A Multi-Dimensional Progressive Perfect Hashing for High-Speed String Matching

Yang Xu, Lei Ma, Zhaobo Liu, H. Jonathan Chao
ECE Department, Polytechnic Institute of NYU
Brooklyn, NY 11201
yangxu@poly.edu, {lma01, zliu01}@students.poly.edu, chao@poly.edu

ABSTRACT
Aho-Corasick (AC) automaton is widely used for multi-string matching in today's Network Intrusion Detection System (NIDS). With fast-growing rule sets, implementing AC automaton with a small memory without sacrificing its performance has remained challenging in NIDS design. In this paper, we propose a multi-dimensional progressive perfect hashing algorithm named P^2-Hashing, which allows transitions of an AC automaton to be placed in a compact hash table without any collision. P^2-Hashing is based on the observation that a hash key of each transition consists of two dimensions, namely a source state ID and an input character. When placing a transition in a hash table and causing a collision, we can change the value of a dimension of the hash key to rehash the transition to a new location of the hash table. For a given AC automaton, P^2-Hashing first divides all the transitions into many small sets based on the two-dimensional values of the hash keys, and then places the sets of transitions progressively into the hash table until all are placed. Hash collisions that occurred during the insertion of a transition will only affect the transitions in the same set. The proposed P^2-Hashing has many unique properties, including fast hash index generation and zero memory overhead, which are very suitable for the AC automaton operation. The feasibility and performance of P^2-Hashing are investigated through simulations on the full Snort (6.4k rules) and ClamAV (54k rules) rule sets, each of which is first converted to a single AC automaton. Simulation results show that P^2-Hashing can successfully construct the perfect hash table even when the load factor of the hash table is as high as 0.91.

Categories and Subject Descriptors
C.2.0 [**Computer-Communication Networks**]: General Security and protection (e.g., firewalls); C.2.6 [**Internetworking**]: Routers

General Terms
Algorithms, Design, Security

Keywords
Aho-Corasick Automaton; Perfect Hash Table; Hash Collision; Multi-string Matching

1. INTRODUCTION
Network Intrusion Detection System (NIDS) has been widely deployed in today's Internet to safeguard the security of network operations. Among the many network-based intrusion detection techniques [2][3][10][11], multi-string matching is commonly used because of its precision and accuracy in attack detection.

Many multi-string matching schemes have been proposed in the past [4][5][6][7][8][9], most of which derive from the classic Aho-Corasick (AC) automaton [1] as its worst case performance is deterministic, linear to the length of the input stream and independent of the rule set size. Therefore it is impossible for an attacker to construct a worst-case traffic that can slow down the NIDS and let malicious traffic escape the inspection. In fact, many popular NIDS and anti-virus systems, such as Snort [12] and ClamAV [13], already implemented AC automaton as their multi-string matching engines. With fast-growing rule sets, implementing AC automaton with a small memory without sacrificing performance becomes a major challenge in NIDS design.

There are many schemes that could be used to efficiently implement dense automatons[1]. We can use a two-dimensional direct-indexed table to store all the transitions, where each row corresponds to a state, and each column corresponds to a symbol. The intersection between each row and each column stores the row ID of the next hop state. In order to reduce memory cost, HEXA [24] was proposed to reduce the number of bits stored in each field of the two-dimensional table using the historical scanning information carried by the input stream. Although a two-dimensional table works fine for the dense automaton, it is not a good solution to implement the sparse automaton (such as AC automaton, which has the transition-to-state ratio normally between 1 and 2), because of the memory waste by the non-existing transitions. Besides the two-dimensional table, another way of implementing the automaton is to store each state as a whole data structure, and connect parent and child states by pointers in the parent states. However, the wide distribution of state sizes (i.e., the numbers of transitions of states) on the AC automaton makes the design of a compact state structure a very difficult task.

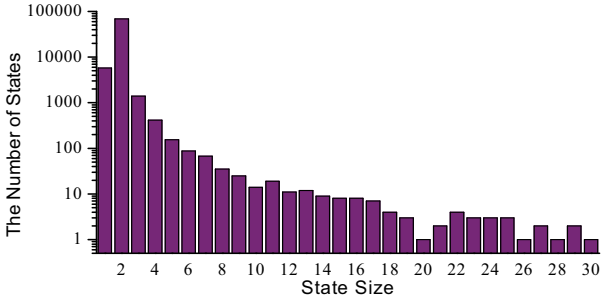

Figure 1. The state size distribution of AC automaton based on the Snort rule set (only shows states with size <= 30)

[1] An automaton is called dense automaton if the ratio of its total transition number to its total state number is close to 256.

978-0-7695-4521-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ANCS.2011.33

Figure 1 shows the distribution of state sizes on the AC automaton based on the Snort rule set. We can see that the distribution is quite wide and unbalanced, and it is very hard to design a compact state structure storing pointers pointing to the child states.

Hash table is a good solution to implement the sparse automaton such as AC automaton, because we no longer need to store the non-existing transitions or keep the complicated state structure. Compared to other AC automaton implementation schemes, such as bitmap-compression AC and path-compression AC [9], storing transitions directly in a hash table can avoid unnecessary memory waste, and simplify the process of making a transition decision. The main problem involved in hash table design is the hash collision, which might increase the memory access times for each transition decision and cause instability of the processing speed. Furthermore, hash collision might be exploited by attackers to degrade system performance. In [8], Lunteren proposes a BFSM-based pattern-matching (BFPM), which uses a hash table construction scheme named Balanced Routing Table (BART) [14] to limit the maximum number of collisions of any hash index by a configurable bound P (P=4 is used in [8]). When a transition decision is made, P transitions are read out from the same entry of the hash table simultaneously. After P parallel comparisons, the correct transition can be decided. Storing multiple transitions in each entry, however, increases the memory bus width and causes unnecessary waste of memory space. Furthermore, P comparisons required for each transition decrease the scheme's efficiency in software implementation.

Therefore, an efficient perfect hashing scheme for AC automaton is desirable in high-performance NIDS design. Although there are many perfect hashing and alternative algorithms available in literature, most of them require multiple memory accesses to generate the hash index (traversing a tree structure) [15][25], or need more than one memory access in the worst case to get the correct hash index for a hash table lookup [16][17][18]. Due to the dependency between two contiguous transitions made on the automaton, one hash query can start only after the previous hash query returns a new current state ID (without the new current state information, the next transition cannot be made). In other words, hash queries have to be performed in serial. The time required to perform one hash query is equal to the sum of the time generating the hash index and the time accessing the hash table. If the hash unit takes too much time generating the hash index or accessing the hash table, the matching speed of the system will be degraded.

Our main contributions in this paper are summarized as follows.

