Fast Firewall Implementations for Software-based Routers

Lili Qiu
lqiu@cs.cornell.edu
Cornell University

George Varghese
varghese@cs.ucsd.edu
University of California, San Diego

Subhash Suri
suri@cs.wustl.edu
Washington University in St. Louis

ABSTRACT

Routers must perform packet classification at high speeds to efficiently implement functions such as firewalls. The classification can be based on an arbitrary number of prefix and range fields in the packet header. The classification required for firewalls is beyond the capabilities offered by standard Operating System classifiers such as BPF [12], DPF [7], PathFinder [1] and others. In fact, there are theoretical results that show the general firewall classification problem has poor worst case cost: for searching over N arbitrary filters using k packet fields, either the worst-case search time is $\Omega((\log N)^{k-1})$ or the worst-case storage is $O(N^k)$.

In this paper, we re-examine two basic mechanisms that have been dismissed in the literature as being too inefficient: backtracking search and set pruning trees. We find using real databases that the time for backtracking search is much better than the worst case bound; instead of $\Omega((logN)^{k-1})$, the search time is only roughly twice the optimal search time[1]. Similarly, we find that set pruning trees (using a DAG optimization) have much better storage costs than the worst case bound; it has memory requirements similar to the RFC scheme of Gupta and McKeown [10]. We also propose several new techniques to further improve the two basic mechanisms. Our major ideas are a novel compression algorithm, the ability to trade smoothly between backtracking and set pruning, and algorithms to effectively make use of hardware if hardware is available. We quantify the performance gain of each technique using real databases. We show that on real firewall databases our schemes, with the accompanying optimizations, are close to optimal in time and storage.

1. INTRODUCTION

As the Internet evolves into a global communication infrastructure, it is increasingly important to provide differentiated services [6] to users with widely varying requirements, allowing users to pay for different levels of service. A key mechanism that enables differentiation in a connectionless network is *packet classification*. In packet classification routers, the route and resources allocated to a packet can be determined by the destination address as well as other header fields of the packet such as the source address and TCP/UDP port numbers. For example, in the emerging diffserv architecture [6] core routers classify packets based only on IP TOS fields; however, edge routers determine Per Hop Behaviors (PHBs) by setting IP TOS bits based on destination, source and even port fields. More importantly (at least today), many router products allow a firewall capability, such as Cisco Access Control Lists (ACLs), which allow packets to be blocked based on the same fields.

[1]The height of the multiplane trie is regarded as optimal search time throughout the paper, unless otherwise specified.

A more abstract framework for packet classification, consistent with Cisco ACLs, is as follows. The packet classification database of a router consists of a potentially large number of filters (or rules in firewall terminology) on key header fields. A general filter consists of arbitrary prefix or range specifications on the destination, source, protocol, port number and possibly other fields. A given packet header can match multiple filters, so each filter is given a cost, and the packet is forwarded using the least cost matching filter. The industry standard, Cisco ACLs, specifies filter costs by entering the filters (sometimes called "lines") in a linear ordering and using the order number as an implicit filter cost. Thus, in Cisco ACLs the first matching filter must be found.

While the diffserv proposal is still not mature, almost every router today has support for ACLs. Edge routers need ACLs to implement firewalls. However, even large backbone routers today implement ACLs to trace denial-of-service and flood attacks. Thus our paper concentrates on techniques for speeding up packet classification for firewalls using properties we have observed in real firewall databases. While we believe our techniques generalize to other filter databases such as diffserv, it is difficult to test this assertion because there are no models of diffserv databases that are generally agreed upon.

The current state of the art in most routers is to either use linear search of the filter database or to use hardware, such as ternary CAMs or other ASICs that perform parallel linear search (e.g., [9]). Other solutions reported in the literature that can be implemented in software (e.g., [16, 10]) are either slow or take too much storage. With the advent of software based routers (e.g., [13]), which are typically aimed at the edge router space where classification is particularly important, it is particularly important to find fast software techniques for fast firewall implementations.

There is evidence that the general filter problem is a hard problem, and requires either $O(N^k)$ memory or $\Omega((\log N)^{k-1})$ search time, where N is the number of filters and k is the number of classified fields [9, 16]. However recent research [10, 11, 17] indicates that such worst case behavior does not arise in real databases. Based on this observation, these papers introduce clever *new* techniques like pruned tuple search [17] and Recursive Flow Classification [10] that exploit the structure of existing databases. However, if real databases have regularities that can be exploited, perhaps even the simplest packet classification algorithms will do quite well.

This question motivates us to re-examine the two simplest packet classification mechanisms we know of: *backtracking search* and *set pruning trees*. There is an interesting duality between these two simple schemes: backtracking search requires the least storage but can have poor worst case search times; set pruning trees have minimal search times but have poor worst case storage. Earlier researchers have dismissed backtracking search as being too slow [17], and dismissed set pruning trees as being suitable only for very small packet classifiers [5, 17].

However, we find using real databases that the time for backtracking search is much better than the worst case bound. Instead of $(\log N)^{k-1}$, the search time is only a constant factor (often only a factor of two) worse than optimal. Similarly, we find that set pruning trees (using a DAG optimization) has much better storage costs than the worst case bound of N^k would indicate.

We also propose several novel techniques to further improve the performance of backtracking and set pruning trees. First, we design a novel compression algorithm that applies to any multiplane trie. Our compression scheme is particularly useful because it allows filters that use port ranges to be stored economically. We evaluate our compression scheme using both theory and experiments. Our results indicate that compression leads to significant reduction in both lookup time and storage.

Given that backtracking search and set pruning trees are at two ends of a spectrum between optimal storage and optimal time, it makes sense to study the tradeoff between these two extremes. As the two schemes are structurally similar and use multiplane tries as their underlying basis, we show that it is possible to smoothly trade storage for time using a new mechanism called selective pushing. Our results show that the tradeoff scheme offers more choices, and can improve the time of backtracking search with only modest increases in storage.

Finally, we investigate the possibility of moving a subset of filters to hardware if it is available. Our results show that by removing only a small number of filters from software for hardware lookup, the storage requirement and query lookup time for the software approach (i.e. backtracking search and set pruning trees) can be greatly reduced. This is significant because they indicate the benefit of adding small ternary CAMs to software-based routers while yet allowing the firewall database to contain a large number of rules.

The paper is organized as follows. We give the standard problem definition in Section 2, and review related work in Section 3. In Section 4 we describe backtracking search and introduce some simple new optimizations to improve search time. We then evaluate its performance, and quantify the effects of each of the optimizations using the real firewall databases. In Section 5 we describe set pruning search, introduce some optimizations to improve storage, and present our experimental results. In Section 6 we propose a novel compression scheme, and evaluate its performance gain both in theory and with experiments. In Section 7 we explore a tradeoff between time and space by starting with backtracking search and using selective pushing. In Section 8 we investigate ways to effectively make use of a limited amount of hardware if it is available. We conclude in Section 9.

2. PROBLEM SPECIFICATION

Packet classification is performed using a packet classifier, which is a collection of filters (or rules in firewall terminology). Each filter specifies a class of packet headers based on some criterion on K fields of the packet header. Each filter has an associated directive, which specifies how to forward the packet matching this filter. We say that a packet P matches a filter F if each field of P matches the corresponding field of F. This can be either an exact match, prefix match, or a range match. The match type is implicit in the specification of the field. For example, if the protocol field is specified as UDP, then it requires an exact match; if the destination field is specified as 11*, then it requires a prefix match; if the port field is a range, such as greater than 1023, then it requires a range match. Typically, destination and source fields use prefix matches, port fields use range matches, and protocol fields use exact or wildcard matches. Since we can represent a range using multiple prefixes [17], we assume for the rest of the paper that each field in a rule is a prefix unless otherwise specified. [2]

Since a packet can match multiple filters in the database, we associate a cost for each filter to determine an unambiguous match. Thus each filter F in the database is associated with a non-negative number, $cost(F)$. Our goal is to find the filter with the least cost matching a packet's header. The key metric is classification speed. It is also important to reduce the size of the data structure to allow it to fit into high speed memory. The time to add or delete filters is often ignored in existing work, but can be important for dynamic filters.

We note that this form of classification is beyond the capabilities of classifier techniques such as BPF [12], PathFinder [1], DPF [7] etc, that are often used in operating systems. Such classifiers do not allow the use of prefixes in every field.

3. RELATED WORK

Many router vendors do a linear search of the filter database for each packet, which scales poorly with the number of filters. To improve the lookup time, some vendors cache the result of the search keyed against the whole header. Caching may work well and have high hit rates [19] but still requires a fast packet classification scheme to handle the $10-20\%$ cache misses. A hardware-only algorithm could employ a ternary CAM (content addressable memory). However ternary CAMs are still fairly small, inflexible and consume a lot of power.

