Advanced Algorithms for Fast and Scalable Deep Packet Inspection

Sailesh Kumar
Washington University
sailesh@arl.wustl.edu

Jonathan Turner
Washington University
jon.turner@wustl.edu

John Williams
Cisco Systems
jwill@cisco.com

ABSTRACT

Modern deep packet inspection systems use regular expressions to define various patterns of interest in network data streams. *Deterministic Finite Automata* (DFA) are commonly used to parse regular expressions. DFAs are fast, but can require prohibitively large amounts of memory for patterns arising in network applications. Traditional DFA table compression only slightly reduces the memory required and requires an additional memory access per input character. Alternative representations of regular expressions, such as NFAs and Delayed Input DFAs (D^2FA) require less memory but sacrifice throughput. In this paper we introduce the *Content Addressed Delayed Input DFA* (CD^2FA), which provides a compact representation of regular expressions that match the throughput of traditional uncompressed DFAs. A CD^2FA addresses successive states of a D^2FA using their content, rather than a "content-less" identifier. This makes selected information available earlier in the state traversal process, which makes it possible to avoid unnecessary memory accesses. We demonstrate that such content-addressing can be effectively used to obtain automata that are very compact and can achieve high throughput. Specifically, we show that for an application using thousands of patterns defined by regular expressions, CD^2FAs use as little as 10% of the space required by a conventional compressed DFA, and match the throughput of an uncompressed DFA.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General – Security and protection (e.g., firewalls)

General Terms

Algorithms, Design, Security.

Keywords

DFA, regular expressions, deep packet inspection.

1. INTRODUCTION

Many network services now process packets based on payload content, in addition to the structured information found in packet headers. Forwarding packets based on content requires new levels of support in networking equipment. Traditionally, deep packet inspection has been limited to comparing packet content to sets of strings. Newer systems, are now replacing string sets with regular expressions, due to their increased expressiveness [Snort, bro, tippingPoint, Cisco]. Cisco, has even the regular expression based content inspection capabilities into its Internetworking Operating System (IOS) [21]. In addition, layer 7 filters based on regular expressions [30] are available for the Linux operating system. While flexible and expressive, regular expressions have traditionally required substantial amounts of memory, which severely limits their applicability in the networking context.

This paper builds on a new technique for parsing regular expressions using *Delayed Input Deterministic Finite Automata* (D^2FA). D^2FAs were introduced in [3] and use *default transitions* to reduce 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, we can replace the common transitions leaving one of the states with a default transition to the other. No state can have more than one default transition, but if the default transitions are chosen appropriately, the amount of memory needed to represent the parsing automaton can be dramatically reduced. Unfortunately, the use of default transitions also reduces throughput, since no input is consumed when a default transition is followed, but memory must be accessed to retrieve the next state. In this paper, we develop an alternate representation for D^2FAs that allows them to be both fast and compact. The remainder of this paper assumes familiarity with D^2FAs [3].

The remainder of the paper is organized as follows. Background on regular expressions and related work are presented in Section 2. Section 3 introduces CD^2FA. Details of CD^2FA construction are presented in Section 4. Section 5 presents optimizations and Section 6 presents the memory packing algorithm. Section 7 reports the experimental results and the paper ends with concluding remarks in Section 8.

2. BACKGROUND AND RELATED WORK

Deep packet inspection is becoming increasingly important as a means to classify and control traffic based on the content, applications, and subscribers. If current trends continue, it is likely that deep packet inspection will become embedded within the network core. Some of the well known Internet applications which use deep packet inspection are listed below:

- *Content-based traffic management and routing*, where packets are classified and processed based upon content. For instance, multimedia traffic originating from a privileged provider can be classified based upon its content and placed into a higher priority queue.
- Network intrusion detection systems (NIDS) generally scan the packet header and payload in order to identify a given set of signatures of well known security threats.
- Layer 7 switches and firewalls provide content-based filtering, load-balancing, authentication and monitoring. Application-aware web switches, for example, provide scalable and transparent load balancing in data centers.

Deep packet inspection often involves scanning every byte of the packet payload and identifying a set of matching predefined patterns. Traditionally, rules have been represented as exact match strings consisting of known patterns of interest. Naturally, due to their wide adoption and importance, several high speed string matching algorithms have been proposed [7,8,9,11,12,13,14,25,26,27,28,29].

In [1], Sommer and Paxson note that regular expressions might prove to be fundamentally more efficient as compared to exact-match strings when specifying attack signatures. Open source NIDS, Snort [5] and Bro [4] already use regular expressions to specify signatures. Commercial security products, such as TippingPoint X505 [20] and security appliances from Cisco Systems [21] also use regular expressions. Consequently, several specialized hardware-based regular expressions matching architectures have been proposed [15,16,17] and specialized commercial engines like RegEx from Tarari [22] report packet scan rates up to 4 Gbps. The recently proposed delayed input DFA (D^2FA) [3] enables a high degree of integration while using embedded memories rather than on-chip logic to store the automata.

