Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Computer Networks 42 (2003) 717–735

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Fast and scalable conflict detection for packet classifiers

F. Baboescu [*], G. Varghese

Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, USA

Received 9 August 2002; accepted 3 January 2003

Responsible Editor: E. Knightly

Abstract

Packet filters provide rules for classifying packets based on header fields. High speed packet classification has received much study. However, the twin problems of fast *updates* and fast *conflict detection* have not received much attention. A *conflict* occurs when two classifiers overlap, potentially creating ambiguity for packets that match both filters. For example, if Rule 1 specifies that all packets going to CNN be rate controlled and Rule 2 specifies that all packets coming from Walmart be given high priority, the rules conflict for traffic from Walmart to CNN. There has been prior work on efficient conflict detection for two-dimensional classifiers. However, the best known algorithm for conflict detection for general classifiers is the naive $O(N^2)$ algorithm of comparing each pair of rules for a conflict. In this paper, we describe an efficient and scalable conflict detection algorithm for the general case that is significantly faster. For example, for a database of 20 000 rules, our algorithm is 40 times faster than the naive implementation. Even without considering conflicts, our algorithm also provides a packet classifier with *fast updates* and *fast lookups* that can be used for stateful packet filtering.

Keywords: Packet classification; Filter conflicts; Classifiers; IP Lookups

1. Introduction

Beyond traditional 32-bit destination IP address lookups, many routers perform packet classification on other IP header fields for purposes such as packet filtering in firewalls, binding flows to MPLS labels for traffic engineering, or binding flows to DiffServ code points to provide QoS. To do so each router keeps a *rule database* which 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 *significant* header field. Three kind of matches are allowed for each packet processed by a router: *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 packet header field—this is useful for blocking access from a specified subnetwork. In a range match, the header values should lie in the range

[*] Corresponding author.
E-mail addresses: baboescu@cs.ucsd.edu (F. Baboescu), varghese@cs.ucsd.edu (G. Varghese).

specified by the rule—this is useful for specifying port number ranges. Ranges, however, can be converted into prefixes as shown in [1,2].

Each rule R_i has an associated action act^i, which specifies how to forward the packet matching this rule. The action may specify if the packet should be blocked or if it is to be forwarded, it specifies the outgoing link on which the packet is to be sent, and perhaps also a queue within that link if the corresponding flow has 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.

A problem may occur when a packet matches multiple filters with conflicting values for the action field. Let us consider the simple example in Table 1. The rules in the tables are associated with actions to guarantee bandwidth. The first rule R_0 assigns all packets that match the tuple $(0*, 10*)$ a bandwidth equal to 10 Mbps, while the second rule R_1 assigns all packets that match the tuple $(00*, 1*)$ a bandwidth equal to 100 Mbps. A conflict occurs in this case because it is unclear what bandwidth (i.e., 10 or 100 Mbps) should be allocated to packets which match the tuple $(00*, 10*)$. We call such a conflict an *overlapping* conflict because there are some packets that match R_0 and not R_1, some that match R_1 and not R_0, and some that match both.

A second type of conflict, a *subset* conflict, occurs between the rules R_2 and R_3. The fields in R_3 describe a strict subset of the fields in rule R_2. The position of the rules in the database in this case is used to decide which of them is to be applied when a packet matches both rules. Assuming the standard firewall rule where the lower the position number, the higher the priority, a packet with a header (1110, 1111) will only be assigned 10 Mbps according to R_2. The last two rules R_4 and R_5 do not have any conflict with any of the other rules in the database.

Table 1
A simple example with 6 rules on two fields

Rule	Field_1	Field_2	Action (Mbps)
R_0	0*	10*	10
R_1	00*	1*	100
R_2	11*	11*	10
R_3	111*	111*	100
R_4	101*	10*	100
R_5	*	01*	10

A seminal paper [3] introduced these two types of conflicts and showed that subset conflicts can be avoided by positioning but overlapping conflicts cannot, in general, be avoided by repositioning. Instead, [3] suggests introducing a new rule for each area that is shared by multiple overlapping rules, for example in the case of R_0 and R_1, the new rule $(00*, 11*)$. In our paper, we will not distinguish between these two types of conflicts but describe an algorithm to identify either all the conflicting pairs of rules in a database, or to identify all rules that conflict with a newly added rule. While some conflicts may be intentional, [3] reports many instances of irreconcilable conflicting actions that indicate erroneous action by managers. Thus flagging conflicts for managers or protocols that insert filters is an important problem.

We believe that conflict detection will become an important problem as router vendors offer larger classifier tables (up to 64K rules in some products) and the rules are used for potentially conflicting purposes such as QoS, security, and Customer Relationship Management (a form of QoS where certain flows are dynamically identified as being important "customers" and given better service). In many of these applications, some service (e.g., Intrusion Detection, stateful filtering, or CRM) may dynamically insert a new rule that can conflict with existing security or QoS policy. While the majority of added filters will not conflict [4], a mechanism to warn managers of potential conflicts seems necessary to avoid breaches of the security or QoS policies.

Clearly, in the examples above the time to add filters and detect conflicts is important, especially for large databases. Thus the ultimate goal is to achieve a scheme that allows both packet classification and rule updates at close to line speed. However, in practice even the most dynamic rule database is unlikely to add new rules more frequently than once every 10–100 packet arrival times. For example, for a stateful filtering application the number 10–100 could represent the number of packets in a conversation because a filter may have to be inserted (and checked for conflicts) when a new conversation starts.

This allows a larger time budget for conflict detection and insertion than for pure lookup (which must complete in a single packet arrival time [4,5]) but is still challenging. We assume that a rule update implies both checking for possible conflicts, and insertion or deletion in the database. Thus besides the goal of fast conflict detection our paper also addresses an important issue: fast rule insertion and deletion.

1.1. Filter conflict detection—problem statement

We give a formal statement of the conflict detection problem. Given a database of filters H containing N filters with k dimensions and a new filter F with fields $(F_1, \ldots, F_k)$ list all the filters P in H such that for all the fields P_i, $i = 1, \ldots, k$, P_i is either a prefix of F_i or an exact match, or F_i is a prefix of P_i.

There are two main factors that we consider in evaluating our implementation. These are the number of *memory accesses* required by an operation (the main limitation in modern computer architectures) and the *memory size* occupied by data structures (because it is important to fit into high speed memory).

2. Previous work

Packet filter classification has received broad attention [1,2,4–10]; from previous work, it appears that the general problem is inherently hard (in a worst-case sense) when the filters contain more than 2 fields. While Ternary CAMs [11] offer a good solution in hardware for small classifiers, they use too much power and do not scale well to large classifiers.

A practical solution for multi-dimensional packet classification problem is given in a paper which we refer to as the original bit vector scheme (BV) [6]. However, it is difficult to scale the scheme to large rule database. Ref. [12] addresses these limitation in the BV scheme and introduces two new ideas, recursive aggregation of bit maps and filter rearrangement, to create an aggregate bit vector scheme (ABV).

None of the papers above addresses the new problem of conflict detection. Moreover, most of these schemes heavily use precomputation to speed up filter search; this makes rule updates slow. The problem of filter classification schemes with fast updates has received only little attention [7,10,13]. Filter conflict detection has received attention only recently [3,14]. The seminal paper [3] describes a fast (linear in the length of each rule) algorithm for two-dimensional classifiers and some other special cases, and a slow $O(N^2)$, where N is the number of filters algorithm for general classifiers. Since real databases often use 5 or more fields, and do not fit the special cases (e.g., some of the special cases in [3] restrict prefix lengths to either 0 or 32), their fast algorithm cannot be used for such databases. A recent paper [14] provides a $O(N^{1.5})$ algorithm for the two-dimensional case only, but for a different priority-based definition of the notion of conflict.

