Fast Firewall Implementations for Software and Hardware-based Routers

Lili Qiu
liliq@microsoft.com
Microsoft Research

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

Subhash Suri
suri@cs.ucsb.edu
University of California, Santa Barbara

Technical Report
MSR-TR-2001-61

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Routers must perform packet classification at high speeds to efficiently implement functions such as firewalls and diffserv. Classification can be based on an arbitrary number of fields in the packet header. Performing classification quickly on an arbitrary number of fields is known to be difficult, and has poor worst-case complexity.

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 tries. 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 tries (using a DAG optimization) have much better storage costs than the worst-case bound. We also propose several new techniques to further improve the two basic mechanisms. Our major ideas are (i) backtracking search on a small memory budget, (ii) a novel compression algorithm, (iii) pipelining the search, (iv) the ability to trade-off smoothly between backtracking and set pruning, and (v) 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 [3] 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. 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.

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 source/destination IP addresses, source/destinations port numbers, 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 ordering 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 (content addressable memory) or other ASICs that perform parallel linear search (e.g., [5]). Such hardware solutions do not scale to large filter databases. Other solutions reported in literature that can be implemented in software (e.g., [9, 6]) are either slow or take too much storage. With the advent of software based routers (e.g., [8]), which are typically aimed at the edge router space where classification is particularly important, it is necessary 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 [5, 9]. However recent research [6, 7, 10] 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 [10] and Recursive Flow Classification [6] 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 two of the simplest packet classification mechanisms: *backtracking search* and *set pruning tries*. 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 tries have minimal search times but have poor worst-case storage. Earlier researchers have dismissed backtracking search as being too slow [10], and dismissed set pruning tries as being suitable only for very small packet classifiers [2, 10].

However, we find that using real databases the time for backtracking search is much better than the worst-

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

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 tries (using a DAG optimization) have much better storage costs than the worst-case bound of N^k indicates.

We also propose several novel techniques to further improve the performance of backtracking and set pruning tries. First, most designers assume that backtracking is infeasible to implement in hardware because of its high computational state (large number of registers) and its inability to be pipelined (without having memory replicated for each pipeline stage). We show, perhaps surprisingly, that both these assumptions are false. We introduce a new backtracking search technique, which reduces the hardware register cost for doing backtracking in hardware to $D + 1$, where D is the number of dimensions. We also show that while some memory must be replicated at different stages, one can pipeline real firewall databases with a very small amount of replicated memory.

Second, 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 reduces the lookup time by a factor of 2 - 5, and reduces the storage by a factor of 4 - 12.

Third, given that backtracking search and set pruning tries 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 tradeoff 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 increase 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 (10% - 20%) from software for hardware lookup, the storage requirement and query lookup time for the software approach (i.e. backtracking search and set pruning tries) 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 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 examine pipelining the search, and evaluate its storage cost. In Section 8 we explore a tradeoff between time and space by starting with backtracking search and using selective pushing. In Section 9 we investigate ways to effectively make use of a limited amount of hardware if it is available. We conclude in Section 10.

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 [10], 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 least cost filter 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.

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

[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.

whole header. Caching may work well and have high hit rates [12] 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.

[5] 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. [10] 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.

[6] proposes a simple multi-stage classification algorithm, called recursive flow classification (RFC). RFC exploits the structure 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. [9] 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. [7] suggests another heuristic based on geometrically partitioning the classification space, which produces fairly good search times and requires less memory than [6].

[4] 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 8 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 a simple algorithm: backtracking search. We start by reviewing the basic mechanism, and then show how it can be augmented by simple optimizations.

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.

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 straight-forward to extend a single dimensional trie to multiple dimensions [10]. 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.

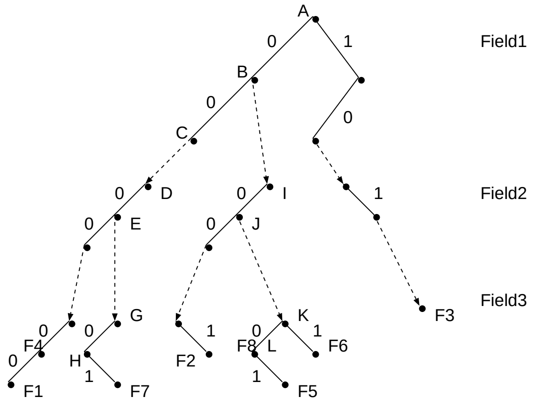

Figure 1: Backtracking search trie.

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.