Hardware based approaches are able to create compact automata by exploiting high degree of parallelism and memory bandwidth available on-chip. While these seem promising for ASIC devices, they are generally not as well suited to a more cost effective implementation approach that uses small on-chip lookup engines (implemented as either hardware or software) together with automata defined by tables in off-chip commodity memory. Performance of these systems is often limited by the memory bandwidth therefore it is critical that on-chip regular expression engines minimize the number of off-chip memory accesses. Memory bandwidth is especially critical in context of deep packet inspection because the number of memory accesses, is often comparable to the number of bytes in the packet payload.

A few recent proposals emphasize the use of commodity memories [32,33,10] however these often require very large memory, which limits their practical application. For example, Cisco commonly employs a gigabyte or more of memory to implement regular expression-based automata; the cost of memory often represents a sizeable fraction of the system cost. In order to implement regular expressions more economically and improve scalability in the number of rules, it is important to reduce the memory consumed without significantly increasing the number of memory accesses.

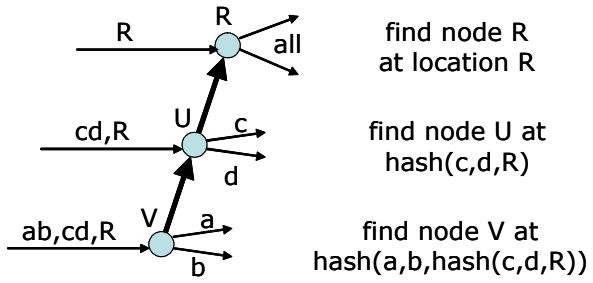


Figure 1. Content-Addressing

3. INTRODUCTION TO CD^{2}FAS

3.1 Content-Addressing

In a conventional DFA implementation, states are identified by numbers and these numbers are used to locate a table entry that contains information defining the given state. Content-addressed D^2FAs replace state identifiers with *content labels* that include part of the information that would normally be stored in the table entry for the state. The content labels can be used to skip past default transitions that would otherwise need to be traversed before reaching a labeled transition that matches the current input character. Using hashing, we can also use the content labels to locate the table entry for the next state.

We illustrate the idea of content addressing with the example shown in Figure 1. This figure shows three states of a D^2FA, *R*, *U* and *V*. The heavy-weight edges in the figure represent default transitions and *R* is the root of one of the trees defined by the default transitions. State *U* has labeled transitions for characters *c* and *d*, in addition to its default transition to *R*. State *V* has labeled transitions for characters *a* and *b*, in addition to its default transition to *U*. The arrows coming in from the left represent transitions from other states to states *R*, *U* and *V*. For each such predecessor state, we store a content label that includes the information shown in the figure, in addition to some auxiliary information that will be discussed later. The content label for transitions entering state *U* is *cd*,*R*. This label tells us that state *U* has outgoing transitions labeled by the characters *c* and *d*, and that its parent is *R*, which is the root of a default transition tree. The content label for transitions entering state *V* is *ab*,*cd*,*R*. This tells us that state *V* has outgoing transitions labeled by the characters a and *b*, and that its parent (in the default transition tree) has outgoing transi-