The bottom line is that previous work describes no efficient conflict detection algorithm for general five-dimensional databases other than the naive one of comparing every pair of rules for conflicts in $O(N^2)$ *time.* For example, for 10 000 rules, assuming that each rules takes five (Destination IP address, Source IP address, Protocol, Dest and Source Port ranges and prefix length information) 32-bit words to store, the naive algorithm must access $10\,000 * 9999 * 5/2$ memory words, which is roughly 250 million memory accesses. Thus it is worth looking for faster algorithms, the subject of this paper.

2.1. Contributions and results

Our paper goes beyond the work in [6,12] by addressing two important new problems: fast packet filter conflict detection and fast rule updates. It also investigates further the effects of aggregation introduced by [12], and shows, perhaps surprisingly, that aggregation can also *reduce* the overall memory size. The algorithms we develop can be used for solving the general k-dimensional problem. We evaluate them on both real firewall databases and synthetically generated five-dimensional databases. Our results show an order of magnitude improvement (e.g., a factor of 40 improvement for a 20 000 rule database) over the naive O(N^2) algorithm as well as simplistic extensions of [6,12].

While our algorithm looks superficially similar to [6] (in the use of bitmaps) and to [12] (in the use of aggregation), we emphasize that both the problem we solve (*conflict detection* vs. *classification*) and our solution (we use a *subtree* semantic for computing bitmaps as opposed to a *path* semantic) are completely different from these previous papers.

3. Towards a new scheme for fast conflict detection

In this section we introduce our ideas for a fast conflict detection scheme. Given that the BV scheme is a fast and practical scheme for packet classification, we start by adapting it for conflict detection. The resulting simplistic scheme has a number of inefficiencies that will motivate our final scheme.

3.1. Simplistic conflict detection using the original bit vector scheme

The BV scheme is a form of divide-and-conquer which divides the packet classification problem into k subproblems, and then combines the results. It builds k one-dimensional tries associated with each field in the original filter database. We assume that ranges are converted to prefixes using techniques shown in [1,2].

For each trie $T_l, l = 1, \dots, k$ a N-bit vector is associated with each node M_l in the trie which corresponds to a valid prefix node. A bit i is set in the bit vector at node M if and only if there is a rule R_i in the database with a field R_i^l which is a prefix (or equal) with the path from root to M in the trie. The result of applying BV for the filter database in Table 2 is shown in Fig. 1. We consider

Table 2
A simple example of a two-dimensional database with 11 rules

Rule	Field$_1$	Field$_2$
R_0	000000∗	111001∗
R_1	1001∗	0000101∗
R_2	10110∗	111∗
R_3	1111∗	0000100∗
R_4	00000100∗	100010∗
R_5	10111∗	000000∗
R_6	10∗	1111∗
R_7	0001010∗	∗
R_8	000111∗	100011∗
R_9	000000∗	111∗
R_{10}	∗	∗

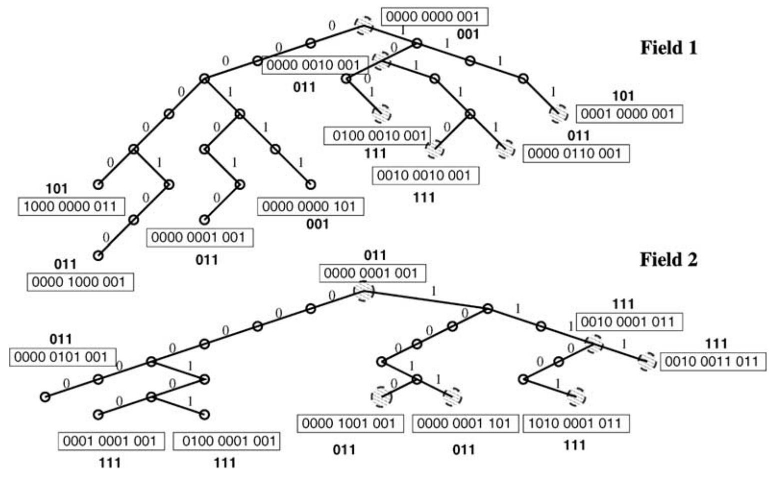


Fig. 1. Two tries associated with each of the fields in the database of Table 2, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with nodes that correspond to valid prefixes. The bits are set according with the semantics in the original bit vector scheme. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right. We mark the nodes that need to be checked for possible conflicts when a rule with the fields (1∗, 1∗) is inserted.

for now only the boxed bit vectors in the figure. For example, the bit vector associated with the rightmost leaf node in the second trie is 00100011011 (the left most bit is associated with R_0) because the prefix $1111*$ associated with this node is matched by both $*$, $111*$ and $1111*$ which corresponds to the values in rules 2, 6, 7, 9 and 10.

Let us assume we want to check if a rule with the tuple $(1*, 1*)$ conflicts with any of the rules in the original database. We traverse the tries until we reach the nodes which are the best match for the prefixes in the rule. In this example, we do not have an exact match on either of the fields. The longest matching prefix is $*$ for both tries. However, the bit vectors that label these nodes specify the rules which have fields that are either an exact match (only if it was an exact match) or prefixes of the one we are looking for. This is insufficient because we also want to consider fields that are *suffixes* of the ones we are looking for. Thus to identify all possible conflicts, we also need to check the descendants of the nodes where we found our best match.

More precisely, the basic algorithm for conflict detection using BV for a k-dimensional filter database is as follows. Trie T_i is associated with field i from the rule database. The trie is built on all possible prefixes that are found in the field i in any rule in the database. A node in trie T_i is associated with a valid prefix P if there is at least one rule in the database which has a value equal to P in field i. Each such node is appended with a bit vector with a size equal to the size of the database. A bit is set in position l in the bit vector if the lth rule in the database has in field i a value which is either a prefix or an exact match of P.

When a new rule $R(H_1, \ldots, H_k)$ needs to be checked for conflicts, a longest matching prefix node is identified in each of the tries for each field i in the rule. If such a node N_i exist in dimension i, its bit vector identifies the rules in the database which for the dimension i contain prefixes of H_i. We need to identify for each dimension i all rules which contain prefixes that are suffixes of H_i. To do so, we compute the union of the already obtained bit vectors *together with the bit vectors of all nodes contained in the subtrie rooted at* N_i. The set of rules that are possible conflicts are then identified by the intersection of all the bit vectors previously built for each trie T_i, $i = 1, \ldots, k$.

The pseudocode for this implementation is:

```
1 **DetectConflictBV** $(R(H_1, \ldots, H_k), T(T_1, \ldots, T_k))$
2   **for** $i \leftarrow 1$ **to** $k$ **do**
3     $N_i \leftarrow longestPrefixMatchNode(T[i], H_i)$;
4     $temp[i] \leftarrow N_i.bitVect$;
5     **for** each valid prefix node $M$ in the subtrie with the root in the node identified by the prefix $H_i$
6       $temp[i] \leftarrow temp[i] \cup M.bitVect$;
7   **return** $\bigcap_{i=1}^{k} temp[i].bitVect$;
```

Unfortunately, this simplistic algorithm may involve a large number of nodes from the subtries in each dimension. A first optimization is to consider only leaf nodes in the subtrie because *for BV the union of all the bit vectors from a subtrie is equal to the union of the bit vectors in the leaves*. Thus line 5 in the pseudocode can be changed using this observation above. But we can do better. We address the limitations of this scheme by focusing on two separate areas:

- how to decrease the complexity of operations on large bit vectors;
- how to reduce the number of bit vectors to be examined by reducing the number of nodes in the trie which need to be checked.

