A Novel Efficient Pattern Matching Packet Inspection by using δ^nFA

[1]N.Kannaiya Raja
CSE Dept. Arulmigu Meenakshi Amman College of Engg, Thiruvannamalai Dt, near Kanchipuram

[2]K.Arulanandam
CSE Department Ganadipathy Tulsi's Jain Engineering College, Vellore

[3]G. Ambika M.E
Arulmigu Meenakshi Amman College of Engg, Thiruvannamalai Dt,near Kanchipuram

Abstract
Deep packet Inspection is an advanced method of packet filtering that functions at the Application layer of the OSI reference model. Deep Packet Inspection is a form of computer network packet filtering that examines the data part of a packet as it passes an inspection point, searching for protocol ,viruses ,spam, intrusions or predefined criteria to decide if the packet can pass or it needs to be routed to a different destination, or for the purpose of collecting statistical information. Deterministic finite automata (DFAs), use large set of rules need a memory amount that turns out to be too large for practical implementation we have presented a new compressed representation for deterministic finite automata, called Delta Finite Automata. The algorithm considerably reduces the number of states and transitions, and it is based on the observation that most adjacent states share several common transitions, so it is convenient to store only the differences between them. In this paper we have presented an improvement to δFA that exploits the Nth-order dependence between states and further reduces the number of transitions by adopting the concept of temporary transition. This schema named as δnFA. Both the schemes are orthogonal to most of the previous solutions, thus allowing for higher compression rates. A new encoding scheme for states has been also proposed(which we refer to as char state), which exploits the association of many states with a few input chars. Such a compression scheme can be efficiently integrated into δFA and δnFA, allowing a further memory reduction with a negligible increase in the state lookup time. The experimental runs have shown remarkable results in terms of lookup speed as well as the issue of excessive memory consumption.

Index Terms
Deep packet inspection, differential encoding, finite automata (FAs), pattern matching, regular expressions.

I. INTRODUCTION

Regular expressions are used in Modern Deep Packet Inspection to define the various patterns of interested data streams in the Network. Deterministic Finite Automata (DFA) is used to parse the regular expressions. Though DFA technique is faster, it consumes large memory space for pattern arising. Traditional DFA table slightly reduces the memory required and access to memory per character. Further improvement on regular expressions such as NFAs and Delayed Input DFAs (D^2FA) reduces memory consumption by sacrificing the throughput.

With respect to [5], the concept of "Temporary transition" is used to improve regular expressions called δFA. Instead of specifying transition set of a state with respect to its direct parents, adopting N-step "ancestors" increases the chances of compression. The best approach to exploit this N^{th}-order dependence is to define the state transitions between ancestors and child as "temporary." Experimental rule set results show that simple approach meets the optimal construction (Memory or transition reduction). As it is an extension to δFA, the method is named as δ^nFA.

Content Addressed Delayed Input DFA (CD^2FA) provides a compact representation of regular expressions which matches the throughput of traditional uncompressed DFAs. Instead of using "content less" identifier, CD^2FA uses their content to address successive states of a D^2FA. By this selected information will be available earlier in the state traversal process. It avoids unnecessary memory access. Based on this content addressing, compact automata can be obtained with high throughput.

CD^2FAs matches the throughput of an uncompressed DFA by using as little as 10% of the space required by conventional DFA. Many network services are processing the packets based on the payload content. As the Deep packet inspection compares the packet to a set of strings, forwarding the packet based on the content requires new level of support in networking devices. New systems are using regular expressions instead of string sets. Cisco has even the regular expression based on content inspection

Manuscript received January 5, 2013
Manuscript revised January 20, 2013

capabilities into its operating system (IOS [21]). Regular expressions can be used in Linux OS to filter the content at Application Layer. Regular expressions are limited in networking context as it requires substantial amount of memory.

A new technique Deterministic Finite Automata (D^2FA) uses delayed input for parsing Regular expressions and default transitions to reduce the memory requirements. A default transition is followed whenever the current input character does not match any of the labeled transitions leaving the current state. If two states have a large number of "next states" in common, the common transitions are replaced leaving one of the states with a default transition to the other. Every state will have only one default transition, so that the amount of memory needed to represent the parsing automata will be reduced dramatically. Default transitions also reduces throughput, since no input is consumed when a default transition is followed, however memory has to be accessed to retrieve the next state.

The remainder of the paper is organized as follows. In Section II, related works about DFAs are discussed. Section III accurately describes construction and analysis of DFA, by starting from a motivating example, Section IV presents the optimization of ∂FA, and Section V proves the integration of the proposed schemes with the previous ones. Finally, Section VI presents the experimental results.

II. RELATED WORK

Traditionally, DFA and NFA are used to search for regexes (Regular expressions). But DFAs have large memory consumption and fixed memory references per character. Although NFAs consumes lower memory, it requires several memory transitions per symbol for all states at a given time. NFAs are used in hardware platforms such as FPGAs while DFA are used in software-based systems such as network processors. In industry, DFA are used to represent regular expression especially for parallel system. As the overall performance of packet processing is affected by the slowest component's processing time. Therefore, industries are adopting for pattern-matching deterministic solutions as DFAs. Large memory consumption of DFA is due to the encoding and state explosion.