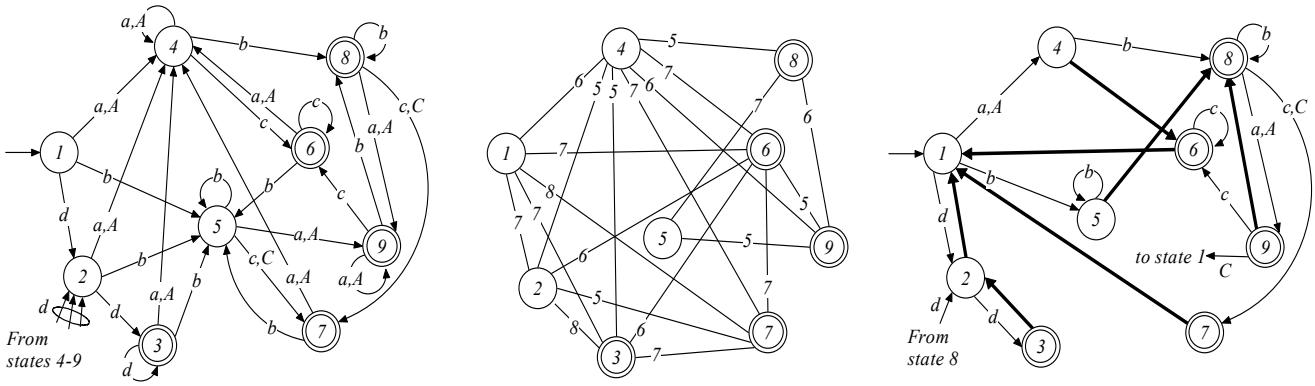


Figure 2. *a*) DFA recognizing patterns $[aA]^+b^+$, $[aA]^+c^+$, $b^+[aA]^+$, $b^+[cC]$, and dd^+ over alphabet {*a*, *b*, *c*, *d*, *A*, *B*, *C*, *D*} (transitions for characters not shown in the figure leads to state 1). *b*) Corresponding space reduction graph (only edges of weight greater than 4 are shown). *c*) A set of default transition trees (tree diameter bounded to 4 edges) and the resulting D²FA.

tions labeled by the characters *c* and *d*, and that its parent's parent is *R*, which is the root of a default transition tree.

Suppose that the current state of the D²FA is one of the predecessors of state *V* and that the current input character selects a content label for a transition to state *V* and that the *next* input character is *x*. While *V* is the next state, since *V* has no labeled transition for *x*, we would like to avoid visiting state *V* so that we can skip the associated memory access. Similarly, we would like to avoid visiting state *U*, since it also has no labeled transition for *x*. Assume that we have a hash function *h* for which $h(cd,R)=U$ and for which $h(ab,U)=V$. Given the content label *ab*,*cd*,*R* (which is stored at the predecessor state), we can determine that neither our immediate next state (*V*) nor its parent (*U*) has an outgoing transition for *x*. Hence, we can proceed directly to *R*. If on the other hand, the next input character is *c* or *d*, then we can proceed directly to *U* by computing $h(cd,R)$. Similarly, if the next input character is *a* or *b*, we can proceed directly to *V* by computing $h(ab,h(cd,R))$.

Summarizing, we associate a content label with every state in a D²FA. Each label includes a character set for the state and each of its ancestors in the default transition tree, plus a number identifying the state at the root of the tree. We augment the content label with a bit string that indicates which of the states on the path from the given state to the root of its tree are matching states for the automaton. In our examples, we use underlining of the character set for a given state to denote that the state is a matching state. So, if state *U* in our example matched an input pattern of interest, we would write the content label for *U* as *cd*,*R* and the content label for *V* as *ab*,*cd*,*R*. Content labels are stored at predecessor states, and hashing is used to map the labels to the next state that we need to visit.

3.2 Complete Example

We now turn to a more complete example. Figure 2*a* shows a DFA that matches the patterns $a[aA]^+b^+$, $[aA]^+c^+$, $b^+[aA]^+$, $b^+[cC]$ and dd^+. Part *b* of the figure shows a corresponding space reduction graph and part *c* shows a D²FA constructed using this space reduction graph. The default transitions are shown as bold edges. Note that states 1 and 8 are roots of their default transition trees and that the longest sequence of default transitions that can be followed without consuming an input character is 2. If we use the D²FA to parse an input string, the number of memory accesses can be as large as three times the number of characters in the input string. Consider a parse of the string *aAcba*. Using the original DFA, we can write this in the form

$$1 \xrightarrow{a} 4 \xrightarrow{A} 4 \xrightarrow{c} \underline{6} \xrightarrow{b} 5 \xrightarrow{a} \underline{9}$$

Here, the underlined state numbers indicate matching states. Using the D²FA, we the parse of the string will be

$$1 \xrightarrow{a} 4 \xrightarrow{A} 614 \xrightarrow{c} 6\underline{6} \xrightarrow{b} 15 \xrightarrow{a} 8\underline{9}$$

Here, we are showing the intermediate states traversed by the D²FA. To specify the CD²FA, we first need to write the content labels for each of the states. These are listed below.

1. 1
2. *d*,1
3. –, *d*,1
4. *b*,*c*,1
5. *b*,8
6. *c*,1
7. –,1
8. 8
9. *cC*,8

Note that since states 3 and 7 have no labeled outgoing transitions in the D²FA, their content labels include empty character sets that are indicated by dashes. The dash in the content label for state 3 is underlined to indicate that state 3 is a matching state.

The complete representation of the CD²FA is shown below. For each state, we list the content labels associated with the character for which there is an outgoing labeled transition from the state. Note that only states 1 and 8 have labeled outgoing transitions for every character and states 3 and 7 have no labeled transitions.

