Packet Classification using Tuple Space Search

V. Srinivasan* S. Suri† G. Varghese‡
cheenu@ccrc.wustl.edu, suri@cs.wustl.edu, varghese@ccrc.wustl.edu
Department of Computer Science, Washington University in St. Louis

Abstract

Routers must perform packet classification at high speeds to efficiently implement functions such as firewalls and QoS routing. Packet classification requires matching each packet against a database of filters (or rules), and forwarding the packet according to the highest priority filter. Existing filter schemes with fast lookup time do not scale to large filter databases. Other more scalable schemes work for 2-dimensional filters, but their lookup times degrade quickly with each additional dimension. While there exist good hardware solutions, our new schemes are geared towards software implementation.

We introduce a generic packet classification algorithm, called *Tuple Space Search (TSS)*. Because real databases typically use only a small number of distinct field lengths, by mapping filters to tuples even a simple linear search of the tuple space can provide significant speedup over naive linear search over the filters. Each tuple is maintained as a hash table that can be searched in one memory access. We then introduce techniques for further refining the search of the tuple space, and demonstrate their effectiveness on some firewall databases. For example, a real database of 278 filters had a tuple space of 41 which our algorithm prunes to 11 tuples. Even as we increased the filter database size from 1K to 100K (using a random two-dimensional filter generation model), the number of tuples grew from 53 to only 186, and the pruned tuples only grew from 1 to 4. Our Pruned Tuple Space search is also the only scheme known to us that allows *fast updates* and fast search times. We also show a lower bound on the general tuple space search problem, and describe an optimal algorithm, called *Rectangle Search*, for two-dimensional filters.

*Research supported in part by NSF Grant NCR-9628145.
†Research supported in part by NSF Grant 9813723
‡Research supported in part by NSF Grant NCR 9813723.

1 Introduction

As the Internet begins to be used for commercial applications, service providers would like routers to provide "service differentiation". Traditional routers do not provide service differentiation because they treat all traffic going to the same Internet destination address identically. Routers with a packet classification [8, 13] capability, however, can distinguish traffic based on destination, source, and application type. Such classification allows various forms of service differentiation: blocking traffic sent by insecure sites (firewalls), preferential treatment for premium traffic (resource reservation), and routing based on traffic type and source (QoS routing).

While more general applications like resource reservation [2]) and QoS routing are likely to be part of future routers, many routers today implement *firewalls* [3] at trust boundaries, such as the entry and exit points of a corporate network. A firewall database consists of a series of packet filters (or rules based on packet header fields) that implement security policies. Despite the progress made in the last year on solutions to the packet classification problem [8, 13], existing firewall software is still slow.

While the general solution in [8] can handle thousands of filters at very high speeds, it is geared towards hardware and uses hardware parallelism and high speed memories. The solutions in [13] have a high worst case figure for the general packet filter problem. Thus there is room for further research especially in the area of *software packet classification.* Existing solutions are also optimized for the case when updates are infrequent. However, many firewall vendors now offer stateful filters [4]. For example, the sending of a UDP request may trigger the addition of a filter addition that allows the response to flow past the firewall. This may require filter insertion in the order of microseconds. Other applications that may require fast filter updates include resource reservation protocols like RSVP [2].

Thus faster software packet classification with fast update times can benefit screening routers and many commercial firewall software packages. It can also be useful for other applications of packet classification implemented in say endnodes, which are unlikely to use FPGA or ASIC based solutions.

A general filter consists of arbitrary prefix or range specifications on the destination, source, protocol, port number and possibly other fields. There is evidence that the general filter problem is a hard problem [8, 13] and requires either memory of N^K or time of $O(N)$, where N is the number of filters and K is the number of dimensions. We confirm this growing body of evidence in our paper with some new lower bounds in a hashing model, which complement the earlier lower bounds on multidimensional range matching quoted in [8, 13].

The lower bounds indicate that to do better one has to *exploit the semantics of actual databases.* Since firewall databases are commonly used, we decided to examine actual firewall databases to see if there were some regularities we could exploit. On examination we found that there were only a few combinations of field lengths used in firewall filters. Intuitively, this follows because most address prefixes are based on Class C (24 bit) and Class B (16 bits) prefixes. Similarly port fields are typically either fully specified port numbers (e.g., port 23), the wildcard range (*), or the single range (> 1024 or ≤ 1023).[1]

This motivated us to examine what we call *Tuple Search.* In its simplest form, Tuple Search examines the space of tuples in a filter database, where a tuple is a combination of field lengths. Each check for a tuple can be efficiently done by hashing. Next, we develop some additional heuristics for speeding up Tuple Search, and use them to construct efficient and practical implementations, some of which have fast update times. We also develop the theory behind Tuple Search. We show that the general filter case is indeed hard in the Tuple Search paradigm as well, confirming the intuition in [8, 13]. For the special case of two-dimensional filters, we describe an optimal Tuple Search scheme called Rectangle Search, whose performance is comparable to the two-dimensional algorithms presented in [8, 13].

This paper is organized as follows. We provide background on firewalls in Section 2, and formally describe the packet classification problem in Section 3. We survey related work in Section 4. We start our discussion of tuple space search in Section 5 with the simplest tuple search. In Section 6, we describe a simple heuristic variant of the basic tuple search called *Tuple Pruning*, that has fast search and update times. While Tuple Pruning appears to be our most practical algorithm (based on our experience with real databases), its worst case case search time for *arbitrary* databases can be bad. In Section 7, we show how to improve the worst case search time of tuple search using markers and precomputation. In Section 8, we also describe an optimal algorithm for 2-dimensional filters. Next, we describe lower bounds on the general tuple search problem in Section 9. In Section 10 we describe another balancing heuristic for computing a good probe sequence for general tuple spaces while exploiting markers and precomputation. We conclude in Section 11.

[1]Because BSD UNIX reserves ports 0 to 1023 for local use only by root, these ports are only used by servers, not clients. Other operating systems have followed this custom. This allows packets sent by servers to be distinguished from packets sent by clients. Filters for X servers are another non-trivial example of a port range (e.g., 60000-61000) but these are less common.

2 A Brief Introduction to Firewalls

While the techniques in our paper are applicable to any application that requires packet classification, we provide some background on firewalls. Firewalls provide a concrete application of packet classification where fast software implementations are currently desired.

Firewalls are implemented using various combinations of two basic techniques [4]: *packet filtering* and *application level gateways* (also known as proxy services). In packet filtering, a so-called *screening router* (also known as a *choke router*) sits between the external and internal worlds, and allows or blocks certain types of packets. Unlike conventional routers, screening routers make their decision based on Layer 3 headers as well as Layer 4 and even application headers. For example, this flexibility allows a screening router to block Telnet (by blocking packets sent to TCP port 23) or to disallow incoming TCP connections from some ports (by inspecting TCP flags).

Application level gateways are specialized server programs that run on firewall hosts that take user requests (e.g., for Telnet and FTP) and forward them according to the site policy. An advantage of proxy services is that they are specialized for each service and can thus implement more sophisticated policies than a screening router by itself. For example, an FTP proxy can allow some users to import files only from some sites. Because they understand the protocol, Proxy Services can also provide more intelligent logging (useful to detect the onset of an attack). On the other hand, one has to construct a proxy service for each possible service one supports and keep it current as implementations change; proxy services usually require some modifications to clients and servers; and proxies don't work with all services [4]. Further, proxy services are only effective with one or more screening routers that restrict communication with the host that implements the proxy service.