DFA encoding introduces the delayed input DFA (D^2FA). The drawback of D^2FA is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions. The number of default transitions taken by a single character is defined as bound B. Larger values of B indicate higher compression.

An improved algorithm called Bec-Cro is implemented in [5] for large number of access per char in D^2FA. Experimental results proved that it reduces bounds on memory bandwidth. It is based on the inspection of all regex evaluations starts at a single initial state, and the majority of transitions among states, back either to the initial state or its closest neighbors. The memory compression of D^2FA is shown as by accessing a single memory per character.

The nonequivalent states can be combined. Combined states with common destinations despite of characters which lead those transitions (unlike D^2FA),it create for more merging and thus we achieve higher memory reduction. Bitmaps for compression purposes which increase the cost by requiring two subsequent memory accesses

CD^2FA is used to increase the speed of D^2FAs by storing a large amount of information (on subsequent reachable transitions) on the edges. It reduces the cost of D^2FAs and requires a construction based on perfect hash functions that may be time-consuming. The idea of storing more information on the edges appears to be a general trend in the literature, and it is implemented in following ways: In [4], transitions carry data on the next reachable nodes; edges have different labels; in [6], a sort of history buffer (i.e., a small and fast cache) stores additional information in order to efficiently follow multiple partially matching signatures, thus yielding the state blow-up; in [5], a finite scratch memory is used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters) in order to keep the transition history and reduce the number of states.

NFAs can improve the memory problem, but it lead to a large bandwidth requirement, in [10] a hybrid DFA-NFA solution is proposed. When constructing the hybrid-DFA, any nodes that would contribute to state explosion retain an NFA encoding, while the remaining nodes are transformed into DFA nodes. Its aim is to reduce size almost equal to NFA, with small memory bandwidth requirements of a DFA.

MEMORY PACKING

When CD^2FA was introduced, there was an assumption of existence of hash function which maps content labels to the original state numbers. There are algorithms to devise such mapping. While associating state numbers to content labels, unique numbers can be directly used as an index into the memory. Thus unique memory address has to be associated to the content label of each state, so that the list of content labels for all labeled transitions leaving the state will be stored at that address. This requires a single memory access per input character. In this section, state numbers are referred as memory address where it is stored and storing a state means storing the content labels for its labeled transitions. As root states are simply stored as a

two dimensional table, attention to be focused on storing non-root states.

The size of the list of content labels for a state depends both upon the number of labeled transitions leaving the state as well as length of their content labels (1 or 2 words). Traditional table compression schemes [11] may be applied to associate a unique address to each state's content label, however these schemes are known to be NP-hard, and they also incur sizeable overheads as they require *i*) additional pointer per state, and *ii*) a marker for every content label. They also require an additional memory access per character, which may reduce the throughput.

A novel method is presented which enables, *i*) an optimal memory utilization with zero space overhead, and *ii*) single memory access per input character. It is based on classical bipartite graph matching, with running time of $O(n3 / 2)$, where n is the number of states. This method proceeds by forming groups of states, so that states with identical memory requirement belong to the same group. Since a non root state is allowed to have at most 5 labeled transitions, the memory requirement of a non-root state can vary from one word to up to ten words; hence there can be up to 10 groups of states. Afterwards, memory is partitioned in 10 regions and states of each group are stored in different regions. In CD^{2}FA, states can be easily associated to their memory regions as the memory requirements of a state can be directly inferred from the states' content label.

PACKING PROBLEM FORMULATION

Let there are n states in a group and each state requires memory words to store its labeled outgoing transitions. Clearly, the group's memory region must contain at least ns words. We consider a slight memory over-provisioning, so the memory region consists of ms words (where $m = n+\Delta$, and Δ/n is the over-provisioning). Content label of all states of the group needs to be uniquely mapped to one of the m memory locations (which become the content labels' state number). We apply a hash function (with co domain = [1, m]) to the content labels to compute this mapping. As traditional hashing is subject to collisions, multiple content labels may be mapped to a single state number. Collision resolution policies can be applied however they are likely to degrade the performance by requiring additional memory accesses. They will also incur space overheads by unnecessarily storing the content labels (as the hash keys).