1. We propose a multi-dimensional progressive perfect hashing algorithm, named P^2-Hashing, that allows transitions of an AC automaton to be placed in a compact hash table without any collision. P^2-Hashing supports both un-optimized and optimized AC automatons.
2. Different from many existing perfect hashing schemes which require additional storage for their own representations, P^2-Hashing requires no storage overhead to implement the perfect hashing function (except for the small fixed 256-entry character translation table). This is achieved by embedding information directly into the AC automaton structure.
3. P^2-Hashing requires no memory access to generate the hash index (a character translation table needs to be accessed one time slot before the generation of the hash index, but it is not on the critical path of the AC automaton operation and therefore can be implemented by a separate pipeline stage). This property is important to AC automaton operation because only one hash query can be performed on the fly due to dependency between two contiguous transitions made on the automaton. A fast hash index generation can speed up the automaton operation.
4. A unified perfect hashing solution has been presented to implement all search tables in the AC automaton implementation, which include transition table and rule ID table. An important advantage of this unified perfect hashing solution is that it avoids pointers which are normally required to connect different tables, so that the memory cost is minimized.

The rest of the paper is organized as follows. Section 2 provides the background of the paper, and reviews the related work. Section 3 gives the problem statement and terms to be used. Section 4 proposes two perfect hash table construction algorithms, including P^2-Hashing and its two-dimensional version. In Section 5, we give the system design of the proposed multi-string matching engine, based on which, in Section 6 we introduce how to construct perfect hash table for rules. Section 7 presents the simulation results. Section 8 discusses an incremental update scheme on the proposed perfect hash table. Finally, Section 9 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 Aho-Corasick Automaton

Aho-Corasick (AC) automaton [1] is one of the most used algorithms in multi-string matching. It is well-known for its deterministic matching throughput, and therefore is not vulnerable to attack traffic. Given a set of string patterns (also called rules in this paper), the construction of an AC automaton consists of two steps. In the first step, a trie structure is created based on the strings. Each state/node on the trie corresponds to a valid prefix of strings. The edges on the trie are called the *goto transitions* of the AC automaton. In the second step, *failure transitions* are added from each state s to a state d if the prefix represented by state d is the longest suffix of the prefix represented by state s. Consider a set of string patterns {hers, he, his, him, me, she}. Its AC automaton is shown in Figure 2, in which the solid arrows represent the goto transitions, while the dotted arrows represent the failure transitions.

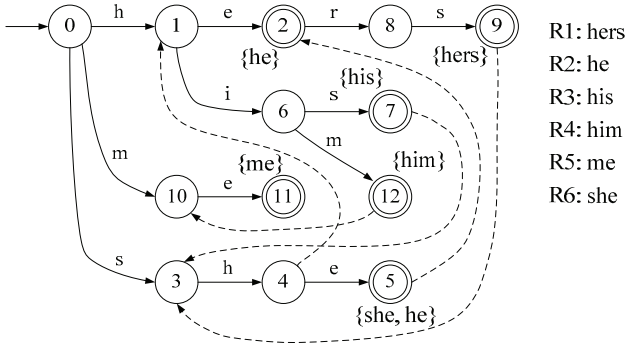

Figure 2. AC automaton for rule set {hers, he, his, him, me, she}. For simplicity, failure transitions to the root state are not shown.

Given a active state s and an input character c, the AC automaton will first check if there is a goto transition from state s labeled with c. If such a goto transition exists, the state pointed by the goto transition will be the active state in the next time slot; otherwise, the active state in the next time slot will be the state pointed by the failure transition of state s and then input character c will be examined again in the next time slot.

Actually, AC automaton has two versions. The one we just introduced is the un-optimized version. The advantage of the un-optimized version is that an AC automaton with N states has only $N-1$ goto transitions and $N-1$ failure transitions; therefore, the storage complexity of transitions is relatively low. For an input stream with length L, the number of state transitions to be made during matching in the worst cast is $2L$.

The optimized version of AC automaton is actually a Deterministic Finite Automaton (DFA). Based on the un-optimized version, the optimized version of AC automaton could be constructed by adding goto transitions for every character from every state and removing the failure transitions. Compared to the un-optimized version, the optimized version only needs to make one state transition for each input character. Therefore its worst-case throughput is twice the throughput of the un-optimized version. The main disadvantage of the optimized version is its huge memory cost, since each state has 256 goto transitions corresponding to 256 characters.

In the remainder of this paper, unless specifically noted, we use AC automaton to denote its un-optimized version, and use AC-DFA to denote the optimized version. For the sake of simplicity, we use the word "transition" to refer the goto transition if there is no confusion in the context.

2.2 Memory Optimization of Aho-Corasick Automaton

Many solutions aiming to reduce the memory cost of AC automaton and AC-DFA have been proposed in literature [8][9][19][20]. Tuck et al. [9] apply bitmap compression and path compression on AC automaton to save the memory cost of non-existing transitions. Tan et al. [20] propose an approach to bit-split the AC-DFA into several small AC-DFAs to reduce the total memory requirement. Song et al. [19] and Lunteren [8] observed that a large fraction of transitions on AC-DFA are backward to states at the first three levels (the root state is at level 1). Based on this observation, Lunteren proposes to remove transitions backward to the first two levels by storing them in a separate 256-entry table; while Song [19] proposes a Cached Deterministic Finite Automate (CDFA) model, based on which backward transitions to states at level 3 can also be removed. The main idea of CDFA is to maintain more than one active state in AC-DFA (one at the root state, one at states at level 2, and one at states at other levels). It has been shown that after the elimination of backward transitions to states at the first three levels, the number of transitions of AC-DFA is approximately equal to the number of transitions of AC automaton. Furthermore, it is observed that the total number of transitions could be significantly reduced if the rule set is partitioned into multiple subsets, and implemented by multiple small AC-DFAs [8][19].

Besides the memory optimization, much research work focuses on accelerating the processing speed of AC automaton/AC-DFA [5][6][7].

2.3 Other Multi-string Matching Schemes

Yu et al. [26] proposed a gigabit rate multi-string matching scheme based on TCAM. Piyachon and Luo [27] proposed a sophisticated memory model for multi-string matching implementation based on Network Processors (NPs).

In addition, there are also many FPGA-based schemes proposed for multi-string matching [21][22][23], which map the rule set directly to the pure logic of FPGA, and can achieve desirable high performance. A main limitation of FPGA-based schemes is that when rules are changed, it takes considerable time to re-synthesize the design and reprogram the FPGA.

3. PROBLEM STATEMENT

In this section, we define the terms to be used in this paper, and give the problem statement.

An AC automaton is formally defined as a 5-tuple $= (Q, \sum, g, f, T)$, which consists of

- A finite set of states, Q, where each state is represented by a number ranging from 0 to $|Q|-1$, among which 0 is the start (root) state;
- A finite input character set, $\sum$, called alphabet;
- A set of accepting states, $T \subseteq Q$;
- A goto transition function that, $g: Q \times \sum \rightarrow Q \cup \{fail\}$;
- A failure function that, $f: Q - \{0\} \rightarrow Q$.

A hash table is a 3-tuple $H = \{K, h, S\}$, consisting of

- A set of keys, K, where each key is used as the input of the hash function to obtain the index of the hash table;
- A table S, which has at least $|K|$ entries, i.e., $|S| \geq |K|$;
- A hash function that, $h: K \rightarrow N$, where N is the set of natural numbers from 0 to $|S|-1$; the hash function is called a perfect hash function if for $\forall a, b \in K$ and $a \neq b$, we have $h(a) \neq h(b)$. In this paper, we call $h(a)$ the hash index of key a.