[9] describes a scheme optimized for implementation in hardware. The scheme computes the closest enclosing range for each dimension. Each range is associated with an N bit vector. The intersection of the K vectors is computed, and the best filter corresponds to the first bit set in the intersection. This scheme works well for up to 8000 filters and should scale further with further hardware improvements. However, it requires specialized hardware. [17] proposes two solutions for multi-dimensional packet classification: grid-of-tries and crossproducting. The former scheme decomposes the multidimensional problem into several 2-dimensional planes, and uses a data structure, grid-of-tries, to solve the 2-dimensional problem. Crossproducting is more general but either requires $O(N^k)$ memory or requires a caching scheme with non-deterministic performance.

[10] proposes a simple multi-stage classification algorithm, called recursive flow classification (RFC). RFC exploits the structure

[2] There can be a large increase in the number of rules during range-to-prefix conversion. Our new compression algorithm will address this issue as shown in Section 6.1.

Filter	$Field_1$	$Field_2$	$Field_3$
F1	00*	00*	00*
F2	0*	00*	1*
F3	10*	1*	*
F4	00*	00*	0*
F5	0*	0*	01*
F6	0*	0*	1*
F7	00*	0*	01*
F8	0*	0*	0*

Table 1: An example of eight 3-dimensional filters.

and redundancy contained in real databases by determining equivalence classes for packet headers. It has much better (but still large) storage for real databases than cross-producting, though its worst-case storage is still high. [16] suggests searching through combinations of field lengths (tuples) and also suggest a heuristic of first doing prefix searches on the individual fields to prune the set of tuples to be searched. [11] suggests another heuristic based on geometrically partitioning the classification space, which produces fairly good search times and requires less memory than [10].

[8] considers a tradeoff between lookup time and storage cost. However, their experimental results are only for 2-dimensional databases, and it is not clear how the algorithm would perform for the higher-dimensional real databases. We describe a completely different algorithm that trades storage for lookup time in Section 7 and evaluate its performance on 5-dimensional real databases.

4. REVISITING BACKTRACKING SEARCH

Given that real firewall filter databases contain considerable structure, in this section we revisit the simplest algorithm we know of: backtracking search. We start by reviewing the basic mechanism, and then show how it can be augmented by simple optimizations.

4.1 Basic Backtracking Search

A trie is a binary branching tree, with each branch labeled 0 or 1. The prefix associated with a node u is the concatenation of all the bits from the root to the node u. It is straightforward to extend a single dimensional trie to multiple dimensions [17]. We illustrate the idea using the following example of a 3-dimensional trie, as shown in Figure 1. The trie corresponds to the filter database shown in Table 1.

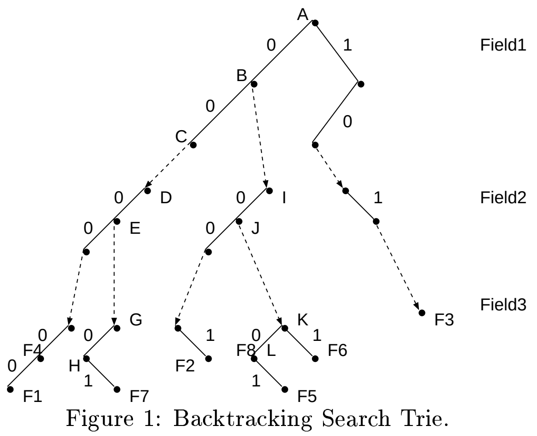


Figure 1: Backtracking Search Trie.

We first build a trie on the prefixes of the first field, $Field_1$. Each valid prefix in the $Field_1$ trie points to a trie containing $Field_2$ prefixes (that follow $Field_1$ prefixes in some filter). Similarly, each valid prefix in the $Field_2$ trie points to a trie containing $Field_3$ prefixes. Since each filter is stored exactly once, the memory requirement for the structure is $O(NW)$, where N is the total number of filters, and W is the maximum number of bits specified in any of the three dimensions.

Backtracking search is essentially a depth first traversal of the tree which visits all the nodes satisfying the given constraints. For the example shown in Figure 1, suppose we search for a lowest cost filter that matches the packet header of the form [00*, 0*, 0*]. We first traverse the path $A - B - C - D - E - G - H$. Then we bracktrack all the way back to node B, and follow the path $I - J - K - L$. Finally, we backtrack to node A. Since there is no subtrie attached to A, we stop and return the best matching filter, which is $F8$.

The lookup time of backtracking search can be as large as $\theta(W^K)$, where K is the number of dimension. This is easy to see using the same 3-dimensional trie example. In the worst case, we may end up searching W two-dimensional tries hanging from $Field_1$. Searching over each two-dimensional trie in turn may involve searching W one-dimensional tries, each at a cost of $O(W)$. This yields altogether $O(W^3)$ in the worst case. The application of switch pointers, introduced in [17], can help to avoid backtracking in the last two dimensions. This reduces the worst-case lookup time to $O(W^{K-1})$. Thus in the 2-dimensional case, the lookup time is $O(W)$.

4.2 Backtracking Search Optimizations

Before we introduce our major optimization ideas, we start by describing some simple (but new) optimizations for backtracking search: considering optimal field order, pruning based on cost, and generalizing switch pointers [17].

Optimal Field Ordering: We have observed that the lookup time in backtracking search is sensitive to the order of fields in the trie. This is illustrated in the following example. Figure 2(a) and (b) correspond to the same database. The only difference is the order: Figure 2(b) exchanges the order of $Field_1$ and $Field_3$ in Figure 2(a). As we can see there is a significant difference in the number of steps involved in backtracking search: the worst case lookup time to search for a header [00*, 00*, 00*] is 17 in Figure 2(a), and only 11 in Figure 2(b).

Based on this observation, we can optimize backtracking lookup time by choosing the best order. A straightforward way to find the best order is to try all possible orders, and pick the one that yields the best lookup time. For our examples that contain 5 fields, there are 120 possible arrangements. However, since the protocol field generally requires either exact match or matching *, it turns out to be nearly optimal to always examine the protocol field first. This leaves only 24 possible field orderings to be examined.

We evaluated the effect of different order on the query lookup time using practical firewall databases. (We will discuss our experimental results in detail in Section 4.3.) Table 2 shows the number of rules in the databases we studied, and the ratio between worst lookup time vs. best lookup time. As we can see, for some databases the effect of different order is small

Database	# Rules	$\frac{worst\ time}{best\ time}$
Database 1	67	111.64%
Database 2	158	164.71%
Database 3	183	130.68%
Database 4	279	145.54%
Database 5	266	127.40%

Table 2: Different field ordering affects the lookup time

Database	# Rules	# Rules in the prefix format
Database 1	67	139
Database 2	157	496
Database 3	183	638
Database 4	279	1177
Database 5	266	1645

Table 3: Firewall databases

(only around 10%), while for other databases the effect is considerably larger. However, for all the databases we have, the difference is within a factor of 2. Moreover, we have found that for almost all the databases, there are about half of the orderings whose lookup times are within 10% different from that of the best ordering. Therefore a more efficient alternative to find a good ordering is to randomly try a few orderings, and choose the best one. This is much faster than exhaustive search, and can achieve comparable performance.

A further enhancement to this approach is to use some heuristic to pre-select some good orderings, and then randomly try some of the selected orders. One possible heuristic we can use is to choose the order that places the field with the most prefix containment [3] as the last field. For example, this heuristic would prefer Figure 2(b) over Figure 2(a) because Field 1 has more prefix containments than the other fields. Other heuristics were suggested in [11, 8].

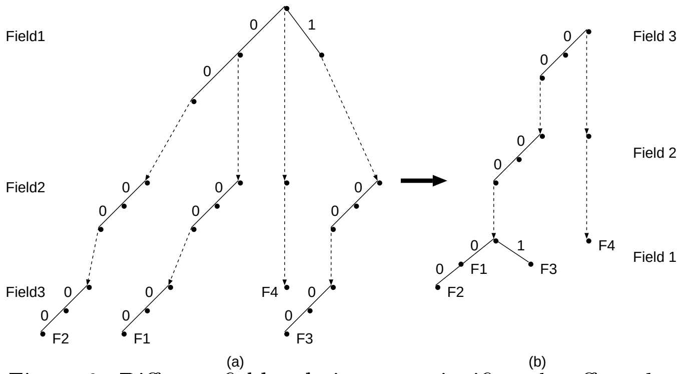


Figure 2: Different field orderings can significantly affect the lookup time in backtracking search.

Pruning Based on Cost: The basic idea behind pruned backtracking is as follows. During the backtracking search, if we encounter a trie node such that the tree beneath the node does not contain any lower cost filter than the current match, then we do not need to search through the trie below that node. (Pruning based on cost is used in other filter algorithms such as [10, 2].).

For example, consider searching for a filter [00*, 00*, 00*] in the trie shown in Figure 3 using backtracking search. Suppose we find that the incoming packet matches filter $F1$, where filter $F1$ is the lowest cost filter in the database. When we backtrack to point P_2, since we know P_2's subtrie does not contain any lower cost filter, we do not need to search P_2's subtrie. Similarly we do not need to search over P_1's subtrie, since it does not contain a lower cost filter than the current match. In general, we use precomputation to annotate every node in the trie with the lowest cost filter in the tree rooted at the node. During backtracking search, if we encounter a node whose annotated match is no better than our current match, we do not search its subtrie.

