet classification more generally, rather than just filtering. This involves mapping packets to a finite set of values instead of just two. Multi-terminal binary decision diagrams (MTBDDs), extensions of BDDs that support multiple terminal nodes, could perhaps be used as a starting point for creating multi-terminal N-ary decision diagrams (MTNDDs). The time and space complexity of this would need to be examined.

References

[1] F. Baboescu and G. Varghese. Scalable Packet Classification. In *Proceedings of SIGCOMM 2001 Annual Technical Conference*, pages 199–210, San Diego, CA, August 2001.

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

[3] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. *Computer Communication Review*, 29(4):147–160, October 1999.

[4] S. Hazelhurst, A. Attar, and R. Sinnappan. Algorithms for Improving the Dependability of Firewall and Filter Rule Lists. In *Workshop on the Dependability of IP Applications Platforms and Networks*, pages 576–585, New York, June 2000. IEEE Computer Society Press. In *Proceedings of the International Conference on Dependable Systems and Networks*.

[5] T. V. Lakshman and D. Stiliadis. High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *Computer Communication Review*, 28(4):203–214, October 1998.

[6] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of USENIX Winter Conference*, pages 259–269, January 1993.

[7] R. Oppliger. *Internet and Intranet Security*. Artech House, Norwood, MA, 1998.

[8] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. *Computer Communication Review*, 29(4):135–146, October 1999.

[9] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. *Computer Communication Review*, 28(4):191–202, October 1998.