A hash table is called a perfect hash table if the hash function associated with the hash table is a perfect hash function. The load factor of a hash table is defined as $\rho = |K|/|S|$, which describes how full the hash table is. Normally, a larger ρ implies a high probability of hash collisions.

We assume that the hash function used in our hash table construction is randomly selected from a universal hash function family and is uniform hashing, i.e., each hash key is equally likely to hash into any of the $|S|$ entries of the hash table, independently of where any other key has hashed to.

Our main objective in this paper is to store all the transitions of AC automaton in a perfect hash table. Each transition on the AC automaton takes one entry of the hash table in the form of "(source state ID, input character) ⟶ destined state ID", where "(source state ID, input character)" stands for the concatenation of "source state ID" and "input character" in binary mode, and works as the key of the hash function, while "destined state ID" is the result we hope to get from the hash table access.

Besides the AC automaton, we hope the proposed perfect hashing algorithm could also be used for AC-DFA. Considering the huge memory cost of AC-DFA, we use the scheme proposed in [19] to eliminate the backward transitions to states at the first several levels, and store only the remaining transitions in the perfect hash table. Actually, AC-DFA could be viewed as a special case of AC automaton, i.e., AC automaton without failure transition.

Therefore, we present the perfect hash table construction algorithm based on the general case: AC automaton.

4. PERFECT HASH TABLE CONSTRUCTION ALGORITHM

4.1 Progressive Perfect Hashing Algorithm

We first present two important observations that will be used in our perfect hash table construction algorithm.

(1) The first is about the hash function. If a hash collision occurs when we try to place a new key into the hash table, the collision might be avoided if we could change the value of the key. This observation is based on the fact that the hash index of a key depends only on the hash function and the value of the key. If the value of the key is changed, the hash index is also changed and, accordingly, the original hash collision may be avoided.

(2) The second observation is that the ID of each state of AC automaton could be named as any values, as long as there are no two states being named with the value.

Based on these two observations, we designed a scheme called Progressive Perfect Hash (P^2-Hashing) to place the transitions of AC automaton in a hash table without collision. To better illustrate our scheme, in this section, we only consider storing goto transitions. In Section 6, we discuss the situation of storing goto transitions, failure transitions, and rule IDs.

The main idea of P^2-Hashing is to divide the goto transitions of a given AC automaton into multiple independent sets according to their source states, and place these transition sets in the hash table in decreasing order of their sizes. The transitions of each set are placed into the hash table as a whole. Any hash collision occurring during the placement of a set causes the set placement failure, and the already-placed transitions in this set are removed from the hash table. Then the source state shared by transitions in this set is renamed, and another set placement trial is performed. The renaming operation repeats until a successful set placement is achieved, and then the placement of the next transition set starts.

Consider the AC automaton shown in Figure 2, which has 12 transitions. The transition sets associated with source states are shown in Table I. Suppose we want to store these transitions into a perfect hash table with 12 entries. With P^2-Hashing, we first place the set associated with state 0, since it has the most transitions. It is easily seen that the success probability that all three transitions in this set are placed into the hash table without collision is $\frac{12}{12} \cdot \frac{11}{12} \cdot \frac{10}{12}$. Suppose the set associated with state 4 is the last set to be placed. Its success probability is $\frac{1}{12}$.

Table I. Transition Sets of Source States

State 0	State 1	State 6	State 2	State 8	State 10	State 3	State 4
(0,h)→1	(1,e)→2	(6,s)→7	(2,r)→8	(8,s)→9	(10,e)→11	(3,h)→4	(4,e)→5
(0,m)→10	(1,i)→6	(6,m)→12					
(0,s)→3							

It should be noted that the sequence of set placements has a great impact on their success probabilities. Consider the above example again, if we place the set associated with state 0 last, the success probability of the set placement is only $\frac{3}{12} \cdot \frac{2}{12} \cdot \frac{1}{12} = 0.0034$. The reason for this low success probability is that we must place all transitions of each set into the hash table without collision simultaneously. If we place the largest set last when the hash table is almost full, the success probability would become very low. That is why P^2-Hashing places larger transition sets into the hash table first.

Formally, the success probability of a set placement is determined by the current load factor of the hash table (ρ) and the number of transitions in the set (W), and could be approximately calculated with the following inequality.

$$P(success\ of\ a\ set\ placement) \leq (1-\rho)^W \quad (1)$$

The success probability of a set placement determines the average number of state renamings required before a successful set placement. Suppose $\rho = 0.5, W = 10$, the success probability is less than $1/1024$, which means that, on average, we need to rename the state 1024 times before achieving a successful set placement. Accordingly, the number of bits to encode the state IDs is expected to be 10. If W increases to 20, we need to rename the state 1 million times on average before achieving a successful set placement, and the number of bits used to represent state IDs increases to 20.

Apparently, if both W and ρ happen to be large during the placement of a transition set, the performance of the P^2-Hashing algorithm would become very poor, not only because of the long running time of state renamings (during each state renaming, many transitions may need to be re-placed), but also the high storage cost required by the long state IDs. Fortunately, AC automatons are normally sparse automatons, especially for large rule sets. As shown in Figure 1, only a few of states have relatively large number of goto transitions (say more than 10 goto transitions), and 99% of states have only three or fewer transitions. By placing large sets first we can avoid the situation that both W and ρ are large.

4.2 Two-Dimensional P^2-Hashing Algorithm

A main limitation of the P^2-Hashing algorithm is that it cannot handle very well the situation in which a few states take the majority of the total transitions (especially for small rule sets). Consider the AC automaton shown in Figure 3, which includes 23 rules with the same length. All of these rules have the same prefix of "*abc*." Suppose we want to use P^2-Hashing to place the 26 transitions of the AC automaton into a hash table with 28 entries. According to P^2-Hashing, we first place the transition set associated with state "3", since it is the largest transition set with 23 transitions.

It is easily seen that the success probability of this set placement is $\prod_{i=0}^{22} \frac{28-i}{28} \approx 1.3 \times 10^{-6}$. That means, on average, we have to rename state "3" by 10^6 times to achieve a successful set placement, and use 20 bits to name each state. Please note that ideally, 27 states of the AC automaton only require 5 bits for unique representation.

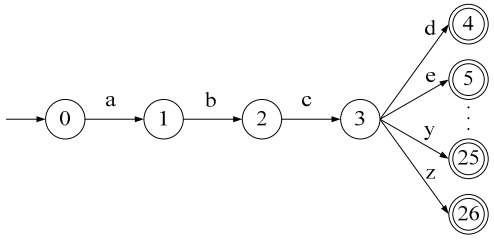

Figure 3. AC automaton for a set of rules with the same prefix. For simplicity, failure transitions are not shown (here all to the root state).

The P^2-Hashing algorithm introduced above changes the hash indexes of transitions by renaming their source states. In fact, the input key of the hash function consists of two dimensions: source state ID and input character. We can achieve the target of changing hash indexes by changing the value of either dimension. When the values of characters are changed, we only need a 256-entry character translation table to record the new encoding of each character.