Throughout the paper, the storage cost is assessed as the total number of nodes in the trie, and the lookup time is assessed as the total number of nodes visited during the search, which corresponds to the total number of memory references during the search. 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 backtrack 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 [10], 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 backtracking on a small memory budget, optimal field ordering, pruning based on cost, and generalizing switch pointers [10].

Backtracking using limited Computation State: Backtracking is considered expensive (especially in hardware where the backtracking state has to be kept in limited register memory) because of the potential need to keep state for all potential backtrack points. We show that for a tree that does backtracking in multiple dimensions, we only need state proportional to the number of dimensions in which there are trees (e.g., 6 registers for the common case of IP 5-tuples).

Consider the backtracking trie shown in Figure 2. There are filters $(D1, S1, P1)$, $(D1, S2, P1)$, $(D2, S2, P1)$, $(D3, S1, P1)$, and $(D3, S2, P1)$. Assume also that $D1$ is a prefix of $D2$ which is a prefix of $D3$, and that $S1$ is a prefix of $S2$.

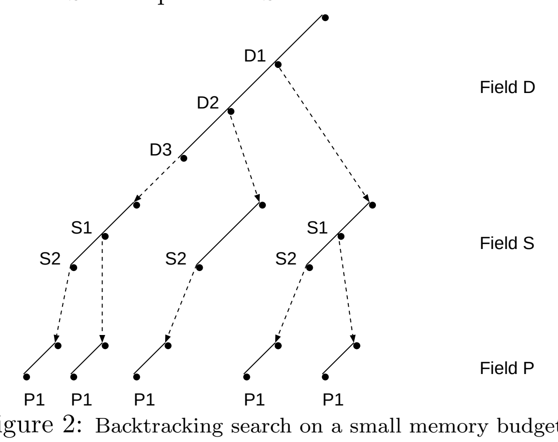

Figure 2: Backtracking search on a small memory budget.

Our usual way of doing backtracking search for a header that matches $(D3, S2, P1)$ is to first traverse the D trie, and remember all matches $D3$, $D2$, and $D1$. Then we traverse the S trie for $D3$ and remember $S1$ and $S2$. In general, the amount we have to remember at each dimension is equal to the number of matches in each dimension which can be $W * D$, where W is the average number of matches per dimensions, and D is the number of dimensions.

We now describe a solution that takes $D + 1$ pieces of state. The key modification is that instead of doing depth-first traversal while remembering all past backtracking points, we go to the first trail we see, and only then return for more exploration. In other words, we visit more general filters first, and more specific filters later. In the previous example, we will explore the D trie until we reach $D1$. Since we see a pointer to a S trie here, we immediately follow it. When we reach $S1$, we see another trail to follow to P trie, we start walking P trie until we reach $P1$. When we come to the end of that (assume no match beyond $P1$), we have to do our first backtrack.

We start walking the S trie for $D1$ further till we hit $S2$. Finally, after walking the P trie for $S2$, we once again try to walk the S trie for $D1$, and fail (no matches beyond $S2$ in the S trie). Then we back up and try to walk the D trie further. Thus the order of filters visited is $(D1, S1, P1)$, $(D1, S2, P1)$, $(D2, S2, P1)$, $(D3, S1, P1)$, and $(D3, S2, P1)$. The standard method visits them exactly in reverse order. While the standard scheme does not keep a lot of state in this simple example, it can in the worst-case. In contrast, the modified backtracking search only takes $D + 1$ pieces of state, since we only need to keep track of the current dimension being explored, the pointers to the roots of the tries (in all dimensions) on the current search path, and the bit positions in the header to start the trie search. Since bit positions only take 5 bits and pointers are less than 27 bits, we can easily keep the state information in a set of $D + 1$ 32-bit registers.

Optimal Field Ordering: We have observed that the lookup time in backtracking search is sensitive to the ordering of fields in the trie. This is illustrated in the following example. Figure 3(a) and (b) correspond to the same database. The only difference is the field ordering: Figure 3(b) exchanges the order of $Field_1$ and $Field_3$ in Figure 3(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 3(a), and only 11 in Figure 3(b).

Based on this observation, we can optimize backtracking lookup time by choosing the best field ordering. A straightforward way to find the best ordering 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 al-

Database	# Rules	$\frac{worst\ time}{best\ time}$
Database 1	67	111.64%
Database 2	158	164.71%
Database 3	183	136.09%
Database 4	279	146.53%
Database 5	266	134.13%

Table 2: Different field ordering affects the lookup time.

ways examine the protocol field first. This leaves only 24 possible field orderings to be examined.

We evaluated the effect of different ordering on the query lookup time using real 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, the difference between the best and the worst orderings ranges from 10% to 65%. 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 ordering that places the field with the most prefix containment [3] as the last field. For example, this heuristic would prefer Figure 3(b) over Figure 3(a) because Field 1 has more prefix containments than the other fields. Other heuristics were suggested in [7, 4].