3.2. Conflict detection using aggregated bit vector scheme

If the bit vectors are sparse (i.e., very few set bits), the BV algorithm has to read all bits, which is a waste. Aggregated bit vector scheme (ABV) [12] addresses this limitation by allocating two bit vectors to each valid prefix node. The first bit vector has N bits for the BV bit vector. The second bit vector is computed from the first one by using aggregation. Using an aggregate size of A, a bit k in this vector is set if and only if there is at least one rule R_n, $A \times k \leqslant n \leqslant A \times k + 1 - 1$ for which P is a prefix of R_n^i. The aggregate bit vector has $\lceil N/A \rceil$ bits. Fig. 1 shows the application of the aggregation for the example database in Table 2 using an aggregate size $A = 4$. The main idea is that the aggregate bit vector provides a compact signature to eliminate redundant reads to words that have no bits set.

With minor modifications, the aggregation scheme can be directly used in the conflict detection algorithm. The *union* and *intersection* operations can be made to avoid redundant reads by considering only words corresponding to bits which are set in the aggregate. Even with aggregation, the algorithm is rather slow because of the need to compute the union of all the leaves in each subtrie defined by the header fields. This sets the stage for our main new idea.

4. A fast conflict detection bit vector algorithm

In this section we describe our new algorithm for fast conflict detection. We start by showing a new semantic for computing bit vectors through which we avoid excessive subtrie traversals to detect conflicts.

4.1. A new semantic for the bit vectors

Consider again the general k-dimensional problem in which k tries are computed. Each valid prefix node in the trie has an associated bit vector. For simplicity we start by not considering aggregation. We later discuss an extension using aggregation.

In each of the tries T_i, $i = 1, \ldots, k$, each node associated with a valid prefix contains two bit vectors. A first bit vector (*bitVect*1) has a bit l set if and only if there is a rule R_l whose field i provides an *exact* match with the node prefix. The second bit vector (*bitVect*2) modifies the semantics of the bit vector in the original bit vector scheme [6] to satisfy the following invariant: for all tries T_i, $i = 1, \ldots, k$, in each valid prefix node N associated with a prefix P, the bit vector $N.bitVect2 = (\bigcup C.bitVect2) \cup N.bitVect1$, where nodes C are all the immediate descendants of N that are also valid prefix nodes. In other words, we only need to explore the subtrie rooted at N till we reach a valid prefix node on each path.

Intuitively, the original BV scheme computes a bit map at node N corresponding to all valid prefix nodes in the path from the root to node N. *Our first bit vector, by contrast, computes a bit vector that only corresponds to exact (and not prefix) matches. Our second bit vector turns the BV bit vector semantics upside down and computes the bitmap at node N corresponding to the union of the bitvectors associated with all valid prefix nodes in the subtrie rooted at node N*. The reader may object at this point "That's just a different form of precomputation". However, we need to show (as we do below) that this new bit semantic can be updated efficiently (fast updates) and can be used to do packet classification (as in the BV scheme).

We consider the filter database in Table 2 to exemplify our new semantic. The appended bit vectors with the new semantics are displayed in Fig. 3 while the bit vectors corresponding to rules with prefixes which are an exact match are shown in Fig. 2. Assume a rule $(1*, 1*)$ which needs to be checked for conflicts with the other rules in the database. For each of the tries in the Fig. 2 we compute the union of all the bit vectors from valid prefix nodes which are prefix or exact match for $1*$ in the first trie and $1*$ in the second trie *(step 1)*. The result is: 00000000001 and 00000001001. Intersection of these bit vectors gives the set of rules which have fields that are either prefixes of the fields in the rule we check or are an exact match. In this example the result is 00000000001, showing that there is one rule R_{10} in the database matching this criteria.

In Fig. 3, for each trie we compute the union of the bitmaps associated with nodes which are valid prefixes and immediate children of the nodes associated with the prefix $1*$ *(step 2)*. These nodes are: $10*$, $1111*$ in the first trie and $100010*$, $100011*$ and $111*$ in the second trie. Please notice that the nodes $1000*$, $10110*$, $10111*$ in the first trie and $111001*$ and $1111*$ in the second trie are not considered. The results of the union operation are 01110110000 and 10101010010 respectively. The values obtained in the previous two steps are combined once again (union) in each dimension (trie) *(step 3)*. The intermediate values are: 01110110001 and 10101011011 respectively. In the end the values are intersected *(step 4)* and the final

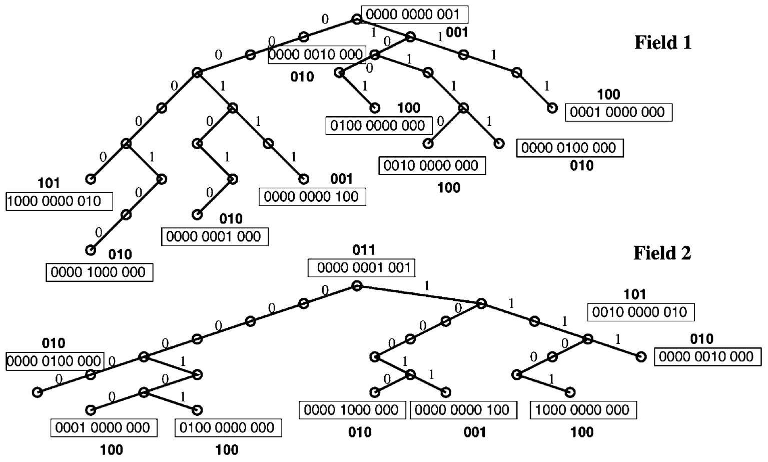


Fig. 2. Two tries associated with each of the fields in the database of Table 2, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with nodes that correspond to valid prefixes. The bits in a node are set only if there is an exact match between the filter prefix and the node. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right.

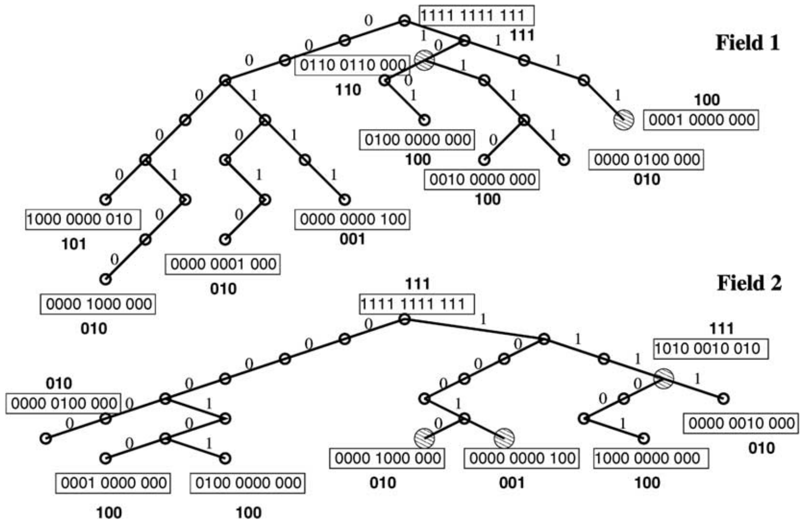


Fig. 3. Two tries associated with each of the fields in the database of Table 2, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with nodes that correspond to valid prefixes. The bits are set according to the new semantic we introduce. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right.We mark the nodes that need to be checked for possible conflicts when a rule with the fields $(1*, 1*)$ is inserted.

result is 00100010001. The result shows that there are three rules (R_2, R_6, R_{10}) which may generate a conflict.

