An Interval Decision Diagram Based Firewall

Mikkel Christiansen, Emmanuel Fleury

An Interval Decision Diagram Based Firewall

Mikkel Christiansen, Emmanuel Fleury

Phone: +45 9635 8900, Fax: +45 9815 9889
BRICS, Department of Computer Science, Aalborg University,
Fredrik Bajersvej 7, 9220 Aalborg OE, Denmark
Aalborg, Denmark
{mixxel,fleury}@cs.auc.dk

***Abstract*— This paper explores the use of Interval Decision Diagrams (IDDs) as the central structure of a firewall packet filtering mechanism. This is done by first relating the packet filtering problem to predicate logic, then implementing a prototype which is used in an empirical evaluation. The main benefits of the IDD structure are that it provides access to boolean algebra over filters, efficient classification time, and potentially a compact representation. Results from the empirical evaluation shows that IDDs are scalable in terms of memory usage: a 50,000 rule filter requires only 3MB of memory, and efficient for packet classification: it is able to handle more rules than the schemes it was compared to without causing a degradation in performance.**

Keywords: Packet Classification, Firewall, Traffic Filtering, Decision Diagrams

I. Introduction

The Internet firewall is one of the key technologies used by network administrators for controlling access to a network. The main reason for its success is that the firewall allows filtering of traffic entering and exiting the protected network at a single centralized point. A central mechanism of the firewall is the packet filter, which decides what packets to pass through the firewall using a filter specification. Filter specification consists of a set of rules describing which policy to apply on which packet based on the values in the packet header fields. In this paper we focus on the packet filtering mechanism, and in particular on how packet filters can be improved in terms of performance.

The primary aspect of packet filtering is the issue of packet classification, which has been subject of much study in recent time, for examples see [1], [2], [3], [4], [5]. The reason being that the ability to classify packets plays a central role in firewalls and routing, for instance.

The requirements of the packet classification scheme are quite different from one application to the other. Issues mainly range from classification speed, size of the representation of the ruleset and complexity of the algorithm building the representation of the rule set. One example is packet forwarding on Internet routers, where it is essential that the classification scheme can handle frequent updates of the rule set. Firewalls, on the other hand, may classify packets using more header fields, than the router, but updates occur less frequently.

The main contributions of the work presented in this paper is a packet classification scheme, that is well suited for use in firewalls, and an empirical evaluation through a prototype implementation. This paper does not address issues such as stateful inspection and application level filters.

The central idea of our packet classification scheme is 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 [6]. However, rather than using the widely known Boolean Decision Diagrams (BDDs) [7] as in [6], we use the less explored Interval Decision Diagrams (IDDs) [8]. IDDs operate on integer ranges rather than booleans thus providing the access to efficient classification of packets on generic CPUs.

The main characteristics of the IDD based scheme can be summarized as follows:

- Access to boolean algebra over filters. This simplifies the description of algorithms used in the scheme because we can use well understood operations like comparison, union and intersection. For instance, as we shall see later, describing the transformation between a filter specification and the actual representation can be expressed using boolean algebra over filters.
- Compact representation and scalability. Decision diagram structures are optimized in space independently of the number header fields used in the specification.
- Efficient classification complexity. Namely, $O(m \cdot \log r)$, where m is the number of fields and r is the maximum range of the fields.
- Static representation of filters. The algorithm for building the representation has polynomial complexity, therefore updating the rule set might not be straight forward.

In the following, we first describe background and related work. Section III describes our model of packet filtering. Section IV continues by introducing IDDs and showing how we can represent filters using IDDs. Section V describes the overall architecture of a firewall using decision diagram based packet filters. Followed by an empirical evaluation where the scheme is studied under a worst case scenario. Finally, section VII states conclusions and describe future work.

II. Related Work

Several papers propose algorithms for packets classification on multiple fields for generic CPUs [9], [3], [10], [11].

Begel *et. al* [9] proposes a fully general packet filter framework. Filters are specified in a declarative predicate language, and they 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. The evaluation of the tool shows good performance. However, only small test cases are applied.

In [11], 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 [1].

In [6], 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 firewall filter specified in a Cisco-like access list language into a BDD, including the handling of issues with overlapping rules. The main focus of BDDs in this paper is a tool that can analyze and test filters. A later paper by Hazelhurst *et. al* [12] focus on 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 [13], 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.

More recently, in [5], Rovniagin and Wool describe an algorithm called Geometric Efficient Matching (GEM). Classifying packets with GEM has a time complexity of $O(m \cdot \log n)$, where m is the number of fields and n is the number of rules. Unfortunately, the worst case space complexity is $O(n^4)$. Indeed, the authors manage to show nice empirical results with this technique. In [4], decision trees are widely used to classify packets. The authors introduce a new technique to split the domain of possible values into nearly optimum division.