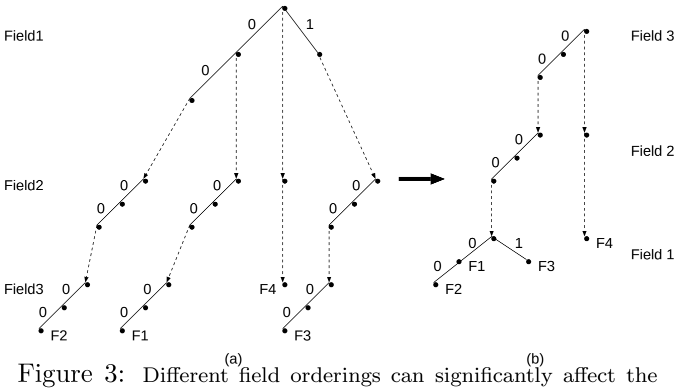

Figure 3: Different field orderings can significantly affect the lookup time in backtracking search.

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 [6, 1].).

For example, consider searching for a filter [00*, 00*, 00*] in the trie shown in Figure 4 using the standard backtracking search [4]. 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.

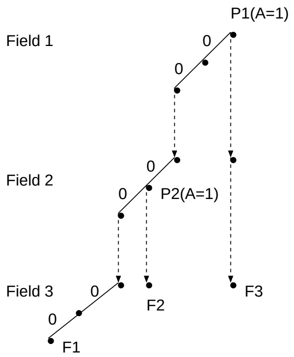

Figure 4: Pruned backtracking search.

Switch Pointers: We further optimize backtracking search with switch pointers. Switch pointers were introduced in [10] 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.

[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] Pruning based on cost is also applicable to our modified backtracking search on a small memory budget, but may prune different branches due to different search orders. The search results in both cases should be the same.

Database	# Rules	# Rules in the prefix format
Database 1	67	127
Database 2	158	418
Database 3	183	531
Database 4	279	949
Database 5	266	1640

Table 3: Firewall databases.

We develop a new algorithm for doing this which is described in the Appendix A.

We use a set of 5 industrial firewall databases that we obtained from various sites for performance evaluation throughout the paper. For privacy reasons, we are not allowed to disclose the names of the companies, or the actual databases. 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 [10]. Column 3 in Table 3 shows the number of rules after converting to the prefix format. The conversion leads to a factor of 2 - 6 increase in the number of rules. Our new compression algorithm, described in Section 6.1, will address this issue.

The databases have the following characteristics:

- *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 ≥ 1024 [5]. Such a range is converted into 6 prefixes using [10], which contributes heavily to the increase in the prefix rules. This will be addressed by our compression algorithm, described in Section 6.1.

- *IP addresses:* The destination and source prefix fields have roughly half the rules that are wildcarded.

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

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 simple 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 [6]. We list the results for the best ordering in Table 4. (The best ordering for all forms of backtracking search is the ordering 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 while using much less storage.

5 Revisiting Set Pruning Tries

Set pruning tries were initially proposed in [2] and briefly examined (and then discarded) in [10]. As with backtracking search, set pruning tries work using multiplane tries. Set pruning tries, however, differ from backtracking search tries by fully specifying all search paths so that no backtracking is necessary. However this is done at the cost of increasing storage. In the worst-case, set pruning tries 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 prun-

[6]Our evaluation is based on the standard backtracking search. The results of modified backtracking search on a small memory budget are the same for basic backtracking, and similar for the other types of backtracking.

Database	Trie Depth	# Memory Accesses			Storage (# nodes)		
		BB	PB	SB	BB	PB	SB
Database 1	86	146	119	117	1848	1782	1782
Database 2	102	153	145	143	4914	4914	4914
Database 3	102	169	149	149	3949	3743	3743
Database 4	102	202	170	170	6785	26006	6630
Database 5	102	208	198	196	6555	13724	16493

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

ing 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.

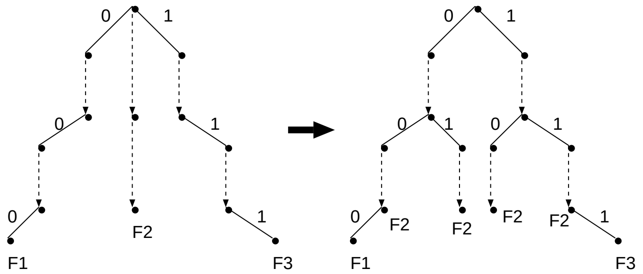

Figure 5: Backtracking search trie versus set pruning trie.

