Run-Based Trie Involving the Structure of Arbitrary Bitmask Rules

SUMMARY Packet classification is a fundamental task in the control of network traffic, protection from cyber threats. Most layer 3 and higher network devices have a packet classification capability that determines whether to permit or discard incoming packets by comparing their headers with a set of rules. Although linear search is an intuitive implementation of packet classification, it is very inefficient. Srinivasan et al. proposed a novel lookup scheme using a hierarchical trie instead of linear search, which realizes faster packet classification with time complexity proportional to rule length rather than the number of rules. However, the hierarchical trie and its various improved algorithms allow only single prefix rules to be processed. Since it is necessary for layer 4 and higher packet classifications to deal with arbitrary bitmask rules in the hierarchical trie, we propose a run-based trie based on the hierarchical trie, but extended to deal with arbitrary bitmask rules. Our proposed algorithm achieves $O(dW^2)$ query time and $O(N^{dW})$ space complexity with $N$ rules of length $dW$. The query time of our novel algorithm doesn’t depend on the number of rules. It solves the latency problem caused by increase of the rules in firewalls.

key words: packet classification, hierarchical trie, decision tree, arbitrary bitmask, theoretical analysis

1. Introduction

The number of certain cyber threats on the Internet, such as denial of service attacks, computer virus distribution, unauthorized computer hacking, and so on, is increasing every year. Packet filtering is one of the countermeasures to defend computer systems against such cyber attacks. Layer 3 and higher network devices have the capability to filter packets, thereby determining whether to permit or discard packets by comparing each header with a set of filtering rules.

Packet classification is a fundamental task in layer 3 and higher routing, filtering, and load balancing. Packet filtering, as dealt with in this paper, is an application of packet classification. The theoretical approach to the packet classification problem focuses on either (1) optimization of the set of rules, or (2) development of a fast algorithm to find the highest priority rule corresponding to a given packet. As it is well-known that the optimal rule ordering problem is NP-complete, many researchers are currently trying to design packet filters with realistic computational complexity.

While various classification algorithms based on linear search have been studied, Srinivasan et al. established a novel lookup algorithm using a hierarchical trie, with complexity proportional to the length of the rules rather than the number of rules [10]. Unfortunately, the hierarchical trie and its improved variations only allow single prefix rules to be processed, or presuppose that bits arrays in every fields keep the form of prefix rules, which excludes arbitrary bitmask rules [1]. Most of the packet classifications dealing with arbitrary bitmask rules are now based on linear search, except for the hardware-based implementations such as ternary content addressable memories (TCAMs) [2], [5], [11]. Although a TCAM-based implementation has the great advantage of fast lookup performance, it suffers from both increased implementation cost and power consumption. It is thus desirable to develop software-based algorithms. Recently, a lookup algorithm was proposed, not based on linear search, but dealing with arbitrary bitmask rules [6]. However, there is a trade-off between time and space complexity in this algorithm and the complexities suffer from exponential explosion.

Packet classification for layer 3 routing makes use of a few header fields, including the source IP address, destination IP address, TTL, and various other fields. The longest prefix match is the basic concept in routing incoming packets to their appropriate destinations and every entry in a routing table must be a single prefix rule. Packet classification for layer 4 and higher filtering makes use of several more header fields apart from those discussed above. These include the source port number, destination port number, protocol number, and other information contained in the incoming packet. Unlike in layer 3 routing, the port number fields in a filtering table usually have a range and arbitrary bitmask rules are suitable for representing such range fields. By incorporating arbitrary bitmask rules, layer 4 and higher packet classification could be useful, for example, to manage session ids, search for virus patterns, control higher layer communication, and so on. To achieve this, it is necessary to develop fast lookup schemes to find a specific rule in a set of arbitrary bitmask rules.

In this paper, we propose a novel lookup algorithm based on the hierarchical trie proposed by Srinivasan et al., but with extensions to deal with arbitrary bitmask rules. Our proposed algorithm achieves $O(dW^2)$ query time and $O(N^{dW})$ space complexity with $N$ rules of length $dW$, on finding the highest priority rule matching an incoming packet.
2. Previous Works

Most of the previous works discussed in this section focused on layer 3 routing with a set of single prefix rules or range rules. First, we give a typical definition for the set of single prefix rules, noting that we can easily translate a range rule into several single prefix rules as needed. According to Rottenstreich and Keslassy [8], every W-bit range rule can be represented by at most W bitmask rules, using not only permit statements, but also deny statements.