Thus in practice, various combinations of the two techniques are used. For example, Telnet and SMTP are handled well using packet filtering. Web services seem best handled using proxies [4] because web proxies are easily available and can improve performance by caching. Passive mode FTP can be handled using packet filtering, but normal model FTP can use port numbers higher than 1023, which can allow access to other services. Thus normal mode FTP is often handled using a proxy.

The most important point to gather from our brief introduction to firewalls is that *software packet filtering routers are an important part of real firewall configurations today.* There are clearly other important components such as proxy and logging services, but packet filtering is important by itself. Our paper is about improving the performance of software packet classification and allowing fast updates. This in turn can allow fast software implementations of screening routers, but can also be used for other packet classification applications such as resource reservation.

3 Problem Statement

Suppose there are K header fields in each packet that are relevant to filtering. Then, each filter F is a K-tuple ($F[1]$, $F[2]$, ..., $F[k]$), where each $F[i]$ is either a variable length prefix bit string or a range. The most common fields are the IP destination address (32 bits), IP source address (32 bits), protocol type (8 bits), and port numbers (16 bits) of destination and source applications, and protocol flags. Since the number of distinct protocol flags, such as TCP ack, are limited, we can combine them into the protocol field itself. (TCP flags are important for packet filtering because the first packet in a connection does not have the ACK bit set while the others do; this allows a simple filter to block TCP connections initiated from the outside while allowing responses to internally initiated connections.)

The filter $F = (128.112.*, \ *, \ \text{TCP}, \ 23, \ *)$, as an example, specifies a rule for traffic flow addressed to subnet 128.112 using TCP destination port 23, which is used for incoming Telnet; a firewall database may disallow Telnet into its network.

A filter database consists of N filters $F_1, F_2, \ldots, F_N$. Each filter F is an array of K distinct fields, where $F[i]$ is a specification on the i-th field. We often refer to the i-th field as the i-th dimension. Each field i in a filter is allowed three kinds of matches: exact, prefix, and range. Each filter F_j has an associated directive: for example, firewall database could specify whether to accept or block a packet. Each filter also has an associated cost; in firewall databases, filters are linearly ranked, and the position of a filter is used as its cost.

We say that a packet P matches filter F if for all packet fields i, $P[i]$ matches $F[i]$. The packet classification problem is to find the lowest cost filter matching a given packet P.

While our examples only use simple fields in the IP and TCP headers, so called "third-generation" filtering products [4] are emerging that can use application header fields, and also other parameters such as the input link and time-of-day. We note that our techniques apply to the use of other fields as well. For the particular case of filters that depend on the input link (useful for preventing forged source addresses), the simplest technique is to use a separate database per input link.

While the cost of inserting a rule may seem less important than search, this is not true for *dynamic* or *stateful* packet filters. This capability is useful, for example, for handling UDP traffic. Because UDP headers do not contain an ACK bit that can be used to determine whether a packet is the bellwether packet of a connection, the screening router cannot tell the difference between the first packet sent from the outside to an internal server (which it may want to block) and a response sent to a UDP request to an internal client (which it may want to pass). The solution used in some products is to have the outgoing request packet dynamically trigger the insertion of a filter (that has addresses and ports that match the request) that allows the inbound response to be passed. This requires very fast update times.

4 Related Work

For several years, packet filters have been used for demultiplexing of incoming packets directly to user processes [10, 1, 6]. The first packet filtering scheme (to our knowledge) that avoids a linear search through the set of filters is PathFinder [1]. However, PathFinder only allows filters that have wild card fields at the end of the filter (for instance, $(D, S, *, *, *)$ is allowed, but not $(D, *, Prot, *, SrcPort)$). For such a restricted case, all filters can be merged into a generalized trie (with hash tables replacing array nodes) and filter lookup can be done in time proportional to the number of packet fields. DPF [6] uses the PathFinder idea of merging filters into a trie but adds the idea of using dynamic code generation for extra performance. However, it is unclear how to handle intermixed wildcards and specified fields, such as $(D, *, Prot, *, SrcPort)$, using these schemes.

Because our problem allows more general filters, the PathFinder idea of using a trie does not work. There does exist a simple trie scheme to perform a lookup in time $O(K)$, where K is the number of packet fields. The basic idea is to consider a trie as a deterministic state machine and to realize that a general filter can be recognized by a non-deterministic automaton (NDA). Such a NDA can be converted to a trie but with an exponential blowup in the storage cost. Such schemes are described (using DAGs instead of trees) in [5, 9]. To the best of our knowledge, such schemes require $\Theta(N^K)$ storage where K is the number of packet fields and N is the number of filters. Thus such schemes are not scalable for large databases. By contrast, our new schemes require only $O(NK)$ storage.

Linear Search and Caching: Another simple way to solve the filter problem is to do a linear search of the set of filters against a header and then to cache the result of the search keyed against the whole header. There are two problems with this scheme. First, the cache hit rate of full IP addresses in the backbones is typically at most 80-90 percent [12, 11]. The poor cache hit rate is caused in part by web traffic, where each flow consists of just a few packets; if a web session sends just 5 packets to the same address, then the cache hit rate is 80 percent. Since caching full headers takes a lot more memory, the cache hit rate should be even worse for the packet classification problem (for the same amount of cache memory). Second, Amdahl's Law shows that even with a 90 percent rate cache, a slow linear search of the filter space will result in poor performance. For example, suppose that a search of the cache costs 100 nsec (one memory access) and linear search of 10,000 filters costs 1000,000 nsec = 1 msec (one memory access per filter). Then the average search time with a cache hit rate of 90 percent is still 0.1 msec, which is rather slow.

Grid-of-Tries and Crossproducting: One of the solutions presented last year for the filter matching problem [13] decomposes the multidimensional problem into several 2-dimensional planes, and uses a data structure, *grid-of-tries*, to solve the 2-dimensional problem. This solution does not scale well as the number of required planes increase. For practical firewall databases, it requires up to 8 planes, with each plane requiring upto 8 memory accesses. This results in a worst case of 64 memory accesses. Another limitation of this scheme is that it does not allow arbitrary ranges.

A second solution in [13] is called *crossproducting*. In this scheme, a longest matching lookup or a range lookup is first performed in each of the dimensions. The results are then concatenated to form a *crossproduct*, which is then mapped to a best matching filter. While crossproducting has good lookup times, it either requires $O(N^K)$ memory or does not provide deterministic search time guarantees.

Hardware solutions: Hardware solutions can potentially use parallelism to gain lookup speed. For exact matches, this is done using Content Addressable Memories (CAMs) in which every memory location, in parallel, compares the input key value to the contents of that memory location. This can be generalized if the CAMs allow certain field positions to be masked out. However, it is difficult to manufacture CAMs with the width required to solve filter problems. Second, hardware solutions run the risk of being made obsolete, in a few years, by software technology running on faster processors and memory.