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 tries is to change the *tree* structure of a multiplane set pruning trie to a *Directed Acylic Graph (DAG)*. This was first suggested in [2]. 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.

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

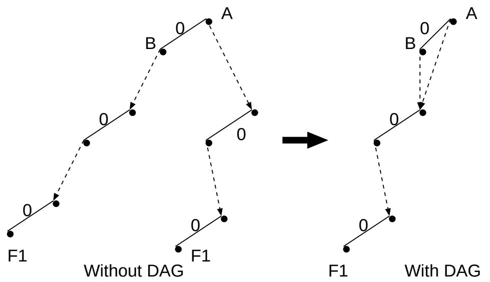

Figure 6: Set pruning trie with the DAG optimization.

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 in the number of filters, its storage cost is usually not a concern.

5.2 Optimizing Storage using Field Ordering

As in backtracking search, we also find that field ordering affects storage requirements significantly. This is illustrated by the example shown in Figure 7. By exchanging the order of $Field_1$ and $Field_3$, the set pruning trie reduces from 11 nodes in Figure 7(a) to 7 nodes in Figure 7(b).

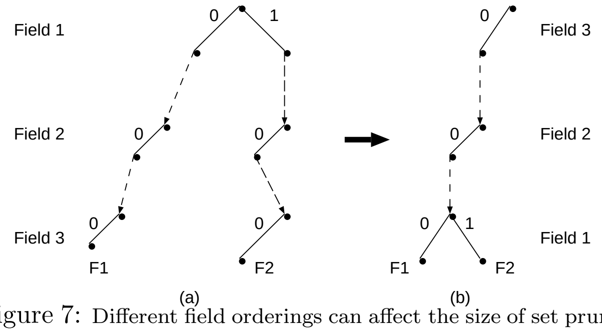

Figure 7: Different field orderings can affect the size of set pruning tries. significantly.

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 - 7. 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

Database	$\frac{Worst\ Storage}{Best\ Storage}$
Database 1	6.7239
Database 2	2.9035
Database 3	2.3335
Database 4	3.4188
Database 5	2.9809

Table 5: Variation in storage using different field orderings.

Database	# memory accesses	Storage (# nodes)
Database 1	86	5541
Database 2	102	51785
Database 3	102	59180
Database 4	102	123951
Database 5	102	165920

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

to this approach is to use some heuristic to pre-select 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 [7, 4] can also be used.

5.3 Experimental Evaluation

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 trie is the ordering 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 tries provide an optimal number of memory accesses at the cost of large storage requirement. In particular, the storage requirement increases to 3 - 25 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 [7].

6 Compression

We further improve backtracking search and set pruning tries using a novel form of compression. A standard compression scheme for tries (e.g., [2]) is to remove all single branching paths so that no node has more than one child). Figure 8 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.

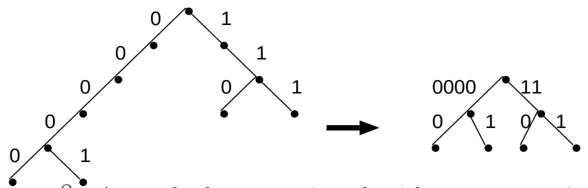

Figure 8: A standard compression algorithm: merge a single branch into one trie node.

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 number of nodes is also equal to $2i+1$. So there are $N-1$ internal nodes, and thus $2N-1$ nodes altogether.

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 standard compression scheme cannot exploit such redundancy. For example, it fails to compress anything for the trie shown on Figure 9(a), since no node in the trie has only a single branch.

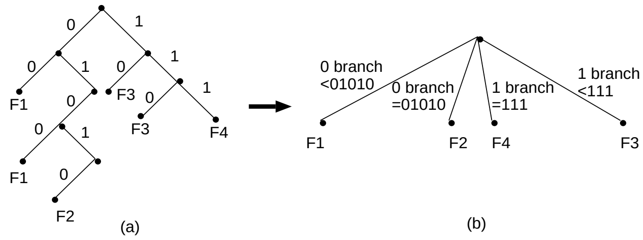

Figure 9: Our new compression algorithm.

However, a closer examination of the trie in Figure 9(a) reveals an interesting property: all nodes on the left 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*

[7]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).

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 9(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 10 to represent a compressed node. We add 3 fields to the original uncompressed trie node: *value*, *len*, and *rangePtr* as shown in Figure 10. 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.

```
#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 10: Data structure for compressed node.

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 11 shows the pseudo-code.

We now examine the details of our general compression scheme. 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:

```
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 11: Search on the compressed trie (compressed using the general compression algorithm).