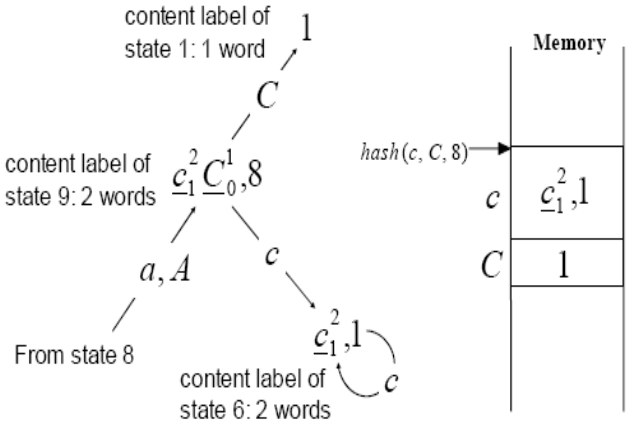


Fig 1 Storing list of content labels for state 9 in memory

This algorithm eliminates both these deficiencies by enabling a collision free hashing, *i.e.* content labels are mapped to unique state number. This is achieved by exploiting the possibility of renaming a content label, without changing its meaning, thus effectively changing its hash value. There are three ways to rename content labels without changing their meanings. *a*) The simplest way is to modify the value of *discriminator*. *b*) An alternative is to change the order in which characters appear in the content label; thus a content label with t characters can have factorial t different possible names. *c*) In fixed size word length restricted content labels, yet another possibility is to pad label shorts by repeating some characters already present in the content label, or by modifying the unused bits. With these facilities to modify the name of a content label without changing its meaning, a naive mapping may arbitrarily rename them whenever a collision occurs. Systematic approach to be developed to select the appropriate names.

The approach progresses by evaluating all possible names (called candidate names) that can be assigned to a content label by employing the three mentioned methods. A hash is then applied to the candidate names, and the result is a set of candidate state numbers for the content label. Once all candidate state numbers are known, a bipartite graph $G = (V1+V2, E)$ is constructed, where vertex set $V1$ corresponds to the n content labels and $V2$ the m state numbers. Edge set E contains all edges (u, v) such that $u \in V1$, $v \in V2$ and v is a candidate state number for u.

After constructing the bipartite graph G, the next step is to seek a *perfect matching*, *i.e.* match each content label to a unique state number. It is likely that no perfect matching exists. A *maximum matching M* in G, which is the largest set of pair wise non-adjacent edges, may not contain n edges, in which case some content labels will not be

assigned any state number. However using theoretical analysis, we show that, when the number of candidate names per content label is $O(\log n)$, then a perfect matching will exist with high probability, even if $\Delta = 0$. As Δ increases slightly, probability of perfect matching grows very quickly, which guarantees that little over-provisioning will always result in a perfect matching.

Once a perfect matching is found, for each content label, we fix its name to the one, for which its state number corresponds to a matching edge. These content labels are guaranteed to enable a collision free hashing during lookup.

AN ILLUSTRATING EXAMPLE

A simple example is considered to illustrate the basic ideas. There are 9 states, and the content labels of labeled transitions entering these states are shown in Figure 2*a*. There are 7 non-root states. States 3 and 7 do not require any memory, as they do not have any labeled outgoing transition (their content labels, however, may be stored at other states, from where a labeled transition enters these states). State 9 is the only state in its group, thus its packing is trivial. States 2, 4, 5 and 6, as shown in Figure 2*b*, each requires one word; therefore these are packed in a memory region containing 4 or more words.

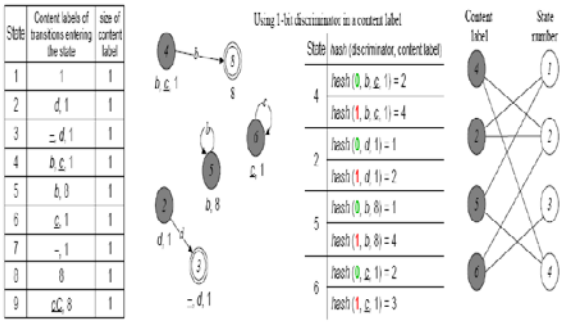


Figure 2. a) Content labels of states of the CD2FA b) Non-root states requiring one word to store the content labels associated with their labeled transitions. c) Candidate content labels (using 1-bit discriminators) and the resulting candidate state numbers. d) Corresponding bipartite graph.

First, we consider no memory over-provisioning ($m = n = 4$), and a single bit discriminator. We limit ourselves to using discriminators to rename content labels and do not use other methods. Thus, there are two candidate names for each state's content label, and the candidate state numbers by applying hash over these are shown in Figure 2*c*. The resulting bipartite graph is shown in Figure 2*d*; there are two perfect matching in this graph, one containing edges, 4-2, 2-1, 5-4 and 6-3 and another containing edges, 4-4, 2- 2, 5-1 and 6-3. Either of these will suffice in mapping unique state numbers to the content labels. Note that, in this case, we have not used memory over-provisioning; indeed, we find that, we can generally avoid memory over provisioning and also avoid discriminators because the other two methods of renaming content labels creates enough edges in the bipartite graph so that a perfect matching most likely exists Analysis of the packing problem

The possibility of an optimal packing depends on the likelihood of finding a perfect matching on the above bipartite graph. A necessary and sufficient condition that a perfect matching exists is due to Hall's Matching Theorem .

Hall's Matching Theorem: Given a set of n items, and a set of identifiers for each item (called its candidate set), each item can be assigned a unique identifier from its candidate set if, and only if, for every $k \in [1, n]$, the union of candidate sets of any k items, contains at least k identifiers.