We define the set of rules as an ordered set with N rules denoted by \( R = \{ r_1, r_2, \ldots, r_N \} \). Every \( r_i \in R \) consists of \( d \) fields, \( F_1, F_2, \ldots, F_d \). Let \( r_i \) be divided into \( d \) fields denoted by \( r_i[F_1], r_i[F_2], \ldots, r_i[F_d] \). We represent the fields by a maximum of W-bit strings, consisting of ‘0’, ‘1’, and ‘*’. where the notation ‘*’ is a don’t-care term or a wild-card term that matches any bit. An example of a set of rules is given in Table 1. In the table, every rule consists of two fields, each of which contains a single prefix rule.

Srinivasan et al. proposed a novel lookup algorithm using a hierarchical trie (h-trie) instead of a linear search [10]. The proposed algorithm achieves efficient matching of incoming packets against a set of rules with multiple fields, with a relatively low memory requirement.

Given a set of rules with \( d \) fields, \( F_1, F_2, \ldots, F_d \), the h-trie builds small subtries according to the d fields. The \( F_1 \) trie is constructed first by placing the bitstrings of the \( F_1 \) field on their corresponding paths of the \( F_1 \) trie. Every time a wild-card is found in the bitstring, the h-trie builds a new \( F_2 \) trie and places a pointer from the node corresponding to the last bit of the string to the root of the \( F_2 \) trie. The set of \( F_2 \) tries is constructed next by placing the bitstrings of the \( F_2 \) field in the same way as for the \( F_1 \) trie. This process is repeated until the set of \( F_d \) tries has been completely built. An example of the 2-field h-trie associated with the set of rules given in Table 1 is shown in Fig. 1. The h-trie traverses all the tries that are connected to one another via links, by backtracking across the tries to follow the bit patterns of the incoming packets. The query time complexity is \( O(W^d) \) due to backtracking with \( d \) fields of maximal length \( W \). The space complexity is \( O(NdW) \), since each rule is stored in its corresponding trie only once.

The set-pruning trie [10], [12], a modified version of the h-trie, reconstructs the set of the h-trie so as to avoid backtracking. The search process goes down a connected path from the root of the \( F_1 \) trie to a node of the \( F_d \) trie as in a single path. Thus, the query time complexity is \( O(dW) \). Subtries on the same layer are duplicated, and are traversed by repeatedly backtracking from their upper trie. Since \( N \) copies of the subtries may overlap at every layer, the space complexity grows to \( O(N^{d-1}dW) \).

The grid-of-tries structure [10] is a modified version of the set-pruning trie, which omits duplicated paths on the set-pruning trie and places pointers from the origins of the omitted paths to their source paths. This scheme performs well in terms of both query time and space complexity for 2-field classifications. However, the query time complexity approaches that of the h-trie as the number of fields used by the scheme increases.

Many geometry-based algorithms have been vigorously discussed. Most of the algorithms partition a rule-space into several regions based on the ranges of the prefix rules, given a set of rules with \( d \) fields. The search process specifies the region that encloses an incoming packet. Overmars and Stappen showed that the theoretical bounds on a point location in \( N \) hyper-rectangle regions are \( O((N^d) \) time and \( O(N^d) \) space or \( O((N^d) \) time with \( O(N) \) space [7].

The cross-producing scheme [10] is one of the heuristics and enables a fast search using a cross-product table, which is computed in a preprocessing step and includes all combinations of \( d \)-tuples, \( \{ r_1, r_2, \ldots, r_d \} \), where \( r_j \in \{ R_i[F_j] \} \). On receiving an incoming packet decomposed into \( d \) fields, this scheme compares the bit pattern in each field with that in each tuple in the cross-product table. Both the length of each field and that of each tuple have a maximal value \( W \). Thus, the query time complexity is \( O(dW) \). While this scheme achieves a fast query time complexity, the space complexity grows exponentially, reaching \( O(N^d) \).