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 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 or 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 [10]. [4] 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 [4] 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 [4].) 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 12 for filter rule ≥ 1024, a very common rule in firewall databases for the port field. As shown in Figure 12, 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 $2N-1$ nodes (according to Theorem 6.1).

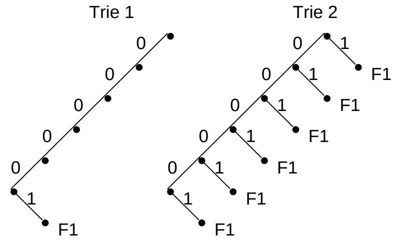

Figure 12: General compression algorithm.

6.2 Experimental results for Compression

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

We list the performance results for the best ordering (defined in Section 4.3.1 and Section 5.3). Compared with the performance results before compression, as shown in Table 4 and Table 7, 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.) [8] For the set pruning trie, compression cuts down storage by a factor of 4 - 12, 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 tries, 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 13.

[8] 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.

Database	# memory accesses				Storage (# nodes)			
	Set Pruning	BB	PB	Linear Search	Set Pruning	BB	PB	Linear Search
Database 1	22	30 (21)	28 (21)	67	1510	429	429	67
Database 2	57	51 (34)	39 (34)	158	5778	912	912	158
Database 3	59	49 (36)	43 (45)	183	6914	1261	1666	183
Database 4	64	98 (58)	85 (58)	279	30322	2951	2951	279
Database 5	55	59 (29)	58 (29)	266	13556	2815	2815	266

Table 7: Performance of compressing backtracking search and set pruning tries (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 larger 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.

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 tries 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.

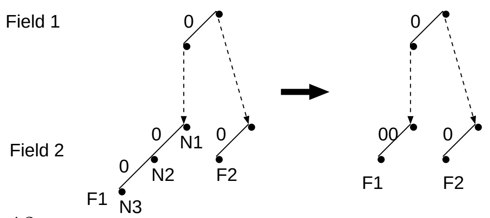

Figure 13: An example to show a path that is compressible in a backtracking search trie may become uncompressible in its corresponding set pruning trie.

We also compare the performance of linear search with backtracking search and set pruning tries in Table 7. 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 2 - 5 times as large as that required by backtracking search or set pruning tries. If we use multi-bit tries, the performance of backtracking search and set pruning tries can be even better. [9]

Furthermore, the lookup time of linear search increases linearly with the size of databases. In contrast, the lookup time of set pruning tries 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 tries for databases with limited prefix containments. Therefore both backtracking search and set pruning tries have more scalable lookup time than linear search.

A few comments follow. First, the storage cost of a compressed node is larger than an uncompressed node. In our implementation, it is about 40% larger than an uncompressed node. On the other hand, compression cuts down the total number of nodes by a factor of 4 - 12 in set pruning tries, and by a factor of 2 - 4 in backtracking tries. So the total storage is significantly reduced after the compression. Second, 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 [10]. However, the total CPU time is actually less because compression cuts down the number of nodes visited by more than 2 times. Moreover, the lookup time is dominated by the number of 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.

To summarize, in this section, we described a new compression algorithm that can explore the redundancy in the multiplane tries effectively. We evaluated the performance of the compression scheme, and showed that it can reduce the storage cost by a factor of 4 - 12, and reduce lookup time by a factor of 2 - 5.

7 Pipelining Backtracking

So far all the algorithms in this paper could be implemented in either hardware or software. However, hardware implementors often wish to use extra hardware gates to improve speed by using parallelism. A natural way to speed up an algorithm that has sequential dependencies (such as backtracking) is to use pipelining.

First, we assume the use of our new backtracking scheme that uses limited computation state; this is cru-

[9] 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.

[10] 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.

cial because it reduces the amount of *register* memory passed between pipeline stages. However, we also have to worry about the amount of main memory (especially if each pipeline stage requires its own memory, and the memory is fast SRAM). SRAM, which corresponds to cache memory in CPUs, is expensive and must be minimized for a feasible solution.

A simple way to pipeline the backtracking search is to store the entire backtracking search trie at every pipelining stage. During the backtracking search, we pass the node to be visited after the previous pipelining stage to the next pipelining stage, so that the search in the next pipelining stage will start from that node. In this case, the main memory storage requirement increases linearly with the number of pipelining stages. However since not all nodes in the backtracking trie will be visited in every pipelining stage, a natural enhancement to the previous approach is to have the pipelining stage i store only the trie nodes that will be visited in the stage i. The details of how to partition the backtracking trie into different pipelining stages are shown in Appendix B.