If we intersected only the results from *step 2*, we would have obtained the set of rules which in all fields have values which have $1*$ as prefix. However, this misses rule R_{10} which may also generate a conflict. Therefore, the set of all the rules which may generate conflicts is given by first doing the union of the bit vectors in the Fig. 2 as in the first case above followed by an union with the bit vectors in the Fig. 3. The k bit vectors are then intersected; a value of 1 in the result identifies rules with a possible conflict.

4.2. An improved conflict filter detection algorithm

Given a rule $R(H_1, \ldots, H_k)$ we want to identify all the possible conflicts it might have with other rules in the database. A trie T_i is built for each dimension i. Each valid prefix node in the trie is appended with two bit vectors. A bit l is set in the first bit vector if and only if the lth rule in the database has its field i value be an exact match with the node prefix (*bitVect*1). The second bit vector (*bitVect*2) has the bits set according with the semantic described in the previous section.

For example, consider the two-dimensional database in the Table 2. We want to check for conflicts a rule R with the fields$(1*, 1*)$. In Fig. 1 we see that using the scheme in which the bit vectors are computed as in the BV scheme there are a total of 10 bit vectors which need to be read from memory, or in an optimized version in which only the leaf nodes are read, a total of 8 bit vectors. If the bit vectors are computed using the new semantic, then $2 + 5 = 7$ bit vectors need to be read from memory. The first value is given by the number of *bitVect*1 to be read (lines 4 and 5 in the pseudocode), while the second value is given by the number of *bitVect*2 to be read (lines 7–9 in the pseudocode).

The pseudocode for the algorithm is:

```
1  NewDetectConflict ($R(H_1, \ldots, H_k), T(T_1, \ldots, T_k)$)
2    for $i \leftarrow 1$ to $k$ do
3      $temp[i] \leftarrow 00\ldots0$;
4      for each valid prefix node $M$ from root until an exact match of $H_i$
5        $temp[i] \leftarrow M.bitVect1 \cup temp[i]$;
6      if $H_i$ matches a valid prefix node $M$ then
7        $temp[i] \leftarrow temp[i] \cup M.bitVect2$;
8          continue [1]
9      $temp[i] \leftarrow \bigcup L.bitVect2$,
         where $L$ designates all the first level children of the node matching $H_i$ that represent valid prefixes
10   return $\bigcap_{i=1}^{k} temp[i]$;
```

[1] It continues execution with the next iteration for i (line 2).

As in the previous algorithm there are a number of bit vector operations which can be more efficiently executed using aggregation. If the algorithm above uses aggregation the pseudocode remains unmodified. However the semantics of both the union and intersection operation must be modified appropriately. Also, both *bitVect*1 and *bitVect*2 now represent a new data structure containing both the original N-bit bit vector and the aggregated one. We call IBV the improved version of the original BV algorithm using the new semantic and we call AIBV the modified version of the IBV using aggregation.

4.3. A fast conflict detection algorithm

The algorithm described so far has the potential to reduce the number of bit vectors to be read compared to a naive use of the BV algorithm. However, it has a limitation. Let us consider again the problem to be solved. Given a k-dimensional database of rules and a rule $R(H_1, \ldots, H_k)$ we need to identify possible conflicts between R and the other rules in the database. In each of the dimensions i, $i = 1, \ldots, k$ if the prefix H_i does not match a valid prefix node in the trie than the step 9 in the algorithm must be executed. In this way all the valid prefix nodes which are first level children of the node matching H_i must be traversed and the bit vectors *bitVect*2 must be read.

Thus in the worst case scenario all these children may be leaves in the trie which makes the algorithm to have performance similar to the BV algorithm. However, we are willing to trade some memory space in order to reduce the search time for possible conflicts. We would like to find a way in which for each rule R, in each trie there is at most one node with enough information regarding possible conflicts.

Consider the same filter database example in the Table 2. Two tries are computed for each of the dimensions. However, all the one-way branches are compressed this time. The resulting compressed tries are displayed in Fig. 4. The valid prefix nodes in the compressed tries carry the same bit vectors (*bitVect*1, *bitVect*2) as in the algorithm before. *However, the main difference is that we also insert the bit vector bitVect2 in all nodes, even if a node does not correspond to a valid prefix*. It is easy to see that by doing so we can at most double the amount of memory because every node other than a leaf has two children in a compressed trie.

However, now the step 9 in the previous algorithm is replaced by reading *bitVect*2 from either a node with a prefix that matches the prefix of the rule to be checked, or from the first node down the path if there is no node with a valid match. All the other steps in the algorithm remain unchanged. By doing so in each search for a conflict of the rule R in each dimension i the algorithm needs to read *bitVect*1 from all the valid prefix nodes that are traversed until the longest prefix match plus the *bitVect*2 from the node that has H_i as a prefix and has the smallest height.

For example, suppose that we want to check a rule $(1*, 1*)$ for possible conflicts with the other rules in our example in Table 2. In this case the total number of bit vectors which are read is: $2 + 2 = 4$ (Fig. 4). The first value is given by the number of *bitVect*1 values to be read (line 4, 5 in the pseudocode), while the second value is given by the number of *bitVect*2 values to be read (line 7 in the pseudocode). Thus we observe that the number of *bitVect*2 values read is k, where k is the number of dimensions. The pseudocode for the algorithm is given below. The tries T_i, $i = 1, \ldots, k$ are all assumed to be compressed.

```
1 FastDetectConflict ($R(H_1, \ldots, H_k), T(T_1, \ldots, T_k)$)
2 for $i \leftarrow 1$ to $k$ do
3       $temp[i] \leftarrow 00\ldots0$;
4       for each valid prefix node $M$ from root
        until an exact match of $H_i$
5         $temp[i] \leftarrow M.bitVect1 \cup temp[i]$;
6       $L \leftarrow$ the smallest height node having $H_i$
        as prefix
7       $temp[i] \leftarrow temp[i] \cup L.bitVect2$;
8 return $\bigcap_{i=1}^{k} temp[i]$;
```

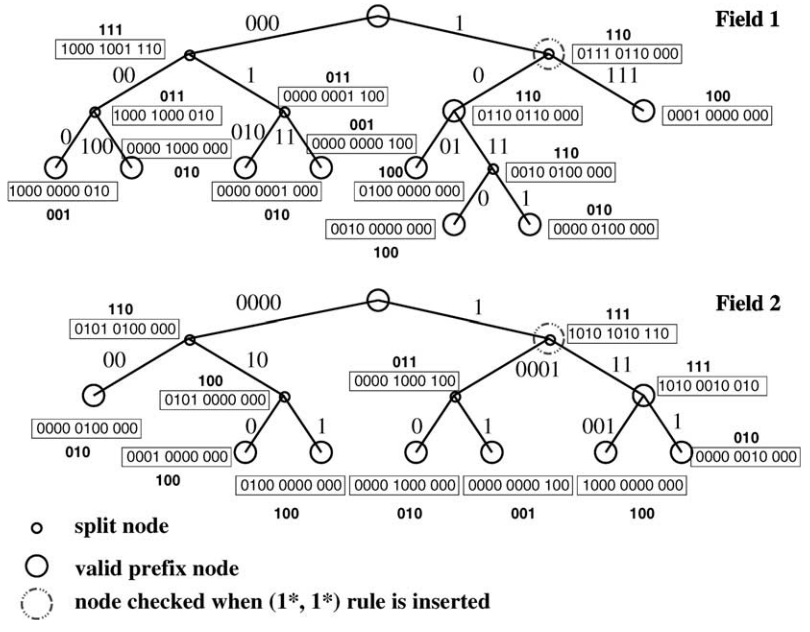


Fig. 4. Two compressed tries associated with each of the fields in the database of Table 2, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with the nodes. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right.

