# Aggregated Bit Vector Search Algorithms for Packet Filter Lookups

Florin Baboescu, George Varghese

Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
{baboescu, varghese}@cs.ucsd.edu

**Abstract**

Packet classification is important for applications such as firewalls, intrusion detection, and differentiated services. Existing algorithms for packet classification reported in the literature scale poorly in either time or space as filter databases grow in size. Hardware solutions such as TCAMs do not scale to large classifiers. However, even for large classifiers (say 100,000 rules), any packet is likely to match a few (say 10) rules. Our paper seeks to exploit this observation to produce a scalable packet classification scheme called Aggregated Bit Vector (ABV). Our paper takes the bit vector search algorithm (BV) described in [10] (which takes linear time) and adds two new ideas, recursive aggregation of bit maps and filter rearrangement, to create ABV (which can take logarithmic time for many databases). We show that ABV outperforms BV by an order of magnitude using simulations on both industrial firewall databases and synthetically generated databases.

## 1 Introduction

Every Internet router today can forward entering Internet messages (packets) based on the destination address. The 32 bit IP destination address is looked up in a table which then determines the output link on which the packet is sent. However, for a competitive advantage, many routers today choose to do additional processing for a specific subset of packets. Such additional processing includes providing differentiated output scheduling (e.g., Voice over IP packets are routed to a high priority queue), taking security-related actions (e.g., dropping packets sent from a certain subnet or flagging suspicious packets for later analysis), load balancing (e.g., routing different packets to different servers) and doing traffic measurement (e.g., measuring traffic between subnet pairs).

Although the details of the additional processing can vary greatly, a common requirement of all the functions above is that routers be able to *classify* packets based on packet headers into equivalence classes called *flows*. A flow is defined by a rule (e.g., packets whose source address starts with prefix bits $S$, the destination address is $D$, and which is sent to the server port for web traffic). Associated with each flow is an action which defines the additional processing (e.g., send to a specific queue, drop, make a copy, update counters).

Thus packet classification routers have a database of rules, one for each flow type that the router wants to process differently. The rules are explicitly ordered by a network manager (or protocol) that creates the rule database. Thus when a packet arrives at a router, the router must find a rule that matches the packet headers; if more than one match is found, the first matching rule is applied.

*Scalable Packet Classification:* This paper is about the problem of performing scalable packet classification for routers at wire speeds even as rule databases increase in size. Forwarding at wire speeds requires forwarding minimum sized packets in the time it takes to arrive on a link; this is crucial because otherwise one might drop important traffic before the router has a chance to know it is important [10]. With Internet usage doubling every 6 months, backbone link speeds have increased from OC-48 to OC-192 (2.4 to 10 Gigabits/second), and speeds up to OC-768 (40 Gigabits/second) are projected. Even link speeds at the network edge have increased from Ethernet (10 Mbit/sec) to Gigabit Ethernet.

Further, rule databases are increasing in size. The initial usage of packet classification for security and firewalls generally resulted in fairly small databases (e.g., the largest database in a large number of Cisco rule sets studied

by [4] is around 1700). This makes sense because such rules are often entered by managers. However, in the very popular Differentiated Services [2] proposal, the idea is to have routers at the edge of the backbone classify packets into a few distinct classes that are marked by bits in the TOS field of the IP header. Backbone routers then only look at the TOS field. If, as seems likely, the DiffServ proposal reaches fruition, the rule sets for edge routers can grow very large.

Similarly, rule sets for edge routers that do load balancing [1] can grow very large. Such rule sets can potentially be installed at routers by a protocol; alternately, a router that handles several thousand subscribers may need to handle say 10 rules per subscriber that are manually entered. For all these reasons, we believe rule databases of up to 100,000 rules are of practical interest.

## 2 Previous Work

Previous work in packet classification [10, 15, 4, 6, 5] has shown that the problem is inherently hard. Most practical solutions we know of either use linear time [10] to search through all rules sequentially, or use a linear amount of parallelism (e.g., Ternary-CAMs as in [11]). Ternary CAMs are Content Addressable Memories that allow wildcard bits. While Ternary-CAMs are very common, such CAMs have smaller density than standard memories, dissipate more power, and require multiple entries to handle rules that specify ranges. Thus CAM solutions are still expensive for very large rule sets of say 100,000 rules, and are not practical for PC-based routers [8]. Solutions based on caching [14] do not appear to work well in practice because of poor hit rates and small flow durations [12], and still need a fast classifier as a backup when the cache fails.

Another practical solution is provided by a seminal paper that we refer to as the Lucent bit vector scheme [10]. The idea is to first search for rules that match each relevant field $F$ of the packet header, and to represent the result of the search as a bitmap of rules that match the packet in field $F$. Then the rules that match the full header can be found by taking the intersection of the bitmaps for all relevant fields $F$. While this scheme is still linear in the size of the ruleset, in practice searching through a bitmap is fast because a large number of bits (up to 1000 in hardware, up to 128 bits in software) can be retrieved in one memory access. While the Lucent scheme can scale to around a reasonably large number of rules (say 10,000) the inherently linear worst-case scaling makes it difficult to scale up to large rule databases.

From a theoretical standpoint, it has been shown [10] that in its fullest generality, packet classification requires either $O(\log N^{k-1})$ time and linear space, or $\log N$ time and $O(N^k)$ space, where $N$ is the number of rules, and $k$ is the number of header fields used in rules. Thus it comes as no surprise that the solutions reported in the literature for $k > 2$ either require large worst case amounts of space (e.g., crossproducting[15], RFC [4], HiCuts [5]) or time (e.g., bit vector search [10], backtracking [15]).

However, the papers by Gupta and McKeown [4, 5] introduced a major new direction into packet classification research. Since the problem is unsolvable in the worst case, they look instead for heuristics that work well on common rule sets. In particular, after surveying a large number of rule sets [4], they have found that *rule intersection* is very rare. In other words, it is very rare to have a packet that matches multiple rules. Since the examples that generate the worst case bounds entail multiple rule sets that intersect, it is natural to wonder whether there are schemes that are provably better given some such structural assumption on real databases.

Among the papers that report heuristics [4, 5, 6], the results on real databases are, indeed, better than the worst case bounds. Despite this, the RFC scheme of [4] still requires comparatively large storage. The HiCuts scheme [5] does better in storage (1 Mbyte for 1700) and requires 20 memory accesses for a database of size 1700. Thus while these schemes do seem to exploit the characteristics of real databases they do not appear to scale well (in time and storage) to very large databases.

Finally, there are several algorithms that are specialized for the case of rules on two fields (e.g., source and destination IP address only). For this special case, the lower bounds do not apply (they apply only for $k > 2$); thus hardly surprisingly, there are algorithms that take logarithmic time and linear storage. These include the use of range trees and fractional cascading [10], grid-of-tries [15], area-based quad-trees [16], and FIS-trees [3]. While these algorithms are useful for special cases (such as measuring traffic between source and destination subnets), they do not solve the general problem and are hence not relevant to the rest of our paper.

We note that the FIS trees paper [3] sketches an extension to $k > 2$ but suggests the "memory usage may be large"; while the authors [3] suggest the use of clustering heuristics could improve multidimensional FIS performance, their paper does not describe a single experiment on a general purpose classifier. Thus, while the extended FIS approach

appears to have merit, it is difficult to evaluate FIS trees as a general purpose approach until it is completely implemented and evaluated on real (even small) multidimensional classifiers.

In summary, for the general classification problem of 3 or more fields, we find that existing solutions do not scale well in one of time or storage. Our paper will use the Lucent bit vector scheme as a point of departure (since it already scales to medium size databases, and is amenable to implementation using either hardware or software). Our Aggregated Bit Vector scheme then adds two new ideas, *rule aggregation and rule rearrangement*, that considerably enhance the scalability of the Lucent scheme. The reader may immediately object that rule rearrangement can lead to incorrect answers because the algorithm could interchange two overlapping rules. Our way out of this dilemma is to *find all matches* and then to use a mapping table to map all matched rule numbers to the rule numbers in the original order. This is efficient if the number of rules matching a given packet is small, a property that has been observed on a large number of real databases [4].

We note that while the HiCuts [5] scheme also does a form of hierarchical aggregation (based on subspaces of the original $d$-dimensional hyper-space), our scheme performs a completely different form of aggregation (based on the position or cost of a rule). We elaborate more on this difference later.

# 3 Problem Statement

We assume that the information relevant to a lookup is contained in $K$ distinct *header fields* in each packet. These header fields are denoted $H_1, H_2, \ldots, H_k$, where each field is a string of bits. For instance, the relevant fields for an IPv4 packet could be the Destination Address (32 bits), the Source Address (32 bits), the Protocol Field (8 bits), the Destination Port (16 bits), the Source Port (16 bits), and TCP flags (8 bits). The number of relevant TCP flags is limited, and so rule databases often combine the protocol and TCP flags into one field—for example, we can use TCP-ACK to mean a TCP packet with the ACK bit set. Note that many rule databases allow the use of other header fields besides TCP/IP such as MAC addresses, and even Application (e.g., http) headers. Thus, the combination $(D, S,$ TCP-ACK $, 80, 2500)$, denotes the header of an IP packet with destination D, source S, protocol TCP, destination port 80, source port 2500, and the ACK bit set.

The *rule database* of a router consists of a finite sequence of rules, $R_1, R_2 \ldots R_N$. Each rule is a combination of $K$ values, one for each header field. Each field in a rule is allowed three kinds of matches: *exact match, prefix match,* or *range match.* In an exact match, the header field of the packet should exactly match the rule field—for instance, this is useful for protocol and flag fields. In a prefix match, the rule field should be a prefix of the header field—this is useful for blocking access from a certain subnetwork. In a range match, the header values should lie in the range specified by the rule—this is useful for specifying port number ranges.

Each rule $R_i$ has an associated action $act^i$, which specifies how to forward the packet matching this rule. The action specifies if the packet should be blocked. If the packet is to be forwarded, it specifies the outgoing link to which the packet is sent, and perhaps also a queue within that link if the message belongs to a flow with bandwidth guarantees.

We say that a packet $P$ *matches* a rule $R$ if each field of $P$ matches the corresponding field of $R$—the match type is implicit in the specification of the field. For instance, let $R = (1010*, *, TCP,$ 1024–1080, $*)$ be a rule, with $act = drop$. Then, a packet with header $(10101\ldots111, 11110\ldots000, TCP,$ 1050, 3) matches $F$, and is therefore dropped. The packet $(10110\ldots000, 11110\ldots000, TCP,$ 80, 3), on the other hand, doesn't match $R$. Since a packet may match multiple rules, we define the matching rule to be the *earliest* matching rule in the sequence of rules.

We wish to do packet classification at wire speed for minimum sized packets and thus speed is the dominant metric. Because both modern hardware and software architectures are limited by memory bandwidth, it makes sense to measure speed in terms of memory accesses. It is also important to reduce the size of the data structure that is used to allow it to fit into the high speed memory. The time to add or delete rules is often ignored, but it is important for dynamic rule sets, that can occur in real firewalls. We show towards the end of our paper that our scheme can be modified to handle fast updates at the cost of increased search time.

# 4 Towards a new scheme

In this section, we introduce the ideas behind our scheme by first describing the Lucent bit vector scheme as our point of departure. Then, using an example rule database, we show our two main ideas: aggregation and rule rearrangement. In the next section, we will formally describe our new scheme.

## 4.1 Bit Vector Linear Search

The Lucent bit vector scheme is a form of divide-and-conquer which divides the packet classification problem into $k$ subproblems, and then combines the results. To do so, we first build $k$ 1-dimensional tries associated with each dimension (field) in the original database. We assume that ranges are either handled using a range tree instead of a trie, or by converting ranges to tries as shown in [15, 6]. An $N-$bit vector is associated with each node of the trie corresponding to a valid prefix. (Recall $N$ is the total number of rules).

Figure 2 illustrates the construction for the simple two dimensional example database in Figure 1. For example, in Figure 1, the second rule $F_1$ has 00* in the first field. Thus, the leftmost node in the trie for the first field, corresponds to 00*. Similarly, the Field 1 trie contains a node for all distinct prefixes in Field 1 of Figure 1 such as 00*, 10*, 11*, 1*, and 0*.