III. 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*'.

Usually, 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 a 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
```

Fig. 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`" (80), if the incoming packet does not match 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.

The current approach is to use this filter specification strait forward. Indeed, this representation of a filter implies that the efficiency of the packet classification will be strongly related to the number of rules in the list.

The worst case complexity of such algorithm is $O(n \cdot m)$, with n the number of rules, m the number of fields to check in the header. This complexity analysis show that the number of rules has a great impact on the performance of the packet filter.

In this section, we consider a filter as a predicate logic formula on integers. This way of specifying a filter define an algebra on filters, in other words, considering filters as logic formulas, allows us to compare them and to compute the intersection or the union of two (or more) filters. Then, by using this algebra, we show that we have the same expressive power than the ordered rule-set representation. The algorithm presented here has already been described in [6], but neither formal scheme, nor proof or evidence that all rule sets can be expressed by this mean was given.

A. Specifying Filters as First-Order Logic Formula

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

Let H be the finite set of all the possible headers, and let $\Pi = \{accept, drop\}$ be the set of the policies. A *rule* is given by a set of headers (η) and a policy (π):

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

For example, a rule which drops 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 $H \times \Pi$:

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

By extension, we define a filter $\varphi = (\eta_i, \pi_{k_i})_{i \leq n}$ as a function that maps any header h to $accept$ and/or $drop$. Formally, the function $\varphi : H \rightarrow \{accept, drop, \{accept, drop\}\}$ 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 $h \in H$, $\varphi(h) = \varphi'(h)$. And we note $\varphi \equiv \varphi'$. As filters are logic formulas, we can easily define the operators $\neg$ (negation), which switch $accept$ to $drop$ and vice-versa, $\vee$ (OR) which computes the union of two filters (what is accepted by one of the two filters is mapped to $accept$), and $\wedge$ (AND) which compute the intersection of two filters (what is accepted by both is mapped to $accept$). Then, we define a *normal form filter* as a filter with only two rules, one with the policy $accept$ and the other one with $drop$. And, finally, we call a *unambiguous filter*, a filter in which the set of headers $(\eta_i)_{i\leq n}$ are a partition of H. A *partition* is defined as follow:

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

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

An *ambiguous filter* might lead to some confusion if a packet header can be classified both in $accept$ and $drop$. In an *ordered filter* ambiguity is avoided by letting the order prioritize among overlapping rules.

B. Ordered Filters vs Predicate Logic Filters

In order to prove the equivalence between an ordered filter and a predicate filter, we first have to define an *ordered filter*.

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

$$(\eta_i, \pi_i) \succ (\eta_j, \pi_j) \Leftrightarrow i > j \qquad (5)$$

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:

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

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

***Proposition** 1:* For any ordered filter $\psi = (\eta_i, \pi_{k_i})_{i\leq n}$, we can build a filter $\varphi = (\eta'_i, \pi'_{k_i})_{i\leq n}$ s.t. ψ and φ are equivalent.

Proof: The proof is straight 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, we can see that this filter is *unambiguous* and semantically equivalent to ψ. ■

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