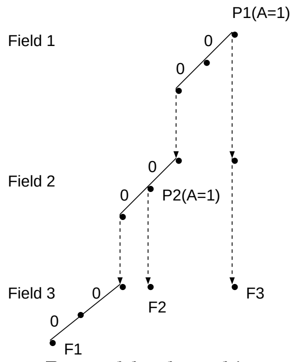


Figure 3: Pruned backtracking search.

Switch Pointers: We further optimize backtracking search with switch pointers. Switch pointers were introduced in [17] but the technique is limited to 2-dimensional packet classification. We extend switch pointers to higher-dimensional packet classification by using it over the last 2 fields, or by avoiding the first backtracking, whichever is more beneficial. We omit details for lack of space.

4.3 Performance Evaluation

In this section, we experimentally evaluate backtracking search to quantify the effects of the various optimizations described above. We use total storage and worst case lookup time as our performance metrics. Total storage is computed as the total number of nodes in the multiplane trie. The worst case lookup time is the total number of memory accesses in the worst case assuming a 1 bit at a time traversal of each trie. Finding worst case backtracking search times is non-trivial. We develop a new algorithm for doing this which is described in the Appendix.

We use a set of 5 industrial firewall databases that we obtained from various sites [4] for performance evaluation throughout the paper. Table 3 shows the number of rules in the databases. The rules in the firewall are specified either as exact match, or as prefix, or as ranges. In order to use a multiplane trie for filter classification, we need to convert all the rules to a prefix format. Rules specified as ranges are converted using the technique of [17]. Column 3 in Table 3 shows the number of rules after converting to the prefix format, which increases by a factor of 2 - 6 times.

The databases have the following characteristics:

[3] Prefix containment is the number of prefixes that are prefixes of a given prefix. For example, suppose we have the following prefixes 0*, 00*, 001*, and 000*. Then the prefix containment of 000* is 2, since 0* and 00* are both its prefixes.

[4] For privacy reasons, we cannot disclose the names of the companies.

- *Prefix containments:* In our databases, no prefix contains more than 4 matching prefixes for each dimension. Most prefixes contain 1, 2, or 3 matching prefixes only. We believe our performance results will be applicable for other filter databases that have a similar number of prefix containments.

- *Prefix Lengths:* The most popular source/destination prefix lengths are 0 (wildcard) and 32. There are also a number of prefixes with lengths 21, 23, 24, 28, and 30. This is very important for the performance of our compression algorithm, described in Section 6.1.

- *Port Ranges:* 5% - 10% of the filters have port fields specified as $\geq$ 1024 [5]. Such a range is converted into 6 prefixes using [17], which contributes heavily to the increase in the prefix rules. Our new compression algorithm, described in Section 6.1, will address this issue.

4.3.1 Performance Results

Our performance results for backtracking are summarized in Table 4. Note that our results are based on searching one bit at a time. A trivial extension is to search multiple bits at a time. Clearly, if we search 4 bits at a a time, then the memory accesses are reduced to $\frac{1}{4}$ of the values reported here but storage could increase by a factor of up to 16.

We compare three algorithms: basic backtracking, pruned backtracking, and pruned backtracking with the switch pointer optimization. We list the results for the best ordering in Table 4. (The best ordering for all forms of backtracking search is the order that minimizes memory accesses. As described in Section 4.2, there are a handful number of good orderings that can give comparable performance to the best performance reported here.)

As we can see, the three backtracking search algorithms have small memory requirements. The exact storage requirements differ by a small amount because the best ordering for the three schemes is not necessarily the same. For all five databases, even the basic backtracking cost is around twice the height of the trie or less. This is somewhat surprising, since backtracking is usually regarded as too slow for packet classification. Thus for real databases with limited number of prefix containments, we believe backtracking can be affordable in practice. Using say an 8 bits at a time trie traversal, backtracking requires around 18 - 26 memory accesses. This is better or as good as any firewall implementations we know using much less storage.

Figure 4 shows the histograms of the percentage-wise improvement over basic backtracking by using pruning and switch pointers for the five databases. As we can see, the performance gain of the two enhancements further reduce the number of memory probes by up to 25%.

5. REVISITING SET PRUNING TREES

Set pruning trees were initially proposed in [5] and briefly examined (and then discarded) in [17]. As with backtracking search, set pruning trees work using multiplane tries. Set pruning tries, however, differ from backtracking search tries by fully specifying all search paths so that no backtracking

[5]This is common because of the convention that the well known ports for standard services such as email etc, use port numbers $<$ 1024.

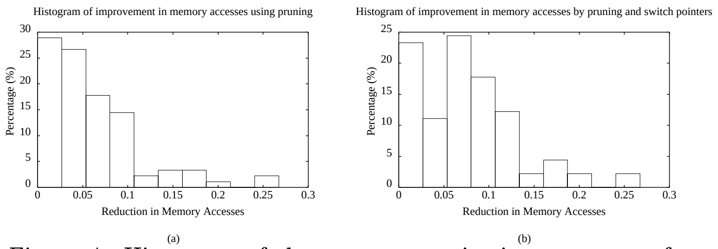


Figure 4: Histogram of the percentage-wise improvement of using the cost-based pruning and extended switch pointers (discussed in Section 4.2).

is necessary. However this is done at the cost of increasing storage. In the worst case, set pruning trees may take up to $O(N^K)$ storage.

We first review some standard terminology, and then explain the process of converting a backtracking search trie to its corresponding set pruning trie.

We say that string S' is a *descendant* of string S if S is a prefix of S'. We say that filter A is a descendent of filter B if for all dimensions $j = \{1, 2, ..., k\}$, string $A(j)$ is a descendant of $B(j)$. (Note that $A(j)$ is allowed to be equal to $B(j)$ and still be a descendant of $B(j)$.)

Converting a backtracking search trie to a set pruning trie is essentially replacing a general filter with its descendent filters. In other words, for every filter F, we "push" F down to all its descendent filters, and then delete F. For instance, in Figure 5, filter [*, *, *] is pushed down to places corresponding to [0*, 0*, *], [0*, 0*, 0*], [0*,1*, *], [1*, 0*, *], and [1*, 1*, *], and [1*, 1*, 1*], all of which are descendent filters of [*, *, *]. After [*, *, *] is pushed down, we can simply delete it, since it is now replaced with six more specific filters. As we can see, the pushdown step may potentially lead to memory blow up. In the worst case, we may need $O(N^K)$ storage for K dimensional filters.

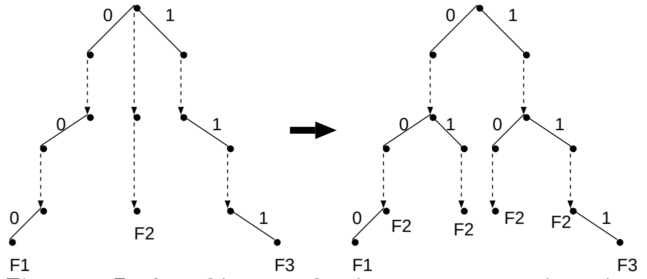


Figure 5: Backtracking search trie versus set pruning trie.

We now consider two optimizations to reduce storage in set pruning tries: the use of DAGs, and the use of optimal field orderings.

5.1 Optimizing Storage using DAGs

A natural technique to reduce the memory of set pruning trees is to change the *tree* structure of a multiplane set pruning trie to a *Directed Acylic Graph (DAG)*. This was first suggested in [5]. We illustrate the idea using a 3-dimensional trie. As shown in Figure 6, the tries that node A and B (both in the first field) are pointing to are identical. So instead of keeping several copies of identical tries, we only need to keep one copy, and have both nodes point to the same 2-dimensional trie. This is done at all field boundaries.

Database	Trie Depth	# Memory Accesses			Storage (# nodes)		
		BB	PB	SB	BB	PB	SB
Database 1	86	146	134	129	1848	2428	2306
Database 2	102	153	146	143	5061	5061	5061
Database 3	102	176	170	164	5367	5367	10567
Database 4	102	202	176	170	6785	26020	6692
Database 5	102	219	204	196	9441	9441	9441

Table 4: Performance of backtracking search using one 5-dimensional trie, where BB, PB, and SB stands for basic backtracking, pruned backtracking, and pruned backtracking with the extended switch pointer optimization (described in Section 4.2).

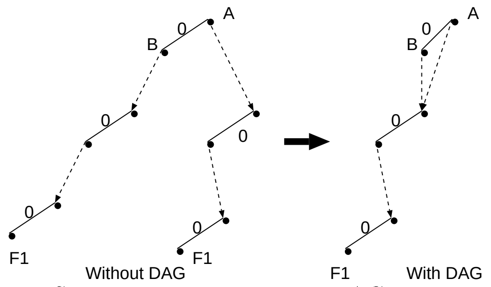


Figure 6: Set pruning trie with the DAG optimization.