Each node in the trie for a field is labeled with a $N$-bit vector. Bit $j$ in the vector is set if the prefix corresponding to rule $F_j$ in the database matches the prefix corresponding to the node. In Figure 1, notice that the prefix 00* is matched by the values 00* and 0*, which correspond to values in rules 1,2,4,5, and 6. Thus the eleven bit vector shown below the leftmost leaf node in Figure 2 is 11001110000. For now, only consider the boxed bit vectors and ignore the smaller bit vectors below each boxed bit vector.

| $Rule$ | $Field_1$ | $Field_2$ |
|---|---|---|
| $F_0$ | $00*$ | $00*$ |
| $F_1$ | $00*$ | $01*$ |
| $F_2$ | $10*$ | $11*$ |
| $F_3$ | $11*$ | $10*$ |
| $F_4$ | $0*$ | $10*$ |
| $F_5$ | $0*$ | $11*$ |
| $F_6$ | $0*$ | $0*$ |
| $F_7$ | $1*$ | $01*$ |
| $F_8$ | $1*$ | $0*$ |
| $F_9$ | $11*$ | $0*$ |
| $F_{10}$ | $10*$ | $10*$ |

Figure 1: A simple example with 11 rules on two fields.

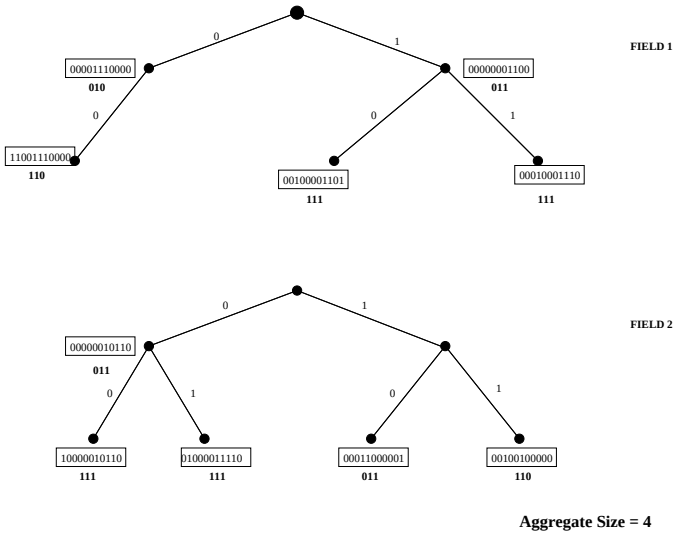


Figure 2: Two tries associated with each of the fields in the database of Figure 1, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with nodes that correspond to valid prefixes. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right.

When a packet header arrives with fields $H_1 \ldots H_k$, we do a longest matching prefix lookup (or narrowest range lookup) in each field $i$ to get matches $M_i$ and read off the resulting bit vectors $S(M_i)$ from the tries for each field $i$. We then take the intersection of $S(M_i)$ for all $i$, and find the lowest cost element of the intersection set. If rules are arranged in non-decreasing order of cost, all we need to do is to find the index of the first bit set in the intersected bit vector. However, these vectors have $N$ bits in length; computing the intersection requires $O(N)$ operations. If $W$ is the size of a word of memory than these bit operations are responsible for $\frac{N \times k}{W}$ memory accesses in the worst case. Note that the worst case occurs very commonly when a packet header does *not* match a single rule in the database.

## 4.2 Using aggregation to reduce memory accesses

Recall that we are targeting the high cost in memory accesses which essentially scales linearly $O(N)$ except that the constant factor is scaled down by the word size of the implementation. With a word size of up to 1000 in hardware, such a "constant" factor improvement is a big gain in practice. However, we want to do better by at least one order of magnitude, and remove the linear dependence on $N$. To this end, we introduce the idea of *aggregation*.

The main motivating idea is as follows. We hope that if we consider the bit vectors produced by each field, the set bits will be very sparse. For example, for a 100,000 rule database, if there are only 5 bits set in a bit vector of size 100,000, it seems a waste to read 100,000 bits. Why do we believe that bit vectors will be sparse? We have the following arguments:

- **Experience:** The databases we have seen have every packet match at most 5 rules. Similar small numbers have been seen in [5] for a large collection of databases up to 1700 rules.
- **Extension:** How will large databases be built? If they are based on aggregating several small classifiers for a large number of classifiers, it seems likely that each classifier will be disjoint. If they are based on a routing protocol that distributed classifiers based on prefix tables, then prefix containment is quite rare in the backbone table and is limited to at most 6 [6]. Again, if a packet matches a large number of rules, it is difficult to make sense of the ordering rules that give one rule priority over others.

While the fact that a given packet matches only a few rules, does not imply that the packet cannot match a large number of rules in all dimensions (with only a few of the matches aligning properly in all dimensions). However, assume for now there is some dimension $j$ whose bit vector is sparse.[1]. To exploit the existence of such a sparse vector, our modified scheme, appends the bit vector for each field in each trie with an *aggregate bit vector*. First, we fix a an aggregate size $A$. $A$ is a constant that can be tuned to optimize the performance of the aggregate scheme; a convenient value for $A$ is $W$ the word size. Next, a bit $i$ is set in the aggregate vector if there is at least one bit $k$, $k \in [i \times A, (i+1) \times A]$. In other words, we simply aggregate each group of $A$ bits in the Lucent bit vector into a single bit (which represents the OR of the aggregated bits) in the aggregate bit vector.

Clearly, we can repeat the aggregation process at multiple levels (forming a tree whose bits are the bits in the original Lucent bit vector for a field). This can be useful for large enough $N$. However, since we deal with aggregate sizes that are at least 32, two levels of hierarchy can handle $32 * 32 * 32 = 32K$ rules. Using larger aggregate sizes will increase the $N$ that can be handled further. Thus for much of this paper, we will focus on one level (i.e., a single aggregate bit vector) or 2 levels (for a few synthetically generated large databases). We note that the only reason our results for synthetic databases are limited to 20,000 rules is because our *current testing* methodology (to check the worst-case search time for all packet header combinations) does not scale. Thus while we believe our algorithm scales to very large classifiers; we hope to prove worst-case times for sizes large than 20,000 after deploying the new testing algorithm we are working on.

Why does aggregation help? The goal is to efficiently construct the bit map intersection of all fields without examining all the leaf bit map values for each field. For example, suppose that a given packet header matches only a small constant number of rules in each field. This can be determined in constant time (even for large $N$) by examining the top level aggregate bit maps; we can then only examine the leaf bit map values for which the aggregate bits are set. Thus, intuitively, we only have to examine a constant number of memory words (for each field) to determine the intersection because the aggregate vectors allow us to quickly filter out bit positions where there is no match. The goal is to have a scheme that comes close to taking $O(log_A N)$ memory accesses, even for large $N$.

Figure 2 illustrates the construction for the example database in Figure 1 using an aggregate size $A = 4$. Let's consider a packet with Field 1 starting with bits 0010... and Field 2 starting with bits 0100.... From Figure 2 one

[1] If this is not the case, as is common, then our second technique of rearrangements can make this assumption more true

can see that the longest prefix match is 00 for the first field and 01 for the second one. The associated bit vectors are: 11001110000 and 01000011110 while the aggregate ones (shown in bold below the regular bit vectors) are: 110 and 111. The $AND$ operation on the two aggregate vectors yields 110, showing that a possible matching rule must be located only in the first 8 bits. Thus it is not necessary to retrieve the remaining 4 bits for each field.

Notice that in this small example, the cost savings (assuming a word size of 4 is only 2 memory accesses, and this reduction is offset by the 2 memory accesses required to retrieve the bit maps. Larger examples show much bigger gains. Also, note that we have shown the memory accesses for *one* particular packet header. We need to efficiently compute the *worst-case* number of memory accesses across *all* packet headers.

While aggregation does often reduce the number of memory accesses, in some cases a phenomenon known as *false matches* can increase the number of memory accesses to being slightly higher (because of the time to retrieve the aggregates for each field) than even the normal Lucent bit vector search technique.

Consider the database in Figure 3 and an aggregation size $A = 2$. $A_1, \ldots, A_{30}$ are all prefixes having the first five bits different from the first five bits of two IP addresses $X$ and $Y$. Assume the arrival of a packet from source $X$ to destination $Y$. Thus the bit vector associated with the longest matching prefix in the Field 1 (source) trie is $1010101 \ldots 101$ and the corresponding bit vector in the Field 2 (destination) trie is $0101010 \ldots 011$. The aggregate bit vectors for both fields both using $A = 2$ are $111 \ldots 1$. However, notice that for all the ones in the aggregate bit vector (except the last one) the algorithm wrongly assumes that there might be a matching rule in the corresponding bit positions.

This is because of what we call a false match, a situation in which the result of an AND operation on an aggregate bit returns a one but there is no valid match in the group of rules identified by the aggregate. This can clearly happen because an aggregate bit set for field 1 corresponding to positions $p, .., p + A - 1$ only means that *some* bit in those positions (e.g., $p + i, i < A$) has a bit set. Similarly, an aggregate bit set for field 2 corresponding to positions $p, .., p + A - 1$ only means that some bit in those positions (e.g., $p + j, j < A$) has a bit set. Thus a false match occurs when the two aggregate bits are set for the two fields but $i \neq j$. The worst case occurs when a false match occurs for every aggregate bit position.

For this particular example there are 30 false matches which makes our algorithm read $31 \times 2$ bits more than the Lucent bit vector linear search algorithm. We have used an aggregation size $A = 2$ in our toy example, while in practice $A$ will be much larger. Note that for larger $A$, our aggregate algorithm will only read a small number of bits more than the Lucent bit vector algorithm even in the worst case.

| $Rule$ | $Field_1$ | $Field_2$ |
|---|---|---|
| $F_1$ | $X$ | $A_1$ |
| $F_2$ | $A_1$ | $Y$ |
| $F_3$ | $X$ | $A_2$ |
| $F_4$ | $A_2$ | $Y$ |
| $F_5$ | $X$ | $A_3$ |
| $F_6$ | $A_3$ | $Y$ |
| $F_7$ | $X$ | $0*$ |
| ... | ... | ... |
| ... | ... | ... |
| $F_{60}$ | $A_{30}$ | $Y$ |
| $F_{61}$ | $X$ | $Y$ |

Figure 3: An example of a database with two-dimensional rules for which the aggregation technique without rearrangement behaves poorly. The size of the aggregate $A = 2$

## 4.3 Why rearrangement of rules can help

Normally, in packet classification it is assumed that rules cannot be rearranged. In general, if Rule 1 occurs before Rule 2, and a packet could match Rule 1 and Rule 2, one must never rearrange Rule 2 before Rule 1. Imagine the disaster if Rule 1 says "Accept", and Rule 2 says "Deny", and a packet that matches both rules get dropped instead of being accepted. Clearly, the problem is that we are rearranging overlapping rules; two rules are said to overlap if there is at least one packet header that can match both rules.

However, the results from [4] imply that in real databases rule overlap is rare. Thus if we know that a packet header can never match Rule 1 and Rule 2, then it cannot affect correctness to rearrange Rule 2 before Rule 1 (they are, so to speak, "independent" rules). We can use this flexibility to try to group together rules that contribute to false matches into the same aggregation groups, so that the cost of false matches (in terms of memory accesses) is reduced.

Better still, we can rearrange rules arbitrarily *as long as we modify the algorithm to find all matches and then compute the lowest cost match.* For example, suppose a packet matched rules Rule 17, Rule 35, and Rule 50. Suppose after rearrangement Rule 50 becomes the new Rule 1, Rule 17 becomes the new Rule 3, and Rule 35 becomes the new Rule 6. If we compute all matches the packet will now match the new rules 1, 3, and 6. Suppose we have precomputed an array that maps from new rule order number to old rule order number (e.g., from 1 to 50, 3 to 17, etc.). Thus in time proportional to the number of matches, we can find the "old rule order number" for all matches, and select the earliest rule in the original order. Once again the crucial assumption to make this efficient is that the number of worst-case rules that match a packet is small. Note also that it is easy (and not much more expensive in the worst-case) to modify a bit vector scheme to compute all matches.