Thus, we have to show that, for every k content labels, the union of their candidate state numbers contains k or more distinct numbers. For k=1, this is obvious, as candidate set of any content label is non-empty. For k>1, Hall's theorem can be unsatisfied. This is due to the use of hashing in determining the state numbers. Even though a content label can have many (say l) names, its candidate set may still contain a single state number, due to collisions. In general, k content labels will have a total of kl random state numbers in the union of their candidate set. Thus, in order to compute the likelihood of a perfect matching, we compute the probability with which a set of kl randomly chosen numbers $\in [1, m]$ contains k or more distinct numbers.

Table 1. Our representative regular expression groups

Source	# of regular expressions	Avg. ASCII length of expressions	% expressions using wild-cards (*, +, ?)	% expressions length restrictions {,k,+}
Cisco	590	36.5	5.42	1.13
Cisco	103	58.7	11.65	7.92
Cisco	7	143.0	100	14.23
Linux	56	64.1	53.57	0
Linux	10	80.1	70	0
Snort	11	43.7	100	9.09
Bro	648	23.6	0	0

The problem of finding perfect matching in such bipartite graphs is well studied. In [12], Motwani shows that a perfect matching in a symmetric bipartite graph with n left and right vertices and with random edges, exists with high probability when the number of edges are $O(n \log n)$. In fact, this threshold is sharp, which means that the

probability of perfect matching increases very quickly, as we add slightly more edges after threshold. In an asymmetric case, (when $m > n$), that the probability of a perfect matching again increases quickly, as m is greater than n. For instance, when $m/n = 1.01$, (implies 1% memory over provisioning), a perfect matching exists with high probability, if there are more than $7n$ edges in the bipartite graph.

With these results we can conclude that if we have flexibility to assign $O(\log n)$ different names to each content label, then we will most likely find a perfect matching without any memory over-provisioning. $O(\log n)$ corresponds to approximately 16 choices of names for each content label in a 64K state CD^{2}FA; this can be easily achieved even without using discriminators. As expected, in our experiments, we found a perfect matching in all CD^{2}FAs without using memory over-provisioning or employing the discriminators.

III. CONSTRUCTION AND ANALAYSIS

Deep packet inspection consists of processing the entire packet payload and identifying a set of predefined patterns. But now we use regular expressions, due to their greater expressive power and flexibility [17]. Regular expressions are searched through DFAs, which have attractive features, such as one transition for each character, which means a fixed number of memory accesses. DFAs have large set of regular expressions can blow up in space, and many recent works have been presented with the aim of reducing their memory.

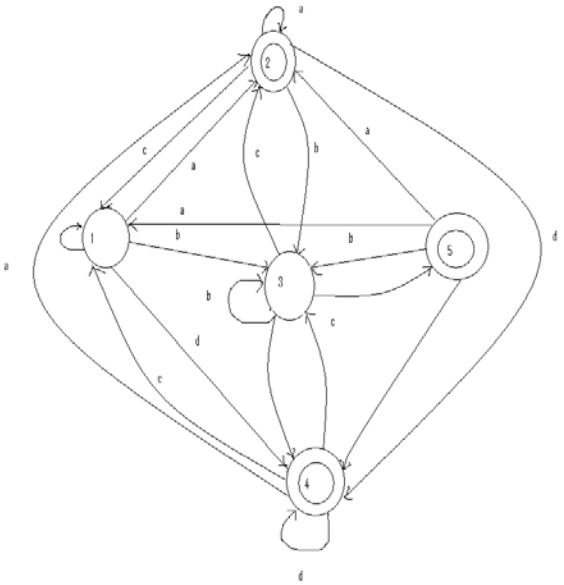


Figure 3. The DFA for (a+), (b+c) and (c∗d+).

In [15], Kumar et al. introduce the Delayed Input DFA (D^{2}FA), a new representation which reduces space requirements. Since many states have similar sets of outgoing transitions, redundant transitions can be replaced with a single default one, this way obtaining a reduction of more than 95%. The drawback is travelling of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions. However, a bound B on the number of default transitions to be taken by a single character in D^{2}FA can be defined: generally, larger values of B (hence many memory accesses per byte) correspond to higher memory compression.

The analysis of DFA shows that it is infeasible as it uses a large set of regular expressions. Although NFAs improve the memory storage problem, it requires large memory bandwidth. It is due to the multiple NFA states which are active and each input character can trigger multiple transitions. Therefore a hybrid DFA-NFA solution is required to combine the advantages of both automata: When constructing the automaton, any nodes that contribute to state explosion retain an NFA encoding, while the others are transformed into DFA nodes.

Kumar et al. [16] also showed how to increase the speed of D^{2}FAs by storing more information on the edges. This appears to be a general trend in the literature even if it has been proposed in different ways: in [16] transitions carry data on the next reachable nodes, in [2] edges have different labels, and even in [14] transitions are no more simple pointers but a sort of “instructions”.