1. *a*: *b*,*c*,1
   b: *b*,8
   c: 1
   d: *d*,1
   A: *b*,*c*,1
   B: 1
   C: 1
   D: 1
2. *d*: –,*d*,1
3.
4. *b*: 8
5. *b*: b,8
6. *c*: *c*,1
7.
8. *a*: *cC*,8
   b: 8
   c: –,1
   d: 1
   A: *cC*,8
   B: 1
   C: –,1
   D: 1
9. *c*: *c*,1
   C: 1

Let's use this representation to parse the input string *aAcba*. In state 1, we find that the content label for the first input character (*a*) is *b*,*c*,1. This tells us that the next state is $h(b,h(c,1))=4$, where h is an assumed hash function that maps content labels to the original state numbers. Since state 4 has no defined transition for the next input character (*A*), we proceed directly to state 1, skipping intermediate states $h(b,h(c,1))=4$ and $h(c,1)=6$. We are now ready to process *A*. We see that its content label is also *b*,*c*,1. In this case however, the parent of the next state does have a defined transition for the next character (*c*), so we proceed to that state, which we find by computing $h(c,1)=6$. In state 6, we process character *c* using the content label *c*,1. Since the label indicates that the next state $h(c,1)=6$ is a match state, we make note of the match, but since state 6 has no labeled transition for the next input character (*b*), we proceed to state 1. Continuing in this way produces the parse

$$1 \xrightarrow{a} 461 \xrightarrow{A} 46 \xrightarrow{c} \underline{6}1 \xrightarrow{b} 58 \xrightarrow{c} \underline{9}$$

If we compare this parse with the parse for the D^{2}FA, we see that the transitional states are simply shifted to the left, reflecting the fact that the CD^{2}FA skips past these states as it processes each input character.

3.3 Details and Refinements

In our examples, we have assumed the existence of a hash function that we could use to map content labels to state numbers. Since the numbers used to identify states are arbitrary, any hash function that produces distinct state numbers for each content label can be used. Note that hash values are only needed for states that are not roots of their default transition trees. The root states can simply be numbered sequentially and since there are only a few such states; the number of bits needed to represent them can also be small.

There are some tricks that can be used to ensure uniqueness of the hash values computed for each content label. Specifically, for each character set in a content label, the order in which the characters are listed is arbitrary. Consequently, we can change the order of the characters in order to avoid conflicting hash values. If content labels are packed into words of fixed size, we can sometimes pad label shorts by repeating some characters, thus changing the hash value without changing the label's meaning. In some cases it may be necessary to augment the hash values with additional bits to ensure uniqueness. We refer to such extra bits as *discriminators*. As we report later, we have found that very few discriminator bits are needed in practice.

To find the content label for the current input character, we need to know where it appears in the list of content labels for the current state. States that are at the roots of their default transition trees have content labels for every symbol in the alphabet, so we can use simple indexing to find the appropriate label in this case. For states that are not roots, we have content labels only for those characters for which there is a labeled outgoing transition. The content label used to reach the state tells us which characters the state has outgoing transitions for. If our next character is the *i*-th one in the list of characters found in the content label at the predecessor state, then the next content label we need to consult will be at position *i* in the list of content labels for the current state. So, given the starting location of the list of content labels for the state, we can use indexing to find the specific content label of interest, without having to scan past the other content labels defined for the current state.

3.4 Memory requirements of a CD^{2}FA

The memory required for a CD^{2}FA depends directly upon the D^{2}FA from which it is constructed and the size of the resulting content labels. If we let $a(s)$ denote the number of ancestors of state s in its default transition tree (including s) and $c(s)$ denote the number of characters for which s and its non-root ancestors have labeled transitions, then we need at least $c(s)b + a(s) + r$ bits to represent the content label for state s, where b is the number of bits needed to represent a character and r is the number of bits needed to represent the identifier for a root. In addition, to identify which characters in the content label correspond to transitions from which ancestors of state s, we can use an additional bit per character, giving $c(s)(b+1) + a(s) + r$. Additional bits may be needed for discriminators, which we ignore for now. Note that if we require that content labels be packed into a fixed size word, then the depth of the default transition trees will be limited by the word size, since both $c(s)$ and $a(s)$ get large for states that are far from the roots of their trees.

If we allow the content labels to have variable lengths, then we can potentially reduce the overall space requirement, since nodes close to the roots of their trees will have smaller content labels. If the nodes with larger content labels have relatively few incoming transitions, then the impact of these larger content labels on the overall memory requirements will be limited. Of course, allowing variable length content labels also means that we have to include length information in content labels, adding to the space needed to represent each label. In our experimental results, we allow content labels of two sizes: 32 bits and 64 bits. This adds approximately $c(s)$ bits to each content label (de-