As in the previous algorithms there are a number of bit vector operations which can be made more efficient using aggregation. If the algorithm above uses aggregation, the pseudocode remains unmodified but the semantics of both the union and intersection operation as well as the data structure used for representing *bitVect*1 and *bitVect*2 need to be changed. We call SBV our algorithm for fast conflict detection and ASBV the modified version of the algorithm using aggregation.

5. Evaluation

In this section we evaluate our conflict detection algorithm vs. the naive algorithm and simplistic extensions of previous bit vector schemes on both real and synthetically created databases. The synthetically created databases are necessary to show the scalability of our algorithm; the real databases we were able to obtain are relatively small.

5.1. Theoretical evaluation of SBV

The metric used to evaluate our algorithm is the number of memory words which are read to determine whether a new rule has a conflict with the existing rules. Conflict detection using algorithms based on the original bit vector semantics have their worst case when a rule containing wildcards in all the fields is checked for conflict. All the valid prefix nodes in all tries (all the leaves in the optimized version) are involved in the computation. *All the bit vectors from these nodes need to be read in order to establish the final answer*.

By contrast, we have the following theorems for the worst case behavior of our new SBV algorithm.

Lemma 1. *Given any database of rules D, and any rule R, a check for conflicts between R and the rules in D using the SBV algorithm requires the read of at most one bitVect2 vector for each dimension of the database.*

Proof. Follows immediately from algorithm description above. □

For a given database, let V denote the maximum number of valid prefix nodes found on a path from root to leaf on the tries built on any of the k dimensions. Then we have:

Lemma 2. *Given any database of rules D, and any rule R, a check for conflicts between R and the rules in D using the SBV algorithm requires at most V bitVect1 vectors to be read for each dimension of the database.*

Proof. Consider again a rule $R(H_i, \ldots, H_k)$ which needs to be checked against possible conflicts with the N rules in the database D. Then, for each dimension i, $i = 1, \ldots, k$ a trie traversal is done based on the prefix H_i and we call X the node in the trie which is the longest matching prefix of H_i. The number of *bitVect*1 vectors which need to be read is equal to the number of valid prefix nodes which are on the path from root to X. □

Studies of both prefix databases and firewall databases [12] show that $V \leqslant 4$ in practice. If we call such databases *common*, we have:

Corollary 3. *Given any common database of rules D, and any rule R, a check for conflicts between R and the rules in D using SBV algorithm requires at most $5 * k * N/W$ memory words from bit vectors to be read, where k is the number of dimensions of the database, N is the number of rules in the database and W is the size of a word, which in a hardware implementation may have values as large as* $500, \ldots, 1000$.

A second corollary is:

Corollary 4. *The conflict detection scheme using the SBV algorithm is at most 5 times slower than the lookup scheme using the original bit vector scheme for common databases.*

Even this extra factor of 5 is a considerable overestimate of the slowdown required for fast insertion for the following reason. The effect of aggregation on the complexity of bit vector operation was investigated in [12]. Since 4 of the 5 bit vectors read in every dimension are the sparse *bitVect*1 values (which are set only for an exact match), these vectors will benefit much more from aggregation than the less sparse *bitVect*2 values. Thus we should achieve great gains from aggregation, and we provide experimental evidence in the next section.

5.2. Experimental evaluation method

We measure speed in terms of memory accesses, the amount of memory used, and the effects of aggregation.

We use two different types of databases. First, we use 4 firewall databases from existing commercial organizations (Table 3). They are five-dimensional databases in which each tuple contains (IP source prefix, IP destination prefix, source port range, destination port range, protocol). We convert the destination and source port ranges to a prefix format using technique shown in [1,2]. The salient features of these databases are that most prefixes have lengths 0 or 32, no prefix contains more than 4 matching subprefixes, the destination and source prefix fields in around half the rules were wildcarded, and roughly half the rules had $\geqslant 1024$ in the port number fields.

The second type of databases we used were randomly generated 5 field (i.e., five-dimensional) databases that are generated as follows.

5.2.1. Synthetic database generation characteristics

In the absence of large public classifiers we used the methodology of [12] to generate random databases that take into account characteristics of the small industrial databases we had, as well as other factors that help stress our algorithm.

Table 3
The sizes of the commercial firewall databases we use in the experiments

Filter	Number of rules specified by	
	Range	Prefix
DB_1	266	1640
DB_2	279	949
DB_3	183	531
DB_4	158	418

The easiest mechanism for generating a synthetic database is to randomly pick IP source and destination prefixes from the core routing tables. The port range fields can also be randomly generated using random numbers between 0 and 65535. The protocol field can be generated either by randomly generating a number between 0 and 255 or by considering only the protocol value numbers for UDP, TCP, ICMP together with a general value OTHER.

However, such a mechanism generates an unrealistic rule database. For example, considering a routing table with 80 000 entries from which we generate IP prefixes, we may not be able to insert even a single prefix of length zero (wildcard) because core routing tables have no default routes. But we have already seen that our commercial databases have a very high percentage of wildcards. Therefore, in addition to randomly inserting prefixes from routing tables our mechanism also randomly inserts zero length prefixes in these fields based on a specified tuning parameter.

A second technique we use is to insert (based on a specified tuning parameter) a set of IP prefixes which share a common subprefix (eg., the sequence $*, 1*, 11*, 110*$). These elements are very rare in a real filter database, however they are crucial for increasing what we call *false matches* [12] and thus increasing the stress on algorithms using aggregation.

5.3. Performance evaluation on commercial firewall databases

We experimentally evaluate our new algorithm SBV with and without aggregation on the four commercial firewall databases described in the beginning of this section. Our algorithm trades memory size for speed by associating two different bit vectors with every single prefix node in the tries. Therefore one would think it should use about three times more the memory space used by the original bit vector scheme. However, this is not true when one includes aggregation, as we see below.

We start by experimentally investigating the impact of the data structures we use on the total memory required. The rules in the databases are converted into a prefix format using techniques described in [1,2]. The memory space occupied by the nodes in the tries is identical for both IBV and BV with or without aggregation. However, our final algorithm SBV uses path compression, and therefore the number of nodes in the trie is reduced.

On the other hand, the memory space occupied by a node in a compressed trie is higher than in a regular trie. We consider a node in the BV algorithm with a regular trie to use 3 memory words (pointers to two children, plus pointer to a bit vector) while a node in the SBV algorithm, using a compressed trie to use 6 memory words.

The results in Table 4 confirm one expected observation: the memory size occupied by the bit vectors in the BV scheme is half the size occupied by those in IBV and about a third the size occupied by those in SBV (recall that SBV also stores bit vectors in nodes which are not associated with valid prefixes). However, the results show several other interesting features:

1. Aggregation considerably reduces the size of the memory occupied by the bit vectors (column 4 vs. 5, column 6 vs. 7 and column 8 vs. 9). There is no reason to store a word which has no bit set in the aggregate vector. The aggregate contains enough information to identify the words containing bits which are set and the position of these words.
2. Aggregation has a larger impact on the memory size occupied by the bit vectors in either the IBV or SBV. This is because a large number of bit vectors are of type *bitVect*1 which corresponds to exact matches, with a very large number of 0s which can be substantially compressed using aggregation.
3. IBV with aggregation uses a smaller amount of memory than the original bit vector scheme(BV) with aggregation while SBV with aggregation uses a slightly larger amount of memory because of additional bit vectors that are inserted.