Now we need to decide the sequence in which transitions should be placed into the hash table, and the dimensions of transitions that should be renamed when hash collisions occur during the placements. The main challenge involved in this process is that when a state or character is renamed, many transitions could be affected, including those already-placed transitions. How can we avoid the fluctuation of the hash table, i.e., transitions keep being placed in and removed from the hash table? Aiming to resolve this problem, we present a two-dimensional P^2-Hashing (2D P^2-Hashing for short) algorithm, which consists of three steps.

(1) In the first step, we model the AC automaton $M = (Q, \sum, g, f, T)$ as a bipartite graph, which is formally defined as a 3-tuple $B = (U, V, E)$, consisting of

- A set of nodes, U;
- A set of nodes, V;
- A set of edges, E; $\forall < u, v > \in E$ satisfies that $u \in U, v \in V$.

In this model, we set $U = Q, V = \Sigma$, and let $E = \{< q, c > | \forall q \in Q, \forall c \in \Sigma, \text{that } g(q, c) \neq fail\}$. In other words, each state in AC automaton corresponds to a node in set U, each character in AC automaton corresponds to a node in set V, and each transition in AC automaton corresponds to an edge in set E. To better illustrate the scheme, we name nodes in set *U state nodes*, and nodes in set *V character nodes*. It is easy to see that storing transitions of the AC automaton in a perfect hash table is equivalent to storing edges of the bipartite graph in the perfect hash table, where the concatenation of u and v of each edge $< u, v >$ is used as the key of the hash function.

Consider the AC automaton in Figure 2 as an example. Its bipartite graph model is shown in Figure 4, in which state node set U includes 13 nodes $\{0 \sim 12\}$, and character node set V includes 6 nodes $\{h, e, r, s, i, m\}$. Each edge in the bipartite graph represents a transition on the AC automaton.

On the bipartite graph, the number of edges connected to each node reflects the impact of the node during the perfect hash table construction. The more edges a node has, the more difficult to rename it to achieve a collision-free placement of all its connected edges.

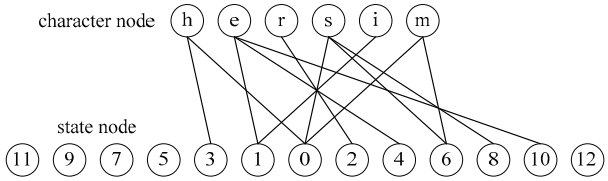

Figure 4. The bipartite graph model of the AC automaton in Figure 2.

Algorithm 1. Bipartite Graph Decomposition

Input:
Bipartite graph $B = (U, V, E)$;

Output:
A sequence number $N(v)$ for every node $v \in U \cup V$;
A dependent edge set $D(v)$ for every node $v \in U \cup V$;

Algorithm:
$N(v) := NULL\ (\forall\ v \in U \cup V)$;
$D(v) := NULL\ (\forall\ v \in U \cup V)$;
for $(j := 1; j \leq |U| + |V|; j + +)$;
{
Among all nodes in bipartite graph B, choose a node, say v, that has the least connected edges; if there are multiple qualified nodes, randomly select one;
$N(v) := j$;
$D(v)$:=the set of edges connected to node v;
Remove node v and its connected edges from the bipartite graph B;
}

(2) In the second step, we decompose edges of the bipartite graph into $|U| + |V|$ sorted edge sets (some sets could be empty), and associate each edge set with a node in $U \cup V$. We call each edge set the dependent edge set of its associated node. The bipartite graph decomposition algorithm is shown in Alg. 1.

The bipartite graph decomposition consists of $|U| + |V|$ phases, and starts with all nodes unassociated. During each phase, among all nodes in the bipartite graph, we choose a node, say v, that has the fewest connected edges (if there are multiple qualified nodes, we choose an arbitrary one). We allocate all edges connected to node v to node v's dependent edge set, and remove node v and its connected edges from the bipartite graph.

After the bipartite graph decomposition, each node is assigned a dependent edge set and a sequence number. For the bipartite graph in Figure 4, the dependent edge sets of nodes and the sequence in which they are removed from the bipartite graph are shown in TABLE II.

TABLE II. Dependent Edge Sets of Nodes After The Bipartite Graph Decomposition (based on the bipartite graph in Figure 4)

Seq	1	2	3	4	5	6	7	8	9	10
Node	11	9	7	5	12	3	h	2	r	4
Dependent edge set						<3,h>	<0,h>	<2,r>		<4,e>
Seq	11	12	13	14	15	16	17	18	19	
Node	10	e	1	i	8	0	s	6	m	
Dependent edge set	<10,e>	<1,e>	<1,i>		<8,s>	<0,s> <0,m>	<6,s>	<6,m>		

The complexity of the bipartite graph decomposition is linear to the number of edges on the bipartite graph, although we need to select a node with the fewest connected edges in each phase. This is due to the following properties of the AC automaton:

- The total number of character nodes is at most 256;
- Although there are many state nodes, the number of edges connected to each state node ranges only from 0 to 256;

- Each time when a node is removed from the bipartite graph, the numbers of edges of its connected nodes are decreases by only one.

According to these prosperities, we can maintain a sorted list for character nodes and 257 linked lists for state nodes with different numbers of connected edges. Based on the 258 lists, the operation required in each phase is proportional to the number of edges removed in the phase.

(3) In the third step, edge sets obtained in the second step are placed into the hash table in reverse order of their removals from the bipartite graph. In other words, the edge set removed from the bipartite graph last is the first placed into the hash table. The perfect hash table construction algorithm is shown in Alg. 2.

Algorithm 2 starts with all nodes of the bipartite graph un-named. Names are assigned to these nodes in the decreasing order of their sequence numbers. Each time we name a node, we place edges of its dependent edge set into the hash table. If hash collision occurs during the placement, we rename the node and re-place all of its dependent edges. This process repeats until all edges in its dependent edge set are successfully placed into the hash table simultaneously. Then we say the name of this node is settled.

Consider the dependent edge sets in TABLE II. Character node m is the first to be assigned a name because it has the largest sequence number. Since node m has no dependent edge, any name for node m is acceptable. After that, state node 6 is named, and its dependent edge $< 6, m >$ is placed into the hash table. Please note that the other endpoint of edge $< 6, m >$ (which is m) has already been named. The next node to be named is s, which has one dependent edge $< 6, s >$. Please also note that the other endpoint of edge $< 6, s >$ (which is 6) is already named. When a hash collision occurs during the placement of edge $< 6, s >$, only node "s" is renamed, while the ID of node "6" will never be changed. This is because some other edges connected to node "6" (the dependent edge set of node "6") are already placed in the hash table. If node "6" was renamed, all these edges would be re-placed again, which might cause further hash collisions. The above process repeats until every node has been named. After this procedure, all edges are placed in the hash table without collision.

2D P^2-Hashing may fail when all names in the name space have been tried before a collision-free placement of a node's dependent edge set could be found. Two measures could be employed to avoid the failure: (1) increase the name spaces of state nodes and character nodes; (2) reduce the load factor of the hash table. However, both measures would increase the memory cost of the perfect hash table.

The 2D P^2-Hashing algorithm has several characteristics, which are summarized as follows.