In conclusion, we managed to define filters as predicate logic formulas and introduce a basic algebra to manipulate them. Finally, we proved that specifying a rule-set as an ordered-list or a predicate logic formula is equivalent. We even provide an algorithm to derive a predicate logic specification of a filter from any ordered list. In the next section we will describe an efficient data-structure for handling predicate logic formulas on integers.

IV. Interval Decision Diagrams

As pointed out in the previous section, the packet filtering problem is equivalent to the evaluation of a predicate logic formula. Indeed, one of the most efficient data-structure, both in matter of space storage and computational time, is *decision diagrams*. The most famous of these are BDDs [14]. Using such a data-structure to represent filters have been already investigated by Hazelhurst in [13], [6]. Although BDDs are extremely efficient for the hardware side, they are ineffective for the software side. Indeed, one main problem in such approach is that BDDs are based on boolean variables only. Therefore, it is mandatory to consider one bit at a time. 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. Moreover, extracting each bit requires specific encapsulation. And cause some overhead in the memory space usage to store these structures. In order to avoid this drawback, we chose to focus on another decision diagram structure called *interval decision diagram* (IDD, [8]). This structure allows us to perform classification on integer numbers within a domain (finite of infinite).

A. Structure of an Interval Decision Diagrams

An IDD is a *directed acyclic graph* (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. Each edge is linked either to another node or to a boolean *terminal* ($True$ or $False$). More formally, 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 predicate 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 gives the integer variable tested on this node:

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

Finally, let $I = ((t_i)_{i\leq n}, \succ)$ be an IDD iff $(t_i)_{i\leq n}$ is a consistent set of IDD nodes and $\succ$ is an order on the integer variables s.t. $\forall t \in (t_i)_{i\leq n}$ 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 also Figure 2):

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

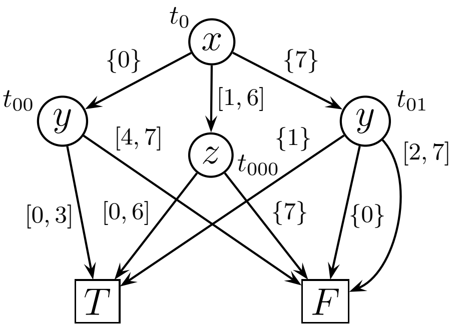

Fig. 2. *Example of an Interval Decision Diagram (IDD).*

IDD structures can easily be used for describing a filter. In Figure 3, we represent a very simple filter as an IDD. This example is testing the 'Source IP' variable that we split into four sub-variables (sip_i) which are easier to test. Note that these IDDs do not contain any non-relevant tests and are optimum in term of operations, if we evaluate the variables in this order.

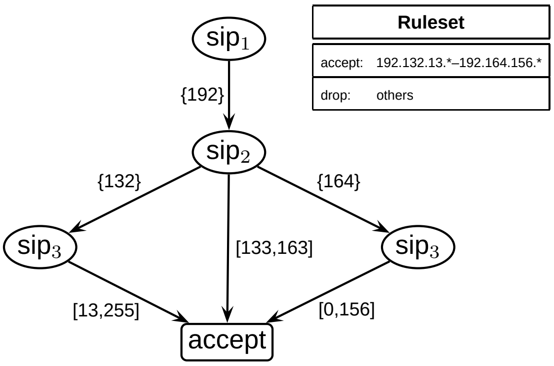

Fig. 3. *IDD representing a filtering rule.*

In Figure 3, $drop$ is assumed to be $\neg accept$, as we handle only boolean terminals. It does not appear because it is assumed that an edge which is not represented leads, by default, to $drop$.

B. Complexity of Packets Classification

As you can see the classification of a packet is done by simply traversing the IDD. From a theoretical point of view, it means that this algorithm has a complexity of $O(m \cdot \log r)$, where m is the number of fields and r is the maximum range of the fields (or the maximum number of intervals that can be in a field). This worst case complexity, when compared to the classical classification scheme ($O(n \cdot m)$ with n the number of rules), appears to be independent to the number of rules. In a matter of facts the number of rules and the number of intervals are related: the more rules you have, the more complex your IDD will be and, therefore, the more intervals you will get. However, this algorithm reduce the complexity from a linear growth to a logarithmic growth[1].

It is also interesting to relate our contribution with [5]. In this paper, Rovniagin and Wool describe an algorithm classifying packets with a time complexity of $O(m \cdot \log n)$. Their results and ours are fairly similar in term of time complexity, but our worst case time complexity get rid of the dependance of the number of rules. Our scheme is tight to the range of each field and do not depend of the number of rules in the filter.

C. Boolean Operations on Interval Decision Diagrams