For example, rearranging the rules in the database shown in the database in Figure 3, we obtain the rearranged database shown in Figure 4. If we return to the example of packet header ($X$, $Y$), the bit vectors associated with the longest matching prefix in the new database will be:$111\ldots11000\ldots0$ and $000\ldots01111\ldots1$ having the first 31 bits 1 in the first bit vector and the last 31 bits 1 in the second bit vector. However, the result of the AND operation on the aggregate has the first bit 1 in the position 16. This makes the number of bits necessary to be read for the aggregate scheme to be $16 \times 2 + 1 \times 2 = 34$ which is less than the number of the bits to be read for the scheme without rearrangement: $31 \times 2 = 62$.

The main intuition in Figure 4 versus Figure 3 is that we have "sorted" the rules by first rearranging all rules that have $X$ in Field 1 to be contiguous; having done so, we can rearrange the remaining rules to have all values in Field 2 with a common value to be together (this is not really needed in our example). What this does is to localize as many matches as possible for the sorted field to lie within a few aggregation groups instead of having matches dispersed across many groups.

Thus our paper has two major contributions. Our first contribution is the idea of using aggregation which, by itself, reduces the number of memory accesses by more than an order of magnitude for real databases, and even for synthetically generated databases where the number of false matches is low. Our second contribution is to show how can one reduce the number of false matches by a further order of magnitude by using rule rearrangement together with aggregation. We also have a third contribution that shows how to make updates faster using aggregated bit maps. In the rest of the paper, we describe our schemes more precisely and provide experimental evidence that shows their efficacy.

| $Rule$ | $Field_1$ | $Field_2$ |
|---|---|---|
| $F_1$ | $X$ | $A_1$ |
| $F_2$ | $X$ | $A_2$ |
| $F_3$ | $X$ | $A_3$ |
| ... | ... | ... |
| $F_{30}$ | $X$ | $A_{30}$ |
| $F_{31}$ | $X$ | $Y$ |
| $F_{32}$ | $A_1$ | $Y$ |
| $F_{33}$ | $A_2$ | $Y$ |
| ... | ... | ... |
| $F_{60}$ | $A_{29}$ | $Y$ |
| $F_{61}$ | $A_{30}$ | $Y$ |

Figure 4: An example of rearranging the database in figure 3 in order to improve the performance of the aggregation technique. The size of the aggregate $A = 2$

# 5 The ABV Algorithm

In this section we describe our new ABV algorithm. We start by describing the algorithm with aggregation only. We then describe the algorithm with aggregation and rearrangement.

## 5.1 Aggregated Search

We start by describing more precisely the basic algorithm for a two level hierarchy (only one aggregate bit vector) and without rearrangement of rules.

For the general $k$-dimension packet classification problem our algorithm uses the $N$ rules of the classifier to precompute $k$ tries, $T_i$, $1 \leq i \leq k$. A trie $T_i$ is associated with field $i$ from the rule database; it consists of a trie built on all possible prefix values that are found in field $i$ in any rule in the rule database.

Thus a node in trie $T_i$ is associated with a valid prefix $P$ if there is at least one rule $R_l$ in the classifier having $R_i^l = P$, where $R_i^l$ is the prefix associated with field $i$ of rule $R_l$. For each such node two bit vectors are allocated. The first one has $N$ bits and is identical to the one that is assigned in the BV algorithm. Bit $j$ in this vector is set if and only if rule $R_j$ in the classifier has $P$ as a prefix of $R_i^j$. The second bit vector is computed based on the first one using aggregation. Using an aggregation size of $A$, a bit $k$ in this vector is set if and only if there is at least one rule $R_n$, $A \times k \leq n \leq A \times k + 1 - 1$ for which $P$ is a prefix of $p_i^n$. The aggregate bit vector has $\lceil \frac{N}{A} \rceil$ bits.

When a packet arrives at a router, a longest prefix match is performed for each field $H_i$ of the packet header in trie $T_i$ to yield a trie node $N_i$. Each node $N_i$ contains both the bit vector ($N_i.bitVector$) and the aggregate vector ($N_i.aggregate$) specifying the set of filters or rules which matches prefix $H_i$ on the dimension $i$. In order to identify the subset $S$ of filters which are a match for the incoming packet, the AND of $N_i.aggregate$ is first computed.

Whenever position $j$ is 1 in the AND of the aggregate vectors, the algorithm performs an AND operation on the regular bit vectors for each chunk of bits identified by the aggregate bit $j$ (bits $A \times j, \ldots, A \times (j+1) - 1$). If a value of 1 is obtained for bit $m$, then the rule $R_m$ is part of set $S$. However, the algorithm selects the rule $R_t$ with the lowest value of $t$.

Thus the simplest way to do this is to compute the matching rules from the smallest position to the largest, and to stop when the first element is placed in $S$. We have implemented this scheme. However, in what follows we prefer to allow arbitrary rearrangement of filters. To support this, we instead compute *all* matches. We also assume that each rule is associated with a cost (that can easily be looked up using an array indexed by the rule position) that reflects its position before rearrangement. We only return the lowest cost (i.e.. the filter with the smallest position number in the original database created by the manager) filter. As described earlier, this simple trick allows us to rearrange filters arbitrarily without regard for whether they intersect or not.

The pseudocode for this implementation is:

```
1  Get Packet P(H_1, ..., H_k);
2  for i ← 1 to k do
3      N_i ← longestPrefixMatchNode(Trie_i, H_i);
4  Aggregate ← 11...1;
5  for i ← 1 to k do
6      Aggregate ← Aggregate ⋂ N_i.aggregate;
7  BestRule ← Null;
8  for i ← 0 to sizeof(Aggregate) − 1 do
9      if Aggregate[i] == 1 then
10         for j ← 0 to A − 1 do
11             if ⋂_{l=1}^{k} N_l.bitVect[i × A + j] == 1 then
12                 if R_{i×A+j}.cost < BestRule.cost then
13                     BestRule = R_{i×A+j};
14 return BestRule;
```

## 5.2 A Sorting Algorithm for Rearrangement

One can see that false matches reduce the performance of the algorithm introduced in the previous section, with lines *10 . . 13* in the algorithm being executed multiple times. In this section, we introduce a scheme which rearranges the rules such that, wherever possible, multiple filters which match a specific packet are placed close to each other. The intent, of course, is that these multiple matching filters are part of the same aggregation group. Note that the code of the last section allows us to rearrange filters arbitrarily as long as we retain their cost value.

Recall that Figure 4 was the result of rearranging the original filter database from Figure 3 by grouping together the entries having $X$ as a prefix on the first field and then the entries having $Y$ as a prefix in the second field. After rearranging entries, a query to identify the filter which matches the header $(X, Y)$ of a packet takes about half the time it would take before rearrangement. This is because regrouping the entries reduces the number of false matches to zero.

To gain some intuition into what optimal rule arrangement should look like, we examined four real life firewall databases. We noticed that there were a large number of entries having prefixes of either length 0 or 32. This suggests a simple idea: if we arbitrarily pick a field and group together first the entries having prefixes of length 0 (such wildcard fields are very common), then the prefixes of length 1, and so on until we reach a group of all size 32 prefixes. Within each group of similar length prefixes, we sort by prefix value, thereby grouping together all filters with the same prefix value. This will clearly (for the field picked) place all the wildcard fields together, and all the length 32 prefixes together. Intuitively, this rule generalizes the transformation from Figure 3 to Figure 4. In the rest of the paper, we refer to this process of rearrangement as *sorting on a field*.

Suppose we started by sorting on field $i$. There may be a number of filters with prefix $X$. Of course, we can continue this process recursively on some other field $j$, by sorting all entries containing entry $X$ using the same process on field $j$. (This clearly leaves the sorting on field $i$ unchanged.)

Our technique of moving the entries in the database creates large areas of entries sharing a common subprefix in one or more fields. If there are entries having fields sharing a common subprefix with different lengths, it separates them at a comfortable distance such that false matches are reduced.

A question each rearrangement scheme should address is correctness. In other words, for any packet $P$ and any filter database $C$ which, after rearrangement is transformed into a database $C'$, the result of the packet classification problem having as entries both $(C, P)$ and $(C', P)$ should be the same. One can see that the ABV algorithm guarantees this because an entry is selected based on its cost. Note that (by contrast) in the BV scheme an entry is selected based on its position in the original database.

Our rearranging scheme uses a recursive procedure which considers the entries from a subsection of the original database identified through the $first$ and $last$ element. The rearrangement is based on the prefixes from the field *col* provided as an argument. The procedure groups the entries based on the length of the prefixes; for example first it considers the prefixes from field 1, and creates a number of groups equal to the number of different prefix lengths in field 1. Each group is then sorted so that entries having the same prefix are now adjacent. The entries having the same prefix then create subgroups; the procedure continues for each subgroup using the next fields that needs to be considered; the algorithm below considers fields in order from 1 to $k$. Note that one could attempt to optimize by considering different orders of fields to sort. We have not done so yet because our results seem good enough without this further degree of optimization.

A pseudocode description of the algorithm is given below. The algorithm is called initially by setting the parameters $first = 1, last = N, col = 1$

ARRANGE-ENTRIES($first, last, col$)

1 **if**(there are no more fields) or ($first == last$) **then return**;
2 **for** (each valid size of prefixes) **then**
3 group all the elements with the same size together;
4 sort the previously created groups. Create subgroups made up
of elements having the same prefixes on the field *col*
5 **for** (each subgroup S with more than two elements) **then**
6 **Arrange-Entries**($S.first, S.last, col + 1$);

# 6 Evaluation

In this section we consider how the ABV algorithm can be implemented, and how it performs on both real firewall databases and synthetically created databases. Note that we need synthetically created databases to test the scalability of our scheme because real firewall databases are quite small.

First, we consider the complexity of the preprocessing stage and the storage requirements of the algorithm. Then, we consider the query performance and we relate it to the performance of the BV algorithm. The speed measure we use is the worst case number of memory accesses to be executed across *all possible packet headers*. Fortunately, computing this number does not entail generating all possible packet headers. This is because packet headers fall into equivalence classes based on distinct cross-products [15]; a distinct cross-product is a unique combination of distinct prefix values for each header field.

Since each packet that has the same cross-product is matched to the same node $N_i$ (in trie $T_i$) for each field $i$, each packet that has the same cross-product will behave identically in both the BV and ABV schemes. Thus it suffices to compute worst case search times for all possible cross-products. Our first algorithm was quite time consuming for large rule databases of around 20,000 rules (one test run can take 6 hours on a modern SPARC), it is feasible. Note also that such long computation time is only required for *testing* the worst-case performance of the algorithms, and not for the preprocessing or running of the actual algorithm itself. However, we have recently improved the algorithm by several orders of magnitude (order of minutes) by using a clever idea exploited in the RFC scheme [4] to equivalence cross-products while computing crossproducts pairwise. We have a number of other ideas to speed up the testing to what we believe are seconds. With the new algorithm in place we hope to test much larger databases of up to a million rules.

One can easily see that because of possible false matches in the rule database, our ABV algorithm may (in theory) have a poorer worst behavior than BVS (because it can potentially retrieve all aggregates as well as all bits in the bit vectors). However through our experiments we show that *ABV outperforms BV by more than an order of magnitude on both real life databases and synthetic databases.* We tried to create synthetic databases by randomly injecting elements (e.g., wildcards) which exacerbate false matches in order to stress ABV as much as we could. Despite this, ABV performed well, as we show below.

## 6.1 ABV Preprocessing

We consider the general case of a $k$ dimension classifier. $k$ tries $T_i$, $1 \leq i \leq k$ are built, one for each dimension. Each trie has two different types of nodes depending if they are associated or not with valid prefixes. The total number of nodes in the tries is on the order of $O(N \times k)$, where $N$ is the number of entries in the classifier (i.e., rule database). Two bit vectors are associated with each valid prefix node. One bit vector is identical with the one used in BV scheme and requires $\lceil \frac{N}{WordSize} \rceil$ words of data. The second bit vector is the aggregate of the first one; it contains $\lceil \frac{N}{A} \rceil$ bits of data which means that it requires $\lceil \frac{N}{A \times WordSize} \rceil$ words of memory ($A$ is the size of the aggregate). Building both bit vectors requires an $O(N)$ pass through the rule database for each valid node of the trie. Thus the preprocessing time is $O(N^2 k)$.

One can easily see from here that the memory requirements for ABV are slightly higher than that of BVS; however for an aggregate size greater than 32 (e.g., software), ABV differs by less than 3%, while for an aggregate size of 500 (e.g., hardware), it is below 0.2%.