Hierarchical Intelligent Cuttings (HiCuts) [3] and Hyper Cuttings (HyperCuts) [9] are also heuristics using decision trees to enable a fast search as in the cross-producing scheme. HiCuts maps the set of range rules with \( d \) fields onto a \( d \)-dimensional search space and recursively partitions the search space into several regions, the number of which gives an optimal threshold for each dimension. HiCuts builds a decision tree of height at most \( d \), which reflects the structure of the partition mentioned above and every external node contains a bounded number of range rules.

| Filter | \( F_1 \) | \( F_2 \) |
|--------|--------|--------|
| \( R_1 \) | \( \ast \) | \( 1 \ast \) |
| \( R_2 \) | \( \ast \) | \( 1 \ast \) |
| \( R_3 \) | \( 1 \ast \) | \( \ast \) |
| \( R_4 \) | \( \ast \) | \( 01 \ast \) |
| \( R_5 \) | \( \ast \) | \( 10 \ast \) |
| \( R_6 \) | \( \ast \) | \( 1 \ast \) |
| \( R_7 \) | \( \ast \) | \( \ast \) |

![Fig. 1](image-url) Hierarchical trie.
On receiving an incoming packet, the search process determines the region in which the incoming packet is located and simultaneously goes down the decision tree to an external node. At the external node, HiCuts performs a linear search to find the highest priority rule. HyperCuts restricts the height of the decision tree by selecting multiple cuts in the 1-dimensional space instead of hyperplanes as in HiCuts. HyperCuts restricts the height of the decision tree by selecting multiple cuts in the 1-dimensional space that partition the rule set into lists of bounded size and reduces the actual search time in real-time packet classifications. HiCuts and HyperCuts both realize theoretical bounds of $O(d)$ query time and $O(N^d)$ space complexity, due to the trade-off between time and space proved by Overmars and Stappen [7].

Recently, Ligatti et al. proposed an interesting algorithm that deals with arbitrary bitmask rules [6]. The basic idea, which they called grouper, is to divide every bitmask rule into approximately $t$ groups and then to generate classification tables according to the group. Grouper expands all wild-cards in each part of the divided rules and gives the relationship between the expanded bit patterns and the $N$-dimensional rule-vectors. The rule-vector corresponding to an entry belonging to one of the $t$ groups identifies the rules that include the same bit pattern as the entry. The search process then only has to perform a bitwise AND on the $t$ rule-vectors corresponding to an incoming packet and a linear search to find the highest priority rule. Thus, the query time complexity is $O(tN)$. The number of rule-vectors in each group is a maximum of $2^{dW/j}$. The space complexity is $O(2^{dW/j} \cdot tN)$.

We show in Table 2 a summary of the packet classification schemes described above, giving the worst case order for query time and space requirement. The heuristic-based schemes achieve acceptable performance in terms of query time, but require relatively large memory. It seems, generally, that packet classification schemes achieve good performance on either query time or space requirement at the expense of the other.

### 3. Run-Based Trie

The proposed data structure is called a run-based trie and is ready to deal with arbitrary bitmask rules. We assume that each bitmask rule is a unified rule in which all the fields have been concatenated. Since each bitmask rule consists of $d$ fields with fixed length $W$, the length of the unified rule is $dW$ and we can compare the run-based trie with other previously proposed schemes in terms of both query time and space complexity. The wild-card term ‘*’ is counted as a single word. An example of a set of arbitrary bitmask rules is given in Table 3.

Although most of the packet classification schemes map the set of rules with multiple fields onto some search space according to the characteristics of the fields, our approach builds subtrees based on runs instead of fields. In this paper, we define a run as a bitstring with maximal length and not containing any wild-cards.

**Definition 3.1:** Let $R = r_1r_2 \ldots r_{dW}$, $r_i \in \{0, 1, *, \}$, be a bitmask rule of length $dW$. We define a run as a substring $s_is_{i+1} \ldots s_{i+k-1}$ ($1 \leq k \leq dW$) on $R$ that satisfies the following conditions:

1. $s_j \in \{0, 1\}$ for $j = i, i + 1, \ldots, i + k - 1$,
2. if $i > 1$ then $s_{i-1} = ‘*‘$,
3. if $i + k - 1 < dW$ then $s_{i+k} = ‘*‘$.

For example, the bitmask rule $1011*001*1**0110$ consists of 4 runs, 101, 001, 1, and 01. Note that these runs begin at the second, the 7th, the 11th, and the 14th bits in the rule, respectively.