As for logic formulas, we can perform all the usual logical operations on IDDs, as *negation* ($\neg$), *and* ($\wedge$), *or* ($\vee$), *equivalence* ($\equiv$) and so on. The advantage of this representation is that we can now manipulate the filters through the boolean algebra. All these operators allows us to build complex operations on filters, *e.g.* the translation of an ordered rule-set into an *normal form* filter. Some basic examples are given in Figures 4 and 5. Figure 4 represents two formulas φ_1 and φ_2. Figure 5 represents the result of $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$. The edges labeled by $*$ denote the complement of all the other edges. For example, if a node has four edges labeled $[0, 2]$, $\{9\}$,$[12, 15]$, $*$ and has a range of $[0, 15]$, then $*$ stand for $[3, 8]$ and $[10, 11]$.

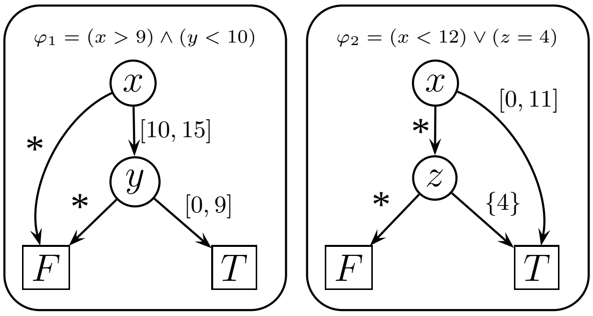
$\varphi_1 = (x > 9) \wedge (y < 10)$
x
$*$
$[10,15]$
y
$*$
$[0,9]$
F
T
$\varphi_2 = (x < 12) \vee (z = 4)$
x
$[0,11]$
$*$
z
$*$
$\{4\}$
F
T

Fig. 4. *Examples of Interval Decision Diagrams.*

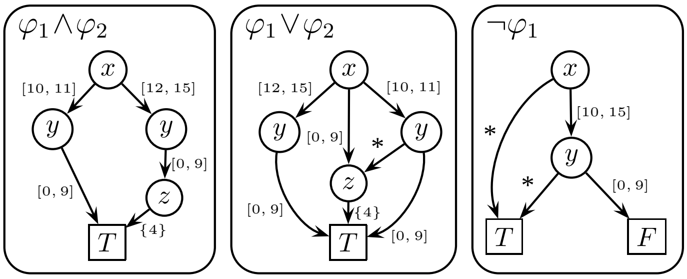

Fig. 5. *Examples of boolean operations on IDDs.*

D. Optimization of Interval Decision Diagrams

In Figure 5, the result of $\wedge$ and $\vee$ is obviously not a strait combination of φ_1 and φ_2. Indeed, optimizations have been performed on the structure in order to prune redundant nodes and sub-trees. The optimization algorithm is quite simple (see [8]). It is performed by listing each node of the IDD and applying the following optimization rules:

1) **Interval Merging**: If two consecutive intervals of a partition lead to the same node, they are merged into one.
2) **Node Pruning**: If a node has only *one* outgoing edge, the node is pruned and the ingoing edges are linked to the node pointed by the previous unique outgoing edge.
3) **Subtrees Merging**: If two nodes are root of two identical subtrees, one subtree is pruned and all the ingoing edges coming to its root are linked to the root of the other.

[1] Note that we are assuming the fact that each new rule added will produce new intervals, which is our worst case behavior.

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 fixed-point have been reached and the optimization process 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 an element and none is adding one). It also guaranty, both, that the number of nodes and the depth of the IDD will be minimal for this given order[2] [8].

In conclusion, we have presented a data-structure (IDDs) to handle with our subset of predicate logic on integer variables, we described an algorithm to optimize in size and depth such data-structures.

V. Architecture of the prototype

The architecture of the packet filtering prototype is shown in Figure 6. The three main components are: the filter, the compiler, and the packet classifier. The flow is similar to that of writing and executing programs using a compiler: A filter is specified using a Cisco-like access list language that allows overlapping rules. A compiler translates the filter to an IDD using the algorithm specified in Section III-B. The compiled filter is then uploaded to a kernel space module which then filters packets according to the specification.

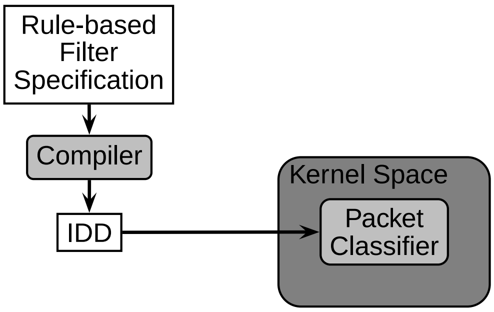

Fig. 6. *Packet Filter Architecture.*

As shown on Figure 6 an actual implementation of the packet classifier runs in kernel-space and serves as the core classification mechanism in a firewall. The majority of kernel space code consists of initializing the IDD data-structure describing the filtering policy, while the classification algorithm is small. Representing the IDD use a directed acyclic graph (DAG) structure, where each partition is sorted and stored in an array, thus allowing fast search of the partition. All partition entries are stored as 32 bit unsigned integers to avoid the overhead of having a comparison function for each different bit-ranges used in the IDD. This gives some overhead for 8 bit and 16 bit fields.

The main strength of this architecture is that the core complexity of the packet filter lies in the compiler that runs in user-space, while the packet classifier that runs in kernel-space is extremely simple. However, a consequence of this design is that the filtering policy is more static because any change requires a recompilation of the filter. We see it as an advantage since the syntax of the filter is checked before actually passing it to the firewall.