The performance results of SBV, IBV and BV together with the naive O(N^2) implementation of conflict search are shown in Table 5. The number represents the total number of memory accesses to check the entire database for conflicts. For naive

Table 4
SBV vs. IBV vs. BV, with and without aggregation: the total number of memory location that are occupied by the algorithm's data structures

Filter	Nodes	Nodes-comp.	BV	ABV	IBV	AIBV	SBV	ASBV
DB_1	2970	2004	9880	4327	19 760	3028	27 248	4884
DB_2	3732	2190	6030	2502	12 060	1987	16 980	3543
DB_3	2493	1320	2108	1337	4216	1141	5848	1887
DB_4	2030	1530	2030	1304	4060	1029	5600	1708

The first two columns represents the total number of memory locations occupied by the nodes in the trie with and without trie compressions, while the next six columns represent the total number of memory locations occupied by the bit vectors (and aggregates) for all three algorithms.

Table 5
SBV vs. IBV vs. BV, with and without aggregation: the number of memory accesses in *kwords* for conflict check and update on a firewall database with rules from four commercial firewall databases

Filter	Naive	BV	ABV	IBV	AIBV	SBV	ASBV
DB_1	6761	4063.6	2709.5	1442.7	365.2	480.9	158.8
DB_2	2249.1	1376.9	1028.1	937.66	271.4	211.1	101
DB_3	682.5	576.9	482.7	429.9	130.2	77.4	44.7
DB_4	435.8	490.6	431.1	304.7	84.4	52.7	32.9

The number of rules are displayed in the second column while the other columns show the total number of memory accesses for the native O(N^2) algorithm, the BV, IBV, and SBV algorithms with and without aggregation. A prefix of A denotes aggregation.

search, we assume the cost of the entire search for the database is the pairwise cost of examining the memory words in every pair of rules which is: $((N * (N - 1))/2) * S$ where S is the size of a rule in words. In our case, 5 is a reasonable number for S for IP 5-tuples, though compression of wildcarded prefixes could reduce this by a factor of around two.

We consider a buildup with rules from the four commercial firewall databases. We add each of the rules in these databases in the order they were in the original database. A conflict check is executed before each rule is inserted. The results in Table 5 which shows the total number of memory words which are accessed during the entire operation. Several conclusions can be drawn:

1. Despite having similar worst case scenario as the original bit vector scheme based algorithm, on average IBV runs faster than BV.
2. Aggregation applied to BV contributes to a reduction in the average conflict search time by a factor of 1.87–2.14.
3. Our SBV conflict search runs on average about 16–28 times faster than the naive $O(N^2)$ algorithm that is the previous best in the literature. An additional 1.5–3 times improvement can be obtained by using SBV with aggregation.
4. SBV conflict search runs on average about 6.5–9.4 times faster than BV. Thus the new bit semantic is clearly very helpful, but the use of aggregation appears also to be essential, buying an extra factor beyond just the use of the new semantic.

5.4. Performance evaluation on five-dimensional synthetic databases

We expect that the gain of our algorithm should increase with the database size, at least when compared to the naive algorithm. To go beyond the small size of the commercial databases we have access to, we now describe tests with larger synthetic databases.

Unfortunately, database size is not the only parameter since we also have other tuning parameters such as the percentage of zero length prefixes, and the number of subprefixes. We note that these parameters stress our algorithm: for example, if no two prefixes are subprefixes of each other, our algorithm will perform extremely well.

As we described earlier in the paper we create our synthetic five-dimensional databases generating the IP prefixes by randomly selecting prefixes existent in routing tables available for public at [15]. The port numbers and protocol numbers are generated through a random selection of these fields from the commercial databases we have. In this paper we display results only for the synthetic databases generated using a view of the MAE-EAST routing table from September 12, 2000. The results for databases with IP prefixes generated for the other four routing tables in [15] are similar and are not reproduced here.

Each database that is created is characterized by the number of rules as well as the percentage of wildcards or *special subprefixes* that are injected. Next we generate a number of rules proportional to the number of rules in the filter database. These rules have the same characteristics as the database which is examined. For each rule we compute the number of memory accesses it takes to identify possible conflicts with the rules already existing in the database. In Tables 6 and 7 we report the average of these values. SBV outperforms BV because it reduces the number of bit vectors which need to be investigated during a conflict detection. We show how this number changes from BV to SBV in Tables 6 and 7.

The following observation limits the maximum performance that may be achieved by a conflict detection algorithm in our definition.

Observation 1. Given a K-dimensional filter database, there is no conflict detection algorithm that can run faster than the fastest packet classification algorithm. (If this were not so, one could use the conflict detection algorithm for lookup.)

As a result, in our measurements, instead of comparing our algorithm with the naive implementation, we compare the performance of SBV with both the original BV-based conflict detection as well as the complexity of packet classification using BV. Note that we are comparing our algorithm for the harder problem of conflict detection

Table 6
The complexity of an update with conflict checking using the original bit vector semantic (BV) and our new semantic (SBV), with and without aggregation, and varying percentages of zero length prefix injection

NR	% of wc	BV lookup	BV			SBV		
			Regular	Aggregate	Nodes	Regular	Aggregate	Nodes
250	0	40	358	347	11	170	158	4
250	1	40	360	349	10	189	172	6
250	10	40	1093	1011	27	169	156	4
250	20	40	1557	1423	34	150	135	4
250	50	40	2209	2154	47	129	121	4
2500	0	395	2352	2100	23	520	378	4
2500	1	395	4866	2886	46	592	369	5
2500	10	395	15 439	6722	141	547	342	5
2500	20	395	25 319	12 606	233	631	390	7
2500	50	395	37 141	22 750	335	575	373	6
10 000	0	1580	10 145	8901	30	1604	1036	4
10 000	1	1580	26 604	10 972	79	2236	1048	6
10 000	10	1580	162 234	41 055	482	2104	1065	6
10 000	20	1580	258 724	79 826	768	2077	1118	6
10 000	50	1580	354 706	179 365	1048	1876	1096	6
20 000	0	3160	19 713	17 695	30	3058	1891	4
20 000	1	3160	86 767	22 398	134	3657	1860	5
20 000	10	3160	519 727	106 478	804	4152	1969	6
20 000	20	3160	872 227	235 500	1348	4014	2060	6
20 000	50	3160	1 384 532	661 270	2137	3706	2099	6

The first column in the table represents the number of rules using port number ranges, while the second column represents the number of wildcards (wc) injected. The third column represents the number of memory words required by a lookup in the filter database using the original BV algorithm to read the one bit vector in each of the five dimensions. The last two groups of columns are associated with the two algorithms BV and SBV respectively. The columns in the group are associated with the number of memory words required by an update with conflict checking with and without using aggregation as well as the total number of bit vectors which are investigated during the operation. A word is made up of 32 bits. The aggregate size is also equal to 32.

with one of the best algorithms (BV) for the easier problem of packet classification.

Effect of zero-length prefixes: We first consider the effect of zero-length prefixes (wildcards) on both schemes with and without aggregation. We investigate both the memory size occupied by the bit vectors as well as the average time it takes for a conflict search. In the SBV scheme with aggregation the overall memory size gets reduced by the insertion of wildcards. This is because when more rules with zero-length prefixes are inserted they only modify the bit vectors associated with the root of the tries. The bit vectors associated with other trie nodes remain very sparse allowing a large compression coefficient to be achieved by using aggregation. The results are displayed in Table 8.