The time required for insertion or the deletion of a rule in ABV is of the same complexity as BV. This is because the aggregate bit vector is updated each time the associated bit vector is updated. Note that updates can be expensive because adding a filter with a prefix $X$ can potentially change the bit maps of several nodes. However, in practice it is rare to see more than a few bitmaps change, possibly because filter intersection is quite rare [4]. Thus incremental update, though slow in the worst case, is quite fast on the average. In the last section, we describe a modified algorithm that can guarantee better worst-case update times.

## 6.2 Experimental Platform

We used two different types of databases. First we used a set of four industrial firewall databases that we obtained from earlier researchers. For privacy reasons we are not allowed to disclose the name of the companies or the actual databases. Each entry in the database contains a $5-$*tuple* (source IP prefix, destination IP prefix, source port

number(range), destination port number(range), protocol). We call these databases $DB_1 \ldots DB_4$. The database characteristics are presented in Table 5.

| $Filter$ | $No.of Rules$ | $No.of Rules in Prefix Format$ |
|---|---|---|
| $DB_1$ | 266 | 1640 |
| $DB_2$ | 279 | 949 |
| $DB_3$ | 183 | 531 |
| $DB_4$ | 158 | 418 |

Figure 5: The sizes of the firewall databases we use in the experiments

The third and fourth field of the database entries are represented by either port numbers or range of port numbers. We convert them to valid prefixes using the technique described in [15].

The following characteristics have important effects on the results of our experiments:

1. Most prefixes have either a length of 0 or 32. There are some other prefixes with length of $21, 23, 24$ and 30.
2. No prefix contains more than 4 matching rules for each field.
3. The destination and source prefix fields in roughly half the rules were wildcarded (by contrast, [3] only assumes at most 20% of the rules have wildcards in their experiments), and roughly half the rules have $>= 1024$ in the port number fields. Thus the amount of overlap within each dimension was large.
4. No packet matches more than 4 rules.

The second type of databases are randomly generated 2 field (sometimes called two dimensional) databases using random selection from five publicly available routing tables ( [7]). We used the snapshot of each table taken on September 12, 2000. An important characteristic of these tables is the prefix length distribution, described in the table 6

| $RoutingTable$ | 8 | $9 \ldots 15$ | 16 | $17 \ldots 23$ | 24 | $25 \ldots 32$ |
|---|---|---|---|---|---|---|
| $Mae-East$ | 10 | 133 | 1813 | 9235 | 11405 | 58 |
| $Mae-West$ | 15 | 227 | 2489 | 11612 | 16290 | 39 |
| $AADS$ | 12 | 133 | 2204 | 10144 | 14704 | 55 |
| $PacBell$ | 12 | 172 | 2665 | 12808 | 19560 | 54 |
| $Paix$ | 22 | 560 | 6584 | 28592 | 49636 | 60 |

Figure 6: Prefix Length Distribution in the routing tables, September 12, 2000

Recall that the problem is to generate a synthetic database that is larger than our sample industrial databases to test the scalability of the ABV and BVS algorithms. The simplest way to generate a two-dimensional classifier on source and destination prefixes of size $N$, for varying values of $N$, would be as follows. We pick randomly two prefixes from any of the five routing tables, one for the source field and one for the destination field. We now iterate this procedure $N$ times to create $N$ rules, for any specified value of $N$.

Unfortunately, such a simple generation technique may be unrealistic. This is because the real routing databases ( [7]) have either no or at most one prefix of length 0. Thus if we use random selection from a routing table of say 80,000 prefixes, we are very unlikely to generate a rule that has a zero length prefix in either field. We have already noted that zero length prefixes are very common in real firewall rule databases. Thus, in addition to random selection from a routing table, we also allow a controlled injection of rules with zero length prefixes, where the injection is controlled by a parameter that determines the percentage of zero length prefixes. For example, if the parameter specifies that 20% of the rules have a zero length prefix, then in selecting a source or destination field for a rule, we first pick a random number between 0 and 1; if the number is less than 0.2 we simply return the zero length prefix; else, we pick a prefix randomly from the specified routing table.

A similar construction technique is also used in [3] though they limit wildcard injection to 20% when our firewall databases have the number of wildcards in a field to be closer to 50%. [3] also uses another technique based on

extracting all pairs of source-destination prefixes from traces and using these as filters. They show that the two methods differ considerably with the random selection method providing better results because the trace method produces more overlapping prefix pairs. We realize that; however, rather than using a trace, we prefer to stress ABV further by adding a controlled injection of groups of prefixes that share a common prefix to produce more overlapping prefix pairs (see next paragraph). Indeed, our second method stresses ABV more, as is consistent with [3]; we prefer the controlled injection because it allows us to investigate the effect of varying the injection rate rather than being limited to that provided by a trace.

We do vary the wildcard injection parameter and see how ABV performs as we increase the percentage of zero length prefix rules. However, we also have another knob that can stress ABV further by increasing the degree of rules that a given rule overlaps with. It is easy to see that groups of prefixes that share a common subprefix are crucial for increasing false matches. Now in practice, such prefixes should be very rare; such prefixes occur very rarely in the databases in [7]. Thus random selection will not create many such prefixes.However, to artificially stress ABV, we found a technique to randomly create databases which have a potentially large number of prefixes that have subprefixes (e.g., the sequence *, 1*, 10*, 101* is a sequence of prefixes, each of which is a subprefix of later prefixes in the sequence).

When we inject a large amount of zero length prefixes and subprefixes, we find that ABV without rearrangement begins to do quite poorly, a partial confirmation that we are stressing the algorithm. Fortunately, ABV with rearrangement still does very well.

Finally, we did some limited testing on synthetic 5-dimensional databases. We generated the source and destination fields of rules as in the synthetic 2-dimensional case; for the remaining fields (e.g., ports) we picked port numbers randomly according to the distribution found in our larger real database. Once again, we find that ABV scales very well compared to Lucent. We will report more complete testing on such 5-dimensional fields in the full paper.

### 6.3 Performance Evaluation on Industrial Firewall Databases

We experimentally evaluate ABV algorithm on a number of four industrial firewall databases described in the figure 5. The rules in the databases are converted into prefix format using the technique described in [9] . The memory space that is used by each of them can be estimated based on the number of nodes in the tries, and the number of nodes associated with valid prefixes. We provide these values in Figure 7. A node associated with a valid prefix carries a bit vector of size equal to $\lceil \frac{N}{32} \rceil$ words and an aggregate bit vector of size $\lceil \frac{N}{32 \times 32} \rceil$ words. We used a word size equal to 32; we also set the size of the aggregate to 32. We used only one level of aggregation.

Our performance results are summarized in Figure 8. We consider the number of memory accesses required by the ABS algorithm once the nodes associated with the longest prefix match are identified in the trie in the worst case scenario. The first stage of finding the nodes in the tries associated with the longest prefix matching is identical in both algorithms ABV and BVS (and depends on the longest prefix match algorithm used; an estimate for the fastest algorithms is around $3-5$ memory accesses per field). Therefore we do not consider it in our measurements. The size of a memory word is 32 bits for all the experiments we considered. Note that in a hardware implementation it is quite easy to have a value of about 500 bits/word, using a wide internal bus.

The results show that ABV without rearrangement outperforms BVS, with the number of memory accesses being reduced by a factor of $27\% \ldots 54\%$. By rearranging the elements in the original database, the performance of ABV can be increased by further reducing the number of memory accesses by a factor of $40\% \ldots 75\%$. Our results also show that for the databases we considered it was sufficient to sort the elements only by the size of the prefix length in one field (and not recursively sort using other fields).

| *Filter* | *No.of Nodes* | *No. of Valid Prefixes* |
|---|---|---|
| $DB_1$ | 980 | 188 |
| $DB_2$ | 1242 | 199 |
| $DB_3$ | 805 | 127 |
| $DB_4$ | 873 | 143 |

Figure 7: The total number of nodes in the tries and the total number of nodes associated with valid prefixes for the industrial firewall databases

| *Filter* | *BVS* | *ABV* | | |
|---|---|---|---|---|
| | | *unsorted* | *One Field Sorted* | *Two Fields Sorted* |
| $DB_1$ | 260 | 120 | 75 | 65 |
| $DB_2$ | 150 | 110 | 50 | 50 |
| $DB_3$ | 85 | 60 | 50 | 50 |
| $DB_4$ | 75 | 55 | 45 | 45 |

Figure 8: The total number of memory accesses in the worst case scenario for the industrial firewall databases. Several cases are taken into consideration: unsorted database (no rearrangement), database sorted one field only, and sorted on two fields.

## 6.4 Experimental Evaluation on Synthetic 2D Databases

Thus on real firewall databases our ABV algorithm outperforms the BVS algorithm. However, for small databases the improvement we can obtain is limited. We also need to evaluate the scalability of our algorithm. In this section we evaluate how our algorithm might behave with larger classifiers. Thus we are forced to synthetically generate larger databases.

However, the size of a classifier is not the only parameter one needs to consider. If we had considered only the size of the classifier and the actual characteristics of the classifiers (as we found in the four real databases) to have most of prefixes grouped in two different groups, one with a length of 0 and another one with a length of 32, than our results would look impressive. In such databases, false matches are very rare. Thus, as said earlier, we injected a controlled number of zero length prefixes as well as a number of prefixes that had subprefixes.

As described earlier, we create our synthetic two-dimensional database of prefixes from routing tables available for public at [7]. The characteristics of the routing tables we used are listed in Figure 6.

*Effect of zero-length prefixes:* We first consider the effect of prefixes of length zero on the number of memory accesses in the worst case scenario. Entries containing prefixes of length zero are randomly generated as described earlier. The results are displayed in Figure 9. The presence of prefixes of length zero randomly distributed through the entire database has a heavy impact on the number of memory accesses which are done to serve a query. If there are no prefixes of length zero in our synthetic database the number of memory accesses for a query using ABV scheme is a factor of $8 \ldots 27$ times less than the BV scheme.

However, by inserting around 20% prefixes of length zero in the database we found that the ABV scheme (without rearrangement) needs to read all the words from both the aggregate and the bit vector; in such a scenario, clearly the BV scheme does better by a small amount. Fortunately, by sorting the entries in the database using the technique described in Section 5.2, the number of memory accesses for the worst case scenario for ABV scheme is reduced to values close to the values of a database (of the same size) without prefixes of length zero.

Figure 10 shows the distribution of the number of memory accesses as a function of number of entries in the synthetic database. The databases are generated using randomly picked prefixes from the MAE-East routing table, and by random injection of prefixes of length zero. Note that the sorted ABV scheme reduces the number of memory accesses by more than 20 times comparing with BVS scheme, with the difference growing larger as the database size gets larger.

*Injecting Subprefixes:* A second feature which may directly affect the overall performance of our algorithm is the presence of entries having prefixes which share common subprefixes. These entries form groups of nodes associated with valid prefixes which share a common subprefix. These groups effectively create subtries. The root of each subtrie is the longest common subprefix of the group. Let $W$ be the depth of the subtrie, and consider a filter database with $k$ dimensions. It is not hard to see that if we wish to stress the algorithm, we need to increase $W$.

Our next experiment attempts to measure the effect of such prefixes on the overall performance of the ABV algorithm. We randomly insert elements from 20 different such groups. In order to do so, we first extract a set of 20 prefixes having length equal to 24. We call this set $L$. There are no two elements in L which share the same $16-$bit prefix. On the second step, for each element in L we insert eight other elements with the length in the range $(24 - W) \ldots 23$. These elements are subprefixes of the element in $L$.