The main technical problem is to decide when two tries are identical. We introduce a new procedure to do this. Our technique is general and works with all our other optimizations. Intuitively, two tries are identical to each other if and only if they have the same geometric shape, and the filter IDs attached at the leaves are the same. We use the following algorithm, whose pseudo-code is shown in Figure 7, to compare if two tries rooted at A and B are identical to each other.

In more detail, a trie node has two major components: pointers to the trie nodes in the same dimension (denoted as *Child*), and a pointer (denoted as *InfoPtr*) to the trie node in the next dimension. The only exception is that nodes at the last dimension/field point to filters. Any of these pointers can be NULL, meaning there are no children or no trie/filters attached to the pointer. Trie i is identical to trie j if and only if **(i)** the ith child of nodes A and B are identical, and **(ii)** the next dimension tries or filters that both nodes point to, if any, are identical. We use *LastField* to denote the last field in the trie.

```
Identical($A$, $B$) {
if (($A == NULL$) and ($B == NULL$))
    return TRUE;
if (($A.Field == LastField$) and ($B.Field == LastField$))
    return($InfoPtr(A) == InfoPtr(B)$);
if (!Identical($InfoPtr(A)$, $InfoPtr(B)$))
    return FALSE;
for ($i = 0$; $i < max\_children$; $i++$)
    if (!Identical($Child(A, i)$, $Child(B, i)$))
        return FALSE;
return TRUE;
}
```

Figure 7: Compare if two tries are identical.

Our results show the DAG optimization helps to reduce memory blow up by two orders of magnitude in the complete set pruning trie. We also experimented with using the DAG optimization at the same bit position *within* a field, but found that the additional saving was insignificant, usually around $5 - 10\%$. Note that DAG optimization can also be applied to backtracking search tries, but the amount of storage saving is much smaller. Moreover, since the memory requirement of backtracking search trie is linear to the number of filters, its storage cost is usually not a concern.

Database	$\frac{Worst\ Storage}{Best\ Storage}$
Database 1	4.1382
Database 2	2.4909
Database 3	2.5359
Database 4	2.4065
Database 5	2.1596

Table 5: Variation in storage using different order

5.2 Optimizing Storage using Field Order

As in backtracking search, we also find that field ordering affects storage requirements significantly. This is illustrated by the example shown in Figure 8. By exchanging the order of *Field*$_1$ and *Field*$_3$, the set pruning trie reduces from 11 nodes in Figure 8(a) to 7 nodes in Figure 8(b).

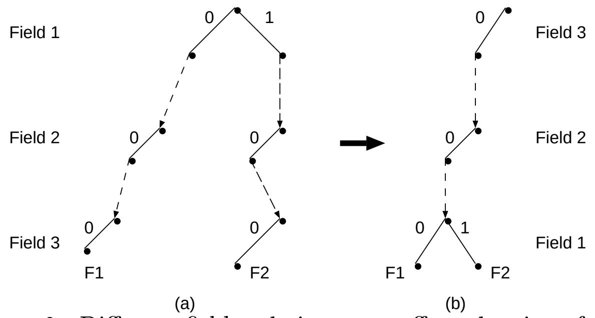


Figure 8: Different field orderings can affect the size of set pruning trees. significantly.

Table 5 shows the ratio between largest storage and smallest storage for the same database. As we can see, the best ordering cuts down the storage cost by a factor of 2 - 4. On the other hand, we find for each database there are a handful of orderings that give comparable performance (within 30% difference) to the best ordering. This suggests we can randomly try several orderings, and pick the best one. A further enhancement to this approach is to use some heuristic to preselect some good candidate ordering, such as placing the fields with fewest branches at the top, and those with most branches at the bottom. Then we can randomly pick a few of the selected orderings. Other heuristics suggested in [11, 8] can also be used.

5.3 Experimental Evaluation

Database	# memory accesses	Storage (# nodes)
Database 1	86	13188
Database 2	102	78604
Database 3	102	92356
Database 4	102	191411
Database 5	102	263882

Table 6: Performance of set pruning trees (with the DAG optimization) using one 5-dimensional trie

In this section, we experimentally evaluate the performance of set pruning trees using the same 5 industrial firewall databases described in Section 4.3.

We list the results for the best ordering in Table 6. (The best ordering for a set pruning tree is the order that minimizes storage.) Again our results are based on searching one bit at a time.

Compared to several forms of backtracking search as shown in Table 4, set pruning trees provide an optimal number of memory accesses at the cost of large storage requirement. In particular, the storage requirement increases to 7 - 28 times as large as what is minimally required by the corresponding backtracking search trie. Note that the databases 2, 3, 4, and 5 all have the same worst-case lookup time of 102, since this is the maximum number of nodes we can ever visit without backtracking [6].

6. COMPRESSION

Our first major new idea is to observe that we can further improve backtracking search and set pruning trees using a novel form of compression. A standard compression scheme for tries (e.g., [5]) is to remove all single branching paths so that no node has more than one child). Figure 9 shows an example: the algorithm compresses the trie on the left into the one on the right by collapsing multiple nodes into a single node when multiple edges succeed each other without any branching. For the performance of this standard compression algorithm, we have the following theorem.

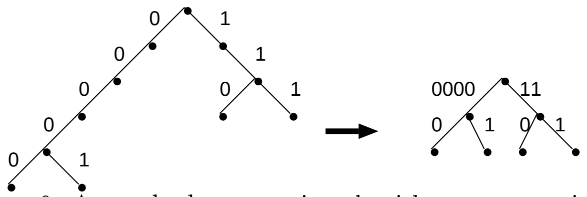


Figure 9: A standard compression algorithm: merge a single branch into one trie node.

THEOREM 6.1. *Consider a 1-dimensional trie with N leaf nodes, and only leaf nodes are associated with filters. After compression, it has $2N-1$ nodes.*

The proof is straight-forward. Suppose there are i internal nodes, and N leaf nodes. Then the total nodes in the trie is $i+N$. Since each node in the compressed trie has two children, the total nodes is also equal to $2i+1$. So there are $N-1$ internal nodes, and thus $2N-1$ nodes altogether.

[6]We have 32 bits each for source/destination address, and 16 bits each for source/destination port. In each database there are less than 16 different protocols, so we use 4 bits to distinguish them. Except the protocol trie, which we search 4 bits at a time, all the other tries are 1-bit at a time. So the maximum number of nodes we can visit without backtracking is 102 (including 5 roots in each dimension).

6.1 General Compression Algorithm

The standard compression scheme is efficient when there is no *redundancy* in the trie nodes. A trie has redundancy when many trie nodes have the same pointer value: either pointing to the same node in the next dimensional trie, or pointing to the same filter. Such redundancy especially arises when filters specified in ranges are converted into those specified in prefixes, or when a more general filter is pushed down to several more specific filters. The basic compression scheme cannot exploit such redundancy. For example, it fails to compress anything for the trie shown on Figure 10(a), since no node in the trie has only a single branch.

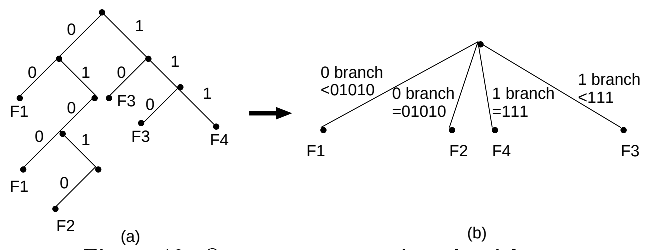


Figure 10: Our new compression algorithm.

However, a closer examination of the trie in Figure 10(a) reveals an interesting property: all nodes on the right of path 01010* (and also on the branches starting with 0) point to filter 1. If we can use range comparison as well as an equality test, we can compress all the branches starting with 0 by creating one *center branch* pointing to filter 2 with value 01010, and one *side branch* pointing to filter 1 with value < 01010. Similarly, we can compress all the branches starting with 1 by creating one *center branch* pointing to filter 4 with value 111, and one *side branch* pointing to filter 3 with value < 111. This leads to a more compact trie as shown on Figure 10(b).

To generalize the above example, if a path AB satisfies the following property, called the *Compressible Property*: (i) all nodes on its left point to the same place L (either the same filter or the same node in the trie), and (ii) all nodes on its right point to the same place R, then we can compress it as follows. Let $\delta(AB)$ denote the string labeling the path from node A to node B. We compress the entire branches by creating three edges: one *center branch* with value $\delta(AB)$ pointing to B, and one *side branch* with value $< \delta(AB)$ pointing to L, and another *side branch* with value $> \delta(AB)$ pointing to R.