VI. Empirical Evaluation

The relevance of using the IDD structure for packet filtering depends on whether performance is competitive with other algorithms for packet filtering. In the following, we describe a set of experiments that we have conducted in order to evaluate the scheme. Focus has been on two issues: space requirements and classification time.

A. Space Requirements of the IDD Structure

The worst case memory requirement of an IDD is exponential in the depth of the IDD, potentially making it unsuitable for packet filtering due to lack of scalability. However, by looking closer at the data stored in the IDD it is quite clear that exponential growth is not a realistic estimate for the actual case. In fact it is quite easy to identify often occurring intervals. For instance, the first bits in an IP-address describes the network address, which in an IDD will be represented as intervals. Another 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.

To provide study the growth rate of IDDs in a realistic setting, we have performed an experiment which allows us to look at sets that are large and yet realistic. The idea is to build filters that describe the traffic of a network backbone by letting each packet header be described by a rule specifying five values: IP protocol, and source, destination IP and ports. Using backbone traces from university network, we were able to generate rule set with sizes ranging from 10-50,000 unique rules. This is worst case scenario for the IDD based scheme because none of the rules contain intervals, only specific values.

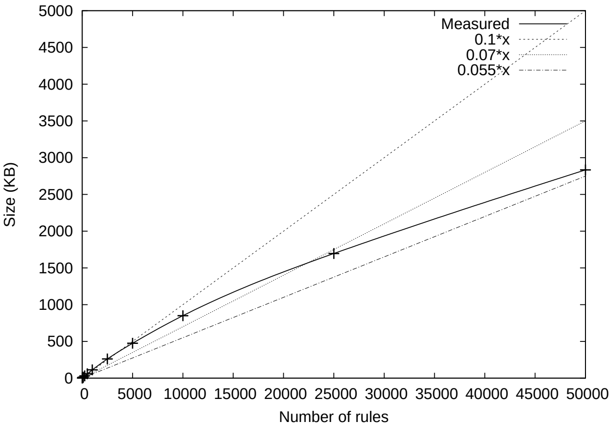

Fig. 7. *Space requirements as a function of number of rules.*

Figure 7 and table I shows the results of the scalability experiments. The graph in figure 7 shows the size of the IDD when loaded by the packet filtering prototype depending on the number of rules in the filter. Initially, we see a linear growth rate of 0.1 per rule, this drops to 0.07 between 0 and 25,000 rules. Finally, between 25,000 and 50,000 rules, the growth rate drops to 0.055. The effect is caused by new rules that produce a minimal change of an existing interval. With 50,000 rules the average size is around 60 bytes per rule. We believe that this can be minimized by more carefully designing the kernel space

[2] Choosing a different order can sometimes lead to some gain.

IDD structure. For instance, we could use indexing rather than pointers when referencing nodes and work on ways to reduce the redundancy of our structure.

#Rules	Time (s)	#Nodes	#Edges	Size (KB)
10	0.01	28	102	2
25	0.03	77	279	4
50	0.11	128	481	7
100	0.44	232	891	14
250	3.44	542	2109	32
500	19.14	998	3952	59
1000	37.72	1884	7544	113
2500	102.1	4289	17493	261
5000	237.0	7678	31880	475
10000	571.4	13420	57417	850
25000	1832	24657	116673	1,696
50000	5221	37416	198754	2,834

TABLE I
IDD RESOURCE REQUIREMENTS (MEASURED ON A 2.6GHZ MACHINE).

As described earlier, the algorithm building the IDDs has a polynomial complexity. To show the effect of this complexity in practice, we briefly discuss the compilation times for transforming a rule-based filter to an IDD. The second column in Table I shows the compile times for filters having sizes ranging from 10 to 50,000 rules. As we can see more realistic filters consisting of up to 1,000 rules, compiles in less than 40 seconds, which is acceptable. Larger filters with more than 10,000 rules take unacceptably long time to compile. However, we have not made any systematic attempts to optimize the compilation process in the current version.

B. Classification time

In this section, we focus on evaluating the IDD based packet filter by studying the performance of the IDD based scheme in a near worst case scenario, and compare the measured performance with that of the packet filtering algorithms currently provided on the Linux platform.

For the experiments we have chosen Linux as our test platform. The main reason for this choice is that Linux already provides two different packet filtering mechanisms, based on two different algorithms, and due to the simplicity of implementing the IDD based scheme as a loadable kernel module.