#### 3.1 Construction of Run-Based Trie

Given a set of rules with $d$ fields, $R = (R_1, R_2, \ldots, R_N)$, the run-based trie is constructed as follows. The basic operation is that the $T_i$ trie is constructed by placing the bit pattern of the run that begins at the $i$-th bit of $R \in R$ on its corresponding path of $T_i$. In addition, we mark $R^j$ on the path if the run is the $j$-th run of $R$, or underlined $R^j$ if the run is the last run of $R$.

The run-based trie completes the construction of at most $dW$ tries by repeating the basic operation explained above for every run of each rule in $R$. The height of $T_i$ is of course less than or equal to $dW - i + 1$. Illustrated in Fig. 2 is the run-based trie based on the rule set given in Table 3. In the figure, as mentioned above, the underlined run $R^j$ on $T_i$ means that there is no successive run derived from $R$ on the other tries below $T_i$.

The run-based trie scheme constructs at most $dW$ tries from $T_i$ to $T_{jW}$, but every rule is stored in its corresponding path only once, similar to the hierarchical trie scheme. Thus, the space complexity remains $O(NdW)$.

#### 3.2 Simple Search

First we describe an intuitive and straightforward implemen-
The simple search prepares an \( N \)-dimensional array \( A \) for storing the indices of runs and a variable \( B \) for storing the highest priority rule. For example, the index \( j \) of \( R_i \) is stored in \( A[j] \). The search process traverses the run-based trie in order from \( T_1 \) to \( T_{dW} \), so that \( T_i \) is processed from its root towards an external node following the bit pattern of an incoming packet from the \( i \)-th bit to the last one, and stores the indices of runs in \( A \) each time it obtains an index from a trie. Every index of \( R_i \) is stored in \( A[j] \) in order from 1 to \( m \) through the traversal, if and only if an incoming packet matches \( R_i \), consisting of \( m \) runs. Having just found an underlined run, the search process compares the priority of the run with that of the non-empty \( B \), and then stores the higher priority rule in \( B \). On completion of the simple search, the highest priority rule will be stored in \( B \). Note that, as mentioned above, in the case that the indices of runs are not stored in order, the simple search excludes those rules as candidates for the highest priority rule.

It is very important for the simple search to sum the complexity of storing the indices of runs in addition to that of traversing the run-based trie. The latter complexity is obviously \( O((dW)^2) \). Bitmask rules of length \( dW \) can contain up to \( \lceil dW/2 \rceil \) runs. Thus, the simple search has the potential to store a maximum of \( N \cdot \lceil dW/2 \rceil \) indices on completion. Storing one index takes constant time, and thus the total time complexity to determine the highest priority rule using the simple search is \( O((dW)^2 + NdW) \).

### 3.3 Decision Tree Search

Here, we discuss a faster classification scheme using a decision tree. The decision tree is constructed as follows. We first generate \( dW \) families of sets, \( S_1, S_2, \ldots, S_{dW} \), from the run-based trie. The elements of \( S_i \) consist of all possible combinations of runs obtained by traversing \( T_i \) from the root to each node. We denote by

\[
S_i = \{ S_i^1, S_i^2, \ldots, S_i^m \}
\]

the family \( S_i \), which contains \( m \) elements. \( S_i \) also contains a special element \( \phi \) for the empty path, since the search process might trace an empty path that does not contain any runs at all, depending on the bit pattern of the given incoming packet. For example, the following four families are obtained from the run-based trie shown in Fig. 2.

\[
S_1 = \{ \{ R_1^3, R_2^3, R_3^3 \}, \{ R_4^3, R_5^3, R_6^3 \}, \{ R_7^3 \}, \phi \},
\]

\[
S_2 = \{ \{ R_1^2 \}, \{ R_1^3, R_2^3, R_3^3 \}, \{ R_4^3 \}, \{ R_5^3, R_6^3, R_7^3 \}, \phi \},
\]

\[
S_3 = \{ \{ R_1^2 \}, \{ R_2^2, R_3^2 \}, \{ R_4^2 \}, \phi \},
\]

\[
S_4 = \{ \{ R_2^2, R_4^2, R_5^2 \}, \phi \}.
\]