To simplify the following discussion, we use the data structure shown in Figure 11 to represent a compressed node. We add 3 fields to the original uncompressed trie node: *value*, *len*, and *rangePtr* as shown in Figure 11. Since some of the elements may be empty, in practice we can have variable size trie nodes which are just large enough to hold the non-empty elements.

The way we use the above data structure is as follows. If the current input is 0, we check to see if it matches $value[0]$ after taking the appropriate bit mask. If so, we follow the pointer to $Child[0]$. Otherwise, we follow its $rangePtr[0]$ if it is less than $value[0]$, or $rangePtr[1]$ if it is larger than $value[0]$. Similarly for input 1. Figure 12 shows the pseudo-code.

We now examine the details of our general compression scheme.

```
#define MAX_CHILD 2 // for binary trie
struct NODE {
   struct NODE * Child[MAX_CHILD]; // center branch
   long value[MAX_CHILD];
   uchar len[MAX_CHILD];
   struct NODE * rangePtr[2*MAX_CHILD]; // side branch
   ... //other data members used in uncompressed trie node
}
```

Figure 11: Data structure for compressed node.

```
start = Level; end = Level + node−> len[currBit] − 1;
value = num[Field]&Mask[start][end];

if ( value == node−> value[currBit] )
   nextNode = node−> child[currBit];
else if (value < node−> value[currBit]) {
   nextNode = NULL;
   nextFieldNode = node−> rangePtr[2 * currBit];
}
else {
   nextNode = NULL;
   nextFieldNode = node−> rangePtr[2 * currBit + 1];
}
```

Figure 12: Search on the compressed trie (compressed using the general compression algorithm).

We need only consider the 1-dimensional case, since compressing a higher-dimensional trie can be achieved by compressing one dimension at a time.

A trie node has up to *MAX_CHILD* paths. Each of these paths can be compressed independently. (This is the same as the standard compression scheme.) So we only need to solve the problem of compressing one path. The basic idea is that at each node we try to decide whether the next immediate step to take can be compressed out. The next step is compressible if and only if it satisfies the invariant that all the nodes on the left of the center path (i.e. the path that will be converted to a center branch) have the same pointer value, and all the nodes on the right of the center path have the same pointer value. Below we refer to the invariant as the *compressible invariant*. To decide if the compressible invariant is maintained, we need to look at the characteristics of the node's *Child*. In particular, we classify a node into the following categories:

1. It has only one child, and none of its children are internal nodes (i.e. nodes whose *Child* are not empty).
2. It has more than one child, and none of its children are internal nodes.
3. Exactly one of its children is an internal node;
4. It has more than one child which are internal nodes.

It is clear that we cannot compress the nodes that are in case 4, since the compressed path can retain the information of either of the paths, but not both. For the other cases, we need to check further to make sure the *compressible invariant* holds. This involves two steps: (i) finding the center path, and (ii) verifying the invariant. The main issue is the first step, finding the center path. The second step is quite straight-forward once the center path is given. The center path is easy to identify in cases 1 and 3, since there is only one path we can possibly take. In case 1, the center path is the branch between the current node and its non-empty child node; in case 3, the center path is the branch between the current node and the child node that is an internal node. In case 2, there is more than one candidate, and we pick one that satisfies the compressible invariant. We omit further details for lack of space.

Analysis: We can show the above algorithm is correct, since for each step we take we maintain the compressible invariant. By induction, the paths that are compressed out will satisfy the compressible property.

As for the performance of the above algorithm, it is clear that Theorem 6.1 holds, since single-branching is a special case in which there is no node on the left/right of the center path. Therefore it satisfies the compressible property.

The general compression algorithm can perform much better than the standard scheme when there is redundancy in the trie nodes. As we described earlier, redundancy arises from range-to-prefix conversion or "pushing down" general filters. Below we quantify the performance of the general compression in the case of range-to-prefix conversion.

Range-to-prefix conversion can lead to a large increase in the number of rules. This is evident in Table 3. The number of rules increases up to a factor of 2 - 6 times using the conversion scheme in [17]. [8] introduces a novel technique that transforms a query of range search with N points in the range into a query of prefix search with $2N$ prefix rules. Our scheme is more general (for example, it works for set pruning tries which adds extra redundancy beyond that created by range to prefix conversions) and achieves the same effect as [8] for ranges.

More specifically, consider a set of 1-dimensional rules specified in a non-overlapping interval. (If there exist overlapping intervals, we first convert them into non-overlapping interval as shown in [8].) The collection of intervals can be specified as a series of left end points of the intervals in the sorted order. Suppose there are N points. The trie representing these N points can be compressed to $2N-1$ nodes using the general compression algorithm. This is stated in Theorem 6.2.

THEOREM 6.2. *For a trie representing N points in the range, we can compress it to* $2N-1$ *nodes using the general compression algorithm.*

We sketch the proof briefly. First, we build a binary trie using the N points. Then, we convert the ranges into prefixes using any range-to-prefix conversion, and build another binary trie using these prefixes. Comparing the two tries, we would find the only difference between the two tries is that the second one has more branches added around some paths. In particular, suppose a filter F_1 specifies $Field_1 \geq j$ (j is an integer, and its bit representation is S_j). Then the second trie has a path corresponding to S_j, which points to F_1; every node on the path that does not have a right child will have appended a right child pointing to F_1. Similarly, suppose a filter F_2 specifies $Field_1 \leq j$. Then the second trie has a path corresponding to S_j, which points to F_2; every node on the path that does not have a left child will have appended a left child pointing to F_2.

This is illustrated with an example shown in Figure 13 for filter rule ≥ 1024, a very common rule in firewall databases for the port field. As shown in Figure 13, trie 1 is built using the end point 1024, and trie 2 is built from the prefixes which ≥ 1024 is converted to. As we can see, the only difference between the two tries is that every node that does not have a right child in trie 1 is appended a right child in trie 2. According to our general compression algorithm, all these newly added branches in trie 2 can be compressed out. Therefore, the compressed version of the two tries are identical. Since the first trie has N leaf nodes, and only leaf nodes are associated with filters, the first trie after compression has $2 * N - 1$ nodes (according to Theorem 6.1).

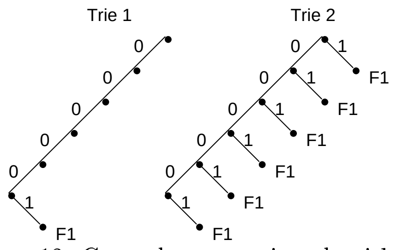


Figure 13: General compression algorithm.

6.2 Experimental results for Compression

In this section, we evaluate our new compression algorithm applied to both backtracking search and set pruning trees using the same five databases shown in Table 3.

We list the performance results for the best order (defined in Section 4.3.1 and Section 5.3). Compared with the performance results before compression, as shown in Table 4 and Table 6, it is evident that compression improves performance significantly. More specifically, compression reduces memory accesses and storage cost by a factor of 2 - 5 for backtracking search with and without cost based pruning. (We haven't implemented compression with switch pointers, but we expect similar performance gain as with the other types of backtracking search.) [7] For the set pruning trie, compression cuts down storage by a factor of 2.8 - 8.7, and cuts down lookup time by a factor of 1.6 - 4.

It is interesting to note that with compression, the lookup time of backtracking search is close to that of set pruning trees, and sometimes even better. This is because the high compression ratio of the paths in the backtracking search trie offsets the cost of a small number of backtracks. More specifically, as Table 7 shows, the total number of memory accesses is usually less than twice the height of the compressed tries. On the other hand, the compression ratio for the backtracking trie is much higher: mostly above 3 or more. This is also higher than the compression ratio of paths in its corresponding set pruning trie. To understand the reason behind this, consider the example shown in Figure 14. As the figure shows, we can compress the backtracking search trie on the left to the one on the right. However in the set pruning trie, filter 2 is pushed down to node N_2, making the path $N_1 - N_2 - N_3$ uncompressible. In general, "pushing down" filters may reduce the overall compression ratio. Careful readers may notice the compression ratio — i.e. the ratio between the storage before compression vs. after compression — of set pruning trees is no less than that of backtracking. This is because a significant part of compression in set pruning comes from the reduction in the number of paths as opposed to the reduction in the length of any individual path.

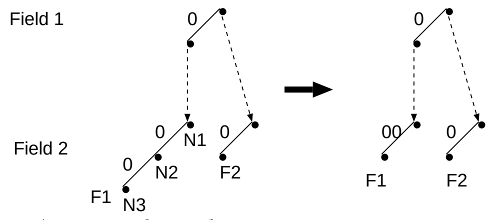


Figure 14: An example to show a path that is compressible in a backtracking search trie may become uncompressible in its corresponding set pruning trie.

We also compare the performance of backtracking search and set pruning trees with linear search. As we would expect, both backtracking search trie and linear search have low storage cost. On the other hand, the lookup time of linear search is 3 - 5 times larger than that required by backtracking search or set pruning trees, as shown in Table 7. If we use multi-bit tries, the performance of backtracking search and set pruning trees can be even better. [8] Furthermore, the lookup time of linear search increases linearly with the size of databases. In contrast, the lookup time of set pruning trees is constant, at most the maximum number of bits in the header fields; the lookup time of backtracking search is a constant factor of that required by the set pruning trees for databases with limited prefix containments. Therefore both backtracking search and set pruning trees have more scalable lookup time than linear search.