A clever hardware approach is presented in [8]. While this scheme is optimized for hardware, it can work quite well in software for moderate sized databases. It involves reading a bitmap of size N bits for each of K dimensions. The scheme starts by computing the best matching prefix (or closest enclosing range) for each dimension. With each such prefix (or range) P is associated an N bit vector B_P. Bit i of B_P is set if P is compatible with the i-th filter in the database. Finally, the intersection of the K bitmaps is computed, and the filter corresponding to the first bit set in the intersection is returned as the result. If we consider 5-dimensional filters and $N = 1000$, this involves reading 5000 bits. With a cache line size of 32 bytes, this is only about 19 memory accesses to main memory. However, in its simplest form, the memory requirement for this scheme is $O(N^2)$. Thus it does not scale well to large databases. The update times are also slow because of the potential need to update all bitmaps when a new filter is added. Some techniques presented in [8] allow these bitmaps to be compressed to $O(N \log N)$, but these schemes involve reading $N \log N$ bits per dimension during search, instead of N.

5 Tuple Space Search

Our scheme is motivated by the observation that while filter databases contain many different prefixes or ranges, the number of distinct *prefix lengths* tends to be small. Thus, the number of distinct combinations of prefix lengths is also small. This observation seems to be validated by our empirical study of some industrial firewall databases. We can define a tuple for each combination of field length, and call the resulting set *tuple space*. Since each tuple has a known set of bits in each field, by concatenating these bits in order we can create a hash key, which can then be used to map filters of that tuple into a hash table.

Suppose we have a filter database FD with N filters, and these filters result in m distinct tuples. Since m tends to be much smaller than N in practice, even a linear search through the tuple set is likely to greatly outperform the linear search through the filter database. Starting with this simple observation, we then develop several optimizations that reduce the search cost further. With this motivation, we first define the notion of tuple space more formally.

5.1 Defining Tuple Space

Consider a filter database FD that contains N filters, each specifying K fields. We will call these *K-dimensional filters*. While our results are general, we will explicitly consider 5-dimensional filters in our examples and experiments, whose fields are IP source, IP destination, protocol type, source port number, and the destination port number. IP source and destination prefixes have at most 32 bits; the protocol is specified by 8 bits; and the port numbers are 16 bit addresses.

A tuple T is vector of K lengths. Thus, for example, [8, 16, 8, 0, 16] is a 5-dimensional tuple, whose IP source field is a 8-bit prefix, IP destination field is a 16-bit prefix, and so on. We say that a filter F *belongs* or *maps* to tuple T if the ith field of F is specified to exactly $T[i]$ bits. For example, considering 2-dimensional filters, both $F_1 = (01*, 111*)$ and $F_2 = (11*, 010*)$ map to the tuple $[2, 3]$.

Filters always specify IP source or destination addresses using prefixes, so the number of bits specified is clear. The port numbers, however, are often specified using ranges, and the number of bits specified is not clear. To get around this, we define the *length* of a port range to be its *nesting level*. For instance, the full port number range $[0, 65535]$ has nesting level and length 0. The ranges $[0, 1023]$ and $[1024, 65535]$ are considered to be nesting level 1, and so on. If we had additional ranges $[30000, 34000]$ and $[31000, 32000]$, then the former will have nesting level 2 and the latter level 3. (We assume that port number ranges specified in a database are non-overlapping.)

While the nesting level of a range helps define the tuple (or hash table) it will be placed in, we also need a key to identify the filter within the hash table. To this end, we use a *RangeId*, which is a unique id given to each range in any particular nesting level. So the full range always has the id 0. The two ranges at depth 1, namely, ≤ 1023 and > 1024, receive the ids 0 and 1 respectively. Suppose we had ranges $200 \ldots 333$, $32000 \ldots 34230$ and $60000 \ldots 65500$ at level 2, then they would be given ids 0,1 and 2 respectively. With range ids, we can now map any 5-dimensional filter to a 5-dimensional tuple. To understand how the *RangeId* works, let us draw an analogy between prefixes and ranges. An address D can have some m prefixes $P_1 \ldots P_m$ in the database that match it, such that P_{i-1} is a prefix of P_i. In the case of a range match, each of these prefix lengths correspond to a nesting depth.

Notice that a given port in a packet header can map to a different ID at each nesting level. For example, with the above ranges, a port number 33000 will map on to three *RangeId* values, one for each nesting depth: Id 0 for nesting depth 0, Id 1 for nesting depth 1, and Id 1 for nesting depth 2. Thus a port number field in a packet header must be translated to its corresponding *RangeId* values before tuple search is performed. In summary, the nesting level is used to determine the tuple, and the *RangeId* for each nesting level is used to form the hash key.

5.2 Searching the Tuple Space

All filters that map to a particular tuple have the same mask: some number of bits in the IP source and destination fields, either a wild card in the protocol field or a specific protocol id, and port number fields that contain either a wild card or a *RangeId*. Thus, we can concatenate the required number of bits from each filter to construct a hash key for that filter. We store all filters mapped to T in a hash table $Hashtable(T)$. A probe in a tuple T involves concatenating the required number of bits from the packet as specified by T (after converting port numbers to *RangeIds*), and then doing a hash in $Hashtable(T)$.

Thus, given a packet P, we can linearly probe all the tuples in the tuple set, and determine the least cost filter matching P. While this is a very naive search strategy, the search cost is proportional to m, *the number of distinct tuples.* On the other hand, the current practice of performing a linear search through the filter database has cost N, which tends to be much larger than m.

We ran tests on 4 industrial firewall databases which we refer to generically as Fwal-1 to Fwal-4. We found that while the number of filters ranged from 68 to 278, the number of tuples ranged from 15 to 41. The following table shows this empirical statistic.

Database	Size	Dest Prefixes	Src Prefixes	Tuples
Fwal-1	278	57	66	41
Fwal-2	158	38	36	28
Fwal-3	183	31	29	24
Fwal-4	68	32	22	15

Table 1: Four Firewall databases.

Thus, even without any additional improvements, tuple space search seems better than linear search. Furthermore, since each filter belongs to a unique tuple, the *update cost* (inserting or deleting a filter) is small; just one memory access for a hash, assuming a perfect hash function that is chosen to avoid hash collisions.

6 Tuple Pruning Algorithm

We now describe our first heuristic for improving tuple space search, which is not only very simple but also gives the best search and update performance of all our heuristics.

The main motivation behind this new heuristic, called *Tuple Pruning*, is that in real filter databases there seem to be very few *prefixes of a given address.* For example, when we examined the Mae-East prefix database [7], we found that no address D has more than 6 matching prefixes. Suppose this were true. If we consider Destination-Source filters, then naive Tuple Search may require searching $32 \times 32 = 1024$ (possible combinations of pairs of lengths for the destination and source prefix lengths) tuples. However, if both destination and source prefixes are taken from Mae-East, then if we first find the longest destination match and the longest source match, there are only at most $6 \times 6 = 36$ possible tuples that are compatible with the individual destination and source matches. Thus we have reduced the number of tuples to be searched from 1024 to 36 at the cost of an extra destination and source prefix match.

In essence, tuple pruning seeks to generalize this empirical observation to arbitrary filters by first doing individual longest prefix (or range matches) in each dimension and then searching only the tuples that are compatible with the individual matches. Tuple pruning will benefit if the reduction in the tuple space afforded by pruning offsets the extra individual prefix (or range) matches on each field. In some sense, this is similar to the Lucent [8] scheme except for the following major difference: while both schemes first do independent matches in each dimension, the Lucent scheme searches through *filters* that are compatible with the individual matches while we search through *tuples* that are compatible with individual matches. Since, as we have said earlier, the number of tuples grows much slower than the number of tuples, we expect tuple pruning to scale better than the Lucent scheme.