The two packet filtering schemes provided in Linux are: the classical packet filtering scheme, Netfilter [15], and a high-performance scheme called Hipac [16] which has been implemented as a Netfilter module. Netfilter performs a simple linear evaluation of the rules in the order they are stored until a matching rule is found. Hipac uses a more advanced algorithm based on the scheme described in [3], [17].

The IDD based packet filtering scheme was implemented as a loadable kernel module. Access to packets is gained by using the hooks which a “netfilter enabled” kernel provides, thus meaning that no changes to the Linux source tree were made to implement our scheme. In addition to the kernel module, a user-space tool was written to allow IDDs to be uploaded to kernel memory. The data-structure for storing the IDD is a DAG. Filtering of packets is done by traversing the IDD from root to terminal and performing binary search of each partition.

For this evaluation, we are primarily interested in studying the effect on performance of a firewall with different sizes of filters and under different traffic loads. Our focus has been on keeping the experiments as simple as possible, yet allowing us to get a clear indication of whether the idea of IDD based firewalls is worth pursuing.

1) Setup and experimental procedure: Figure 8 shows the experimental setup. At the core of the network, we have the firewall which filter traffic arriving from the traffic generators (TG1-20). The traffic is filtered and accepted traffic is routed to a sink which simply drops the incoming traffic. All nodes are connected using two switches: a Cisco Catalyst 2950 and a Cisco Catalyst 3500XL. The 2950 allows 100Mbps links from traffic generators to be aggregated onto a 1Gbps link, while the 3500XL switch allows the firewall to be connected to the Catalyst 2950 and the sink. To monitor the performance on the link without influencing the experiments, we use SPAN port monitoring on the catalyst 3500XL which forwards all traffic on a specific port toward a monitoring port. A machine with two gigabit interfaces is connected to the SPAN enabled ports allowing us to monitor both input and output traffic of the firewall being tested.

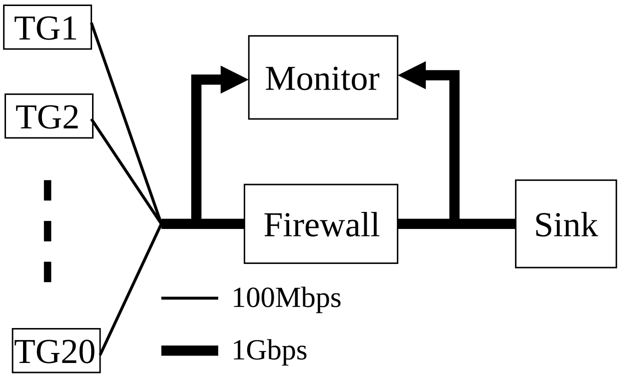

Fig. 8. *Logical structure of the experimental setup.*

Each experiment consists of choosing a filtering scheme (Netfilter, Hipac, or the IDD based scheme), and uploading a packet filter to the chosen scheme. A traffic load is then generated using the header trace matching the installed packet filter. Packet filters and header traces were generated using a method similar to the method used for the growth rate study described in the previous section. However, for each unique rule, we also store the unique packet header in a trace. Therefore, when replaying the header trace on a traffic generator machine we are sure that all packets are accepted by the filter and all rules are matched uniformly. This corresponds to a worst case scenario for the IDD based filtering scheme, because all parts of the filter is used continuously, and because each packet requires a full search of the IDD.

In one way the generated traffic is different from the traffic described in the replayed header traces. The payload of each packet is kept fixed throughout the experiment. This allows us to compare the performance measured with two different filters. If we used the original payload size, then each header trace would have a different payload size distribution thus making it difficult to compare performance.

During each experiment the traffic generators run for a five minute period during which we monitor the performance of the router. At load up to 75Kpps (thousands of packets per second), we are able to monitor traffic on both interfaces of the router, however due to limited disk bandwidth at higher loads, we are only able to monitor the outbound link of the two links on R with M, however traffic generators measure the rate at which traffic is generated ensuring that we can detect any variations in the input load to the firewall. Throughout the experiments, switch statistics has also been monitored to ensure that switches are not congested with traffic during any of our experiments.

CPU	AMD Athlon(TM) MP 2000+ (1700MHz)
NICs	SysKonnect SK9821 V2.0 (64bit, 66MHz)
NIC driver	Driver sk98lin v6.03, NAPI enabled
Kernel	Linux-2.4.20

TABLE II
FIREWALL CONFIGURATION.