(1) By breaking edges of the bipartite graph into small independent sets, the impact of hash collision during the placement of an edge is limited to a relatively small range. Consider the AC automaton in Figure 3; after step 2 of the 2D P^2-Hashing, transitions of the AC automaton (i.e., edges of the biparitite graph) will be divided to 26 independent single-transition sets. In step 3 of the 2D P^2-Hashing, these 26 single-transition sets will be placed into the perfect hash table seperately. The failure of the placement of a transition set only affects one transition (resulting in a replacement of the single transition). As a result, the success probabilities of set placements are significantly increased.

Algorithm 2. Perfect Hash Table Construction

Input:
A sequence number $N(v)$ for every node $v \in U \cup V$;
A dependent edge set $D(v)$ for every node $v \in U \cup V$;
Name space NS_{state} and $NS_{character}$ // *contain available IDs for state nodes and character nodes, respectively*.

Output:
A perfect hash table H;
A Character Translation Table CTT, indexed by the ASCII codes of characters;

Algorithm:
Set H, CTT, and STT empty;
Sort nodes in $U \cup V$ in decreasing order of their sequence numbers;
for every node u in the sorted set $U \cup V$ **do**
//Without loss of generality, suppose u is a state node (the following code should be changed accordingly if u is a character node);
{
(1) Among all available IDs in NS_{state}, randomly choose an ID, say $id1$, which hasn't been tried by node u; if all IDs in NS_{state} have already been tried by node u, an error is returned;
Name node u as $id1$ and place all edges of $D(u)$ into hash table H; *//for every edge $< u, v >$ in $D(u)$, it's guaranteed that v has already been named;*
if no hash collision occurs during the placement of $D(u)$
remove $id1$ from NS_{state};
else
goto (1);
}

(2) With the 2D P^2-Hashing, once the name of a node is settled, it will never be changed again. This way, we can avoid the fluctuation of the hash table.

(3) When an edge set is about to be placed into the hash table, every edge in the set has one settled end node and one unsettled end node (which is a common node shared by all edges in the set). When hash collisions occur during the set placement, only the common unsettled node needs to be renamed. Consider the edge set dependent on node "0" in TABLE II; it has two edges, which are $< 0, s >$, and $< 6, m >$, respectively. When this edge set is about to be placed in the hash table, node "s" and "m" are already settled. If any collision occurs during the placement of the two edges, only their common unsettled node (i.e., "0") will be renamed.

(4) Due to the principles used in step 2 and 3 of the 2D P^2-Hashing algorithm, large edge sets are likely to be placed in the hash table at the very beginning when the hash table is almost empty, while the edge sets placed to the hash table at the end are very small. This way, the 2D P^2-Hashing can achieve higher success probabilities for large set placements.

In addition, the proposed 2D P^2-Hashing algorithm can be easily extended to support more than two dimensions in the hash key, i.e., replace the bipartite graph model with a multipartite graph model.

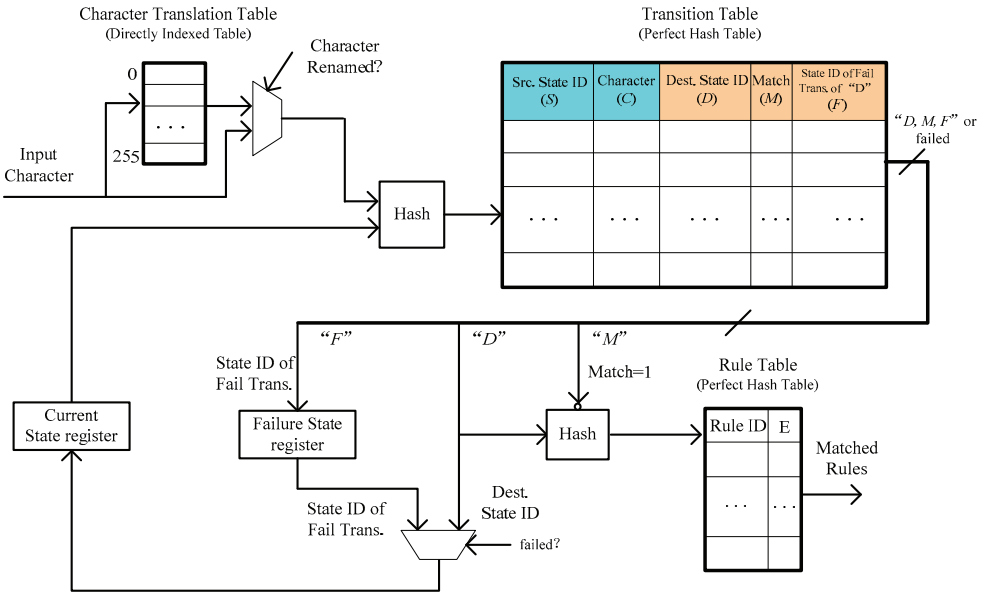

Figure 5. The architecture of multi-string matching engine.

5. SYSTEM DESIGN

In this section, we present the architecture of the proposed multi-string matching engine, as shown in Figure 5. There are three main tables in the architecture, including two perfect hash tables and one directly indexed table.

Character Translation Table (CTT) is used to translate input characters from ASCII codes to the internal encodings. CTT is used only for the 2D P^2-Hashing algorithm. To support both P^2-Hashing and 2D P^2-Hashing in our architecture, a selector is used to decide if CTT is used. Please note that the number of entries in CTT is fixed, i.e., 256 (one for each ASCII char).

Transition Table (TT) is used to store goto transitions and failure transitions and implemented as a perfect hash table. Each entry of TT represents one goto transition of the AC automaton and includes five fields. The first two fields, source state ID ("*S*") and character ("*C*"), are used as the hash key to search the table. The third field "*D*" is the ID of the destination state pointed by the goto transition. The fourth field "*M*" is used to indicate if the state in column "*D*" matches any rules ("1" means match and "0" means no match). The last field "*F*" records the state ID pointed by the failure transition derived from the state in field "*D*".

There are several properties about the TT worth being mentioned.

(1) Although each state on the AC automaton may occur multiple times on the first column (due to multiple goto transitions derived from the state), it can only occur once on column "*D*" (because each state is pointed by one goto transition).

(2) Each state on the AC automaton has only one failure transition.

Because of the above two properties we are able to store the failure transition derived from each state (say *d*) at the same entry where the goto transition pointing to state *d* is stored.

The matching rules are stored in Rule Table (RT). Every time we visit a state associated with rules, we use the state ID as the hash key to get the index of RT. To use memory efficiently, each entry of RT only stores one rule. If a state associates with multiple matching rules, we store all its associated rules in continuous entries starting at the location pointed by the hash index, and use one bit in each entry to indicate if the entry is the last rule associated with the state. For instance, state 5 in Figure 2 associates with two rules (rule 2 and 6). So we store rule 2 and 6 in two continuous entries, and use the ID of state 5 as the hash key to get the index of the first rule. Please note that one rule may have multiple instances in the rule table if it associates with multiple states. The details about how to construct table RT will be presented in the next section.

Based on the architecture in Figure 5, the procedure that a character (say *c*) is processed is explained as follows.