This behavior sharply contrasts with the behavior of the original BV scheme in which a value of 10% or higher of wildcards injected results in all the bits of all of the aggregates being set. This is why the memory size by using BV with aggregation reaches a ceiling equal to $(1 + 1/A) * L$, where A is the size of the aggregate and L is the total memory size occupied by the bit vectors in the BV.

We next investigate the effect of zero-length prefixes on the average conflict search time. We show the results in Table 6. For example, checking conflicts using SBV in a database with about 20 000 rules with 20% wildcards injected [2] is about 217 times faster than the original BV algorithm. It is also about 430 times faster if it is used together with aggregation. This is because there are only, in average, 6 bit vectors which need to be investigated

[2] We have noticed that it is very common for a filter database to contain about 20% wildcards.

Table 7
The complexity of an update with conflict checking using the original bit vector semantic (BV) and our new semantic (SBV), with and without aggregation, and with varying percentages of prefixes which share a common subprefix

NR	% of prefixes	BV lookup	BV			SBV		
			Regular	Aggregate	Nodes	Regular	Aggregate	Nodes
250	1	40	329	322	11	176	169	4
250	10	40	332	327	11	156	157	4
250	20	40	295	283	10	145	135	4
250	50	40	319	303	10	166	157	4
2500	1	395	2397	2151	24	527	380	4
2500	10	395	2288	1995	23	496	359	4
2500	20	395	2834	2556	29	565	388	5
2500	50	395	2256	1971	22	684	410	6
10 000	1	1580	9940	8712	29	1526	999	4
10 000	10	1580	10 016	8877	30	1782	1030	5
10 000	20	1580	10 239	8949	30	1919	1014	5
10 000	50	1580	10 322	8924	31	2706	1054	5
20 000	1	3160	19 713	17 695	30	3053	1809	5
20 000	10	3160	20 089	17 657	31	3404	1796	5
20 000	20	3160	20 596	18 062	31	4008	1858	7
20 000	50	3160	20 562	17 776	31	5299	1941	8

The first column in the table represents the number of rules using port number ranges, while the second column represents the number of prefixes sharing a common subprefix injected. The third column represents the number of memory words required by a lookup in the filter database using the original BV algorithm to read the one bit vector in each of the five dimensions. The last two groups of columns are associated with the two algorithms BV and SBV respectively. The columns in the group are associated with the number of memory words required by an update with conflict checking with and without using aggregation as well as the total number of bit vectors which are investigated during the operation. A word is made up of 32 bits. The aggregate size is also equal to 32.

in SBV while in BV this number is 432. Much more, despite increasing the number of rules in the database, the average number of bit vectors which need to be checked has not been greater than 7.

Effect of 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. We randomly generate elements from 50 different groups. (This methodology is justified more carefully in [12].) The IP prefixes in the synthetic database are created either by randomly picking elements from these groups or from the public routing tables from [15]. The port number ranges and protocol numbers are also generated by randomly picking values from the commercial firewall databases.

These elements may contribute to an increase in the number of memory accesses required by algorithms using aggregation through what we call *false matchings* as well as, in the case of SBV, through an increase in the number of bit vectors that may need to be examined.

Table 7 shows that these prefixes do not have a large impact in the overall performance of the algorithms. Also the memory size used by SBV does not increase significantly when the injection rate increases up to 20% (Table 9). This is also true when aggregation is used.

6. Fast rule insertion and deletion

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

Table 8
The effect of aggregation on the total size of the memory occupied by the appended bit vectors in both the original bit vector scheme and our scheme (SBV)

NR	% of wc	BV			SBV		
		Regular	Aggregate	Nodes	Regular	Aggregate	Nodes
250	0	1456	708	6093	4224	1772	2076
250	1	1624	894	7035	4680	1933	2292
250	10	1360	814	5592	3888	1630	1896
250	20	1192	683	5103	3408	1428	1662
250	50	1192	998	4983	3368	1367	1632
2500	0	110 837	15 607	39 942	329 035	29 882	16 572
2500	1	105 070	17 629	37 965	311 734	28 760	15 696
2500	10	92 193	18 733	34 695	273 814	25 541	13 794
2500	20	96 301	32 471	34 734	285 348	26 311	14 358
2500	50	75 524	38 242	28 359	223 175	21 228	11 214
10 000	0	1 472 978	102 681	106 338	4 381 061	218 262	55 746
10 000	1	1 594 109	143 796	111 624	4 730 369	233 541	60 120
10 000	10	1 433 540	263 743	103 083	4 256 800	212 476	54 120
10 000	20	1 323 051	340 302	96 519	3 930 028	195 974	49 974
10 000	50	971 552	428 862	75 201	2 886 173	148 135	36 702
20 000	0	5 894 416	303 294	177 528	17 456 636	735 008	110 820
20 000	1	5 655 910	359 773	172 731	16 771 792	706 347	106 542
20 000	10	5 332 268	902 307	165 516	15 799 614	664 737	100 326
20 000	20	4 838 980	1 190 362	154 194	14 327 888	606 927	90 948
20 000	50	3 891 842	1 664 650	128 907	11 527 790	493 130	73 188

The filter databases are synthetically generated using injection of zero length prefixes (wildcards) with different rates $(0, \ldots, 50)$. The other entries in the database are generated using random selection of prefixes from the MAE-EAST routing table and port domains and protocol numbers extracted from our commercial firewall databases. A word is made up of 32 bits. The first column NR represents the number of rules using port number ranges. The results in the table show both the memory space occupied by the bit vectors with or without aggregation as well as the memory space occupied by the nodes.

of the memory required by the algorithm. For certain applications, such as stateful filters and dynamic insertion of QoS filters, better worst-case update times may be necessary. We add the following ideas to ABV to allow fast insert/delete operations. We used these rules implicitly in all the experiments above but summarize our new ideas here:

- *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. We did exactly this when we defined *bitVect*1 above.
- *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 or-

Table 9
The effect of aggregation on the total size of the memory occupied by the appended bit vectors in both the original bit vector scheme and our scheme (SBV)

NR	% of prefixes	BV			SBV		
		Regular	Aggregate	Nodes	Regular	Aggregate	Nodes
250	1	1800	878	7473	5216	2134	2562
250	10	1328	736	5616	3840	1626	1884
250	20	752	461	3105	2120	1007	1026
250	50	1272	573	5013	3608	1544	1752
2500	1	100 646	15 385	36 849	298 067	27 318	14 994
2500	10	98 829	13 868	35 376	290 562	27 015	14 562
2500	20	117 315	17 865	37 527	336 303	31 317	16 632
2500	50	97 091	15 658	27 105	260 937	27 943	12 444
10 000	1	1 507 095	101 785	107 967	4 471 831	222 140	56 832
10 000	10	1 546 220	106 599	103 551	4 493 428	225 782	56 496
10 000	20	1 387 216	102 554	92 181	3 964 145	205 862	49 398
10 000	50	1 077 033	98 494	69 414	2 981 012	177 339	36 498
20 000	1	5 894 416	303 294	177 528	16 512 002	696 396	104 760
20 000	10	5 091 884	277 033	153 528	14 769 844	634 375	92 760
20 000	20	5 319 748	292 784	155 679	15 302 570	669 256	95 682
20 000	50	3 714 684	251 874	115 128	10 549 352	515 096	65 508

The filter databases are synthetically generated using injection of prefixes which are sharing a common subprefix with different rates (0, ..., 50). The other entries in the database are generated using random selection of prefixes from the MAE-EAST routing table and port domains and protocol numbers extracted from our commercial firewall databases. A word is made up of 32 bits. The results in the table show both the memory space occupied by the bit vectors with or without aggregation as well as the memory space occupied by the nodes.

der (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 can find all matches and map back to the old order number using the techniques in [12].