To set up Tuple Pruning, for each destination prefix D in the database, we compute a *tuple list* (or bitmap) containing the names of tuples that have a filter with destination equal to D or a prefix of D. Similarly, for each source address S in the database, we compute a list containing the names of tuples that have a filter with source equal to S or a prefix of S. We can do this for the protocol and port number fields as well, but our implementation results seem to indicate that (at least for the databases we had) the results do not improve much by using additional fields.

For instance, suppose $D = 1010*$. Suppose all the filters whose destination is a prefix of D belong to tuples $[1,4]$, $[1,1]$, and $[2,3]$. Then, the tuple list of D contains these 3 tuples. Similarly suppose $S = 0010*$. Suppose all the filters whose source is a prefix of S belong to tuples $[2,4]$, $[1,1]$, and $[2,5]$.

Now, our search algorithm works as follows. Given a packet header P, we first compute the longest matching prefix PD of the destination address $P[1]$, and the longest matching prefix PS of the source address $P[2]$. We now take the tuples lists stored with PD and SD, and find their common intersection. For instance, if $PD = D$ and $PS = S$ in the above example, the intersection list only includes tuple $[1,1]$. We now probe the tuples in this intersected list.

The update algorithm is also quite simple. First, we refine our description of the search process to describe how the the tuple list associated with a prefix D is computed. Rather than store the tuple list of D and all its prefixes with D, we store only the tuples associated with D. We can obtain the tuple lists associated with prefixes of D if we do a trie search for D because such a search must necessarily encounter all the prefixes of D.

When a new filter is added into the database, we add its destination and source prefixes (say D and S) to the destination and source tries. Next, for each prefix we add (for example D), we maintain an augmented list of tuples corresponding to filters that contain D. The augmented list is a list of tuples (as before) but each tuple T also contains a reference count of the number of distinct filters that have destination D and map to T. The reference count is useful for deletion of filters which decrements the reference count in an analogous way. When the reference count reaches zero, the corresponding tuple is removed from the tuple list of the associated prefix. Note that the augmented tuple lists containing counts need not be part of the search data structure but must be maintained by the update process. Thus the update time requires 2 best matching prefix operations plus a constant time to update the augmented tuple lists for each of three fields.

Table 2 shows the effect of Tuple Space Pruning on our firewall databases. We calculated the size of the worst case set of pruned tuples for all possible combinations of source, destination, and protocol matches. This pruned size is shown in the Pruned Tuples column. Note that this simple heuristic prunes the tuple space by a factor of at least three. We tried pruning on port fields as well, but found that we did not get better numbers. However, when we used the two port fields instead of the address fields for pruning, we obtained similar results. Thus for the small databases we examined pruning based either on destination and source addresses or destination and source ports produced the best results. However, for larger databases, we expect that pruning on all fields will be beneficial.

Database	Size	Tuples	Pruned Tuples
Fwal-1	278	41	11
Fwal-2	158	28	6
Fwal-3	183	24	7
Fwal-4	68	15	5

Table 2: The worst-case number of hash probes produced by the Pruned Tuple Search Method.

6.1 Experiments with random filters

We believe that as the filter database grows larger the additive cost of doing a separate best matching prefix on the destination and source fields will remain a constant, but the relative gains of Pruned Tuple search will remain. Since there does not seem to be any publicly available large filter database, we experimented by creating random filters to see how well the tuple pruning algorithm scales. We generated N source-destination filters, where the source and the destination prefix of each filter was chosen uniformly at random from the MaeEast database [7]. We tried pruning on the destination field, the source field, and then on both the fields. We present our experimental results in Table 3. When the number of filters is large (> 10000), we could not strictly test the worst case pruned tuple size for all possible crossproducts. Thus for those two cases, we did a statistical sampling test by considering a million randomly chosen crossproducts, and finding the worst case size of the pruned tuples across all the random samples.

Size	Tuples	Dest-Pruned	Src-Pruned	Both-Pruned
1000	53	1	1	1
5000	92	2	2	2
10000	104	3	3	2
50000	151	4	6	3
100000	186	6	8	4

Table 3: Number of tuples found in a randomly generated filter database, and the effect of pruning. Prefixes were randomly chosen from the MaeEast database.

While this test does not in any way guarantee that Tuple Pruning will scale as well for large databases in practice, reducing the tuple space from say 104 (naive tuple search) to 2 (pruned tuple) at the cost of say 8 more memory accesses, does suggest good scaling behavior. More importantly, the Pruned Tuple Search algorithm has a very fast update time unlike the Balancing Heuristic algorithm we describe later (and all the other techniques in the literature). Thus Pruned Tuple Search is the only scheme we know of that provides fast search times and yet can be used for applications like Stateful Packet filters and RSVP filters.

7 Improved Tuple Space Search via Precomputation

Naive Tuple search can have a large search time; pruned tuple search can improve the search time dramatically but it depends on assumptions on the structure of existing databases. Can we improve the performance of tuple search and provide some worst case guarantees without making such assumptions? The remainder of this paper is devoted to answering this question. In essence, our results will show that we can do fairly well for the case of two dimensional filters by using precomputation (which increases filter update time), but the worst case improvement is only marginal for the general filter problem.

The ideas described below can be considered to be a generalization of the ideas in [14] for finding longest matching prefix. For example, simple tuple space search will take $O(W)$ time for finding the longest matching prefix of a W length destination address field, but the techniques of markers and precomputation can be used to reduce the search time to $O(\log W)$ at the cost of slower insertion times [14]. We will show that for two-dimensional filters we can reduce the search time from W^2 memory accesses to $2W$ accesses using similar techniques. We will also describe lower bounds to show that this algorithm is optimal.

We now consider such strategies for reducing the search space. The main idea is that a probe into a tuple T_i can be used to eliminate a subset of the tuple space. In particular, if a probe succeeds, then we can eliminate all the tuples that are *coordinate-wise shorter* than T_i, because we can precompute the best matching filter from those tuples and store the answers with filters in T_i. Similarly, if a probe fails in T_i, then we can eliminate all the tuples that are *coordinate-wise longer* than T_i, because each filter in those tuples can leave a marker filter in T_i. This is a time-space tradeoff, because markers require additional memory, but the search cost can be reduced. We now describe the marker and precomputation ideas in more detail.

7.1 Markers and Precomputation

Consider a tuple $T_i = [l_1, l_2, \ldots, l_K]$. We can partition the remaining tuple space into 3 disjoint parts, $Short(T_i)$, $Long(T_i)$ and $IC(T_i)$ (where IC stands for *incomparable*). The set $Short(T_i)$ contains all those tuples whose length vector is coordinate-wise shorter than T_i. That is, a tuple $T_j = [h_1, h_2, \ldots, h_K]$ belongs to set $Short(T_i)$ if and only if $h_i \leq l_i$, for all $i = 1, 2, \ldots, K$, and $T_j \neq T_i$.

Similarly, $Long(T_i)$ contains all those tuples whose length vector is coordinate-wise longer than T_i. Specifically, a tuple $T_j = [h_1, h_2, \ldots, h_K]$ belongs to set $Long(T_i)$ if and only if $h_i \geq l_i$, for all $i = 1, 2, \ldots, K$, and $T_j \neq T_i$.