We generate the filter database by randomly picking prefixes from both the routing table and from the new created set $L$. We can control the rate with which elements from $L$ are inserted in the filter database. We measure the effect of different tries heights $W$ as well as the effect of having different ratios of such elements. The results are displayed in Figures 11, 12, and 13.

| *Size* | *DB* | *BVS* | *Percentage of prefixes of length zero; sorted(s)/usorted(u)* | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1$u$ | 1$s$ | 2$u$ | 2$s$ | 5$u$ | 5$s$ | 10$u$ | 10$s$ | 20$u$ | 20$s$ | 50$u$ | 50$s$ |
| 1$K$ | *AADS* | 64 | 8 | 18 | 8 | 24 | 8 | 48 | 12 | 66 | 10 | 66 | 10 | 66 | 10 |
| 1$K$ | *EAST* | 64 | 8 | 12 | 10 | 26 | 10 | 54 | 10 | 66 | 12 | 66 | 12 | 66 | 10 |
| 1$K$ | *WEST* | 64 | 6 | 12 | 8 | 24 | 10 | 56 | 10 | 62 | 12 | 66 | 10 | 66 | 10 |
| 1$K$ | *PB* | 64 | 6 | 12 | 10 | 24 | 8 | 48 | 10 | 64 | 10 | 66 | 8 | 66 | 10 |
| 1$K$ | *PAIX* | 64 | 8 | 12 | 8 | 24 | 10 | 48 | 10 | 66 | 10 | 66 | 8 | 66 | 10 |
| 2$K$ | *AADS* | 126 | 10 | 24 | 12 | 32 | 12 | 86 | 14 | 118 | 14 | 130 | 12 | 130 | 12 |
| 2$K$ | *EAST* | 126 | 10 | 28 | 14 | 58 | 12 | 84 | 14 | 126 | 14 | 130 | 14 | 130 | 14 |
| 2$K$ | *WEST* | 126 | 10 | 28 | 12 | 38 | 12 | 80 | 12 | 126 | 12 | 130 | 12 | 130 | 12 |
| 2$K$ | *PB* | 126 | 10 | 22 | 12 | 42 | 12 | 86 | 12 | 126 | 12 | 130 | 14 | 130 | 14 |
| 2$K$ | *PAIX* | 126 | 10 | 18 | 12 | 40 | 10 | 86 | 12 | 126 | 12 | 130 | 14 | 130 | 14 |
| 5$K$ | *AADS* | 314 | 16 | 50 | 18 | 86 | 20 | 216 | 20 | 306 | 22 | 324 | 20 | 324 | 20 |
| 5$K$ | *EAST* | 314 | 16 | 50 | 18 | 76 | 18 | 216 | 20 | 298 | 20 | 324 | 22 | 324 | 18 |
| 5$K$ | *WEST* | 314 | 16 | 48 | 18 | 114 | 20 | 224 | 18 | 310 | 20 | 324 | 20 | 324 | 20 |
| 5$K$ | *PB* | 314 | 16 | 38 | 18 | 72 | 20 | 226 | 18 | 304 | 22 | 324 | 20 | 324 | 20 |
| 5$K$ | *PAIX* | 314 | 16 | 40 | 20 | 100 | 20 | 226 | 18 | 310 | 18 | 324 | 18 | 324 | 18 |
| 10$K$ | *AADS* | 626 | 26 | 92 | 30 | 186 | 28 | 426 | 28 | 600 | 30 | 646 | 32 | 646 | 32 |
| 10$K$ | *EAST* | 626 | 26 | 78 | 30 | 196 | 28 | 426 | 34 | 588 | 34 | 644 | 32 | 646 | 30 |
| 10$K$ | *WEST* | 626 | 26 | 82 | 30 | 146 | 28 | 420 | 28 | 594 | 28 | 646 | 30 | 646 | 30 |
| 10$K$ | *PB* | 626 | 26 | 82 | 30 | 146 | 28 | 432 | 30 | 610 | 28 | 646 | 30 | 646 | 30 |
| 10$K$ | *PAIX* | 626 | 26 | 78 | 28 | 146 | 28 | 432 | 30 | 610 | 30 | 646 | 30 | 646 | 28 |
| 20$K$ | *AADS* | 1250 | 48 | 158 | 50 | 332 | 52 | 832 | 52 | 1202 | 50 | 1292 | 50 | 1292 | 50 |
| 20$K$ | *EAST* | 1250 | 48 | 148 | 48 | 346 | 50 | 860 | 52 | 1212 | 54 | 1288 | 52 | 1292 | 52 |
| 20$K$ | *WEST* | 1250 | 48 | 156 | 50 | 296 | 50 | 806 | 48 | 1228 | 54 | 1290 | 52 | 1292 | 52 |
| 20$K$ | *PB* | 1250 | 46 | 138 | 52 | 336 | 50 | 858 | 52 | 1186 | 52 | 1290 | 52 | 1292 | 50 |
| 20$K$ | *PAIX* | 1250 | 46 | 158 | 48 | 336 | 48 | 878 | 52 | 1200 | 54 | 1292 | 52 | 1292 | 52 |

Figure 9: The total number of memory accesses in the worst case scenario for synthetic two-dimensional database of various sizes, with a variable percentage of prefixes with length zero.

The figures shows that, at least for a model of random insertion *the height W does not have a large impact on the number of false matches*. A slight increase in this number can be seen only when there are about 90% of such elements inserted in the measured database. We consider next the ratio of these elements in the total number of prefixes in the database. Their impact on the total number of memory accesses is lower than the impact of prefixes of length zero. When their percentage is roughly 50%, the number of memory accesses using ABV algorithm (without sorting) is about 10 times lower than the number of memory accesses using the BVS algorithm. This number is further improved by sorting the original database by a factor of about 30%. These numbers were for a database with $20K$ entries.

### 6.4.1 Evaluating ABV with Two Levels of Aggregation

So far our version of ABV for 2D databases has used only 1 level of aggregation. Even for a $32,000$ rule database, we would use an aggregate bit vector of length equal to $32,000/32 = 1000$. However, if only a few bits are set in such an aggregate vector, it is a waste of time to scan all 1000 bits. The natural solution, for aggregate bit vectors greater than $A^2$ (1024 in our example), is to use a *second* level of hierarchy. With $A = 32$, a second level can handle rule databases of size equal to $32^3 = 32K$. Since this approaches the limits of the largest database that we can test (for worst-case performance), we could not examine the use of any more levels of aggregation.

To see whether 2 levels provides any benefit versus using 1 level only, we simulate the behavior of the 2 level ABV algorithm on our larger synthetic databases. (It makes no sense to compare the performance of 2 levels versus one level for our small industrial databases.). For lack of space, in Figure 14 we only compare the performance of two versus one level ABV on synthetic databases (of sizes 5000, 10000, and 20000) generated from MAE-EAST by

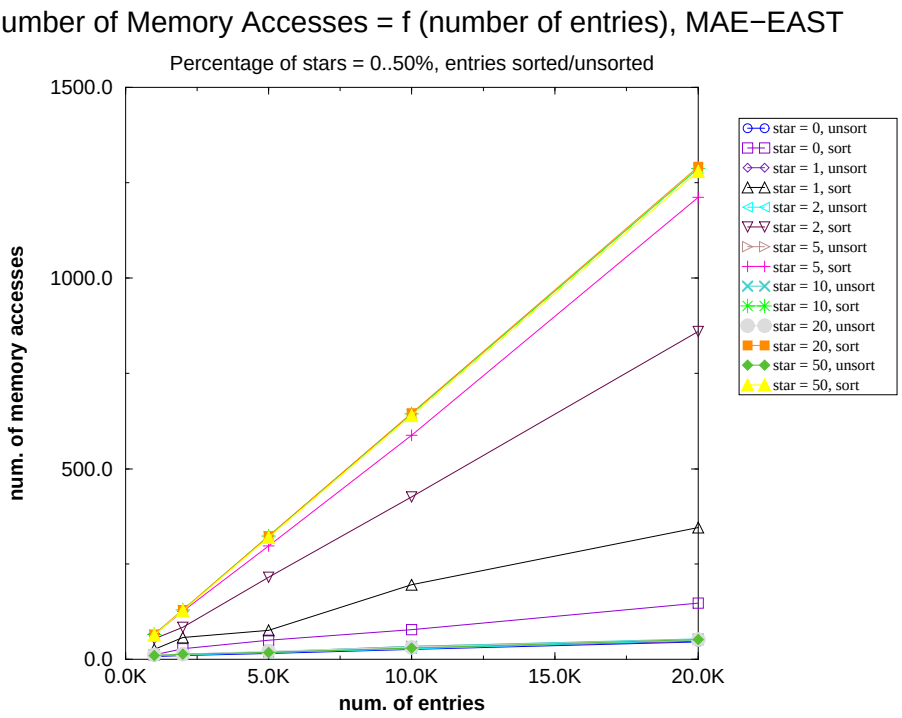


Figure 10: The number of memory accesses as a function of number of database entries. ABV outperforms BVS scheme by a factor greater than twenty on a sorted synthetic database having prefixes of length zero randomly inserted. The synthetic databases were generated using the MAE-EAST routing table [7]

injecting 0% to 50% prefixes of zero length. In all cases we use the ABV algorithm with rearrangement (i.e., the best case for both one and two levels).

The results show that using an extra level of aggregation reduces the worst number of memory accesses by 60% for the largest databases. For the smallest database (5000) the improvement is marginal, which accords with our intuition — although the algorithm does not touch as many leaf bits for the database of size 5000, this gain is offset by the need to read another level of aggregate bits. However, at a database size of 10,000 there is a clear gain. While we need much more work to validate our hypothesis, the results do suggest that the number of memory accesses for a general multilevel ABV can scale logarithmically with the size of the rule database, allowing potentially very large databases.

## 6.5 Performance Evaluation using Synthetic 5-dimensional databases

So far, we have tested scalability only on randomly generated 2-dimensional database. However, there are existing schemes such as grid-of-tries and FIS trees that also scale well for this special case. Thus in this section we briefly describe initial results of our tests for synthetic 5-dimensional databases. The testing is still not complete because we only recently improved our test methodology to check the worst case (note that for 5 dimensions the number of crossproducts grow as $N^5$ and so even very small databases of size 5000 were hard to do). However, with the recent use of ideas in [4] we were able to cut down testing time and obtain the following promising results. We will expand on the complete set of results in the final paper.

We investigated the scalability of the ABV scheme on five dimensional databases. The industrial firewall databases we use have a maximum size of 1640 rules which limits the possibility to be used in order to show the scalability of our scheme. To avoid this limitation we generated synthetic generated five dimension databases using the IP prefix addresses from MAE-EAST as in the two-dimensional case, and port number ranges and protocol fields using the distributions of values and ranges found in the industrial firewall databases.

| *Size* | *DB* | *BVS* | *W = 4* | | | | | *W = 6* | | | | | *W = 8* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 10 | 20 | 50 | 90 | 1 | 10 | 20 | 50 | 90 | 1 | 10 | 20 | 50 | 90 |
| 1K | *AADS* | 64 | 8 | 12 | 18 | 38 | 48 | 8 | 14 | 18 | 36 | 54 | 8 | 12 | 18 | 38 | 52 |
| 1K | *EAST* | 64 | 8 | 10 | 20 | 40 | 52 | 8 | 12 | 26 | 38 | 56 | 8 | 12 | 20 | 36 | 52 |
| 1K | *WEST* | 64 | 8 | 12 | 18 | 36 | 52 | 8 | 14 | 22 | 34 | 52 | 8 | 10 | 18 | 36 | 56 |
| 1K | *PB* | 64 | 8 | 14 | 16 | 38 | 50 | 6 | 12 | 16 | 36 | 50 | 8 | 10 | 20 | 38 | 54 |
| 1K | *PAIX* | 64 | 6 | 12 | 18 | 38 | 50 | 8 | 12 | 16 | 38 | 50 | 8 | 10 | 20 | 38 | 52 |
| 5K | *AADS* | 314 | 16 | 30 | 54 | 134 | 152 | 16 | 28 | 56 | 132 | 156 | 16 | 28 | 62 | 134 | 154 |
| 5K | *EAST* | 314 | 16 | 28 | 56 | 124 | 144 | 16 | 32 | 56 | 126 | 148 | 16 | 30 | 50 | 120 | 162 |
| 5K | *WEST* | 314 | 16 | 34 | 48 | 124 | 158 | 16 | 34 | 56 | 124 | 154 | 16 | 38 | 56 | 130 | 158 |
| 5K | *PB* | 314 | 16 | 32 | 58 | 134 | 154 | 16 | 32 | 58 | 134 | 152 | 18 | 30 | 52 | 130 | 188 |
| 5K | *PAIX* | 314 | 16 | 32 | 50 | 138 | 174 | 16 | 30 | 56 | 144 | 170 | 16 | 32 | 48 | 136 | 172 |
| 10K | *AADS* | 626 | 26 | 52 | 98 | 232 | 202 | 26 | 50 | 96 | 192 | 226 | 26 | 50 | 92 | 214 | 236 |
| 10K | *EAST* | 626 | 28 | 54 | 96 | 228 | 214 | 26 | 50 | 96 | 244 | 234 | 26 | 50 | 94 | 194 | 226 |
| 10K | *WEST* | 626 | 28 | 50 | 96 | 186 | 246 | 26 | 52 | 104 | 230 | 214 | 26 | 50 | 86 | 196 | 222 |
| 10K | *PB* | 626 | 26 | 52 | 94 | 198 | 230 | 28 | 54 | 104 | 212 | 208 | 26 | 58 | 98 | 202 | 232 |
| 10K | *PAIX* | 626 | 26 | 52 | 96 | 208 | 262 | 26 | 50 | 96 | 204 | 258 | 26 | 52 | 90 | 222 | 234 |
| 20K | *AADS* | 1250 | 48 | 94 | 172 | 234 | 306 | 46 | 88 | 170 | 352 | 310 | 48 | 92 | 156 | 300 | 320 |
| 20K | *EAST* | 1250 | 48 | 88 | 168 | 308 | 254 | 48 | 90 | 154 | 274 | 292 | 48 | 92 | 176 | 304 | 326 |
| 20K | *WEST* | 1250 | 48 | 102 | 164 | 284 | 274 | 48 | 96 | 176 | 352 | 300 | 48 | 96 | 178 | 334 | 304 |
| 20K | *PB* | 1250 | 48 | 92 | 168 | 354 | 280 | 48 | 94 | 172 | 280 | 288 | 48 | 90 | 168 | 286 | 280 |
| 20K | *PAIX* | 1250 | 48 | 96 | 180 | 306 | 318 | 46 | 94 | 178 | 274 | 312 | 48 | 86 | 172 | 290 | 280 |