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. The bit vector introduced above is identical with the one we introduced in the previous sections for fast conflict detection. Therefore, the scheme described above has two features: it allows fast updates and it also allows fast conflict detection.

Fig. 2 illustrates the modified trie construction for the simple two-dimensional example database in Table 2. For example, in Fig. 2, the bit vector associated with the rightmost node corresponding to prefix 1111∗ in the second field is now 00000010000 instead of 00100011011 in Fig. 1. On the other hand, a search for prefix 1111∗ would yield three valid prefixes ∗ (with bitmap 00000001001), the prefix 111∗ (with bitmap 00100000010) and the prefix 1111∗ (with bitmap 00000010000) and the OR of these bitmaps would yield the same answer found in Fig. 1 which is 00100011011.

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 10. The table shows the worst case update time (measured in memory accesses) and the worst case lookup time for 3 algorithms: the original BV algorithm, 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 increased by up to a factor of

Table 10
ABVI vs. ABV vs. BV: the total number of memory locations that are modified by an update operation in the worst case, and the worst case lookup time

Filter	Modified memory locations			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

two when compared to the BV scheme. This may be an acceptable tradeoff. However, for larger databases, ABVI lookups are faster than the BV scheme though slower than ABV. In the case of a synthetic 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 our scheme with aggregation comparing with 1250 memory accesses in the case of BV. Also, our implementation for ABVI does not do any sorting (see [12] for an explanation of 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 [16]) will make ABVI more competitive with ABV in search times.

7. Conclusions

The bit vector scheme introduced by Lakshman and Stiliadis from Lucent is a seminal scheme with an efficient hardware or software implementation. However, this scheme only scales to medium size databases, does not allow fast updates, and the naive extension to handle conflict detection requires subtrie traversal and is thus very slow. The scheme described in [12] scales to large databases but has the second and third problems.

Our paper addresses all three of the above problems in the original bit vector scheme (BV) [6]. Recognizing that subtrie traversal is a bottleneck, we introduce *two* new bit vector semantics: one based on subtrie matches, and one based on exact matches. To handle fast insertion, we added three other ideas: making search compute the union of all valid bitmaps on the path (this slowdown is mitigated greatly by aggregates), leaving holes after deletes that can be filled by later inserts, and (for search) computing all matches and mapping back to the original manager-specified order.

By putting together this package of ideas, we provide a scheme which has all three features: (1) packet classification that is only a small constant slower than the fastest packet classification algorithms, (2) fast updates, and (3) conflict detection that is an order of magnitude faster than the best general purpose algorithms described in the literature.

We evaluated our implementation on both industrial firewall databases and synthetically generated databases. We studied the effect of wildcard injections on both schemes BV and SBV. The average conflict detection time for a database with about 20 000 rules increases by a factor of 44 in BV (or 13 times with aggregation) when the ratio of wildcards inserted in the database increases from 0 to 20%. On the other hand, in the case of our SBV algorithm this increase is insignificant for the same databases. This is because SBV limits the number of bit vectors which are read. Overall, for a database with 20 000 rules and 20% wildcard injection, our scheme with aggregation runs on average 50 times faster than the best scheme in the existing literature, and 420 times faster than simplistic extensions of the bit vector schemes that we use as a point of departure.

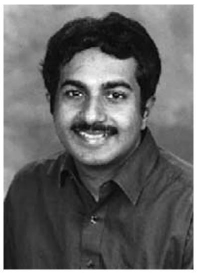
The additional bit vector we introduced for conflict detection also proved useful for allowing fast update operations. In our scheme an update operation modifies this bit vector in only one node per trie in all the cases while in BV with or without aggregation a worst case scenario for update may modify all the valid prefix nodes in the tries. However, our scheme has lower performance results for Search than BV scheme with or without aggregation for a small number of rules but it can perform better when the number of rules increases. For example, in the case of a synthetic 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 our scheme with aggregation comparing with 1250 memory accesses in the case of the BV scheme.

Finally, we note that the algorithms in this paper could be used solely for conflict detection, in which case its one disadvantage, a very small slowdown in search, is not even a factor. We believe that conflict detection, though largely ignored commercially today, will become an important problem in the future as router classifiers grow in size and use dynamic rule insertion to provide better QoS and security guarantees. We believe the algorithm described in this paper can provide a fast solution for this important problem.

References

[1] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast scalable level four switching, in: Proceedings of ACM Sigcomm, 1998.
[2] V. Srinivasan, S.G. Varghese, Packet classification using tuple space search, in: Proceedings of ACM Sigcomm, 1999.
[3] A. Hari, S. Suri, G. Parulkar, Detecting and resolving packet filter conflicts, in: Proceedings of Infocom, 2000.
[4] P. Gupta, N. McKeown, Packet classification on multiple fields, in: Proceedings of ACM Sigcomm, 1999.
[5] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, in: Proceedings of Hot Interconnects VII, Stanford, 1999.
[6] T.V. Lakshman, D. Stidialis, High speed policy-based packet forwarding using efficient multi-dimensional range matching, in: Proceedings of ACM Sigcomm, 1998.
[7] M.M. Buddhikot, S. Suri, M. Waldvogel, Space decomposition techniques for fast layer-4 switching, in: Proceedings of the Conference on Protocols for High Speed Networks, 1999.
[8] L. Qiu, G. Varghese, S. Suri, Fast firewall implementation for software and hardware based routers, in: Proceedings of the 9th International Conference on Network Protocols (ICNP), 2001.
[9] V. Sahasranaman, M.M. Buddhikot, Comparative evaluation of software implementation of layer-4 packet class schemes, in: Proceedings of the 9th International Conference on Network Protocols (ICNP), 2001.
[10] A. Feldman, S. Muthukrishnan, Tradeoffs for packet classification, in: Proceedings of Infocom vol. 1, 2000, pp. 397–413.
[11] Memory–memory, http://www.memorymemory.com, 2000.
[12] F. Baboescu, G. Varghese, Scalable packet classification, in: Proceedings of ACM Sigcomm, 2001.
[13] V. Srinivasan, A packet classification and filter management system, in: Proceedings of Infocom, 2001.
[14] D. Eppstein, S. Muthukrishnan, Internet packet filter management and rectangle geometry, in: Proceedings of the 12th ACM-SIAM Symposium Discrete Algorithms, 2001, pp. 827–835.
[15] M. Inc., Ipma statistics, http://nic.merit.edu/ipma, 2000.
[16] D. Shah, P. Gupta, Fast updates on ternary-cams for packet lookups and classification, in: Proceedings of Hot Interconnects VIII, Stanford, 2000.

Florin Baboescu has a Ph.D. in Computer Science from University of California, San Diego and a M.Sc in Computer Engineering from University Politehnica, Bucharest, Romania. He works on the design of algorithms for IP lookups and packet classification as well as scheduling mechanisms. His other research interestes are in the area of peer to peer service design, scalable services and protocols for robust wide-area mobile internetworking.



George Varghese worked at DEC for several years designing DECNET protocols before obtaining his Ph.D. in 1992 from MIT. After working from 1993–1999 at Washington University, he joined UCSD in 1999 where he currently is a professor of computer science. He works on efficient protocol implementation and protocol design. He won the ONR Young Investigator Award in 1996, and is an ACM Fellow. Together with colleagues, he has 12 awarded and several pending patents. Several of the algorithms he has helped develop (e.g., IP Lookups, timing wheels, DRR) have found their way into commercial systems.