A tuple T_j, where $T_j \neq T_i$, that is neither in $Short(T_i)$ nor in $Long(T_i)$ belongs to the *incomparable set* $IC(T_i)$. Figure 1 shows this partitioning of the tuple space into the three sets.

As an example, consider the tuple $T = [2, 3, 2]$. Then tuple $[1, 1, 1]$ belongs to $Short(T)$; the tuple $[5, 3, 3]$ belongs to $Long(T)$; and tuple $[1, 4, 1]$ belongs to $IC(T)$.

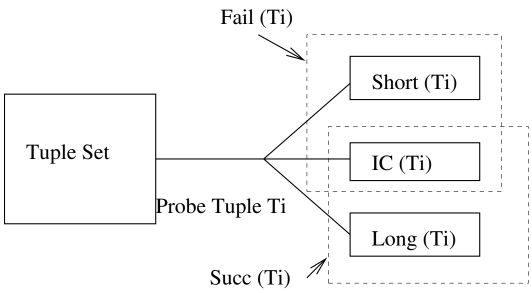

Figure 1: Figure showing the three sets obtained when a probe is done at tuple T_i. The three sets are mutually exclusive and every tuple other than T_i falls into one of the three sets. The sets $Fail(T_i)$ and $Succ(T_i)$ are defined later.

Since each tuple in $Short(T)$ is less specific than T on all fields, it is possible to *precompute* the best matching filter information for the set $Short(T)$ and store it with the filters in T. Specifically, let F be a filter in T. We can precompute the best filter matching F in the set of filters than belong to $Short(T)$. Thus, if we get a match with filter F during the probe in T, we no longer need to search $Short(T)$. In other words, if the probe at T returns a match, we can restrict our search to the tuples in $IC(T)$ and $Long(T)$. Their union is the set $Succ(T)$ illustrated in Figure 1.

Similarly, we have each filter F in $Long(T)$ leave a *marker*, which is the filter obtained by using only l_i bits of the ith field of F, where $[l_1, l_2, \ldots, l_K]$ is the length vector for T. Since each filter in $Long(T)$ has at least l_i bits specified in field i, this is possible. (As an example, a filter $F = (1010*,\ 110*)$ of tuple $[4, 3]$ leaves the marker $F' = (10*, 11*)$ in the tuple $[2, 2]$.) Now, if we probe T and do not get a match, we can easily eliminate the set $Long(T)$—if any filter in that set matched the packet header, its marker entry in T would have produced a match. Thus, when the probe in T fails, we can restrict our search to the tuples in $IC(T)$ and $Short(T)$. Their union is the set $Fail(T)$ illustrated in Figure 1.

While this general strategy seems promising, we need to find a specific instantiation of the strategy (which would specify the sequence of tuples to be probed) that can provide an improvement *in the worst-case*. We start by showing a specific search strategy that works well for the two-dimensional case. Later we show some lower bound results that shed some light on the difficulty of the general problem. Finally, we return to the general problem and show a heuristic search strategy that attempts to compute an *optimal probe sequence for a given filter database*, using precomputation and markers.

8 Rectangle Search: An Optimal Algorithm for 2-D Filters

One can argue that two-dimensional filters (especially, destination-source filters) are interesting in their own right for multicast forwarding or virtual private networks (VPNs) [8]. A very simple application of a two dimensional filter database (suggested by Jon Turner) is a network monitor that computes traffic statistics between all source and destination subnetworks. In this section, we will restrict ourselves to two dimensional filters.

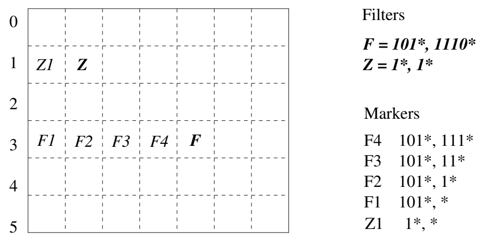

Figure 2: Illustration of markers and precomputation.

Several efficient packet classification algorithms exist for the two-dimensional problem, such as range matching based on fractional cascading [8] or grid-of-tries for prefix matching [13]. Is there a similar efficient algorithm for the 2D case in the tuple space model? We show below that the answer is yes.

We give a simple algorithm than computes the best matching filter in a $W \times W$ tuple space using $2W - 1$ probes, which is optimal in view of a lower bound we show later. The algorithm uses markers and precomputation to eliminate subsets of the tuple space after each probe. We call this algorithm *Rectangle Search*, and it uses a different marker and probing strategy than the heuristics of Section 7.1. We now describe the algorithm.

A filter leaves a marker at all the tuples to its left in its row. So, a filter F that belongs in tuple $[i, j]$ leaves a marker in tuples $[i, j-1], [i, j-2], \ldots, [i, 1]$. Each filter (or marker) also precomputes the least cost filter matching it from among the tuples above it in its column. That is, a filter (or marker) in tuple $[i, j]$ precomputes the least cost filter matching it from the tuples $[i-1, j], [i-2, j], \ldots, [1, j]$.

Figure 2 shows an example of precomputation and markers, using two filters F and Z. The marker F_2 precomputes the best matching filter among the entries in the column above it, which in this example is Z.

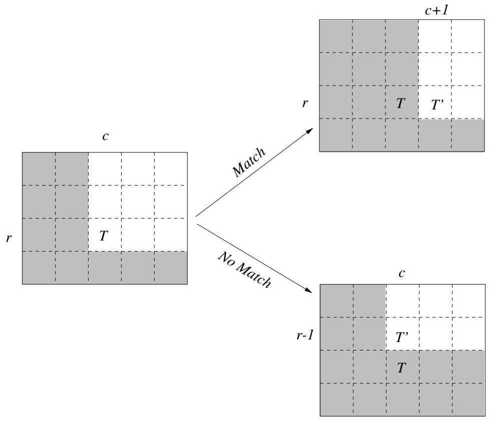

Figure 3: If a probe in tuple T results in a match, then the entries in the column above T are eliminated; if the probe results in no match, then the entries in the row to the right of T are eliminated. In both cases, the next probe is done in tuple T'.

Given these markers and precomputation, we can now describe a search strategy that outputs the best matching filter after $2W-1$ worst-case probes. We start by probing the lower-left tuple, namely, $[W, 1]$. At each tuple, if the probe returns a match, we move to the next tuple in the same row. If the search returns no match, we move up one row, in the same column. See Figure 3.

When we get a match in a tuple, the matching filter's precomputed information makes searching the tuples above it in its column unnecessary, and so we can eliminate them from the search. This allows us to move to the next column. If we fail to get a match, the marker rule tells us that there is no filter to the right of the current tuple; because otherwise that filter's marker would have produced a match. In this case we can eliminate the tuples to the right of the current tuple, and move to the row above. Thus, each probe eliminates a row or a column. The search terminates when the we reach the rightmost column or the first row. Since there are W rows and W columns, the number of probes needed is at most $2W-1$.

8.1 Time-Space Tradeoffs and Generalized Rectangle Search

Rectangle search requires $O(NW)$ memory, since each filter leaves at most W markers. This performance is comparable to other $O(\log N)$ search algorithms using $O(N \log N)$ memory, since $W \approx \log N$. We can tradeoff memory for speed by using fewer markers. For a filter in column i, we can leave markers in every column to the left of i that is a multiple of some k, and in each column after the largest multiple of k smaller than i. This increases the required number of hashes in the worst case from $2W$ to $3W$, but reduces the memory to $O(NW/k)$. Choosing $k = \sqrt{W}$ gives us an $O(W)$ worst-case algorithm for the 2-dimensional filters with $O(N\sqrt{W}) memory$.