Figure 11: The total number of memory accesses in the worst case scenario for a synthetic two-dimension database having injected a variable percentage of elements which share a common subprefix. The database is *not sorted*. $W$ is is the depth of the subtrie created by these elements. The values under the BVS estimates the number of memory accesses using the BV scheme. All the other values are associated with the ABV scheme.

Our results are shown in Figure 15in which the ABV scheme outperforms the BVS scheme by more than one order of magnitude. The only results we have shown use no wildcard injection. The results for larger wildcard injections appear to be similar to before (though sorting on possible multiple fields appears to be even more crucial). Note that for a 5 dimensional database with 21,226 rules the Lucent scheme required 3320 memory accesses while ABV with an aggregation size of 32 required only 140 memory accesses.

# 7 Theoretical Worst Case Bounds for ABV

We try to find an upper bound for the maximum number of memory accesses in the worst case for the ABV algorithm for a $K$ field, $N$ rules classifier $\{R_i\}_{0 \le i \le N-1}$. To get an intuition let's consider first the figure 16. The pattern on the left identifies a 2 dimension sorted database. For simplicity we assume that the maximum length of the prefixes is 4. One can easily notice that a packet with a header *(X,Y)* does not find any match in this database once $X$ starts with an 0 and $Y$ starts with an 1. However there are a number of at least four false matches if an ABV scheme is used and the aggregate window size is equal with 2. Let's consider now a sorted 3 dimension filter database like the one shown on the right in figure 16 having four different lengths of prefixes and a packet with the header fields *(Z, X, Y)*. There is no matching filter for this packet in the database, however in the ABV scheme there is a number of $4 \times 4 = 16$ false matchings. Generalizing the observation above:

**Lema 1** *There is a $K$ dimension database with the number of different length of prefixes equal with $W$ for which under a conveniently chosen aggregation window the number of false matches is $2 \times W^{k-1} - 1$ and this is maximum.*

**Proof 1** *The proof is by induction. Let $T(k)$ the maximum number of false matches for a database with k dimensions and $W$ different prefix lengths. For $k = 2$, $T(2) = 2 \times W - 1$ a false match may exist both between entries having the same prefix length on the first dimension or between entries having adjacent prefix lengths on the first dimension. For the general case, $T(k) = W \times T(k-1) + (W-1)$ which can be immediately shown that implies $T(k) = 2 \times W^{k-1} - 1$.*

| *Size* | *DB* | *W = 4* | | | | | *W = 6* | | | | | *W = 8* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 10 | 20 | 50 | 90 | 1 | 10 | 20 | 50 | 90 | 1 | 10 | 20 | 50 | 90 |
| 1K | *AADS* | 8 | 10 | 14 | 32 | 48 | 8 | 12 | 18 | 32 | 52 | 8 | 12 | 16 | 36 | 56 |
| 1K | *EAST* | 6 | 12 | 16 | 34 | 54 | 8 | 12 | 18 | 36 | 48 | 8 | 12 | 16 | 36 | 48 |
| 1K | *WEST* | 8 | 10 | 14 | 32 | 46 | 8 | 10 | 18 | 34 | 50 | 8 | 10 | 14 | 34 | 52 |
| 1K | *PB* | 8 | 12 | 16 | 38 | 52 | 8 | 10 | 18 | 34 | 50 | 8 | 10 | 14 | 34 | 52 |
| 1K | *PAIX* | 8 | 10 | 18 | 36 | 52 | 6 | 10 | 16 | 32 | 52 | 6 | 10 | 16 | 36 | 52 |
| 5K | *AADS* | 16 | 30 | 46 | 116 | 134 | 16 | 32 | 46 | 120 | 140 | 16 | 32 | 44 | 122 | 154 |
| 5K | *EAST* | 16 | 26 | 48 | 106 | 136 | 16 | 30 | 44 | 112 | 136 | 16 | 30 | 46 | 116 | 138 |
| 5K | *WEST* | 16 | 28 | 50 | 122 | 116 | 16 | 30 | 50 | 120 | 122 | 16 | 30 | 52 | 106 | 126 |
| 5K | *PB* | 16 | 30 | 52 | 104 | 116 | 16 | 32 | 52 | 110 | 132 | 16 | 28 | 46 | 114 | 146 |
| 5K | *PAIX* | 16 | 34 | 52 | 106 | 122 | 16 | 32 | 48 | 116 | 134 | 16 | 28 | 44 | 114 | 146 |
| 10K | *AADS* | 26 | 42 | 80 | 176 | 130 | 26 | 50 | 88 | 164 | 160 | 26 | 48 | 76 | 170 | 148 |
| 10K | *EAST* | 26 | 46 | 82 | 176 | 154 | 26 | 52 | 80 | 166 | 176 | 26 | 48 | 84 | 198 | 178 |
| 10K | *WEST* | 26 | 46 | 78 | 180 | 172 | 26 | 50 | 86 | 184 | 220 | 28 | 52 | 82 | 162 | 196 |
| 10K | *PB* | 26 | 46 | 84 | 158 | 132 | 26 | 52 | 84 | 156 | 170 | 26 | 48 | 76 | 190 | 218 |
| 10K | *PAIX* | 28 | 46 | 80 | 198 | 130 | 26 | 48 | 80 | 186 | 200 | 26 | 48 | 76 | 190 | 218 |
| 20K | *AADS* | 48 | 90 | 132 | 236 | 172 | 48 | 82 | 142 | 214 | 180 | 48 | 86 | 134 | 230 | 214 |
| 20K | *EAST* | 48 | 78 | 146 | 212 | 138 | 48 | 100 | 142 | 224 | 208 | 48 | 88 | 136 | 232 | 170 |
| 20K | *WEST* | 48 | 86 | 142 | 202 | 172 | 48 | 90 | 148 | 208 | 192 | 48 | 98 | 148 | 254 | 206 |
| 20K | *PB* | 46 | 88 | 138 | 224 | 158 | 48 | 86 | 148 | 202 | 176 | 48 | 90 | 140 | 204 | 224 |
| 20K | *PAIX* | 48 | 86 | 144 | 196 | 186 | 48 | 84 | 142 | 228 | 184 | 48 | 94 | 144 | 218 | 188 |

Figure 12: The total number of memory accesses in the worst case scenario for a synthetic two-dimension database having injected a variable percentage of elements which share a common subprefix. The database is *sorted*. $W$ is is the depth of the subtrie created by these elements. All the values are associated with the ABV scheme.

**Lema 2** *The maximum number of memory accesses for the ABV scheme with an aggregate size A for a K dimension database with N entries with W different prefix lengths, is equal with* $(2 \times W^{K-1} + 1) \times (K \times \lceil \frac{A}{M} \rceil) + \lceil \frac{N}{A \times M} \rceil$*, where M is the size of an word of memory.*

# 8 Providing Fast Worst Case Update Times

ABV and BV appear to have reasonably fast updates on the average; however it is possible to insert a rule $R$ that has wildcards in all fields which causes a bit to be set in *every* bit vector because $R$ matches all rules. This will require touching most of the memory required by the algorithm. For certain applications, such as stateful filters, worst-case update times may be necessary. We add the following ideas to ABV to allow for fast insert/delete operations:

- **Reduced Precomputation:** In the current algorithm, a bit $j$ is set for a prefix $P$ in a Field $k$ trie if the value of Field $k$ of Rule $R_j$ *matches (i.e., is a prefix of)* $P$. In the new algorithm, a bit $j$ is set for a prefix $P$ in a Field $k$ trie if the value of Field $k$ of Rule $R_j$ is *exactly equal to* $P$. For example, if $P = 101*$ and the Field $k$ value of Rule $R_j$ is $*$, then the original algorithm would have the bit set while the new one will not. Intuitively, this simple modification avoids large worst-case computation caused by examples such as the insertion of a filter of all wildcards.

- **Increased Search Time:** Despite the reduced precomputation above, we still need to collect all rules that match Field $k$ of a packet header for algorithm correctness. To do so, when traversing the trie for field $k$ for a value $P$, we must take the OR of all bit maps associated with $P$ and all valid prefixes of $P$ in the trie. However, each of the prefix nodes also have associated aggregate bit maps; thus we can ignore an aggregate at a prefix node if the summary bit is a 0.

- **Avoiding excessive reordering:** If we delete rule 5, and we have to push up the order number of all rules with number greater than 5, then every bit map will have to change. Similarly, if we insert a new rule 5 and wish all rules no less than 5 downwards, we have a similar problem. Our solution is to simply leave a hole (that can be filled later) for a delete, and to insert in arbitrary order (either to fill the first hole left by a delete, at the end, or to help incremental sorting). Notice that this is possible because we find all matches and map back to the old order number.

Thus in summary the main idea is to reduce precomputation associated by recording all matches associated with prefixes and replacing it with more work to collect these prefix matches during search. If the number of prefixes in a path is no more than 4, then this slows down search by at most a factor of 4, while allowing an order of magnitude speedup in worst-case insertion time. This may be worthwhile for some applications or a portion of the database that needs to be dynamic.

Figure 17 illustrates the modified trie construction for the simple two dimensional example database in Figure 1. For example, in Figure 17, the bit vector associated with the leftmost node corresponding to prefix 00* is now 11000000000 instead of 11011100000 in Figure 2. On the other hand, a search for prefix 00* would yield two valid prefixes 0* (with bitmap 11000000000 and the prefix 00* (with bitmap 11000000000) and the OR of these bitmaps would yield the same answer found in Figure17 which is 11011100000.

Since the new algorithm reflects a tradeoff between insert/delete times and search time (the new algorithm also adds memory for more bitmaps but this can at most double the number of bitmaps), we evaluated this tradeoff in Table 18. The table shows the worst case update time (measured in memory accesses) and the worst case lookup time for 3 algorithms: the Lucent Algorithm (BV), the original aggregated bit vector (ABV), and the modified ABV with fast insertion times (ABVI) for the four commercial databases we used.

Notice that the worst-case insert-delete costs are cut by nearly three orders of magnitude while the search time is now up to twice as comparable to the Lucent scheme. This may be an acceptable tradeoff. However, we expect for larger databases (we will finish this test for the final paper) ABVI lookups will be faster than the Lucent scheme though slower than ABV. We have also not implemented incremental sorting; thus insertion and deletion increase the number of false matches. We believe that implementing incremental sorting (such sorting can be done proportional to the number of distinct prefix lengths [13]) will make ABVI more competitive with ABV in search times.

# 9 Conclusions

The Lucent Bit Vector scheme [10] is a seminal scheme that is very amenable to hardware or software implementation. Despite the fact that it is fundamentally an $O(N)$ scheme, the use of an initial projection step allows the scheme to work with bitmaps. Taken together with memory locality, the scheme allows a nice hardware (or software) implementation. However, the scheme only scales to medium size databases.