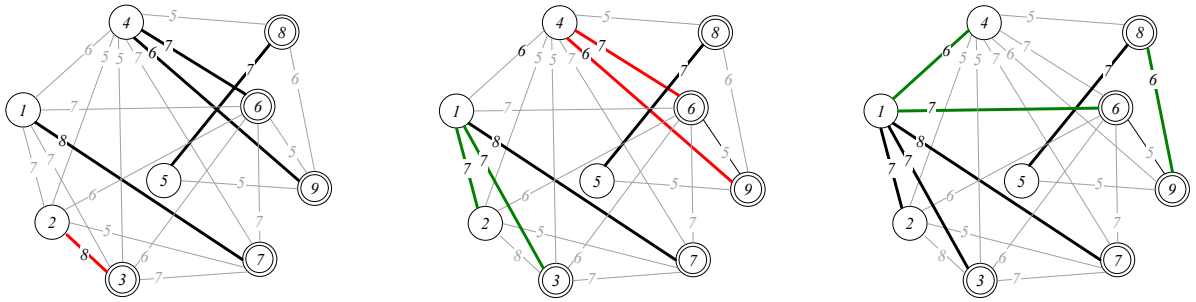


Figure 3. *a*) A set of default transition trees created by Kruskal's algorithm with tree diameter bounded to 2. *b*) After dissolving tree 2-3 and joining its vertices to root vertex 1. *c*) After dissolving tree 9-4-6 and joining its vertices to root vertices 1, 1 and 2.

tails in section 5.2), but does lead to a significant overall space savings.

This discussion makes it clear that the problem of constructing D^{2}FAs that lead to small CD^{2}FA is non-trivial. As shown in [3], bounding the default paths to a small constant in general leads to larger D^{2}FAs than if we allow the depth to become large. However, small depth D^{2}FAs will have relatively small content labels. The use of variable length content labels adds another dimension to the problem, since it makes it desirable for states with many incoming transitions to have small content labels. Hence, it makes sense to position these states close to the roots of their default transition trees. Unfortunately, we don't know in advance, which states will have large numbers of incoming transitions, since the introduction of default transitions can dramatically change the number of labeled transitions entering a state. In the next section, we focus on a simple heuristic approach, which we have found produces good results experimentally.

4. CONSTRUCTION OF GOOD D^{2}FAS

In this section, we attempt to construct D^{2}FAs, which lead to compact CD^{2}FAs. We need to ensure that size of content labels are bounded so that they can be fetched in a single memory access (in our experiments, we bound them to 64-bits), hence we only consider edges of the space reduction graph, whose weights are sufficiently large (in our case larger than 252). Our general objective is to create compact CD^{2}FAs and not compact D^{2}FAs (which can be created by solving a maximum weight spanning tree problem [3]), therefore we take proper care that default paths do not grow too deep and content labels do not become too big.

To meet these objectives, we have developed simple yet effective heuristic called *CRO*, which runs in three phases, called *creation*, *reduction* and *optimization*. Creation phase creates a set of initial default transition trees whose default paths are bounded to one default transition. Reduction phase reduces the number of trees by iteratively *dissolving* and *merging* trees, while maintaining the default length bound of one transition. Optimization phase attempts to reduce the space requirements further, by allowing some default paths to grow longer than one default transition.

4.1 Creation phase

During the creation phase, a collection of trees on the space reduction graph is constructed using a variant of Kruskal's algorithm [24]. We refer to these trees as spanning forest; diameter of all trees in this forest is bounded to two edges, thus the default paths contain a single default transition. Kruskal's algorithm examines edges in order of decreasing weight; if adding an edge to a tree neither creates a cycle nor violates the diameter bound of two then it is added to the spanning forest. The variation in the Kruskal's algorithm is aimed to ensure that states, where more labeled transitions enter, are more likely to become tree roots. Therefore, an *in-degree*, equal to the total number of labeled transitions entering a state, is assigned to all states. As Kruskal's algorithm progresses, from among all unexamined edges of equal weights, the ones, whose joining vertices have higher in-degrees are examined earlier than examining those edges whose joining vertices have relatively lower in-degrees.

In Figure 3*a*, we illustrate the outcome of creation phase, when applied to the space reduction graph shown in Figure 2*b*. Four default transition trees form, three of which contain a single edge. In general, creation phase forms a large number of trees which contain a single edge because, once such a tree forms, they can not be linked to other trees containing one or more edges (because diameter bound is 2). Consequently, the weight of the forest can be increased further by reducing the number of trees. For instance, if, instead of selecting the edge 2-3 in Figure 3*a*, we select edges 1-2 and 1-3, then the weight can be increased by 6, while maintaining the diameter bound. Therefore, creation phase follows with a reduction phase, which reduces the number of trees.