In a further comprehensive work [14], Kumar et al. analyze three main limitations of the traditional DFAs. First, DFAs do not take advantage of the fact that normal data streams rarely match more than a few initial symbols of any signature; its propose is to split signatures such that only one portion needs to remain active, while the remaining portions can be “put to sleep” (in an external memory) under normal conditions. Second, the DFAs are extremely inefficient in following multiple partially matching signatures and this yields the so-called *state blow-up*: a new improved Finite State Machine is proposed in order to solve this problem. The idea is to construct a machine which remembers more information, such as encountering a closure, by storing them in a small and fast cache which represents a sort of history buffer. This class of machines is called History-based Finite Automaton (H-FA) and shows a space reduction close to 95%. Third, DFAs are incapable of keeping track of the occurrences of certain sub-expressions, thus resulting in a blow-up in the number of state: Introducing some extensions to address this issue in the History-based counting Finite Automata (H-cFA).

The idea of adding some information to transitions, consequently reduced the number of states, has been retrieved, where another scheme, named extended FA

(XFA), is proposed. In more details, XFA augments traditional finite automata with a finite scratch memory used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters and other instructions attached to edges and states). Experiments performed with a large class of NIDS signatures showed time complexity similar to DFAs and space complexity similar to or better than NFAs.

IV DELTA FINITE AUTOMATON: δFA

A. A motivating example

In this section we introduce the principles of δFA [18] by analyzing the same example brought by Kumar et al. in [8]: the fig. 3 represents a standard DFA on the alphabet *{a, b, c, d}*that recognizes the regexes $(a+)$,$(b+c)$ and $(c*d+)$. In fig. 4 the D^2FA for the same set of regular expressions is shown, where the memory footprint of states is reduced by storing only a limited number of transitions for each state and by taking a default transition for all input chars for which a transition is not defined. The total number of transitions was reduced to 9 (less than half of the equivalent DFA which has 20 edges), thus achieving a remarkable compression.

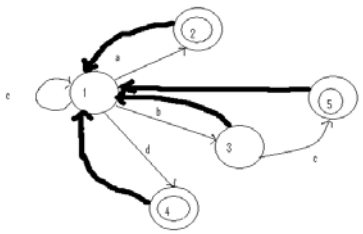


Figure 4. D^2FA

However, observing the graph in fig. 3, it is evident that most transitions for a given input lead to the same state, regardless of the starting state; in particular, adjacent states share the majority of the next-states associated with the same input chars. Then if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (because for each character they lead to the same set of states as 1). This means that state 2 can be described with a very small amount of bits. The result of what we have just described is depicted in fig.5 (except for the local transition set), which is the δFA equivalent to the DFA in fig. 3. We have 8 edges in the graph (as opposed to the 20 of a full DFA) and every input char requires *a single state traversal* (unlike D^2FA).

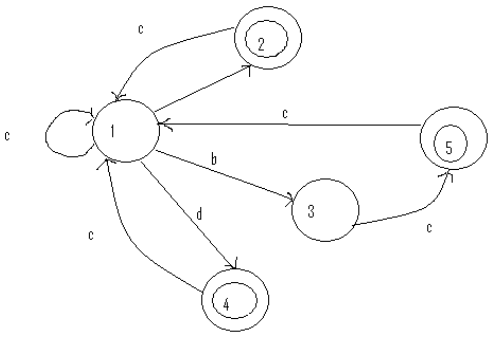


Figure 5. δFA for
Automata recognizing $(a+)$, $(b+c)$ and $(c*d+)$

B. The main idea of δFA

The idea of δFA comes from the following observations:
• a state is defined by its transition set and by a small value signaling if it is an accepting state;

• in a DFA, most transitions for a given input char are directed to the same state.

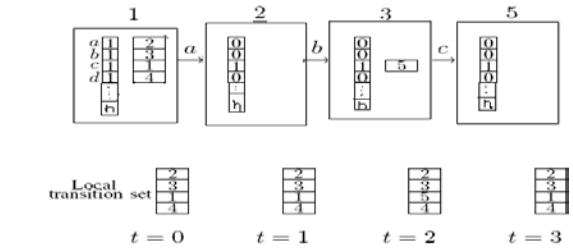
By elaborating upon the last observation, it becomes evident that most adjacent states share a large part of the same transitions. adjacent (or, better, "parent-child"1) states This requires, however, the addition of a supplementary structure that locally stores the transition set of the current state. The idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA in fig. 5 only stores the transitions that *must* be defined for each state in the original DFA.
In [18] we also proposed a new encoding scheme for transitions (named *Char-State* compression), which exploits the association of many states with a few input characters. Such a compression scheme can be efficiently integrated into the δFA algorithm, allowing a further memory reduction with a negligible increase in the lookup time.