2) Results: In the overall set of experiments, we have explored several combinations of input load (100, 250, and 500Mbps). However, for clarity we only show results from experiments with a sustained input load of 500Mbps.

Figure 9 and 10 shows the results of experiments with a sustained input load of 500Mbps, fixed payload size of 300bytes (∼180 Kpps) and number of rules ranging from 10-10,000. In the interval from 10-100 rules the performance of the packet filters are quite similar. However, both Hipac and Netfilter perform lower than expected with 25 rules. For Netfilter, throughput performance starts to degrade as the number of rules surpasses 100 rules, and continues to degrade until 2500 rules where the throughput reaches zero. Hipac performs significantly better than Netfilter. Performance also degrades once the filter grows larger than 900 rules, however at a slower rate than with Netfilter. Finally, the IDD based classification scheme can handle up to 3,000 rules before performance begins to degrade. Between 5,000-10,000 performance of Hipac and the IDD based scheme is nearly unchanged. In essence, the figures clearly demonstrates the complexity difference between the approach used in Netfilter and the IDD based classification scheme.

Figure 11 (x-axis is now logarithmic) shows a similar set of experiments for runs with a fixed payload size of 600bytes (∼95 Kpps). Behavior mimic the previous set of experiments except that the performance degradation occurs with larger rule sets. An interesting aspect is that, between 10-100 rules, Netfilter actually performs better than Hipac. We expect this to be due to a processing overhead of using Hipac, however further investigation is necessary to determine the exact cause. Experiments with a frame size of 1440 were also performed. But, under that load, neither of the filtering mechanisms had problems handling the 500Mbps load. Probably due to the fairly low frame rate.

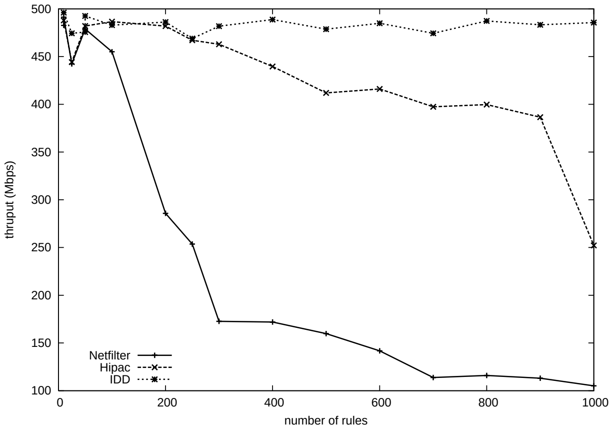

Fig. 9. *Firewall throughput as a function of the number of rules (0-1000 rules). Input load is fixed at 500Mbps load and payloads fixed to 300 bytes.*

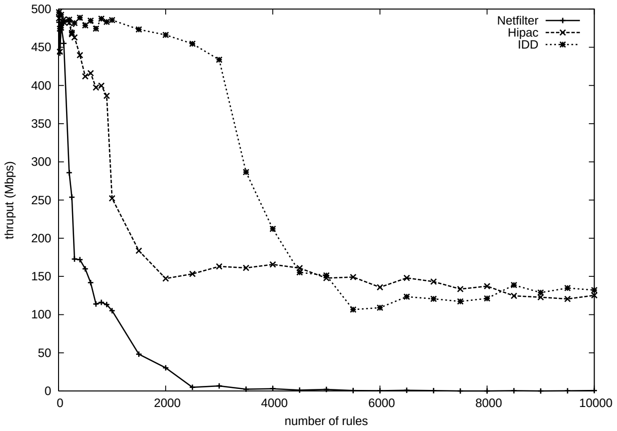

Fig. 10. *Firewall throughput as a function of the number of rules (0-10000 rules). Input load is fixed at 500Mbps load and payloads fixed to 300 bytes.*

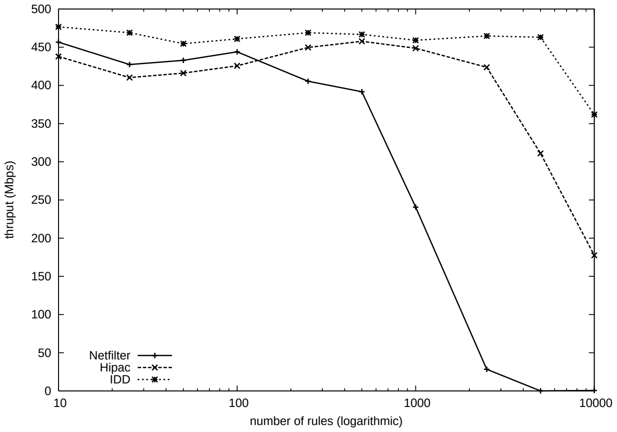

Fig. 11. *Firewall throughput as a function of the number of rules (x-axis is logarithmic). Input load 500Mbps and payloads are fixed to 600 bytes.*

In total, these experiments shows significant performance potential of the IDD base packet filtering scheme. The experiments with payloads of 300 bytes is the most interesting, we were able to handle three times more rules than Hipac before reaching the performance limit with 3,000 rules. One could argue the only few firewalls will ever have filters consisting of more than 500 rules. Nevertheless, it is important to realize that even though we get similar performance with smaller rule sets, the IDD based scheme overall, seems to use less CPU resources than both Netfilter and Hipac, leaving more resources to other tasks allowing the use of a slower and cheaper CPU.

The conducted experiments have a limited scope and many questions remain open. For instance the described experiments do not take into account that in a realistic setting some rules will be matched more often than other, thus allowing the approach used in Netfilter to be optimized by setting the most used rules first in the filter. In future experiments, we hope to explore performance issues related to this aspect and many others.

VII. Conclusion

In this paper we have described the use of Interval Decision Diagrams (IDDs) as the central structure in the packet filtering mechanism of a firewall. The work includes a mapping of traditional filter specification to predicate logic, and an empirical evaluation of the scheme, comparing it to two other schemes provided on the Linux platform.

The primary advantages of using IDDs, in a packet filtering scheme, is that an IDD describes a predicate logic expression, thus providing access to boolean algebra over filters. Furthermore, IDDs allow efficient classification of packets on generic CPUs, by avoiding tests of individual bits. A disadvantage, on the other hand, is that the build algorithm has polynomial complexity, this means that it is quite time consuming to rebuild the structure over a certain size.