4.2 Reduction phase

During reduction phase, the number of trees is reduced in an attempt to increase the weight of the spanning forest. Trees in the current forest are repeatedly examined in an order of decreasing weight (sum of weight of all edges in the tree). For any tree under examination, it is first dissolved and all its edges are removed from the forest. After-

wards, each vertex u of the dissolved tree is joined to the root vertex r of one of the tree among all trees in the forest, so that edge (u, r) has the highest weight. If the result of dissolving and reconnecting the vertices does not lead to an increase in the weight of the forest then the dissolved tree is restored. Reduction phase stops when the forest remains unaffected after a pass of examination of all trees in it.

Outcome of the reduction phase is illustrated in Figure 3*b* and 3*c*. Initially, tree 5-8 is examined, however it is not dissolved because dissolving it and connecting its vertices to one of the tree roots doesn't increase the weight of the forest. Afterwards, tree 2-3 is dissolved and vertices 2 and 3 are joined to the root vertex 1. This increases the weight of the forest by 6. Thereafter, tree 4-6-9 is dissolved, and its vertices 4, 6, and 9 are joined to root vertices 1, 1, and 8 respectively. The weight again increases by 6. None of the two remaining trees can be dissolved further, therefore reduction phase stops.

Reduction phase, in this instance, has reduced the number of trees from 4 to 2 and increased the weight of the forest from 36 to 48. Thus, the total number of labeled transitions in the D^{2}FA has been reduced from 36 to 24. In large automata, reduction phase is much more effective in reducing the number of default transition trees and therefore the total number labeled transitions in a D^{2}FA.

CD^{2}FA synthesized immediately after reduction phase are generally compact as *i*) there is reduced number of labeled transitions in the D^{2}FA and *ii*) all default paths are bounded to a single default transition leading to compact content labels. However, even more compact CD^{2}FA can be created by allowing longer default paths for certain states, specifically the states where not many labeled transitions enter. Optimization phase carries out these optimizations, where diameter of the certain parts of the trees is expanded.

4.3 Optimization phase

Prior to the optimization phase, a CD^{2}FA is constructed and content labels are associated with all labeled transitions. At this point, some states may have many labeled outgoing transition because their default paths are bounded to single default transition. We may reduce the number of labeled transitions at these states by allowing them to have longer default paths. This, however, may increase the size of content label of transitions entering into those states, as labels associated with those transitions will store all characters for which transitions are defined at all states along the default path. Therefore, it is important to selectively increase the default path of only those states which results in an overall space reduction.

To accomplish this, optimization phase proceeds with an assignment of *in-degree* (size of content label of transitions entering into the state) and *out-degree* (size content label of transitions leaving the state) to all states. The eligible candidates for the default path expansion are those states which have high *out-degree* and low *in-degree*. Therefore, states are repeatedly examined in decreasing order of their (*out-degree* – *in-degree*) values. For a state under examination, a new default state from among all states, whose default path contains a single default transition, is evaluated. If one such default state results in an overall reduction in the memory, then it becomes the new default transition of the examined state. The time needed to examine a state is $O(n)$, thus the time to once examine all n states is $O(n^2)$.

After all states are examined once, default paths contain up to two default transitions. The procedure is repeated until a pass doesn't result in any reduction in the total memory. Note that during a pass, default paths grow by at most one default transition. In practice, we found that the algorithm stops after 1-2 passes, thus the resulting default paths contain at most 2-3 default transitions and the asymptotic run time of optimization phase remains $O(n^2)$.

5. OPTIMIZING CONTENT LABELS

In this section, we present optimizations to compactly encode CD^{2}FA content labels. We begin these with an attempt to reduce number of symbols in the alphabet.

5.1 Alphabet reduction

A CD^{2}FA consists of root states, which do not have a default transition, and non-root states, which have a valid default transition. Even though, root states have labeled transitions defined for all characters in the alphabet, a large fraction of these leads to the same next state. We refer to the most frequently occurring "next state" from any given state as its *common transition*. If we let A denote the original alphabet, set $C(s)$ denote the characters, for which common transitions are present at a root state s, then its alphabet can be reduced to A-$C(s)$, if we explicitly store the common transition of the state. In general, alphabet of the root states can be reduced to $A - \bigcup_{s \in \text{root states}} C(s)$. For example, root states 1 and 8 of the D^{2}FA in Figure 2*c*, have common transitions (over characters B and D) leading to state 1. Note that these transitions are not explicitly shown in the figure, assuming that all transitions which are not explicitly shown in the figure leads to state 1. Once we remove the characters for these common transitions from the alphabet, it can be reduced to $\{a,b,\text{c},\text{d},A,C\}$.

It turns out that in all CD^{2}FAs we consider in our experiments, the reduced alphabet of the root states contains less than 128 characters. Moreover, even though there are up to a thousand root states, there are less than 64 distinct common transitions at these states; thus, we only need a 64 entry table to store the content labels associated with the common transitions. We also need to associate each root to one of the table entries, which can be done efficiently by appropriately numbering the root states. For instance if all

root states with identical common transition are assigned a series of contiguous integers, then we only need to associate the first and last integer value to the common transition.