Our paper introduces the notion of aggregation and rule rearrangement to make the Lucent bit vector (BV) scheme more scalable, creating what we call the ABV scheme. The resulting ABV scheme is at least an order of magnitude faster than the BV scheme on all tests that we performed. The ABV scheme appears to be equally simple to implement in hardware or software. While both schemes have a poor worst-case insertion time (essentially comparable), the average worst-case insertion time is small.

In comparing the two heuristics we used, aggregation by itself is not powerful enough. For example, for large synthetically generated databases with 20% of the rules containing zero length prefixes, the performance of ABV without rearrangement grew to be slightly worse than BV. However, the addition of sorting again made ABV faster by an order of magnitude. A similar effect was found for injecting subprefixes. However, a more precise statement of the conditions under ABV does well is needed.

A simple (and correct) condition is that if the number of possible matches in some field is limited to a constant, then ABV takes logarithmic time, where the logarithm uses a large radix of at least 32. However, this may be too restrictive a condition (because there could be a large number of wildcarded values in each field), and can probably be generalized.

We evaluated our implementation on both industrial firewall databases and synthetically generated databases. While we attempted to inject prefixes that could cause bad behavior, it is likely that further work is needed to find other ways to randomly generate databases that will stress ABV even further. Using only 32 bit memory accesses, we were able to do a 20,000 rule random 2 dimensional databases (with almost half the entries being wildcards) using 20 accesses using 2 levels of hierarchy. By contrast, the Lucent algorithm took 1250 memory accesses on the same

database. Similarly, for a random 5 dimensional database of 20,000 rules the Lucent scheme required 3320 memory accesses while ABV with one level of hierarchy required only 140 memory accesses. Taken together with wider memory accesses possible using either cache lines in software or wide busses in hardware, we believe our algorithm should have sufficient speed for OC-48 links even for large databases using SRAM.

We note that the hardware implementation of our algorithm can be done using similar techniques to that of the Lucent algorithm described in [10]. In particular, the initial searches on the individual tries can be pipelined with the remainder of the search through the bitmaps. The searches in the levels of the bitmap hierarchy can also be pipelined.

We also introduced a modified version of ABV, we called it ABVI, in order to allow fast update operations. In our scheme an update operation modifies only one node per trie in all the cases while in both BVS and ABV schemes an worst case scenario for update may modify all the valid prefix nodes in the tries. The scheme has lower performance results than ABV and BVS for a small number of rules but can perform better when the number of rules increases. For example, in the case of a synthetic 2D database with $20K$ entries having injected 10% elements having a common subprefix the worst case lookup time does not exceed 720 memory accesses in the case of ABVI comparing with 1250 memory accesses in the case of BVS. Also, we note that we have not yet implemented incremental sorting in ABVI; this should make the numbers for ABVI much better than BV and more comparable to ABV.

While most of the paper used only one level of hierarchy, we also implemented a two level hierarchy for the large synthetically generated databases. The second level of hierarchy does improve the number of memory accesses for large classifiers, which suggests that the scaling of ABV is indeed logarithmic. It also suggests that ABV is potentially useful for the very large classifiers that may be necessary to support such applications as DiffServ and content-based Load Balancing that are already being deployed.

# References


[1] Cisco ArrowPoint Communications. In *http://www.arrowpoint.com*, 2000.

[2] IETF Differentiated Services (diffserv) Working Group. In *http://www.ietf.org/html.charters/diffserv-charter.html*, 2000.

[3] A. Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings of Infocom vol. 1*, pages 397–413, march 2000.

[4] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM Sigcomm'99*, september 1999.

[5] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings of Hot Interconnects VII, Stanford*, august 1999.

[6] V.Srinivasan S.Suri G.Varghese. Packet classification using tuple space search. In *Proceedings of ACM Sigcomm'99*, september 1999.

[7] Merit Inc. Ipma statistics. In *http://nic.merit.edu/ipma*, 2000.

[8] R. Morris E. Kohler J. Jannotti and M. F. Kaashoek. The click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, december 1999.

[9] M.Waldvogel G.Varghese J.Turner and B.Plattner. Scalable high speed ip routing lookups. In *Proceedings of ACM Sigcomm'97*, october 1997.

[10] T. V. Lakshman and D. Stidialis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM Sigcomm '98*, september 1998.

[11] Memory-memory. In *http://www.memorymemory.com*, 2000.

[12] C. Partridge. Locality and route caches. In *Proceedings of NSF Workshop, Internet Statistics Measurement and Analysis*, february 1999.

[13] D. Shah and P. Gupta. Fast updates on ternary-cams for packet lookups and classification. In *Proceedings of Hot Interconnects VIII, Stanford*, august 2000.

[14] J. Xu M. Singhal and J. Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. In *Proceedings of Infocom*, march 2000.

[15] V.Srinivasan G.Varghese S.Suri and M.Waldvogel. Fast scalable level four switching. In *Proceedings of ACM Sigcomm'98*, september 1998.

[16] M. M. Buddhikot S. Suri and M. Waldvogel. Space decomposition techniques for fast layer-4 switching. In *Proceedings of the Conference on Protocols for High Speed Networks*, august 1999.

# 10 Appendix

We try to illustrate our algorithm on an imaginary firewall database. Consider the firewall database in the figure 19. The database has 5 dimensions and 32 entries. Let's consider that $IP_1 \ldots IP_{25}$ are 32 bit IP addresses which are not having either $N_i$ or $M_i$, $1 \leq i \leq 3$ as prefixes. $N_i$ and $M_i$ are choosed such that $N_i$ is a prefix of $N_j$ and $M_i$ is also a prefix of $M_j$ for $i \leq j$. $N_i$ and $M_i$ do not share a common prefix.

Five tries are generated based on the database in figure 19, one trie is associated with each dimension. Consider the worst case scenario for the BVS algorithm in which a lookup needs to be done for a packet with the header $(M_{32}, N_{32}, 2500, 80, TCP)$. $M_{32}$ and $N_{32}$ have $M_2$ and $N_2$ respectively as subprefixes. A longest prefix match is done in each of the tries and five bit vectors are identified for each of the five dimensions of the database (figure 20).

The matching filter in the BVS algorithm is found by doing a bit by bit AND between the five bit vectors. The operation in this case requires the traversal of the whole bit vector in order to find the matching filter. Let's assume for simplicity that the size of an word is of 4 bits. In this case we need to read $5 \times \frac{32}{4} = 40$ words of memory.

Assume now the use of the ABVS algorithm. As we mentioned before two vectors are associated with each valid prefix node. The first one is the bit vector, similar with the one in the BVS algorithm. The second vector is the aggregate bit vector which is computed based on the information in the first one. Figure 20 shows the aggregated bit vector for this example. The size of the aggregation window is 4. The ABVS algorithm computes a bit by bit AND of the aggregated vectors and for each value of 1 in the result computes a bit by bit AND of the aggregated areas in the original vectors. In this case the result has values of 1 in all the positions which implies that it needs to read all the words from the original bit vectors. Therefore the total number of memory accesses is equal with $2 \times 5 + 5 \times \frac{32}{4} = 50$ which is greater than the number of memory accesses in the worst case of the BVS algrithm.

Let's consider now a rearrangement of the database in which we are trying to group together entries having wildcards on the same dimension. The result is displayed in the figure 21. The worst case scenario for the BVs algorithm for this new firewall database corresponds to a packet of the type $(AnyIP, AnyIP, AnyPort, 258, TCP)$. It takes a number of 40 memory accesses to serve a lookup request for such a packet. However, in the case of the ABVS algorithm, the aggregated bit vector for the *destination port* dimension is 00110001 while the one for the *sourceIP* dimension might be 00000111 which makes the total number of memory accesses to be made equal with $5 \times 2 + 5 \times 1 = 15$ words of memory. This is the minimum value for the worst case scenario in the ABVS algorithm.

Applying the rearranging scheme we introduced in section 5.2 the new firewall database look like the one in figure 22. For this case the worst case scenario has the same number of memory accesses like in the previous example.

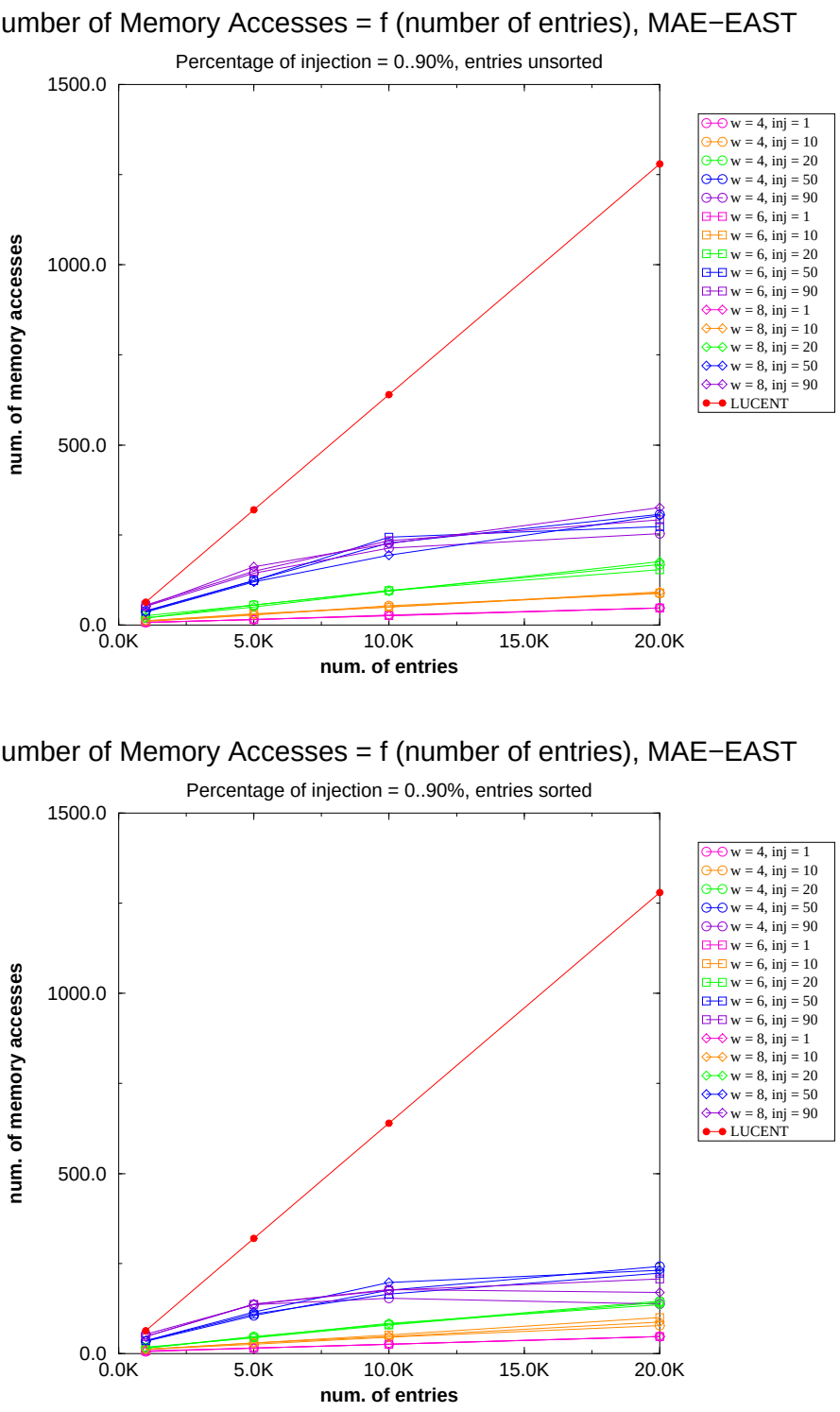


Figure 13: The number of memory accesses as a function of number of database entries. ABV scheme outperforms the BVS scheme with a factor of $2 \ldots 4$ if the database is not sorted and with a factor of $2 \ldots 7$ if the database is sorted. Synthetic database generated using MAE-EAST routing table and by randomly inserting group of elements which are sharing a common subprefix. $W$ is the depth of the subtrie created by these elements

| *Experiment* | *No. Of Entries = 5000* | | *No. Of Entries = 10000* | | *No. Of Entries = 20000* | |
|---|---|---|---|---|---|---|
| | *One Level* | *Two Levels* | *One Level* | *Two Levels* | *One Level* | *Two Levels* |
| *0% stars* | 16 | 14 | 26 | 14 | 46 | 18 |
| *1% stars* | 18 | 14 | 30 | 20 | 52 | 22 |
| *5% stars* | 20 | 14 | 30 | 18 | 52 | 26 |
| *10% stars* | 22 | 20 | 32 | 22 | 50 | 22 |
| *50% stars* | 20 | 18 | 30 | 18 | 50 | 20 |

Figure 14: Comparison between the ABV algorithm with one and two levels of aggregation. The filter database is *sorted* and it is generated using the MAE-EAST routing table.

| *Size* | *BVS* | *ABV - 32* |
|---|---|---|
| 3722 | 585 | 40 |
| 7799 | 1220 | 65 |
| 21226 | 3320 | 140 |

Figure 15: ABV vs. BVS scheme for a five dimension synthetic generated database. Synthetic database generated using MAE-EAST routing table and port number ranges and protocol numbers from the industrial firewall databases. We consider an aggregate size of 32.