(1) Character *c* is used to index table CTT to get the internal encoding of *c*. This step is required only when 2D P^2-Hashing is used in the construction of the perfect hash tables.

(2) The concatenation of current state ID (stored in current state register) and the encoding of *c* is used as the hash key sent to the hash unit, which returns the index to table TT. The current state ID and character *c* are compared with the first two fields of the indexed entry.

(2.1) if they are matched, we say a goto transition is found and we (a) update the current state register using field "*D*" of the indexed entry; (b) update the failure state register using field "*F*" of the indexed entry; (c) search table RT (using field "*D*" as the hash key) to find matched rules if field "*M*" is equal to 1.

(2.2) if they are not matched, a failure is returned and we (a) update the current state register using the state ID stored in the failure state register; (b) go back to the beginning of step (2) and repeat the procedure.

Based on the above procedure, it is easy to see that the major operations involved in the architecture are hash calculations and table accesses. Therefore, the architecture is suitable for both hardware and software implementations.

6. PERFECT HASH TABLE CONSTRUCTION ALGORITHM WITH RULE TABLE SUPPORT

In Section 4, we discuss the perfect hash table construction for transition table. In this section, we consider a more complicated situation in which two perfect hash tables (TT and RT) are constructed simultaneously.

Let's first review hash keys used in the two hash tables. The hash key of TT is the concatenation of source state ID and input character, while the hash key of RT is only the source state ID. To generalize the perfect hash table construction problem, we suppose that each rule R_i corresponds to a virtual character γ_i. Values of these virtual characters are all NULL. With the introduction of virtual characters, hash keys of the two hash tables are unified to the same form, i.e., the concatenation of source state ID and character.

We now modify the 2D P^2-Hashing algorithm to support the constructions of the two perfect hash tables (TT and RT).

(1) The purpose of step 1 of the 2D P^2-Hashing algorithm is to convert the AC automaton $M = (Q, \sum, g, f, T)$ to a bipartite graph $B = (U, V, E)$. With the consideration of two tables, we let $U = Q$, and $V = \Sigma \cup \{\gamma_1, \ldots \gamma_l\}$, where l is the number of rules. We let the edge set E as the union of two subsets: E_1 and E_2. Each edge in E_1 corresponds to a goto transition, i.e., $E_1 = \{< q, c > \mid \forall q \in Q, \forall c \in \Sigma,$ that $g(q, c) \neq fail\}$. Each edge in E_2 corresponds to a pair of state and matched rule, i.e., $E_2 = \{< q, \gamma_i > \mid$ if state q matches rule i $\}$.

Consider the AC automaton in Figure 2. Its bipartite graph model with the consideration of matching rules is given in Figure 6, where gray nodes correspond to virtual characters.

(2) Since $\gamma_1, \ldots \gamma_l$ are virtual characters, they cannot be renamed to help avoid hash collisions. The reason for representing them in the bipartite graph is to help determine the degrees of state nodes, which imply the difficulties of renaming the state nodes to achieve collision-free placements in both perfect hash tables. In the second step, the bipartite graph is decomposed to small edge sets. Since virtual characters cannot be renamed, they are treated as fixed nodes, and never participate in the procedure of decomposition. One possible decomposition result of the bipartite graph in Figure 6 is given in TABLE III.

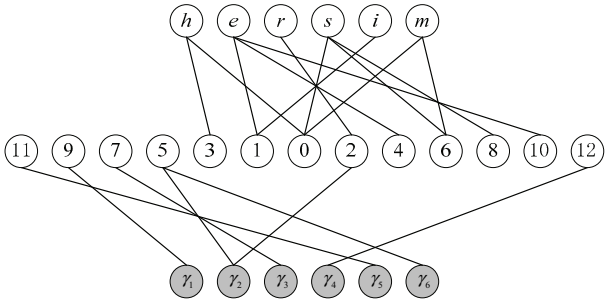

Figure 6. The bipartite graph model of AC automaton in Figure 2 with the consideration of matching rules. (Nodes in the first and third rows are in node set V.)

TABLE III. Decomposition result of the bipartite graph in Figure 6

Seq	1	2	3	4	5	6	7	8	9	10
Node	11	9	7	12	3	h	r	2	i	1
Dependent edge set	<11,γ_5>	<9,γ_1>	<7,γ_3>	<12,γ_4>	<3,h>	<0,h>	<2,r>	<2,γ_2>	<1,i>	<1,e>
Seq	11	12	13	14	15	16	17	18	19	
Node	4	e	10	8	5	0	s	6	m	
Dependent edge set	<4,e>	<10,e>		<8,s>	<5,γ_2> <5,γ_6>	<0,s> <0,m>	<6,s>	<6,m>		

(3) Step 3 is similar to what described in Section 4, except that each dependent edge set here might have two different types of edges. During the placement of each dependent edge set, edges are placed into the corresponding hash tables according to their types. Any hash collision that occurs during the placement of a dependent edge set causes the renaming of the associated node, and the re-placements of all edges in the set. It's worth noting that the definition of hash collision in the rule table is different from that in the transition table. Consider state node 5 in TABLE III. It is associated with two rules, R2 and R6. According to our system design, we use the ID of state node 5 as the hash key to get the hash index, and place the two rule instances in two continuous entries starting at the index. The placement is successful if both of the two entries are available; otherwise a hash collision occurs.

7. PERFORMANCE EVALUATION

In this section, we evaluate P^2-Hashing and 2D P^2-Hashing in terms of hash table construction time and memory cost. The evaluated algorithms are implemented in C++ and tested on an AMD Athlon 64 X2 5200+ 2.60-GHz computer with 3-GB memory. Three string rule sets are used in the evaluation. The first is extracted from the Snort rule set (June 2009), and includes 6.4K string rules; the second is extracted from the ClamAV rule set (June 2009), and includes 54K string rules; the third one is a subset of the first rule set including only 20 short rules with the same prefix. Given the three rule sets, we first prepare the automatons used in the evaluation. We convert each rule set into an AC automaton, and an AC-DFA. Due to the huge memory cost of AC-DFA, we eliminate its backward transitions to states at the first four levels according to the scheme in [19]. To maintain the functionality of the AC-DFA after the transition elimination, we adopt the scheme proposed in [19], i.e., cache two states in registers to keep track of the destination states pointed to by the eliminated transitions. TABLE IV shows the statistics of the three rule sets and their automatons.