The second step is to reduce the alphabet of non-root states, which have a small number of labeled transitions and a default transition. If $L(s)$ is the set of characters for which labeled transitions are present at a non-root state s, then the alphabet of non-root states can become $\bigcup_{s \in non-\text{root states}} L(s)$. Using this procedure, alphabet of the non-root states of the D^{2}FA shown in Figure 2*c* can be reduced to $\{b,c,d,C\}$.

If we take the union of the reduced alphabet of root and non-root states, the resulting set (referred as A_r) still contains much fewer characters than the ASCII alphabet, thus characters of the reduced alphabet may require fewer bits to represent. For instance we were able to represent them by 7-bits in our experiments, as A_r contained between 64 and 128 characters. In order to translate a character from ASCII alphabet to the reduced alphabet, an *alphabet translation table* is needed, which contains 256 entries and is indexed by the ASCII character. Entries, which correspond to the characters in the reduced alphabet, contain a 7-bit translated character, while entries, for which a character is not present in the reduced alphabet, contain a special symbol. This table requires less than 256 bytes and therefore can be easily stored either in the data cache or in the instruction cache via a function call.

5.2 Optimizing content label of non-root states

Content labels of non-root states may have variable length, which depends upon the number of labeled transitions leaving the state and its non-root predecessors. In this paper, we intend to restrict the content labels to two words (8-bytes), so that they can be fetched in a single memory access[1]. Thus, a content label may require 3 additional bits to store its length. We can perform an optimization by considering that memories often allow addressing at 4-byte boundaries; in other words, memory is organized as 4-byte words, in which case content labels will either be one or two words long, and a single bit will be sufficient to store its length.

When content labels are variable length, a complication may arise, as we need to know, where the content label for an input character appears in the list of content labels for the current state. In order to appropriately index this list, with the content label of each state, we need to store the length of the content labels of the states where its labeled transition enters. Thus, the content label of a state s with $c(s)$ labeled transitions requires $c(s)$ additional bits. As we have already mentioned, we need two additional bits per content label in the list, one to indicate whether it associates with a match and another to indicate the depth of the associated next state, in its default transition tree.

Consider an example, where we seek to store the list of content labels for state 9 of the CD^{2}FA in Figure 2*c*. State 9 has 2 labeled transitions, one leading to state 1 on input *C* and another leading to state 6 on input *c*. If we assume that that content label of the first transition is 1 word long, while the second is 2 words, then the content label of state 9 (more specifically of labeled transitions entering state 9), will be $\underline{c}_1^2\underline{C}_0^1,8$; here we indicate length of the content label as a superscript and the depth (in its default transition tree) of the next state as a subscript. The resulting memory structure is shown in Figure 4; state 9 requires total 3 memory words at an address determined by applying a hash on its content label (we discuss more about hashing in section 6).

With ASCII alphabet, (8-bit characters), content label for a state will require 11-bits per labeled transition it has, plus $\lceil \log_2 t \rceil$ bits to represent the root of its default transition tree (t is the number of default transition trees). In our experiments, we reduce the alphabet to fewer than 128 characters, thus, 7-bits represent a character, which enables us to use only 10-bits per labeled transition. Also there are fewer than a thousand root states, thus 10-bit are enough to represent them. Therefore, content label of a state, which has up to 2 labeled transitions, fits in a 4-byte word, while states with between 3 and 5 labeled transitions require two words. Note that, we only allow a state to have up to five labeled transitions, thus, we only consider those edges of the space reduction graph, whose weight is higher than 251.

5.3 Numbering root states

With only $\lceil \log_2 t \rceil$ bits to represent root states, transitions leaving root states are stored in a two dimensional table with t rows and $|A_r|$ columns. The table is indexed using the root state number as row index and the input character as the column index. Each cell of the table is two words long (even though content label of many transitions may be just 1 word long); thus a root state requires $2|A_r|$ memory words.

6. MEMORY PACKING

When we introduced CD^{2}FA, we assumed that there exists a hash function that maps content labels to the original state numbers. In this section, we present algorithms to devise such mapping. While associating state numbers to content labels, we are interested in not only associating unique numbers but also such numbers that can be directly used as an index into the memory. Thus, we would like to associate a unique memory address to the content label of each state, so that the list of content labels for all labeled transitions leaving the state is stored at that address. This will truly enable us to require a single memory access per input character. Throughout this section, we refer to the state number as the memory address where it is stored and storing a state means storing the content labels for its labeled transitions. We focus on storing non-root states, as

[1] These schemes can be easily extended to memory technologies, where minimum access size is different from 8-bytes.