C. Lookup

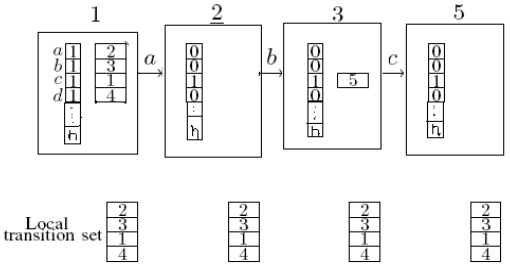
In the first step of the lookup process, the current state must be read with its whole transition set. Then it is used

to update the local transition set: for each transition defined in the set read from the state, we update the corresponding entry in the local storage. Finally, the next state is computed by simply observing the proper entry in the local storage. As obvious, the algorithm relies on wide memory accesses which are very common in DRAMs nowadays. The lookup algorithm requires a maximum of *C* elementary operations (such as shifts and logic AND or pop counts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases, the computational delay is negligible with respect to the memory access latency.

In fig. 6(a) we show the transitions taken by the *δ*FA in fig. 5 on the input string *abc*: a block represents a state and its internals include the transition set and a bitmap. The bitmap and the transition set have been defined during construction. We start Therefore we can store only the differences between(t = 0) in state 1 that has a fully-specified transition set. This is copied into the local transition set (below). Then we read the input char *a* and move (t = 1) to state 2, that specifies a single transition toward state 1 on input char *c*. This is also an accepting state (underlined in figure). Then we read *b* and move to state 3. Note that the transition to be taken now is not specified within state 2 but it is in our local transition set. Again state 3 has a single transition specified, that this time changes the corresponding one in the local transition set. As we read *c* we move to state 5 which is again accepting.



(a) δFA internals



(b) (δ^nFA)*

Figure 6. Automata internals: a lookup example

V. DELTA FINITE AUTOMATON: (δ^nFA)*

In the following section we describe *(δ^nFA)** main idea, the lookup process and its construction.

A. Main idea

The extension of *δ*FA can be explained by using same DFA [15],[18] shown in fig 3.From fig 5 the number of transition are reduced but it have some limits in compression. In example ,all the transitions for character *c* are specified (and hence stored) for all the 5 states, because of a single state 3 that defines a different transition. Because of strict rules in *δ*FA the transition set of a state is stored as the difference with respect to all its direct parents.

D^2FA have long default-transitions paths compresses better than a bounded D^2FA with *B*=2 [15], by definition of "parents" to "grandparents" (i.e., 2-N step neighbor nodes) the effectiveness of the *δ*FA approach increases because of the larger number of possibilities.

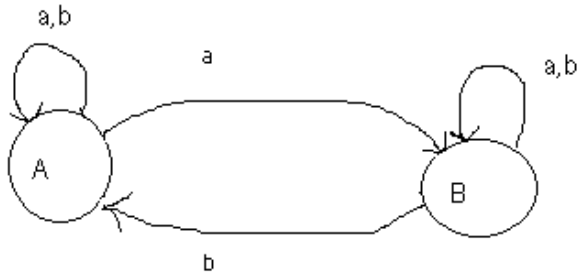


Figure 7. (δ^nFA)*

This concept does not provide better results in *δ*FA: for example, in fig. 5 defining the transitions for *c* as difference with respect to all the "grandparents" still would not allow to eliminate any new transition. Moreover this scheme would require to store 2 local transition sets (doubling the local memory needed).In fig 7 the state A can send char a to B and state B send char b to state A also we indicate loop as a.b in both states .

Lookup

The way to handle temporary transitions in*(δ^nFA)** have some slight variation here temporary transitions are valid within their state but they are not stored in the local transition set. Fig. 6(b) shows an example of the lookup process for a *(δ^nFA)**: the whole transition set of state 1 (where we start at time t = 0) is copied into the local transition set. Then by char *a*, we move (t = 1) to state 2 which does not specify any transition. When we read *b* (t = 2), we move to state 3, where a temporary transition (dashed box) is specified: this transition is valid only

within state 3. Finally (t = 3) we read c, take the temporary transition, and end up in state 5.

Algorithm Pseudo-code for the creation of the transition table of a δ^nFA from the transition table t of a δFA

```
1:    t2←t
2:    for all states s in δ^nFA do
3:      for all char c do
4:        if t[s,c] ≠ LOCAL_ TX then
5:          S4←{parents of s}
6:          if t[sj,c]  ∀ sj ∈ S4 are equal and specified
          then
7:            S1←{children of s}
8:            if ∃ sj ∈S1 s.t t[sj,c]==LOCAL_TX
          then
9.              break
10.           if ∃  sj ∈ S1 s.t t[sj,c]== t[S4,c] then
11.           S2← { parents of sj } \ s
12.         if  t[S2,c]==t[sj,c]= =t[S4,c]≠t[s,c] then
13.           t2[s,c]← TEMP_TX
14.           delete  t2[sj,c]
15.       else
16.           ∃ sj ∈ S1 s.t  t[sj,c]== t[S4,c]
17.       return 0;
```

C. Construction

The construction process of (δ^nFA)* use δFA to be constructed before and it is used as input. It work is based on the subsets of nodes where a transition for a given character can be defined as temporary. In fig 8 nodes are shown as divided into sets according to their parent-child relationships and their transitions.