TABLE IV. Three rule sets and their Automatons

	Snort	ClamAV	Small Set
Rule #	6.4K	54K	20
total character #	105K	6.49M	82
state # of AC automaton	77K	6.24M	24
(failure) transition # of AC automaton	77K	6.24M	23
trans. # of AC-DFA after trans. elim.	118K	7.62M	/
# of bits in state ID	19	25	7
# of bits in Character ID (if applicable)	9	9	6

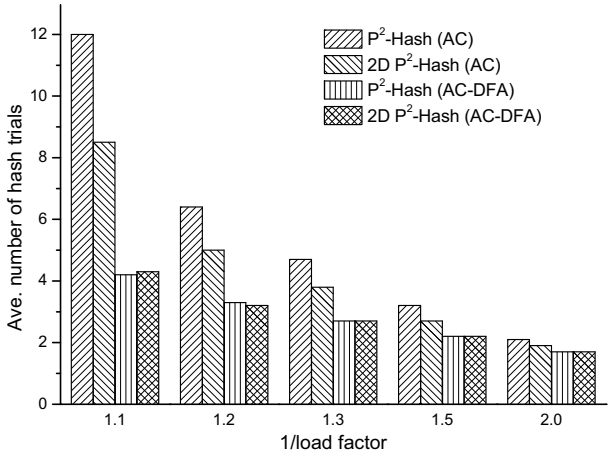

Figure 7. Average number of hash trials required for each transition placement under Snort rule set

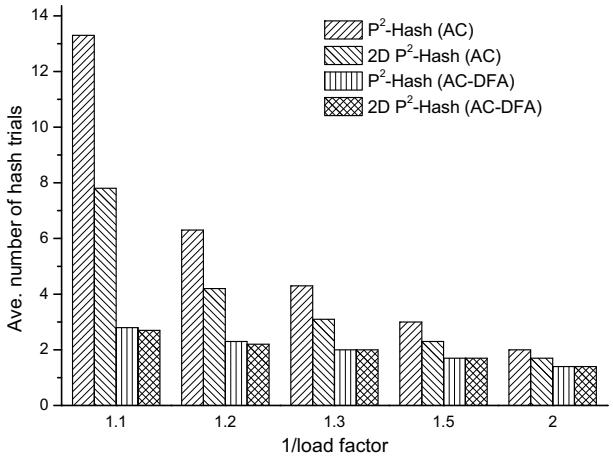

Figure 8. Average number of hash trials required for each transition placement under ClamAV rule set

The name space of states is set four times as large as the state number, i.e., we use two more bits in state IDs than what we really need to uniquely represent the states. This allows us a certain amount of unused names to select from when hash collisions occur. The name space of characters is set twice as large as the character number. For both AC automaton and AC-DFA, the evaluated algorithms construct two hash tables (TT and RT). During the experiment, all hash tables are configured with the same load factor.

7.1 Perfect Hash Table Construction Time

The hash table construction time could be measured by the average number of hash table insertion trials (hash trials for short) required for every transition placement. Figure 7 and Figure 8 show the average number of hash trials required for each transition placement when two automatons are built under different rule sets. Both P^2-Hashing and 2D P^2-Hashing can successfully construct the perfect hash tables with the given name space sizes, even when the load factor of the hash table is set as high as $1/1.1 = 90.9\%$.

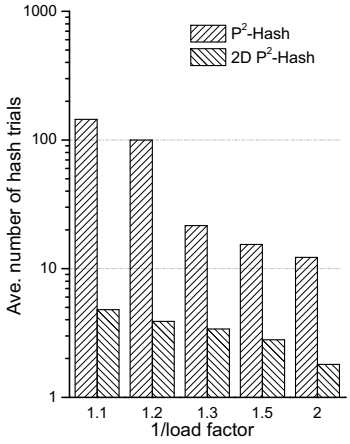

Figure 9. Average number of hash trials for each transition placement under small rule set for building AC automaton

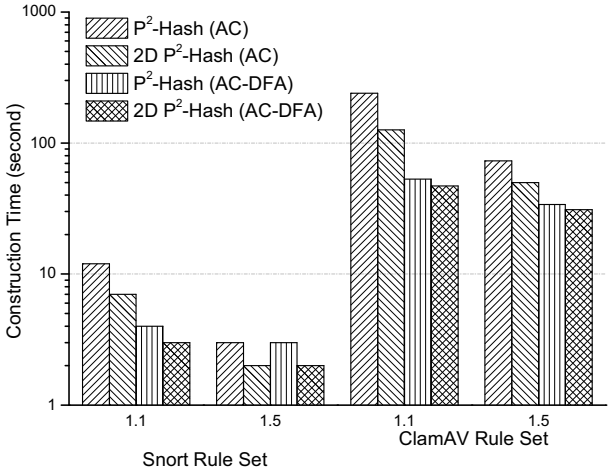

Figure 10. Perfect hash table construction time for different rule sets (load factors are set to 1/1.1 and 1/1.5)

It's easy to see that 2D P^2-Hashing performs slightly better than P^2-Hashing. The better performance is because when compared to P^2-Hashing, 2D P^2-Hashing can decompose transitions to much smaller transition sets. Taking the AC automaton of the Snort rule set as an example, the largest transition set in P^2-Hashing that has to be placed into the hash table as a whole includes 178 transitions, while the largest transition set in 2D P^2-Hashing includes only 14 transitions (details are not shown due to space limitations). Therefore, 2D P^2-Hashing can achieve higher success probabilities when placing transition sets into the hash table. In Figure 9, we compare P^2-Hashing and 2D P^2-Hashing under the small rule set. We can see that the performance gap between P^2-Hashing and 2D P^2-Hashing has widened.

From these figures, it is clear that the average number of hash trials drops quickly when the load factor of the hash table decreases. Figure 10 shows the actual running times of the evaluated algorithms to construct the perfect hash tables with different rule sets, automatons, and hash-table load factors. Under the Snort rule set, when the load factor of the hash tables is 90.9%, the 2D P^2-Hashing algorithm takes only 7 seconds to construct the perfect hash tables for AC automaton, and 3 seconds for AC-DFA.

Both construction times drop to only 2 seconds when the load factor of the hash tables is reduced to 66.6%.

Due to the large size of the ClamAV rule set, 2D P^2-Hashing requires about 120 seconds and 50 seconds to build the perfect hash tables for AC automaton and AC-DFA, respectively, when the load factor of hash tables is 90.9%. If the load factor reduces to 66.6%, 2D P^2-Hashing requires only 50 seconds to build perfect hash tables for AC automaton, and 30 seconds for AC-DFA. The reason that the construction time of AC-DFA is shorter than that of the AC automaton is because the construction of AC-DFA doesn't need to avoid hash collisions for failure transitions.

7.2 Storage Requirement

The perfect hash tables constructed by P^2-Hashing and 2D P^2-Hashing are very compact. We compare the memory cost of our multi-string matching engine against the existing work in TABLE V and TABLE VI. In the evaluation, the load factors of hash tables are assumed to be 90.9%. It can be seen that under the snort rule set, our scheme is among the most compact implementations of AC automaton. Compared with CDFA and B-FSM, our solutions store all rules in a single AC automaton without rule-set partitioning, and therefore are suitable for both hardware and software implementations. Under the ClamAV rule set, the memory costs of our schemes are about 1.5~1.8 times that of CDFA, which is currently the best known scheme for the full ClamAV rule set. To achieve the very low memory cost, CDFA needs to partition the rule set into 32 subsets, and implements each of them with an individual AC automaton. The large number of parallel AC automatons makes CDFA unsuitable for software implementation and hardware off-chip memory implementation (due to the pin limitation of the chip). Furthermore, hardware on-chip implementation is also infeasible, because CDFA requires 26.8MB memory, which is far beyond the capacity of on-chip memory.