Rectangle Search is designed for square tuple spaces. What can be done for non-square tuple spaces? If we have a $R \times C$ rectangular tuple space, then Rectangle Search gives a worst-case bound of $R + C - 1$. However, notice that if $R \ll C$, then binary search in each row can achieve $O(R \log C)$, which may be much better. We can extend our Rectangle Search with a new idea, called *doubling search*, which allows us to search a $R \times C$ tuples space in $O(R \log(C/R))$ probes. We do not describe doubling search here for lack of space. The bound for doubling search nicely interpolates the optimal bounds at the two extremes: $O(\log W)$ for the linear tuple space, and $O(W)$ for the square tuple space.

In table 4 we compare several 2-dimensional lookup algorithms and their worst case search and memory complexities.

Scheme	Memory	Search
Linear Search	$O(N)$	$O(N)$
Binary search	$O(N)$	$O(W + \log N)$
Grid-of-tries	$O(NW)$	$O(W)$
Rectangle Search	$O(N\sqrt{W})$	$O(W)$

Table 4: Complexity of 2-dimensional lookup algorithms.

9 Some Lower Bounds

The preceding sections have described several algorithms for the best matching filter problem in the tuple space search paradigm. The attractiveness of our algorithms is based on the premise that the tuple space of practical databases is quite sparse. Several researchers have also investigated packet classification algorithms with good worst-case guarantees [8, 13]. Although some nice results are known for the two-dimensional filters, none of the known algorithms scale well to multi-dimensional filters. As the dimension increases, either the search time degrades quickly, or the memory requirement grows exponentially.

In this section, we argue that the packet classification problem indeed suffers from the curse of dimensionality, and formally establish a lower bound on the number of hash probes needed to find the lowest cost matching filter in the tuple space model. Let us first consider the simple case of 2-dimensional (source-destination pair) filters. What is the best possible bound for searching the $W \times W$ tuple space? Is $O(\log W)$ possible? We establish a lower bound showing that at least $2W - 1$ probes must be made in the worst case. In fact, we prove a more general result: if the tuple space has R rows and C columns, then roughly $R \log(1+\frac{C}{R})$ probes are necessary in the worst case, where $C \geq R$. This shows that our rectangle search algorithm is essentially optimal in the tuple search paradigm.

We will use the "decision tree model" of computation for our lower bound argument. The decision tree model is an abstraction of branching programs, in which each internal node represents a decision, which has a binary outcome. (For instance, in sorting the node corresponds to a comparison test between two elements. In our case, the test involves checking whether the header of a packet matches any of the filters in a tuple, which takes one hashed memory access.) The algorithm starts by performing the test at the root of the decision tree. Based on the outcome, the program branches to one of the children, and continues until a leaf node is reached, at which point the algorithm produces its output. The execution of the algorithm on a specific instance corresponds to a path in the decision tree from the root to some leaf. The worst-case running time of the algorithm, therefore, corresponds to the *height* of the decision tree. Showing a lower bound of L on a problem's complexity requires one to show that there are at least 2^L "configurations" of the input where no two configurations correspond to the same leaf node in the decision tree. We show that the problem of searching a $R \times C$ tuple space, where $C \geq R$, has roughly $(\frac{C+R}{R})^R$ distinct configurations, which will lead to a $\Omega(R \log \frac{C}{R})$ lower bound.

Consider a fixed packet header with say source address S and destination address D. Our argument establishes a lower bound for the search cost of any decision tree algorithm for this fixed header (S, D) over all possible input instances, where the input is a set of filters. Given an instance of the filter database, imagine coloring all those tuples *red* that contain a filter matching the header (S, D). Starting from the lower-left corner, we can draw a unique *leftmost staircase* that contains all the red tuples on or above it. Figure 4 (i) illustrates this concept. In this manner, each instance of the filter database can be associated with a staircase. The key point is that, for each instance, none of the tuples below the staircase contain a filter matching the pair (S, D).

We now show that if two instances of the filter database have distinct staircases, then the decision tree algorithm must follow different paths on these inputs. Indeed, suppose we have two instances corresponding to staircases Z_1 and Z_2. Then, there is a tuple T that lies on or above one staircase but below the other. Without loss of generality, suppose T lies above Z_1, and below Z_2. (See Figure 4 (ii), where the staircase Z_1 is shown in dashed lines.) Clearly, if the algorithm probes the tuple T during its course, then the search path for Z_1 and Z_2 would diverge at that node—since we get a match for Z_1 and no match for Z_2. Thus, the search algorithm must not have probed T.

A common decision path for Z_1 and Z_2 means that the algorithm probes the same set of tuples, makes the same branching decision at each node, and therefore outputs the same tuple as its answer. A simple consequence of this observation is that on this particular search path, the algorithm cannot output tuple T as the best matching filter tuple—since the search path is common to both

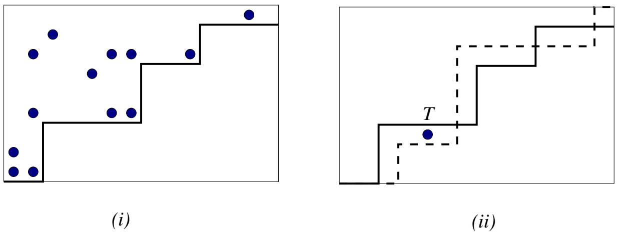

Figure 4: (i) A staircase. The tuples that match the header (S, D) are shown with solid disks. The staircase is shown in thick lines. (ii) Shows two staircases. Tuple T lies above the staircase shown with dashed lines and below the one with solid lines.

Z_1 or Z_2, the output tuple must be a red tuple for both. But then if we simply put the least cost filter in T for a Z_1 instance, *the algorithm's output will be incorrect.* Thus, every staircase must correspond to a distinct leaf node in the decision tree.

Next, we establish a lower bound on the number of distinct staircases. This requires a simple combinatorial argument. If we have a grid of dimensions $R \times C$, where $C \geq R \geq 2$, then the number of staircases connecting the leftmost and rightmost corner is at least

$$\binom{C+R-1}{R-1}$$

One simple way to see this is as follows: each staircase can be uniquely identified by the positions of the vertical (unit) steps. There are $R-1$ are vertical *unit* steps needed to go from bottom row to the top row. Now, we are ready to prove our lower bound. The worst-case search cost of the algorithm is at least as large as the height of the decision tree. If a binary tree has M leaves, it has height at least $\log_2 M$. Thus, the height H of the decision tree has the following lower bound:

$$\begin{aligned} H &\geq \log \binom{C+R-1}{R-1} \\ &\geq \log \left(\frac{C+R-1}{R-1}\right)^{R-1} \\ &\geq (R-1)\log\left(\frac{C}{R-1}+1\right) \\ &= \Omega\left(R \log\left(1+\frac{C}{R}\right)\right). \end{aligned}$$

This completes our proof of the lower bound for the two-dimensional tuple space. While the technique presented above is quite general, it does not yield the best possible lower bounds. For instance, when the tuple space is a $W \times W$ square, the proof above gives a lower bound of about $W-1$, though it is possible to obtain a tighter bound of $2W-1$ using an adversary based argument, which we now present. However, the staircase lower bound provides a bound for a rectangular tuple space.