The decision tree reflects the structure of the Cartesian product of the non empty families, \( |S_1| \times |S_2| \times \cdots \times |S_{dW}| \). Each path of the decision tree is equivalent to a search path obtained by traversing the run-based trie in order from \( T_1 \) to \( T_{dW} \). We can systematically compute the highest priority rule on every path of the decision tree in advance. We show in Fig. 3 the decision tree obtained from the run-based trie illustrated in Fig. 2. In the figure, we omit a number of nodes because the decision tree becomes very large, eventually comprising \( 5 \times 6 \times 4 \times 2 = 240 \) external nodes! The highest priority rule on each path is underlined.
The maximum cardinality of $S_i$ is equal to the number of paths from the root of $T_i$ to the nodes. Thus, we have

$$|S_1| + |S_2| + \cdots + |S_{dW}| \leq NdW. \quad (6)$$

From the inequality of the arithmetic and geometric means, we also have

$$\sqrt[4]{|S_1| \times |S_2| \times \cdots \times |S_{dW}|} \leq \frac{|S_1| + |S_2| + \cdots + |S_{dW}|}{dW}, \quad (7)$$

and finally,

$$|S_1| \times |S_2| \times \cdots \times |S_{dW}| \leq N^{dW}. \quad (8)$$

Thus, the space complexity of the decision tree is $O(N^{dW})$. Generation of the families $S_1, S_2, \ldots, S_{dW}$ has the same complexity as a preorder traversal of the run-based trie. The computational complexity to construct the decision tree is $O(dW^2 + N^{dW})$.

The search process alternates between the run-based trie traversal and the decision tree traversal. In the run-based trie traversal, all incoming packets do not reach the nodes of the run-based trie on which some runs are marked. In most cases, they go out of the nodes on which no runs are marked. Even if an incoming packet goes out of a node $v$ on which no runs are marked, each of the marked runs lying on the path from the root to $v$ could be a candidate rule matching the packet. The elements of $S_i$ indicate the set of candidate rules on $T_i$. For example, in Fig. 2, there are 5 nodes on the leftmost path of $T_1$, which are associated with the elements of $S_1$, specifically $\phi$, $[R_3^1, R_4^1]$, $[R_3^1, R_4^1]$, $[R_3^1, R_4^1]$, and $[R_3^1, R_4^1]$. When receiving a 4-bit packet $0010$, the run-based trie goes out of the third node on the leftmost path of $T_1$, and obtains the candidate rule set $\{R_3^1, R_4^1\}$ at the node.

The decision tree traversal is performed in the following way. After the $T_i$ trie traversal, we can obtain the candidate rule set $x \in S_1$ marked on the $T_i$ trie. The search process specifies one node at level $i$ of the decision tree corresponding to $x$. The next step is similar in that the search process specifies one node at level $i + 1$ of the decision tree by the $T_{i+1}$ traversal in the same way. Thus, a unique path is determined in the decision tree through the run-based trie traversal in order from $T_1$ to $T_{dW}$, and the highest priority rule, which will be on the external node of the determined path depending on the incoming packet, is obtained.

An example of the search process in the run-based trie and the decision tree is illustrated in Figs. 2 and 3, respectively. When receiving a 4-bit packet $0010$, the search process traverses 4 subtrees by following the bit pattern: $T_1$ with $0010$, $T_2$ with $010$, $T_3$ with $10$, and $T_4$ with $0$. After the traversal, the search process obtains 4 candidate rule sets, $S_1^0 = \{R_3^1, R_4^1\}$, $S_2^1 = \{R_3^1, R_4^1\}$, $S_3^2 = \{R_3^1, R_4^1\}$, and $\phi$. The search process specifies the unique path, $S_1^0 - S_2^1 - S_3^2 - \phi$ in the decision tree, and finally, obtains the highest priority rule $R_3^1$.

The total time complexity to determine the highest priority rule using a decision tree search is the sum of the complexities of traversing both the decision tree and the run-based trie. It is obvious that the latter complexity includes the former complexity. Thus, the total query time complexity is only $O((dW)^2)$.

4. Experimental Results

To confirm the efficiency of our method, we implemented the rule search program based on run-based trie in C language under the Cent OS Release 6.2 on intel Core i7-980X 3.33 GHz with 24GB main memory.

4.1 The Memory Size and Time Required for Building and Search

In every cases from 3 bits to 17 bits rule length, a set of 1000 rules was generalized randomly. We measured the memory sizes and times required for constructing run-based tries and decision trees which correspond to every rule sets. The memory sizes required for construction are shown in the Figs. 4 and 5. The horizontal axis represents the bit length of rules and the vertical axis represents the size of memory. The times for construction are shown in Figs. 6 and 7. The
horizontal axis represents the numbers of rules and the vertical axis represents the times. In the case of run-based tries, the memory sizes and constructing times are both linearly limited. The decision trees requires both exponential memory size and constructing times.