TABLE V. Memory usage comparison for Snort rule set

	AC Types	Rules #	# of Partitions	Total Characters	Total Memory	mem/ char
(2D) P^2-Hash	AC	6.4K	1	105K	699KB	7.6B
(2D) P^2-Hash	AC-DFA	6.4K	1	105K	767KB	8.33B
CDFA [17]	AC-DFA	1,785	2	29.0K	129KB~ 256KB	4.45B~ 8.2B
B-FSM [8]	AC-DFA	1.5K	4	25.2K	188KB	7.4B
Bitmap Compression [9]	AC	1.5K	1	18.2K	2.8MB	154B
Path Compression [9]	AC	1.5K	1	18.2K	1.1MB	60B

TABLE VI. Memory usage comparison for ClamAV rule set

	AC Types	Rule #	# of Partitions	Total Characters	Total Memory	mem/ char
(2D) P^2-Hash	AC	54K	1	6.49M	72.1MB	11.1B
(2D) P^2-Hash	AC-DFA	54K	1	6.49M	61.8MB	9.53B
CDFA[17]	AC-DFA	50K	32	4.44M	26.8MB	6.1B

8. DISCUSSION AND FUTURE WORK: INCREMENTAL UPDATE

Since the insertion and deletion of a rule can be decomposed to multiple insertions or deletions of transitions, we only consider how to delete a transition from and insert a transition into the hash table. Deleting a transition from the hash table is trivial, and is similar to performing a hash table lookup. Inserting a transition, however, is not that natural, because hash collisions may occur during the insertion. According to the main principle of this paper, when hash collision occurs, we should rename the source state or the labeled character of the conflicting transition, and re-place all related transitions. Given the fact that each character usually associates with tens of thousands of transitions in large rule set, renaming characters is infeasible. So the only choice is to rename the source state of the conflicting transition until all of its associated transitions are placed into the hash table without collision. However, there are some states associated with a large number of transitions, say 100 transitions (we call these states large states, and call the numbers of associated transitions the sizes of the states). The probability of placing such a large number of transitions into an almost full hash table without any collision is low.

Actually, instead of renaming the large state until all of its associated transitions are placed into collision-free locations, we can let its transitions kick out the transitions currently resident in the conflicted locations (similar to the scheme used in Cuckoo hashing [16]), if the states associated with these resident transitions are all relatively small. In cases, some states that are kicked out are still too large. They can continue to kick out smaller states, until the states to be renamed are small enough. Then we just need to rename these small states to achieve collision-free placements. This way a complicated problem is decomposed to many simpler problems. In fact, analysis on the AC automatons of Snort and ClamAV rule sets shows that more than 99% of states have only three or fewer transitions (as shown in Figure 1). Therefore, the above scheme seems feasible. In our future work, we will study and investigate the incremental update scheme.

9. CONCLUSION

This paper proposes a multi-dimensional perfect hash table construction algorithm named P^2-Hashing, based on which the well-known AC automaton can be implemented by very compact perfect hash tables. P^2-Hashing requires no memory access to generate the hash index and guarantees to return the hash result within the time of exact one memory access. The processing of each character therefore requires only one memory access in a pipelined architecture. This property is very important for NIDS to survive under the attack of malicious traffic. It should be also noted that the use of character translation table won't change the above property, since the character translation table is not on the critical path of the AC automaton pipeline operation and works independently to the hash tables.

10. REFERENCES

[1] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," Communications of the ACM, vol. 18, no. 6, pp. 333-340, 1975.

[2] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in DISCEX, 2003.

[3] L. Spitzner, Honeypots: Tracking Attackers. Addison-Wesley, 2002.

[4] S.Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[5] S. Dharmapurikar and J. W. Lockwood, “Fast and scalable pattern matching for network intrusion detection systems,” IEEE Journal of Selected Areas in Communications, vol. 24, no. 10, 2006.

[6] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, “A memory-efficient parallel string matching architecture for high-speed intrusion detection,” IEEE Journal of Selected Areas in Communications, vol. 24, no. 10, 2006.

[7] N. Hua, H. Song, T.V. Lakshman, “Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection”, in IEEE INFOCOM, 2009.

[8] J. van Lunteren, “High-Performance Pattern-Matching for Intrusion Detection,” in IEEE INFOCOM, 2006.

[9] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection,” in IEEE INFOCOM, 2004.

[10] M. Becchi and P. Crowley, “ Efficient Regular Expression Evaluation: Theory to Practice,” In Proceedings of the 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), San Jose, CA, November, 2008.

[11] F. Yu , “High speed deep packet inspection with hardware support,” PhD dissertation of University of California at Berkeley, Berkeley, CA, 2006

[12] A free lightweight network intrusion detection system for UNIX and Windows. [Online]. Available: http://www.snort.org.

[13] ClamAV. [Online]. Available: http://www.clamav.net.

[14] J. van Lunteren and A.P.J. Engbersen, “Fast and scalable packet classification,” IEEE Journal of Selected Areas in Communications, vol. 21, no. 4, pp. 560-571, May 2003.

[15] N. S. Artan and H. J. Chao, “Tribica: Trie bitmap content analyzer for high-speed network intrusion detection,” in IEEE INFOCOM, 2007

[16] R. Pagh and F. F. Rodler, “Cuckoo hashing,” in ESA, 2001.

[17] S. Kumar, J. Turner, and P. Crowley, “Peacock hashing: Deterministic and updatable hashing for high performance networking,” in IEEE INFOCOM, 2008.

[18] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond bloom filters: from approximate membership checks to approximate state machines,” in ACM SIGCOMM, 2006.

[19] T. Song, W. Zhang, D. Wang, and Y. Xue, “A memory efficient multiple pattern matching architecture for network security,” in IEEE INFOCOM, 2008.

[20] L. Tan, T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” in 32nd Annual International Symposium on Computer Architecture, ISCA, 2005.

[21] Z.K. Baker, V. K. Prasanna, “High-throughput linked-pattern matching for intrusion detection systems,” in symposium on Architecture for Networking and Communications Systems (ANCS), Oct. 2005.

[22] I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis, “Scalable multigigabit pattern matching for packet inspection,” IEEE Trans. VLSI Syst., 16(2):156–166, 2008.

[23] Y.-H. E. Yang and V. K. Prasanna, “Memory-efficient pipelined architecture for large-scale string matching,” in 17th Annual IEEE FCCM, April 2009.

[24] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, “HEXA: Compact Data Structures for Faster Packet Processing,” In Proceedings of the Fifteenth IEEE International Conference on Network Protocols (ICNP), pp. 246-255, 2007.

[25] N. S. Artan, M. Bando, and H. J. Chao, “Boundary Hash for Memory-Efficient Deep Packet Inspection,” IEEE International Conference on Communications (ICC 2008), Beijing, PRC, May 19-23, 2008.

[26] F. Yu, R. H. Katz, and T. V. Lakshman, “Gigabit Rate Packet Pattern-Matching Using TCAM,” In Proceedings of the Fifteenth IEEE International Conference on Network Protocols (ICNP), 2004.

[27] P. Piyachon and Y. Luo, “Efficient memory utilization on network processors for deep packet inspection,” In symposium on Architecture for Networking and Communications Systems (ANCS), 2006.