9.1 An adversary-based lower bound

Consider the following set of tuples:

$$M_1 = \{(l_1, l_2) \mid l_1 + l_2 = W\}.$$

Since $0 \le l_1, l_2 \le W$, there are precisely W tuples in M_1, namely, $(0, W), (1, W-1), \ldots, (W, 0)$. These tuples in fact correspond to the *diagonal* entries of the square tuple space. Using the definition of Section 7.1, the tuples of M_1 are *pairwise incomparable*. That is, if $T = (l_1, l_2)$ and $T' = (l_1', l_2')$, where $l_1 + l_2 = l_1' + l_2' = W$, then $l_1 < l_1'$ implies $l_2 > l_2'$. Thus, probing one tuple does not yield information about the other—neither markers or precomputation help, since each tuple has a longer coordinate than the other. Similarly, we can define a second set of tuples

$$M_2 = \{(l_1, l_2) \mid l_1 + l_2 = W + 1\},$$

which corresponds to the tuples along the *second diagonal* of the tuple space. The set M_2 has precisely $W - 1$ tuples. See Figure 5 for illustration.

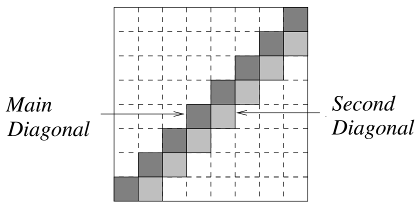

Figure 5: Adversary lower bound.

Now, consider an arbitrary packet header (S, D). We create W filters, one per tuple of M_1. Specifically, the filter $F(i, j)$ corresponding to the tuple (i, j) is constructed by taking the i-bit prefix of S and j-bit prefix of D. For instance, assuming $W = 3$, $S = 101$, and $D = 011$, the filter for tuple $(1, 2)$ is $(1*, 01*)$. Each of these filters may be assigned an arbitrary cost. The adversary's strategy is as follows: if the algorithm probes a tuple in M_1, the adversary returns the matching filter; if the algorithm probes any other tuple, the adversary returns "no match." Notice that this strategy is consistent—the matches along the main diagonal only eliminate the upper triangular tuple space (precomputation), while the lack of matches along the second diagonal only eliminates the lower triangular tuple space (markers).

We claim that any tuple space algorithm must probe at least $2W - 1$ tuples to correctly find the best matching filter for the header (S, D). Clearly, the algorithm must probe the W tuples of M_1—otherwise, since we can arbitrarily choose the filter costs, the filter in the unprobed tuple can be made the cheapest filter, thus foiling the algorithm. The key observation here is that since the tuples of M_1 are pairwise incomparable, probing one does not reveal any information about any other tuple in M_1.

In addition to the tuples of M_1, the algorithm must also probe all tuples in M_2. The adversary returns "no match" on any tuple of M_2 probed. But if the algorithm fails to probe a tuple, say, $(i, j+1)$, where $i + j + 1 = W + 1$, then we can put a least cost filter matching (S, D) in it, and foil the algorithm. Thus, in order for the algorithm to correctly determine the least cost filter matching (S, D), at least $2W - 1$ probes must be made.

Note that this argument cannot be extended to a third diagonal—probing a cell of the middle diagonal will either eliminate some entries on the diagonal above (if a match is found) or it would eliminate some entries on the lower diagonal (if no match is found).

9.2 Extension to multi-dimensional filters

The adversary lower bound argument can be extended to k-dimensional filters, as follows. Consider the set of tuples

$$M = \{(l_1, l_2, \ldots, l_k) \mid l_1 + l_2 + \cdots + l_k = W\}.$$

Clearly, the tuples in set M are *pairwise incomparable*—if T is shorter than T' in some dimension, then T must be longer than T' in some other dimension to keep the sum constant. Now, given a packet header $(H_1, H_2, \ldots, H_k)$, we can create a filter for tuple $(l_1, l_2, \ldots, l_k)$ by taking the l_1-bit prefix of H_1, the l_2-bit prefix of H_2 and so on. Each of these $|M|$ filters match the packet, and since these filters are pairwise incomparable, an adversary can force the algorithm to search all $|M|$ tuples in order to find the least cost filter. So, how many tuples are in the set M? An easy inductive proof shows that $|M| \ge \frac{W^{(k-1)}}{k!}$. Thus, in the hashing model of tuple space search, packet classification among arbitrary filters requires at least $\frac{W^{(k-1)}}{k!}$ tuple probes in the worst case.

10 Applying Markers and Precomputation for General Tuple Search

So far, we have introduced the notion of using markers and precomputation to improve worst case search time in tuple search (Section 7), we have shown an optimal two-dimensional algorithm (Section 8), and we have shown lower bounds showing that markers and precomputation cannot improve the worst case significantly for general filter databases (Section 9). However, just as Tuple Pruning (Section 6) can improve the performance of naive tuple search for *specific databases*, the question still remains: can we find an optimal search strategy using markers and precomputation for specific databases. We already know how to do this for 2D databases, but this certainly does not apply to the real firewall databases Fwal-1 to Fwal-4 that we used earlier. We now address this question.

10.1 A Dynamic Programming Algorithm

Before reading on, the reader may wish to review the general strategy in Section 7.1 and the terminology used there.

Given a tuple space TS, we wish to determine the optimal sequence of probes for computing the best matching filter, using markers and precomputation to eliminate tuples after each probe. This essentially corresponds to constructing a *decision tree* in which each node is labeled with a probe, and the algorithm branches to one or the other subtree based on the probe outcome. Suppose we probe tuple T_i in the first step; then a match in T_i results in searching the set $Succ(T_i)$, while a fail in T_i results in searching $Fail(T_i)$. The optimal cost of searching the tuple space TS can then be written recursively as follows:

$$Opt(TS) = 1 + \min_{i=1}^{|TS|} \max\{Opt(Succ(T_i)), Opt(Fail(T_i))\}.$$

The recurrence is based on the fact that after the probe done in the first step, one of the two tuple subsets, $Succ(T_i)$ or $Fail(T_i)$, needs to be solved optimally, and the best first probe is the one that minimizes the *maximum* of the two costs.

Unfortunately, since the subsets $Succ(T_i)$ and $Fail(T_i)$ are not disjoint, and can have significant overlap, computing $Opt(TS)$ takes exponential time in the worst case. Specifically, in the worst-case, the recurrence for computing $Opt(TS)$ can be written as

$$f(m) = 2mf(m-1) + m^2,$$

where m is the number of tuples in the tuple space and m^2 is the complexity of forming the sets $Succ(T_i)$ and $Fail(T_i)$. This gives $f(m) \geq 2^m$, which is exponential.

10.2 Tuple Search using a Balancing Heuristic

Given the prohibitive complexity of computing $Opt(TS)$, we use some heuristic algorithms. Rather than computing $Opt(Succ(T_i))$ and $Opt(Fail(T_i))$ recursively, we simply use some estimates on their costs. One simple estimate is the number of tuples in the set. That is, as the first probe we pick the tuple T_i that minimizes the larger of $|Succ(T_i)|$ and $|Fail(T_i)|$, and then recursively work on the sets $Succ(T_i)$ and $Fail(T_i)$ using the same heuristic. Unfortunately, even this heuristic ends up being exponential! Indeed, the recurrence for this heuristic is $f(m) = 2f(m-1) + m^2$, which is better than the previous one, but still leads to exponential time. The main difficulty remains that the total size of the two subproblems, $Succ(T_i)$ and $Fail(T_i)$, is still large.