Figure 14 shows the main memory storage requirement as a function of the total number of pipelining stages for both uncompressed and compressed tries [11]. As we can see, the storage increases only moderately with the number of pipelining stages in both cases.

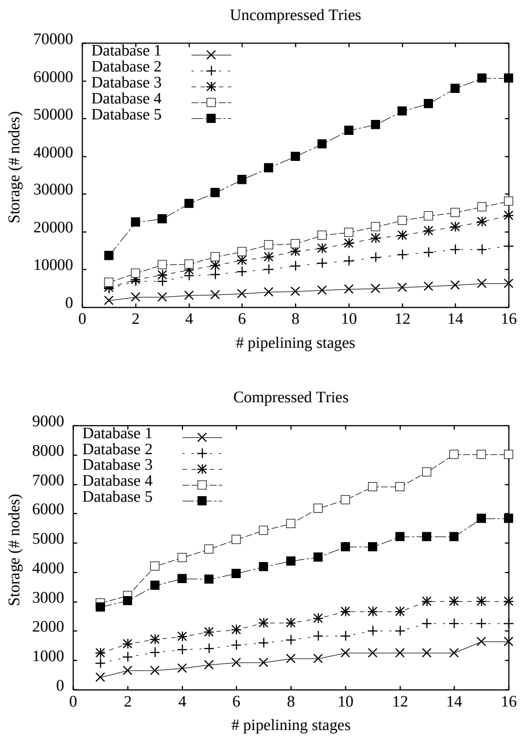

Figure 14: Storage requirement for pipeline.

[11] Our evaluation is based on the modified backtracking search on a small register memory budget. Similar results are observed for the standard backtracking search.

Similarly, we can also apply pipelining to set pruning tries for improvement in search time. The total storage requirement in this case is equal to the size of the original set pruning tries, since every node is stored at exactly one pipelining stage.

8 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 tries can perform much better than the worst case bounds, especially using several optimization 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 tries. 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 tries are no longer in the tradeoff region, for they neither yield the best storage requirement nor the best lookup time.