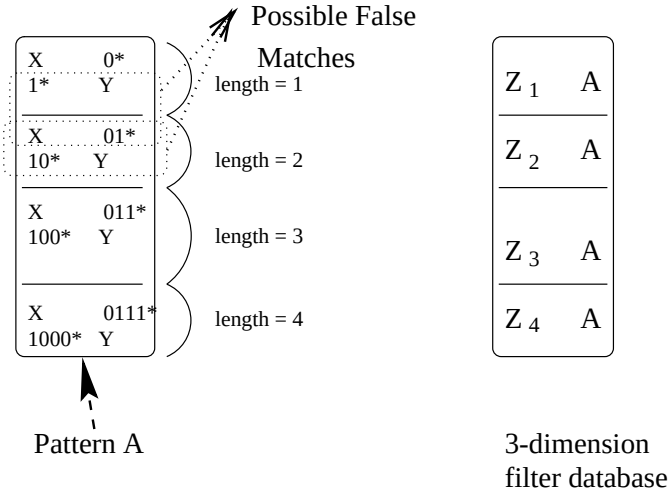


Figure 16: An example of a 2 and 3 dimension database with the number of different lengths of prefixes equal with 4 for which the number of false matchings is equal with 4 and 16 respectively.

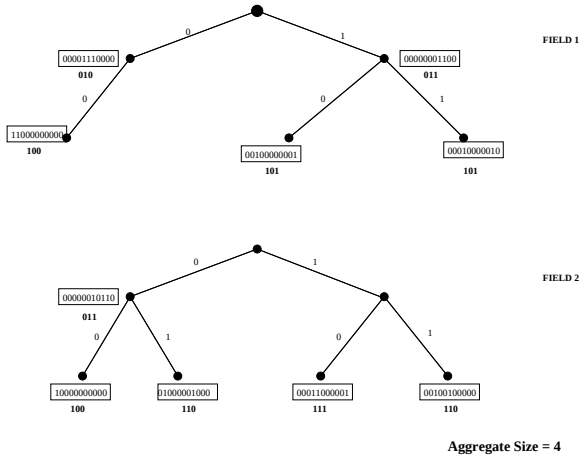


Figure 17: Two tries associated with each of the fields in the database of Figure 1 in the ABVI Algorithm. Compare the bitmaps with those of the ABV algorithm in Figure 2.

| *Filter* | *No. of Modified Mem. Loc. by Update* | | | *Lookup Time* | | |
|---|---|---|---|---|---|---|
| | *BV* | *ABV* | *ABVI* | *BV* | *ABV* | *ABVI* |
| $DB_1$ | 9776 | 384 | 10 | 260 | 120 | 260 |
| $DB_2$ | 5970 | 396 | 10 | 150 | 110 | 336 |
| $DB_3$ | 2159 | 254 | 10 | 85 | 60 | 154 |
| $DB_4$ | 2002 | 286 | 10 | 75 | 55 | 192 |

Figure 18: ABVI vs ABV vs BV: the total number of memory location that are modified by an update operation in the worst case and the worst case lookup time.

| *Source IP* | *Dset. IP* | *Source Port* | *Dest. Port* | *Protocol* |
|---|---|---|---|---|
| $*$ | $IP_1$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $IP_2$ | $N$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $IP_3$ | $*$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $M_2$ | $IP_4$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $IP_5$ | $N_1$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $*$ | $IP_6$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $*$ | $*$ | $*$ | 512 | $TCP$ |
| $M_1$ | $IP_7$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $IP_8$ | $*$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $M_2$ | $N_2$ | $*$ | 256 | $TCP$ |
| $*$ | $IP_9$ | $*$ | 256 | $TCP$ |
| $*$ | $*$ | $*$ | 257 | $TCP$ |
| $IP_{10}$ | $*$ | $1024 \ldots 65535$ | 80 | $TCP$ |
| $M_2$ | $N_2$ | $*$ | $*$ | $UDP$ |
| $IP_{11}$ | $IP_{12}$ | $*$ | $*$ | $UDP$ |
| $IP_{28}$ | $N_2$ | $*$ | $*$ | $UDP$ |
| $*$ | $*$ | $*$ | 258 | $TCP$ |
| $IP_{13}$ | $IP_{14}$ | $*$ | 80 | $TCP$ |
| $M_2$ | $N_1$ | $*$ | 259 | $TCP$ |
| $M_2$ | $N$ | $*$ | 260 | $TCP$ |
| $M_2$ | $*$ | $*$ | 261 | $TCP$ |
| $IP_{15}$ | $IP_{16}$ | $*$ | 261 | $TCP$ |
| $M_1$ | $N_1$ | $*$ | 262 | $TCP$ |
| $IP_{17}$ | $IP_{18}$ | $*$ | $*$ | $*$ |
| $M$ | $N$ | $*$ | 264 | $TCP$ |
| $IP_{19}$ | $IP_{20}$ | $*$ | $*$ | $*$ |
| $IP_{21}$ | $IP_{26}$ | $*$ | 264 | $TCP$ |
| $IP_{22}$ | $IP_{27}$ | $*$ | 264 | $TCP$ |
| $*$ | $IP_{23}$ | $*$ | 264 | $TCP$ |
| $*$ | $IP_{24}$ | $*$ | 264 | $TCP$ |
| $*$ | $IP_{25}$ | $*$ | 264 | $TCP$ |
| $M_2$ | $N_2$ | $*$ | 80 | $TCP$ |

Figure 19: A 32 rules, 5 dimensions firewall database

| *Dimension* | *Bit Vector* | *Aggregate* |
|---|---|---|
| *Source IP Pref.*$= M_2$ | 10010111011101001011101010001111 | 11111111 |
| *Dest. IP Pref.* $= N_2$ | 01101010110111011011101010110001 | 11111111 |
| *Source Port = 1024 /10* | 11111111111111111111111111111111 | 11111111 |
| *Dest. Port = 80* | 11111101100011110100000010100000 1 | 11111111 |
| *Protocol = TCP* | 11111111111110001111111111111111 | 11111111 |

Figure 20: A 32 rules, 5 dimensions firewall database

| *Source IP* | *Dset. IP* | *Source Port* | *Dest. Port* | *Protocol* |
|---|---|---|---|---|
| $IP_2$ | $N$ | 1024...65535 | 80 | $TCP$ |
| $M_2$ | $IP_4$ | 1024...65535 | 80 | $TCP$ |
| $IP_5$ | $N_1$ | 1024...65535 | 80 | $TCP$ |
| $M_1$ | $IP_7$ | 1024...65535 | 80 | $TCP$ |
| $IP_{13}$ | $IP_{14}$ | * | 80 | $TCP$ |
| $M_2$ | $N_1$ | * | 259 | $TCP$ |
| $M_2$ | $N$ | * | 260 | $TCP$ |
| $IP_{15}$ | $IP_{16}$ | * | 261 | $TCP$ |
| $M_1$ | $N_1$ | * | 262 | $TCP$ |
| $M$ | $N$ | * | 264 | $TCP$ |
| $IP_{21}$ | $IP_{26}$ | * | 264 | $TCP$ |
| $IP_{22}$ | $IP_{27}$ | * | 264 | $TCP$ |
| $M_2$ | $N_2$ | * | 80 | $TCP$ |
| $M_2$ | $N_2$ | * | 256 | $TCP$ |
| $M_2$ | $N_2$ | * | * | $UDP$ |
| $IP_{11}$ | $IP_{12}$ | * | * | $UDP$ |
| $IP_{28}$ | $N_2$ | * | * | $UDP$ |
| $IP_{17}$ | $IP_{18}$ | * | * | * |
| $IP_{19}$ | $IP_{20}$ | * | * | * |
| $IP_3$ | * | 1024...65535 | 80 | $TCP$ |
| $IP_8$ | * | 1024...65535 | 80 | $TCP$ |
| $IP_{10}$ | * | 1024...65535 | 80 | $TCP$ |
| $M_2$ | * | * | 261 | $TCP$ |
| * | $IP_1$ | 1024...65535 | 80 | $TCP$ |
| * | $IP_6$ | 1024...65535 | 80 | $TCP$ |
| * | $IP_9$ | * | 256 | $TCP$ |
| * | $IP_{23}$ | * | 264 | $TCP$ |
| * | $IP_{24}$ | * | 264 | $TCP$ |
| * | $IP_{25}$ | * | 264 | $TCP$ |
| * | * | * | 512 | $TCP$ |
| * | * | * | 257 | $TCP$ |
| * | * | * | 258 | $TCP$ |

Figure 21: A 32 rules, 5 dimensions firewall database after rearranging the entries by grouping the ones having wildcards as prefixes).

| *Source IP* | *Dset. IP* | *Source Port* | *Dest. Port* | *Protocol* |
|---|---|---|---|---|
| $IP_2$ | $N$ | 1024...65535 | 80 | $TCP$ |
| $IP_5$ | $N_1$ | 1024...65535 | 80 | $TCP$ |
| $IP_{13}$ | $IP_{14}$ | * | 80 | $TCP$ |
| $IP_{15}$ | $IP_{16}$ | * | 261 | $TCP$ |
| $IP_{21}$ | $IP_{26}$ | * | 264 | $TCP$ |
| $IP_{22}$ | $IP_{27}$ | * | 264 | $TCP$ |
| $IP_{11}$ | $IP_{12}$ | * | * | $UDP$ |
| $IP_{28}$ | $N_2$ | * | * | $UDP$ |
| $IP_{17}$ | $IP_{18}$ | * | * | * |
| $IP_{19}$ | $IP_{20}$ | * | * | * |
| $IP_3$ | * | 1024...65535 | 80 | $TCP$ |
| $IP_8$ | * | 1024...65535 | 80 | $TCP$ |
| $IP_{10}$ | * | 1024...65535 | 80 | $TCP$ |
| $M_2$ | $IP_4$ | 1024...65535 | 80 | $TCP$ |
| $M_2$ | $N_2$ | * | 80 | $TCP$ |
| $M_2$ | $N_2$ | * | 256 | $TCP$ |
| $M_2$ | $N_2$ | * | * | $UDP$ |
| $M_2$ | $N_1$ | * | 259 | $TCP$ |
| $M_2$ | $N$ | * | 260 | $TCP$ |
| $M_2$ | * | * | 261 | $TCP$ |
| $M_1$ | $IP_7$ | 1024...65535 | 80 | $TCP$ |
| $M_1$ | $N_1$ | * | 262 | $TCP$ |
| $M$ | $N$ | * | 264 | $TCP$ |
| * | $IP_1$ | 1024...65535 | 80 | $TCP$ |
| * | $IP_6$ | 1024...65535 | 80 | $TCP$ |
| * | $IP_9$ | * | 256 | $TCP$ |
| * | $IP_{23}$ | * | 264 | $TCP$ |
| * | $IP_{24}$ | * | 264 | $TCP$ |
| * | $IP_{25}$ | * | 264 | $TCP$ |
| * | * | * | 512 | $TCP$ |
| * | * | * | 257 | $TCP$ |
| * | * | * | 258 | $TCP$ |

Figure 22: A 32 rules, 5 dimensions firewall database after rearranging the entries by grouping together the entries having the same dimension for prefixes on each dimension and then sorting the elements for every dimension).