As a final comment, search on our compressed tries involves both equality test and range comparison. Thus the CPU cost per node is a little higher with the compressed trie. Without special optimization, the CPU time spent per node almost doubles after compression [9]. However, the total CPU time is actually less because compression cuts down the number of nodes visited by mostly 3 times or more as shown in Table 7. Moreover, the query time is dominated by the memory accesses, which is much less after the compression. Therefore the performance gain of compression is almost the amount of the reduction in memory accesses, and the additional CPU overhead with compression is negligible.

In summary, in this section, we described a new compression algorithm that can explore the redundancy in the multiplane tries effectively. We evaluate the performance of the compression scheme, and show that it can reduce the storage cost and lookup time by a factor of 2 - 8.

7. TRADING STORAGE FOR TIME

In the previous section, we showed that real databases contain significant structure, and that simple mechanisms like backtracking search and set pruning trees can perform much better than the worst case bounds, especially using several op-

[7]Since the compressed trie nodes are bigger than standard trie nodes, each access to a trie node should strictly be charged twice the number of memory accesses shown (to access the value and then follow the pointer). However, since most processors prefetch a whole cache line, the second access should be essentially zero cost as long as the node fits in a cache line. We use the same assumption for linear search.

[8]We are working on using multi-bit lookup on the compressed trie. We expect using k bits at a time, the speedup in the lookup time of backtracking search will be close to, but smaller than, a factor of k.

[9]The performance result is based on looking up one header filter 1000000 times on an otherwise idle UNIX machine. We conducted the experiments for different header filters and using different databases, and the performance is similar.

Database	# memory accesses				Storage (# nodes)			
	Set Pruning	BB	PB	Linear Search	Set Pruning	BB	PB	Linear Search
Database 1	22	30 (14)	28 (14)	67	1510	429	429	67
Database 2	46	51 (33)	42 (33)	158	22049	912	933	158
Database 3	49	49 (25)	43 (44)	183	32886	1261	1666	183
Database 4	64	99 (35)	85 (35)	279	52093	2831	4147	279
Database 5	45	59 (27)	58 (27)	266	81924	2815	2815	266

Table 7: Performance of compressing backtracking search and set pruning trees (with the DAG optimization) using one 5-dimensional trie, where BB and PB stand for basic backtracking and pruned backtracking (described in Section 4.2). The numbers in the parentheses are the height of the compressed trie. Sometimes the height is smaller than the memory accesses of pruned backtracking, since some nodes can not be reached after pruning. For linear search, we assume each filter fits in a cache line, and a filter rule takes 1 memory accesses.

timization techniques (in particular, generalized compression) we proposed. These two algorithms are at two extremes: one has small storage requirement with suboptimal lookup times, and the other offers good lookup time at the expense of a suboptimal memory requirement. Ideally we would like to have a smooth tradeoff between the two extremes. That is, if we can afford larger memory, we would like to have correspondingly better lookup times. Similarly if the lookup time requirement is relatively large, we would like to be able to use cheap machines (with less memory) to do filter search. Such a *tunable algorithm* would give designers more choices and flexibility.

Note that, with compression, backtracking search may sometimes have better lookup time than set pruning trees. In this case, we can still use the same technique, described below, to tradeoff memory for lookup time. The only difference is that compressed set pruning trees are no longer in the tradeoff region, for they neither yield the best storage requirement nor the best lookup time.

7.1 Selective Pushing

As we have seen earlier in Section 5, set pruning tries eliminate all backtracking by "pushing" down *all* filters to its descendents. So searching for a filter in a set pruning trie is simply searching for the longest matching prefix in every dimension. There is no need to do any backtracking. An important observation is that eliminating all backtracking can be potentially storage intensive. If we are willing to afford small amounts of backtracking, however, we can "push" down fewer filters, which can reduce storage requirements. Below we describe a heuristic called *selective pushing* which decides which filters to push down.

The basic idea is that we only "push down" the filters with high worst case backtracking times, and leave the other filters intact. The code in Figure 15 shows the skeleton of the selective push algorithm. Basically it computes the search cost for each header class, where a header class (defined in Appendix) corresponds to a search path in the set pruning trie. If the search time for the header class (itself represented as a filter) exceeds our required time bound, then we insert the filter into the trie. Note that the header class filters may be different from the original filters in the database.

After filter F is inserted, then our search for F will be exactly the same as in a set pruning trie: we simply search for the longest prefix match in each dimension, and the leaf will be the matching filter; no backtracking is necessary. Therefore we must annotate the leaf of the pushed down filter to indicate that there is no need for backtracking search after encountering this leaf.

```
foreach header class (represented as Filter(i))
    cost = BacktrackSearch(trie,header);
    if ($cost > bound$)
        insertFilter(trie,Filter(i));
        annotate the leaf so that search can stop at this leaf
end
```

Figure 15: Find worst-case search time.

Pushing down a filter makes search time for that particular filter $O(KW)$, where K is the number of fields, and W is the maximum length of any field. However, as a side effect, adding some more paths to the trie (during the pushdown) may make searching for some other filters longer.

For the example shown in Figure 16, if we require the lookup time of all filters to be no more than 11 memory accesses, then the filter [0*, 0*, 000*] with search cost of 12 memory accesses is pushed down as shown in Figure 16(b). However this makes the lookup time of filter [0*, 0*, 001*] increase from 11 to 12 memory accesses, so we also need to push down the filter [0*, 0*, 001*] as shown in Figure 16(c). Therefore we need to *iteratively* push down: we first get rid of the longest path; if this push-down produces new long paths, then we need to get rid of these as well. The algorithm stops either when the worst case lookup time is below the required time bound (specified as an input), or the memory grows to the size of a set pruning trie, corresponding to the state where all filters get pushed down.

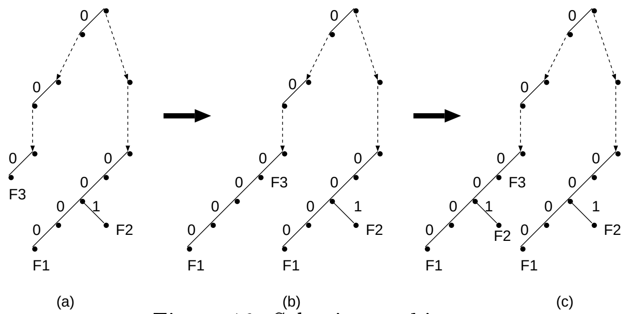


Figure 16: Selective pushing.

Selective pushing can be applied to both uncompressed and compressed tries. The side effects of applying selective pushing to the compressed trie are two fold: (i) adding more branches may increase the search time for other filters as shown in Figure 16, and (ii) adding more branches may reduce the compression ratio, which may in turn increase search time for a large number of filters. Therefore we need to be more conservative when applying selective pushing to a compressed trie. Our experiments suggest that a better heuristic in this case is to push the filters with largest search time in the current iteration, and do it iteratively.

7.2 Performance Results of Selective Pushing

We evaluate the performance of selective pushing applied to both uncompressed and compressed tries using the same 5 databases. For the following evaluation, we pick some arbitrary field orderings.

For the uncompressed trie, we apply selective pushing on basic backtracking augmented with cost-based pruning and extended switch pointers, described in Section 4.2. The results are shown in Figure 17. As we can see, we can reduce the query lookup time with little increase in storage when the lookup time is large. For example, for database 1, there is almost no increase in storage when the lookup time is reduced from 144 to 130 memory probes. Further decrease in the lookup time is achieved at higher cost in storage. Similar behavior is observed for database 2. Also for both databases, when the lookup time comes close to the optimal, the storage costs saturate at those of the corresponding set pruning tree, as we would expect.

For the compressed trie, we evaluate the performance of selective pushing on basic backtracking search. (We haven't implemented it for other types of backtracking, but we believe the performance should be similar if not better.) The results are shown in Figure 17. As before, when the lookup time is large, it drops rapidly at very small cost in storage. For instance, in database 1, we reduce the lookup time from 110 to 87 with little extra storage. Further decrease in lookup time incurs higher cost in storage after we reach the "knee" of the curve. The lookup time and storage cost eventually saturates at those of its corresponding set pruning trees. For database 2, we reduce the lookup time from 71 to 66 with little increase in storage. Further decrease in lookup time is achieved at much higher cost in storage.

To conclude, in this section we proposed a new mechanism, called selective pushing, to smoothly tradeoff storage for lookup time. We find selective pushing is useful to improve backtracking search times by around 10-25% with only a small increase in storage. Once the knee of the tradeoff curve is reached, however, further reduction in lookup time is achieved at the cost of a large increase in the storage.