8.1 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 A) 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.

```
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.

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.

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.

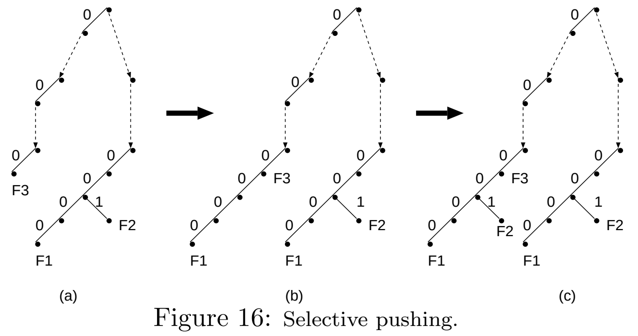

Figure 16: Selective pushing.

Selective pushing can be applied to both uncompressed and compressed tries. The side effects of applying selective pushing for 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.

8.2 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 use the best field orderings as defined in Section 4.3 and Section 5.3.

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, the lookup time is reduced from 117 to 95 with moderate increase in storage. Further decrease in the lookup time is achieved at higher cost in storage after we reach the "knee" of the curve. Similar behavior is observed for the other databases. Also for all databases, when the lookup time comes close to the optimal, the storage costs saturate at those of the corresponding set pruning trie, as we would expect.

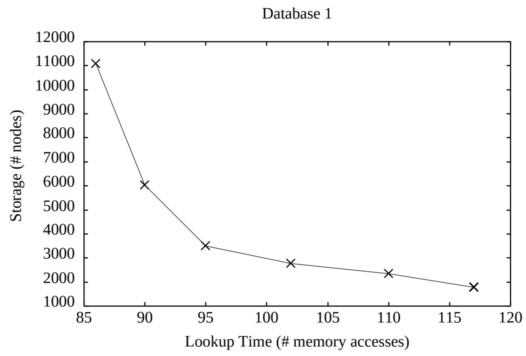

Figure 17: Selective pushing applied to uncompressed trie.

For compressed tries, 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 18. As before, when the lookup time is large, it drops rapidly at very small cost in storage. For instance, in database 2, we reduce the lookup time from 51 to 44 with little extra storage. Further decrease in lookup time incurs higher cost in storage after we reach the "knee" of the curve. Similar behavior is observed for the other databases.

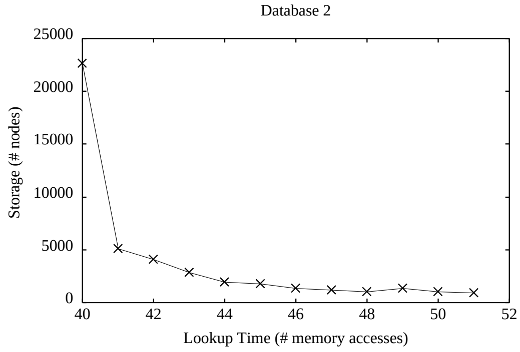

Figure 18: Selective pushing applied to compressed trie.

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 10-20% with only a small increase in storage. Once the knee of the tradeoff curve is reached, however, further reduction in lookup time is achieved only at the cost of a large increase in the storage. As part of our future work, we plan to further examine the selective pushing to choose the knee point automatically.

9 Using Minimal Hardware

We have discussed a set of algorithms that can be implemented in either software or hardware. With the exception of the pipelining techniques, all the algorithms described can be implemented in software in a router. Rather than consider two extremes, pure hardware (e.g., with pipelining) or pure software, in this section we consider implementations that use software with a *minimal* amount of hardware support.

For example, if we assume that ternary CAMs are available for a small number of filters, but we have large filter databases, 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 tries. 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 attempts to quantify the storage cost for each filter rule by counting the exact number of occurrences of each filter in the set pruning tries.

We evaluate performance using both heuristics on set pruning tries. Figure 19(a) shows the results for uncompressed set pruning tries. 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 20% after removing 10 rules, and 40% after removing 30 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 tries. As we can see, using heuristic 2, the storage cost drops 40% after removing 10 rules, and 53% after removing 30 rules. The performance of heuristic 1 comes close to that of heuristic 2 in this case.

To optimize the query lookup time, we use the same heuristics as for reducing the storage, except changing the second heuristic to "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 30 filter rules reduces the lookup time by 10%, and removing 40 filter rules reduces the lookup time by 20%. 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), using heuristic 2, the reduction in lookup time is 32% after removing the first 10 filter rules, and around 38% after removing the first 30 filter rules. The performance of heuristics 1 and 2 is similar in this case.

10 Conclusion

This paper has five contributions. First, we showed experimentally that the performance of simple trie based filter schemes is much better than worst-case figures predict. We also improved the basic trie based filter schemes with a number of optimizations, including a new backtracking search technique, which requires only $D+1$ pieces of state, where D is the number of dimensions. Second, we proposed a novel compression algorithm that further reduces the lookup time and storage

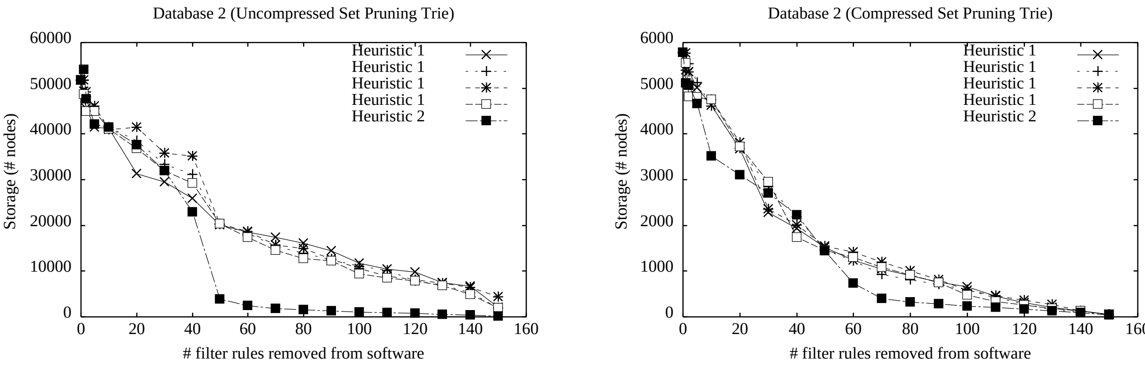

Figure 19: Moving a subset of filters from software to hardware for storage savings.

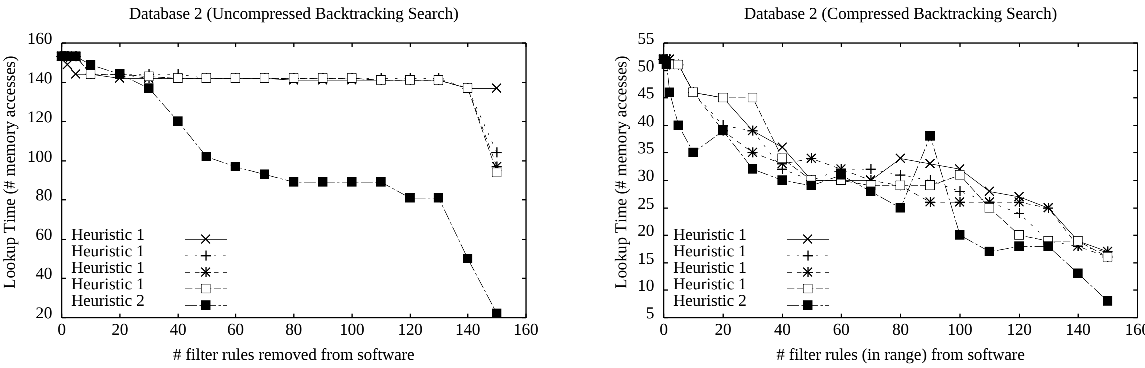

Figure 20: Moving a subset of filters from software to hardware for better lookup time.

Approach	Description	Performance Gain
Optimize backtrack search	backtrack on a small memory budget	reduce the memory cost to $D+1$ when doing backtracking search
	optimal field ordering	up to 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 tries	DAG	two orders of magnitude saving in storage