The presented empirical evaluation focused on two issues: growth-rate of the size of the IDD and the efficiency of the packet classification algorithm. Experiments on growth-rate showed that IDD based structure grows less than linearly in the number of rules, when rules describe individual packet from a packet-header trace. This is a promising result, considering the fact that worst-case growth rate of an IDD is exponential.

To evaluate the packet classification algorithm, a prototype was implemented on Linux. This allowed us compare performance with two other packet classification schemes provided on the Linux platform. The IDD based scheme was successful by being able to handle filters with significantly more rules before throughput of the firewall decreased.

Overall, the work presented in this paper suggests that the IDD structure is a strong and efficient foundation for packet filtering on firewalls.

Further work includes a more elaborate evaluation of the IDD based scheme, for instance by comparing it to other schemes. Also, testing the scheme with real filters is important as well as extending the scheme to allow more than two policies. Finally, issues on working on optimizing space requirements of the actual data-structure and minimizing compile times should be addressed as well. Moreover, we are now working on a new prototype for Linux, called Netkeeper [18], which is freely downloadable.

VIII. Acknowledgments

We would like to thank the DIRT group at University of North Carolina, Chapel Hill, and in particular Felix Hernandez-Campos for providing access to relevant network traces. Furthermore, we would like to thank Jesper Sloth Christensen, Martin Dalum, and Lars Riis Olsen for making the initial implementation of the packet filter as a Linux kernel module.

References

[1] T. V. Lakshman and D. Stiliadis, "High Speed Policy-based Packet Forwarding Using Efficient Multidimensional Range Matching," in *Proc. of ACM SIGCOMM*, Vancouver, Canada, September 1998, pp. 203–214.

[2] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *Proc. of ACM SIGCOMM*, Cambridge, MA, USA, August 1999, pp. 147–160.

[3] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *Proc. of IEEE INFOCOMM*, Tel–Aviv, Israel, March 2000, pp. 1193–1202.

[4] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," in *Proc. of ACM SIGCOMM*, August 2003, pp. 213–223.

[5] D. Rovniagin and A. Wool, "The geometric efficient matching algorithm for firewalls," Tech. Rep., Dept. of Electrical Engineering Systems, Tel Aviv University, Ramat Aviv 69978 Israel, 2003.

[6] Scott Hazelhurst, "Algorithms for Analysing Firewall and Router Access Lists," Tech. Rep. TR-WitsCS-1999-5, Department of Computer Science, University of the Witwatersrand, South Africa, 1999.

[7] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C–35, no. 8, pp. 677–691, August 1986.

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

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

[10] V. Srinivasan, "A Packet Classification and Filter Management System," in *Proc. of IEEE INFOCOMM*, Anchorage, AK, USA, April 2001.

[11] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *Proc. of ACM SIGCOMM*, San Diego, CA, USA, August 2001, pp. 199–210.

[12] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for Improving the Dependability of Firewall and Filter Rule Lists," in *Proc. of the International Conference on Dependable Systems and Networks*, New York, NY, USA, June 2000, pp. 576–585.

[13] A. Attar and S. Hazelhurst, "Fast Packet Filtering Using N-ary Decision Diagrams," Tech. Rep., School of Computer Science, University of Witwatersrand, 2002.

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

[15] Martin Josefsson, Kadlecsik Jozsef, Harald Welte, James Morris, Marc Boucher, and Paul "Rusty" Russell, "NetFilter Homepage," `http://www.netfilter.org`, 2003.

[16] M. Bellion and T. Heinz, "Hipac (High Performance Packet Classification for Netfilter)," `http://www.hipac.org`, 2003.

[17] M. Bellion and T. Heinz, ," `http://www.hipac.org/firewall_documentation/packet_filter_faq.htm#l17`, 2003.

[18] M. Christiansen and E. Fleury, "Linux Netkeeper," `http://www.cs.auc.dk/~fleury/netkeeper/`, 2003.