4.2 Decision Tree Search

For the sets of randomly generalized 8 bits rules from 1 to 40 bits, we constructed run-based tries and decision trees. We measured the time required for searching appropriate rules for 1000 packets. Figure 8 shows the average time of 10 times trials. Although the time in run-based tries linearly increase with respect to the number of rules \( N \), the time of decision trees don’t depend on \( N \), which represents the efficiency of our method.

In every lengths rules of of 4, 8, 12 bits, we constructed the run-based tries and decision tress correspond to the set of 1 to 25 rules. We measured the rules search time for 1000 randomly generalized packets. Figure 9 shows the average search times of 10 times trials in run-based tries. Figure 10 shows the average search times in decision trees. The time of run-based trie is limited within \( O(NdW) \) and that of decision trees doesn’t depend on the number of rules as expected. Since the query time of our decision trees doesn’t depend on the number of rules, it solves the latency problem caused by increase of the rules in firewalls.

5. Conclusion

In this paper, we proposed novel packet classification methods based on “run-based trie”. Simple search achieves worst case query time complexity of \( O((dW)^2 + NdW) \). Decision tree search achieves a worst case query time complexity of \( O((dW)^2) \) with space complexity of \( O(NdW) \). The proposed algorithms have several advantages in that it is easy to im-
plement in software because the basic structure of the search process is very simple. Our algorithms can be used for single prefix rules without any modification. The latter yields query time complexity of $O(dW)$ and space complexity of $O(N^d)$, which is the same as the cross-producting scheme.

The decision tree in our algorithm can grow into a very large tree. To reduce its size, it is necessary to prune unnecessary paths or merge some of the redundant paths in the decision tree. Solving the tight bound on the space complexity remains an issue for future work. The empirical efficacy of our methods are to be demonstrated experimentally in compared with conventional method which can treat arbitrary bitmask rules [6].

Acknowledgments

This work was partially supported by the Japan Society for the Promotion of Science (JSPS), Grant-in-Aid for Scientific Research (C) (23500099).

References

[1] H.J. Chen, “Next generation routers,” Proc. IEEE, vol.90, no.9, pp.1518–1558, Sept. 2002.
[2] V. Grout, J. McGinn, and J. Davies, “Real-time optimisation of access control lists for efficient Internet packet filtering,” J. Heuristics, vol.13, no.5, pp.435–454, Oct. 2007.
[3] P. Gupta and N. McKeown, “Classifying packets with hierarchical intelligent cuttings,” IEEE Micro, vol.20, no.1, pp.34–41, Jan./Feb. 2000.
[4] P. Gupta and N. McKeown, “Packet classification on multiple fields,” Proc. ACM SIGCOMM, pp.147–160, 1999.
[5] H. Hamed and E. Al-Shaer, “Dynamic rule-ordering optimization for high-speed firewall filtering,” Proc. ACM Symposium on Information, Computer and Communications Security, pp.332–342, 2006.
[6] J. Ligatti, J. Kuhn, and C. Gage, “A packet-classification algorithm for arbitrary bitmask rules, with automatic time-space tradeoffs,” Proc. 19th IEEE Int. Conf. on Computer Communication Networks, Zurich, Switzerland, 2010.
[7] M.H. Overmars and A.F. van der Stappen, “Range searching and point location among fat objects,” J. Algorithms, vol.21, no.3, pp.629–656, Nov. 1996.
[8] O. Rottenstreich and I. Keslassy, “Worst-case TCAM rule expansion,” Proc. IEEE INFOCOM, pp.456–460, 2010.
[9] S. Singh, F. Baboescu, G. Varghese, and J. Wang, “Packet classification using multidimensional cutting,” Proc. ACM SIGCOMM, pp.213–224, 2003.
[10] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, “Fast and scalable layer four switching,” Proc. ACM SIGCOMM, pp.191–202, 1998.
[11] K. Tanaka and S. Ito, “Reconstruction method of filtering rules in order to mitigate load of network gears,” IEICE Trans. Commun. (Japanese Edition), vol.J88-B, no.5, pp.905–912, May 2005.
[12] P.F. Tsuchiya, “A search algorithm for table entries with non-contiguous wildcarding,” unpublished report, Bellcore.