	optimal field ordering	a factor of 2 - 7 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 4 - 12 saving in storage
Pipelining the search	Split the search into multiple pipelining stages, and each stage is responsible for a portion of search	increase throughput with marginal increase in memory cost
Selective pushing	"push down" the filters with high worst-case backtracking times to speed up lookup time	10 - 20% 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 rules cuts storage cost by up to 40%, or lookup time by up to 30%

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

cost. Third, we investigated pipelining the search, and found the storage cost for pipelining increases only moderately with the number of pipelining stages. Fourth, 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 with a limited number of prefix containments, 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 pruning based on cost, finding optimal field orderings, using switch pointers at the lowest two dimensions) 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.

Second, we can further improve the efficiency of backtracking search in hardware using pipelining. Our results show the perhaps surprising result that storage (main memory) cost for pipelining grows only moderately with the number of pipelining stages. Notice that our new backtracking technique, which requires only a small number of registers to hold backtracking state, is also crucial for a hardware implementation of backtracking. It is our experience that most implementors dismiss backtracking as a scheme that cannot be pipelined: we believe this is a false assumption.

A third observation is that our compression algorithm can reduce 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 - 20% 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 tries. However, we have only used a simple greedy heuristic. We are currently investigating better tradeoffs using optimization techniques.

Finally, we find that 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 a software implementation to effectively take advantage of any limited hardware when it becomes available.

Despite the fact that the storage numbers for optimized set pruning tries are reasonable, our final message is that backtracking (with compression, selective pushing, and possible hardware support) offers a more reasonable time-space tradeoff than set pruning tries, and should be seriously considered by router implementors.

References

[1] 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.

[2] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner. Router Plugins A Software Architecture for Next Generation Routers. In *Proc. of SIGCOMM'98*, Sept. 1998.

[3] IETF Differentiated Services Working Group. http://www.ietf.org/html.charters/diffserv-charter.html.

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

[5] T. V. Lakshman and D. Stidialis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *Proc. of SIGCOMM'98*, pp. 191-202, Sept. 1998.

[6] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proc. of SIGCOMM'99*, pp. 147-160, Sept. 1999.

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

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

[9] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *Proc. of SIGCOMM'99*, pp. 135 - 146, Sept. 1999.

[10] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. In *Proc. of SIGCOMM'98*, pp. 203-214, Sept. 1998.

[11] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of SIGCOMM'97*, Sept. 1997.

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

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*].

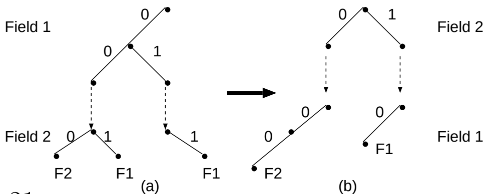

Figure 21: Different field orderings can have different header classes significantly.

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 tries 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.

B Finding Storage Requirement for Pipelining

As described in Section 7, each pipelining stage stores only the trie nodes that will be visited during that stage. We use the following algorithm to determine the storage requirement when there are P pipelining stages.

First, we use the algorithm described in Appendix A to find the worst-case backtracking search time, denoted as W. Then each pipelining stage is responsible for up to $\lceil \frac{W}{P} \rceil$ steps in the backtracking search. Now for every input class, we do backtracking search. During the search, we annotate every trie node encountered with the stage at which the node is being visited using a bit vector, $stagesVector$. When a node is visited at the stage i, we set its $stagesVector[i]$ to 1. After searching over all the input classes, we can compute the total storage requirement by traversing the trie, and summing over the storage cost incurred by all the trie nodes, where a trie node with k bits set in its $stagesVector$ account for k nodes in total storage, since there are k pipelining stages that require storing the node.