8. USING MINIMAL HARDWARE

Although this paper has largely focused on software techniques for firewalls, an interesting question is the sensitivity of our results to the availability of a small amount of hardware support. For example, if we assume that small ternary CAMs are available for a small number of filters, which filters should we move to the ternary CAM. Intuitively, we might expect that moving filters with a large number of wildcarded filters should help. However, we show below that the choice of which filters should be moved to hardware may depend on what metric we wish to optimize.

Let us first consider the policies for reducing storage cost. We propose two heuristics. (1) Remove from the software the filters with the largest number of wildcarded fields. For filters with the same number of wildcarded fields, we pick randomly among them. (2) Remove from the software the filters that occur at largest number of nodes in the set pruning trees. The intuition behind the first heuristic is that filters with a large number of wildcarded fields are more likely to lead to memory blow up during the "push down" stage. For example, filter [*, *, *, *, *] is pushed almost everywhere in the set pruning trie, which contributes a large portion of storage cost. The second heuristic tries to quantify the storage cost for each filter rule by counting the exact number of occurrences of each filter in the set pruning trees.

We evaluate performance using both heuristics on set pruning trees. Figure 19(a) shows the results for uncompressed set pruning trees. It has four curves using heuristic 1 with different random seed, and one curve using heuristic 2. As we can see, using both heuristics, the storage cost initially drops rapidly as more filter rules are removed from software. For example, using heuristic 2, the storage cost drops to $\frac{2}{3}$ after removing 10 rules, and less than one half after removing 20 rules. As we would expect, heuristic 2 out-performs heuristic 1, since the former quantifies the storage taken by each filter, while the latter just gives a coarse estimation based on the number of wildcarded fields. Figure 19(b) shows the results for compressed set pruning trees. As before, the storage decreases rapidly as more filters are removed. On the other hand, heuristic 2 no longer out-perform heuristic 1. We conjecture that this is because neither heuristic takes into account the compression ratio, which is critical to the performance of compressed tries. We are currently investigating better heuristics for compressed tries.

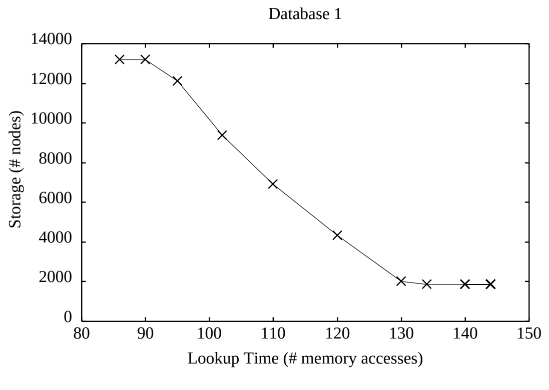
To optimize the query lookup time, we propose two heuristics. (1) Remove from the software the filters with the largest number of wildcarded fields. For filters with the same number of wildcarded fields, we just pick randomly among them. (2) Remove from the software the filter rules that incur the largest number of memory accesses.

We apply the heuristics to backtracking search, and Figure 20 summarizes our results. As shown in Figure 20(a), there is a big difference in performance between the two heuristics. The lookup time drops relatively fast when heuristic 2 is used. For example, removing 10 filter rules helps to reduce the lookup time by 15%. In comparison, the lookup time changes very little when heuristic 1 is used. We also apply the heuristics to the compressed tries. As shown in Figure 20(b), the reduction in lookup time is marginal after removing the first 20 filter rules. We are currently investigating better heuristics for compressed trie to take into account of compression ratio.

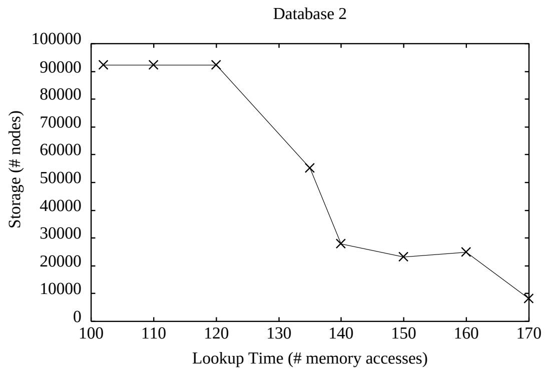
Compared with the reduction in storage, it is evident that reduction in lookup time by removing filters from software is less significant. This is because a small number of filter rules consume a large portion of the storage, while a considerable number of filter rules have relatively large worst-case lookup time. This is confirmed using the probability density function of storage and worst case lookup time taken by filter rules. (We omit the figures in the interest of brevity.)

9. CONCLUSION

This paper has four contributions. First, we showed experimentally that the performance of simple trie based filter schemes

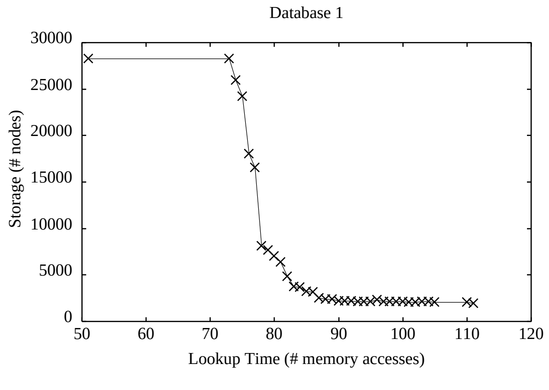


(a)

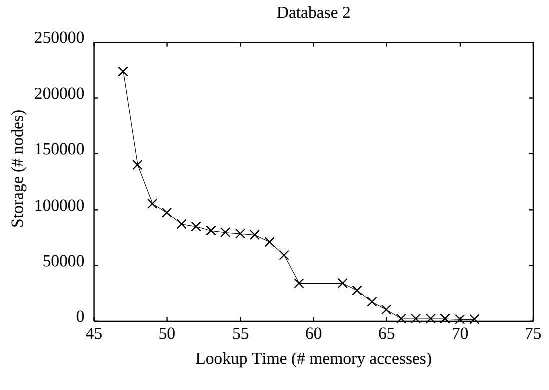


(b)

Figure 17: Selective pushing applied to uncompressed trie.

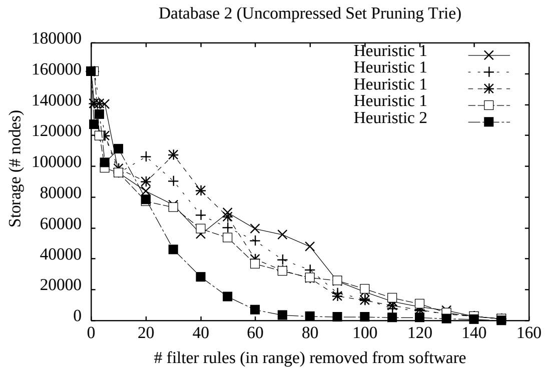


(a)

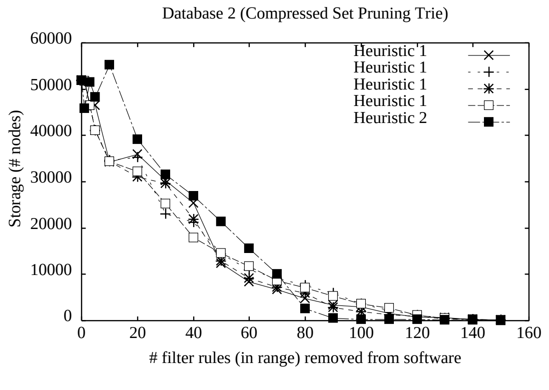


(b)

Figure 18: Selective pushing applied to compressed trie.

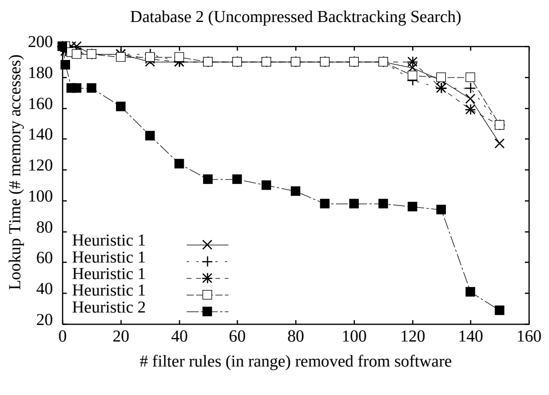


(a)

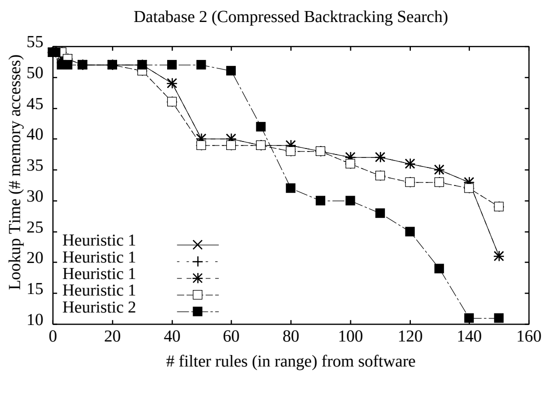


(b)

Figure 19: Moving a subset of filters from software to hardware for storage savings.