In our final heuristic, we treat the common elements of the two sets, namely, $Succ(T_i) \cap Fail(T_i)$, separately. In other words, when we choose a tuple T_i to probe, we divide the set into three parts as shown in Figure 6. We then use a simple balancing heuristic for the cost function:

$$Cost(T_i) = |IC(T_i)| + \max\{|Short(T_i)|, |Long(T_i)|\}.$$

The tuple T_i with smallest $Cost$ is used as the first probe. We then recursively solve the problem for the three subsets $IC(T_i)$, $Short(T_i)$ and $Long(T_i)$. This heuristic runs in polynomial time, and can be analyzed as follows.

Initially, we start with m tuples. (See Figure 6.) After $O(m^2)$ computation, we decide the tuple that will be probed first in the search sequence. This leaves us with $m-1$ tuples in the next level. We do $O(m^2)$ computation at each level to choose the tuples to be probed at the next level. The maximum number of levels is m, since we can choose each of the tuples only once. This gives us a total time complexity of $O(m^3)$.

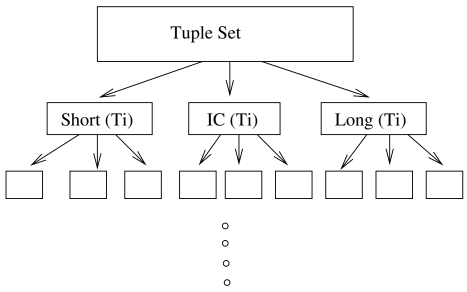

Figure 6: Each level has at most m tuples, and there are at most m levels. Processing each level takes $O(m^2)$ time, for a total of $O(m^3)$.

10.3 Implementation and Measurements

We demonstrate the performance of our heuristic on the 4 industrial firewall databases shown in Table 1. In Table 5, we show for each of the firewall databases the worst-case number of probes, as found by the heuristic.

Database	Size	Tuples	Probes	Time
Fwal-1	278	41	23	0.4
Fwal-2	158	28	17	0.3
Fwal-3	183	24	14	0.2
Fwal-4	68	15	10	0.1

Table 5: Number of probes is the worst-case search length found by the balancing heuristic. The time is the time needed by the heuristic in seconds to compute the probe decision tree. As the table shows, the heuristic seems to reduce the number of probes by about 40%.

When comparing the results of Pruned Tuple Search (Table 2) to that of the Balancing Heuristic (Table 5), we may at first conclude that the Balancing Heuristic is much worse (for example 23 tuples versus 11 for the largest database). However, we have not accounted for the cost of computing the best matching prefix on the Destination and Source fields in Pruned Tuple Search. Using good algorithms, this can add 4 hashes for Destination, 4 for Source, and 1 for Protocol. This would yield more comparable numbers.

11 Conclusion

In this paper, we have presented a new packet classification algorithm that we call tuple space search. Simple tuple space algorithm searches through the field length combinations found in the filter set. It is motivated by the observation that the number of tuples in real databases is much smaller than the number of filters. Our experimental results were limited by the moderately size databases we had access to, the largest of which had 278 filters and 41 tuples. However, we believe that even for very large filter database (say a million filters), the tuple space is unlikely to grow beyond a few hundred. This is because most databases use only a few prefix lengths corresponding to Class A, Class B, and Class C addresses and a small number of port ranges.

Even if the total tuple space were to grow into the thousands (using 32 possible destination and source prefix lengths), we argue that pruned tuple search will produce a much smaller set of pruned tuples. We have examined the Mae-East IP prefix database and have found that no prefix D has more than 6 prefixes that are prefixes of D in the worst case. Thus even the number of D-S tuples is probably bounded by 36 instead of $32 * 32 = 1024$. Because our empirical evidence shows a substantial reduction in the pruned set of tuples, we expect this behavior to be valid for larger databases. Pruned Tuple Search is also the only scheme we know of that has fast update times. It seems appropriate for software firewall implementations that require dynamic updates.

Despite the apparent utility of Pruned Tuple Search, Pruned Tuple search does not guarantee a good worst case search time for arbitrary databases. Thus we spent a significant portion of this paper investigating whether techniques based on precomputation and markers could improve the worst case search time of tuple space search at the cost of increased update times. As shown in [14], for the IP address lookup, which can be thought of as the one-dimensional packet classification problem, markers and precomputation improve the worst case search cost from W to $\log W$. Our paper shows that similar techniques improve the search cost from W^2 to $2W-1$ in the two-dimensional packet classification. Our lower bounds demonstrate that for K-dimensional filters, where $K > 2$, the search time is $O(W^{K-1})$. Thus there is a point of diminishing returns beyond two dimensions where the use of markers and precomputation does not improve worst-case search times significantly.

While the lower bounds preclude the ability to find an algorithm based on tuple search that has a fast search time on all databases, it does not preclude algorithms that work well on specific databases. Thus to complete the investigation, we also examined a Balancing Heuristic for generalized tuple search based on markers and precomputation. Our experimental results for the Balancing Heuristic were not encouraging, and the Balancing Heuristic does not seem to scale as well to large databases as does the Tuple Pruning Heuristic. However, the Balancing Heuristic may outperform Tuple Pruning search if the tuple space becomes sufficiently dense to allow more tuples to be eliminated by precomputation and markers. Another alternative for a dense tuple space is to break up the space into multiple rectangles, and perform Rectangle search.

While our paper has emphasized software implementation, we note that tuple pruning search has a simple parallel implementation where each tuple can be probed in parallel. Thus, in conclusion, we believe that tuple pruning search is simple and scalable, has fast update times and has a simple parallel implementation. Finally, rectangle search provides an optimal algorithm for two-dimensional filters.

Acknowledgement

We thank Jonathan Turner for an observation that led us to the tuple pruning algorithm. Marcel Waldvogel independently invented a specialized form of tuple search, called *line search*. We also thank Paul Vixie for providing us with firewall databases.

References

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

[2] J. Boyle. Internet Draft: *RSVP Extensions for CIDR aggregated data flows*. Internic, 1997.

[3] W. Cheswick and S. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1995.

[4] D.B. Chapman and E.D. Zwicky. *Building Internet Firewalls*. O-Reilly & Associates, Inc., 1995.

[5] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *Proc. of ACM Sigcomm*, 1998.

[6] D. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. *Proc. of ACM Sigcomm*, 1996.

[7] Merit Inc. *IPMA Statistics*. http://nic.merit.edu/ipma.

[8] T. V. Lakshman and D. Stidialis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *Proc. of ACM Sigcomm*, 1998.

[9] G. Malan and F. Jahanian. An Extensible Probe Architecture for Network Protocol Measurement. *Proc. of ACM Sigcomm*, 1998.

[10] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. *USENIX Technical Conference Proceedings*, 1993.

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

[12] C. Partridge. Locality and Route Caches. *NSF Workshop on Internet Statistics Measurement and Analysis*, 1996.

[13] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. *Proc. of Sigcomm*, 1998.

[14] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. *Proc. of Sigcomm*, 1997.