The nodes have the same transition for a given char x share the same color ,sets $S1$, $S2$ and $S4$ all provide the same transition for char x, while $S3$ defines a different next state for x. If we set all the transitions for x in $S3$ as temporary, we can avoid storing the transition for x in $S1$.

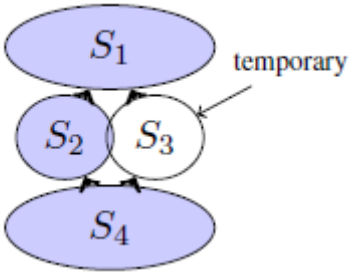


Figure 8. Schematic view of the problem. Same color means same properties. If the properties of $S3$ are set temporary, the ones in $S1$ can be avoided.

Dataset	Cisco30	Cisco50	Snort24	Snort31	Bro217
Del. ratio	97%	89%	100%	99%	99%
Temp. ratio	84%	76%	100%	98%	99%

Table 2 Simple VS. Optimal Approach: Ratio of deleted and Temporary Transitions.

In a real implementation, in order to recognize the nodes where a transition for a given character can be defined as temporary, for each char x of each state s, if the corresponding transition $t[s, x]$ in the δFA is stored (i.e., it is different from that $t[p, x]$ of all its parents) the following steps are required:

• a search is performed in all the children of s: whenever at least a child has the same transition $t[p, x]$ of its "grandparents", the second step follows;
• check all the other parents (except for s) of such a subset of children in order to check if they have the same transition $t[p, x]$;
• in this case, the transition $t[s, x]$ in s can be set as temporary and the process ends.

A few remarks (which ultimately result in constraints in the construction process) can be explained by referring to fig. 8(where the transitions for x in $S3$ are set temporary):

1) no state in $S4$ can have a temporary transition for x. The reason is simple: a temporary transition for x in the parents $S4$ means that such a transition does not modify the local transition table and therefore we have no way to "remember" the next-state when (after some hops) we reach the children $S1$

2) all children states in $S1$ must have specified transitions for x, because if the transitions in $S3$ are temporary and an un-specified transition exists in a state sj $S1$, the ultimate result is that $t[sj, x] = t[S4, x]$ while sj was meant to inherit $t[S3, x]$.

Hence, this process introduces some constraints and, as usual when dealing with constraints on graphs, this creates new problems: as described above, when setting a subset *y* of transitions as temporary, we must rely on some other transitions (the grandparents of *y*) to be non-temporary. This can be classified as a graph-coloring problem which is known to be NP-hard.

Oblivious construction: we construct the $(\delta^n FA)^*$ in a single run by observing all the transitions and setting all the transitions that satisfy the above-mentioned constraints as temporary. This solution is very fast because it does not explore the whole solution domain and simply gives up the idea of optimality. While this may appear unusual and is certainly non-optimal, it is however motivated by a number of experimental results (reported in the following section), where this approach does not differ significantly from the optimal setting (if ever reachable) in terms of transitions reduction. Moreover, notice that the optimal construction would require an exhaustive search of all the solution domain, thus questioning the advantages of the optimal setting.

VI. EXPERIMENTAL RESULTS

Regular Expression sets from IDS/IPS (Intrusion Detection System / Intrusion Prevention System) are taken from the real-world security devices of Snort, BRO and Cisco networks. These experimental results are reported.

The experiment is conducted to reduce the number of transitions as in δFA. The comparison between the δFA and $(\delta^n FA)^*$ are expressed in terms of deleted transitions. Tabular Column 2 shows the ratio of maximum number of deleted and temporary transitions. The values in the table shows that our $(\delta^n FA)^*$ approach is effective as it is reaching the maximum number of deleted transitions almost in all cases.

The comparison among δFA and $(\delta^n FA)^*$ (which include also the *Char-State* encoding scheme for further memory compression, as explained in [18]) and the most efficient previous solutions based on the performance is shown in Tabular column 3. For D^2FA and BEC-CRO, the code of *regex-tool* [10] is used, which builds a standard DFA. Different algorithms are used to reduce states and transitions. In D^2FA, the code runs with two different values of the bound *B* (i.e., 2 and ∞), which is a parameter that affects the structure size and the average number of state-traversals per character [15].

Tabular column 3(a) shows the memory compression, which is expressed as the ratio of number of deleted transitions and the original ones. Tabular column 3(b) shows the memory compression, which is expressed by considering the overall memory consumption, different structures and state sizes. Our algorithms achieve a degree of compression and it is comparable to that of D^2FA and BEC-CRO, while allowing for a higher lookup speed by preserving one transition per character. This is the main strength of our scheme, as it reduces lookup time by exploiting the adoption of wide memory accesses which are very common in DRAMs. The results shows that $(\delta^n FA)^*$ provides an improvement to δFA, since it requires a minimal change in the lookup algorithm.

.