(a)



(b)

Figure 20: Moving a subset of filters from software to hardware for better lookup time.

Approach	Description	Performance Gain
Optimize backtrack search	optimal field ordering	a factor of 2 saving in lookup time
	cost-based pruning	5% - 25% saving in lookup time
	switch pointer	5% - 25% saving in lookup time
Optimize set pruning trees	DAG	two orders of magnitude saving in storage
	optimal field ordering	a factor of 2 - 4 saving in storage
New compression algorithm	compress multiplane tries and effectively exploit redundancy in trie nodes	a factor of 2 - 5 saving in lookup time, a factor of 2.8 - 8.7 saving in storage
Selective pushing	"push down" the filters with high worst case backtracking times to speed up lookup time	10 - 25% decrease in time with only a small increase in storage
Removal of filters from software	heuristics to remove a small number of filters to improve storage/lookup time	removing 10 - 20 rules cuts storage cost by $\frac{1}{3}$ - $\frac{1}{2}$, or lookup time by $10\% - 20\%$

Table 8: Summary of techniques introduced in the paper, and their performance gain for the firewall databases we used.

(together with simple optimizations) is much better than worst-case figures predict. Second, we proposed a novel compression algorithm that further reduces the lookup time and storage cost. Third, we introduced a simple mechanism for trading memory to improve the search time of backtracking search. Finally, we also investigated ways to effectively utilize hardware by moving a small subset of filters rules from software to hardware. Our contributions are summarized in Table 8. Based on our results, we make five observations.

First, we observe that in existing databases simple backtracking search (together with multi-bit trie traversal and compression) works quite well, providing search times within a factor of two of optimal. In particular, simple backtracking search should be adequate for software firewall implementations because it is fast and also has fast insertion times. Inserting filter F simply involves first placing the first field of F in the first field trie, and then placing the second field of F in the second field trie pointed to by the leaf of the first field trie, etc. Thus insertion is as fast as K insertions into a standard trie.

Of course, the backtracking optimizations we introduced (such as using switch pointers at the lowest two dimensions, finding optimal field orderings, and cost pruning) will slow down insertion. But the contributions of these optimizations to lowering search time are small enough to be ignored if insertion times are important.

A second observation is that set pruning trees (with the DAG and other optimizations) come close in storage performance to the RFC [10] scheme. Unfortunately, we could not do a head-on comparison with identical databases. Although the two schemes look similar, it is not possible to theoretically compare them in all cases. If the RFC scheme uses a simple linear combining tree, then the DAG scheme can emulate the performance of RFC. However, this is not true for more general RFC combining trees; RFC allows fields to be combined in a tree form unlike set pruning trees. On the other hand, unlike RFC, a set pruning trie needs storage for only those filters that have not been eliminated so far in the path. Thus one can construct examples of filter databases where RFC takes less storage, and databases where set pruning tries take less storage. A more fruitful approach would be to experimentally compare the two schemes.

A third observation is that our compression algorithm can cut down the lookup time and storage by an order of magnitude. This is done by effectively exploiting redundancy in multiplane tries.

Fourth, we observe that selective pushing allows a tunable storage-time tradeoff. Selective pushing is useful to improve backtracking search times (for both uncompressed and compressed tries) by around 10 - 25% with only small increases in storage. Once the knee of the tradeoff curve is reached however, the storage needs climb rapidly for even small decreases in search time, quickly reaching the storage of set pruning trees. However, we have only used a simple greedy heuristic. We are currently investigating better tradeoffs using optimization techniques.

Finally, we find by removing only a small number of filters from software to hardware, we can substantially cut down the storage requirements and the search times of the software approach. This enables us to effectively take advantage of any limited hardware when it is available. Our schemes can also potentially be useful for hardware implementations. Even backtracking search can be pipelined across S stages using at most a factor of S increase in storage.

10. REFERENCES

[1] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proc. of the 1 st Symposium on Operating System Design and Implementation*, November 1994.

[2] M. M. Buddhikot, S. Suri, and M. Waldvogel. Space Decomposition Techniques for Fast Layer-4 Switching. IFIP Sixth International Workshop on Protocols For High-Speed Networks, Aug. 1999.

[3] B. Chazelle. Lower bounds for orthogonal range searching, I: The reporting case. *Journal of the ACM*, 37, pp. 200-212, 1990.

[4] B. Chazelle. Lower bounds for orthogonal range searching, II: The Arithmetic model. *Journal of the ACM*, 37, pp. 439-463, 1990.

[5] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner. Router Plugins A Software Architecture for Next Generation Routers. In *Proc. of SIGCOMM*, Sept. 1998.

[6] IETF Differentiated Services Working Group Home Page, http://www.ietf.org/html.charters/diffserv-charter.html

[7] D. R. Engler, and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proc. of SIGCOMM*, Aug. 1996.

[8] A. Feldmann, and S. Muthukrishnan. Tradeoffs for Packet Clasification. In *Proc. INFOCOM*, March 2000.

[9] T. V. Lakshman and D. Stidialis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *Proc. of SIGCOMM*, pp. 191-202, Sept. 1998.

[10] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proc. of SIGCOMM*, pp. 147-160, Sept. 1999.

[11] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings, Proceedings Hot Interconnects VII, Aug. 1999.

[12] S. McCanne, and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan. 1993), USENIX.

[13] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek. The Click Modular Router. In *17th ACM Symposium on Operating Systems Principles*, Dec. 1999.

[14] P. Newman, G. Minshall, and L. Huston. IP Switching and Gigabit Routers. In *IEEE Communications Magazine*, Jan. 1997.

[15] C. Partridge. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis*, San Diego, CA, USA, Feb. 1996.

[16] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *Proc. of SIGCOMM*, pp. 135 - 146, Sept. 1999.

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. In *Proc. of SIGCOMM*, pp. 203-214, Sept. 1998.

[18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of SIGCOMM*, Sept. 1997.

[19] J. Xu, M. Singhal J. Degroat, A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds. In Proc of Infocom, March 2000.

APPENDIX

A. COMPUTING BACKTRACKING SEARCH TIME

Since we care about worst-case search times, we need to compute the worst-case search time for backtracking search on a given filter database. It may appear trivial to find the worst case search times in a backtracking search trie by using a bottom-up procedure and annotating each node with the worst-case search time downwards from the node. Then the worst-case search time for a parent node P might be conjectured to be the sum of two quantities: first, the worst-case search time of any child of P (computed recursively); and second, the worst-case search time to do a backtracking search starting at P and moving to some next dimensional trie pointed to by P. Unfortunately, the two worst-case times can occur with different packet headers. Thus this simple procedure will overestimate search times.

One way to find the true worst case search cost is to compute the search cost for each possible header. If there are K fields, each with length W, then the number of possible headers is $O(2^{KW})$, which is prohibitive. Therefore it is necessary to design an algorithm to reduce the header space into a smaller, but equivalent, space. Our reduction algorithm is based on the following observation. Many of these headers follow the same search path in backtracking search. If the search path two headers take are the same during backtracking search, then they are indistinguishable as far as cost is concerned. Therefore we divide the whole header space into a number of classes. Two headers belong to the same class if and only if they follow the same paths in backtracking search. We only need to find the maximum search cost across all header classes.

It is not difficult to see that two headers will follow the same path in backtracking search if and only if they follow the same path in the set pruning trie. Therefore, we can represent a header class as a filter describing its search path in the set pruning trie. Different search paths represent different header classes. For the example shown in Figure 21(a), the header classes are [00*,0*], [00*,1*], and [01*,1*].

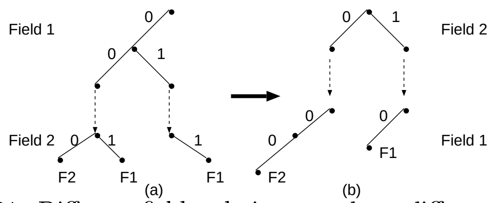


Figure 21: Different field orderings can have different header classes significantly.

A few comments follow. First, there is no overlap between different header classes. That is, if there exists a packet that belongs to header class A and B, then $A = B$. Second, the union of all header classes is the entire search space the filter database covers. So we have essentially converted an $O(2^{KW})$ search space into a smaller space represented by header classes. Third, the header class is specific to the ordering of different fields. For the example in Figure 21, if we swap $Field_1$ and $Field_2$ shown in Figure 21(b), then the header class changes to two header classes: [0*, 00*] and [1*, 0*]. Fourth, since constructing the set pruning trees is done offline for performance evaluation purposes, the memory blowup is less of a concern. (If storage is a concern, one can partially push filters.)

Finally, having obtained the header classes, we can now find the worst cast lookup time in backtracking by using backtracking search to search for every header class, and recording the largest lookup time.