Dataset	D^2FA		BEC CRO	δFA	δ^2FA	$(\delta^nFA)^*$
	DB=	DB=				
Snort24	98.92	89.59	98.7	96.3	96.8	98.88
Cisco30	98.84	79.35	98.7	90.8	92.0	95
Cisco50	98.76	76.26	98.6	84.1	86.1	93
Cisco10	99.11	74.65	98.9	85.6	86.9	90.6
Bro217	99.41	76.49	99.3	93.8	94.3	97.21

(a) Transitions reduction (%)

Dataset	D^2FA		BEC-CRO	δFA	δ^2FA	$(\delta^nFA)^*$
	DB=∞	DB=2				
Snort24	95.92	67.17	95.36	95.02	95.90	98.2
Cisco30	97.20	55.50	97.11	91.07	92.65	95.5
Cisco50	97.18	51.06	97.01	87.23	89.03	92.6
Cisco100	97.93	51.38	97.58	89.05	90.3	95.5
Bro217	98.37	53	98.23	92.79	93.4	99

(b)Memory compression (%)

Table 3 Compression of different Algorithms. In (B) the Results For ∂FA And δ^2FA include char-state compression

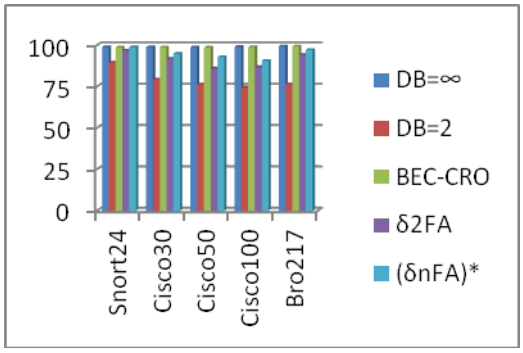


Figure 8. Mean number of memory Accesses

Fig. 8 shows the average number of memory accesses required to perform pattern matching through the compared algorithms. It is worth noticing that, while $(\delta^n FA)^*$ (just as δFA) needs about < 1.05 accesses (more

than 1 because of the integration with the *Char-State* scheme), the other algorithms require more accesses, thus increasing the lookup time.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an extension to δFA, a compressed representation for DFA. It takes an advantage of the second order dependence between states in a DFA, hence it is named as $(\delta^n FA)^*$. Also it reduces the number of transitions. As the adjacent (and 2-step neighbors) states are sharing common transitions, memory usage is reduced. It is achieved by simply storing the differences between them. Furthermore, it is orthogonal to previous solutions and allows higher compression rates. Since the $(\delta^n FA)^*$ requires a state transition per character only (as DFAs), fast string matching is allowed. The experimental results shows that $(\delta^n FA)^*$ is an effective and simple improvement to δFA both in terms of memory consumption and lookup speed. The future experiment can be performed on (δ(δnFA)*)* to reduce the memory consumption and time taken for transitions.

REFERENCES

[1] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proc. ACM CCS*, 2003, pp. 262–271.

[2] "Snort: Lightweight intrusion detection for networks," Source fire, Inc., Columbia, MD [Online]. Available: http://www.snort.org/

[3] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. IEEE INFOCOM*, 2007, pp. 1064–1072.

[4] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. ANCS*, 2006, pp. 81–92.

[5] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, 2008.

[6] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. ACM ANCS*, 2007, pp. 155–164.

[7] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[8] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, 2004, pp. 333–340.

[9] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM ANCS*, 2007, pp. 145–154.

[10] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT*, 2007, pp. 1–12.

[11] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979

[12] R. Motwani, "Average-case analysis of algorithms for matching and related problems," J. of the ACM, 41:1329--1356, 1994.

[13] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proc. of INFOCOM 2007*, May 2007.

[14] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. of ANCS '07*, pages 155–164. ACM.

[15] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. of SIGCOMM '06*, pages 339–350. ACM.

[16] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proc. of ANCS '06*, pages 81–92. ACM.

[17] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. of CCS '03*, pages 262–271.

[18] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. DiPietro. An improved dfa for fast regular expression matching. *SIGCOMM Computer Communication Review*, 38(5), 2008.

N.Kannaiya Raja received degree MCA from Alagappa University and ME from Anna University Chennai in 2007 joined assistant professor in various engineering colleges in Tamil Nadu affiliated to Anna University and has eight years teaching experience. His research work in deep packet inspection. He has been session chair in major conference and workshops in computer vision on algorithm, network, mobile communication, image processing papers and pattern recognition. His current primary areas of research are packet inspection and network. He is interested to conduct guest lecturer in various engineering in Tamil Nadu.

K Arulanandam received PhD doctorate degree in 2010 from Vinayaka Missions University. He has twelve years teaching experience in various engineering colleges in Tamil Nadu which are affiliated to Anna University and his research experience network, mobile communication tworks, image processing papers and algorithm papers. Currently working in Ganadipathy Tulasi's Jain Engineering College Vellore .

G.Ambika received degree B.Tech Information Technology from Anna University Chennai in 2008. Now pursuing ME Computer Science and Engineering in Arulmigu Meenakshi Amman College of Engineering Kanchipuram affiliated to Anna University Chennai.