We focus on storing non-root states, as root states are simply stored as a two dimensional table.

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 [2] 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.

We present a novel method 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 $O(n^{3/2})$, where n is the number of states. Our method proceeds by forming groups of states so that states with identical memory requirement belong to the same group. Since we allow a non-root state 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. Note that, in a 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.

Afterwards, our algorithm handles a group at a time and, as described below, stores its states into its memory region.

6.1 Packing problem formulation

Let there are n states in a group and each state requires s 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 codomain = [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).

Our 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

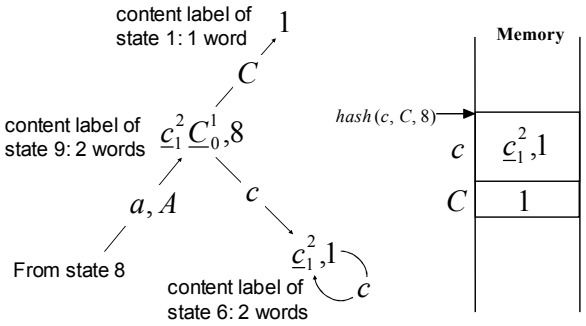


Figure 4. Storing list of content labels for state 9 in memory

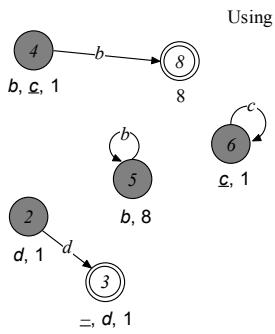
out changing their meanings. *a*) The simplest way is to modify the value of *discriminator*. *b*) An alternative is to change the order in which characters appear in the content label; thus a content label with t characters can have factorial t different possible names. *c*) In fixed size word length 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 naïve mapping may arbitrarily rename them whenever a collision occurs. We develop a more systematic approach to select the appropriate names.

Our 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 $G = (V_1+V_2, E)$ is constructed, where vertex set V_1 corresponds to the n content labels and V_2 the m state numbers. Edge set E contains all edges (u, v) such that $u \in V_1$, $v \in V_2$ 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 pairwise 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.

State	Content labels of transitions entering the state	size of content label
1	1	1
2	*d*, 1	1
3	=, *d*, 1	1
4	*b*, *c*, 1	1
5	*b*, 8	1
6	*c*, 1	1
7	–, 1	1
8	8	1
9	*cC*, 8	1



Using 1-bit discriminator in a content label

State	*hash* (discriminator, content label)
4	*hash* (0, *b*, *c*, 1) = 2
	hash (1, *b*, *c*, 1) = 4
2	*hash* (0, *d*, 1) = 1
	hash (1, *d*, 1) = 2
5	*hash* (0, *b*, 8) = 1
	hash (1, *b*, 8) = 4
6	*hash* (0, *c*, 1) = 2
	hash (1, *c*, 1) = 3

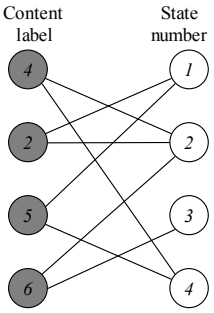


Figure 5. *a*) Content labels of states of the CD²FA shown in Figure 2. *b*) Non-root states requiring one word to store the content labels associated with their labeled transitions. *c*) Candidate content labels (using 1-bit discriminators) and the resulting candidate state numbers. *d*) Corresponding bipartite graph.

6.2 An illustrating example

Before presenting the analysis of our memory packing, we consider a simple example to illustrate the basic ideas. We consider the CD²FA shown in Figure 2*c*. There are 9 states, and the content labels of labeled transitions entering these states are shown in Figure 5*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 5*b*, each requires one word; therefore these are packed in a memory region containing 4 or more words.

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 5*c*. The resulting bipartite graph is shown in Figure 5*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.

6.3 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 [18].

***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.

The problem of finding perfect matching in such bipartite graphs is well studied. In [23], 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 $m > n$), [34] shows 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 CD²FA; this can be easily achieved even without using discriminators. As expected, in our experiments, we found a perfect matching in all CD²FAs without using memory over-provisioning or employing the discriminators.

Table 1. Our representative regular expression groups

Source	# of regular expressions	Avg. ASCII length of expressions	% expressions using wildcards (*, +, ?)	% 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

7. EXPERIMENTAL EVALUATION

In order to evaluate the effectiveness of a CD^{2}FA, we performed experiments on regular expression sets used in a wide variety of networking applications. Our most important dataset are the regular expression sets used in deep packet inspection appliances from Cisco Systems [19], which contains more than 750 moderately complex expressions. We also considered the regular expressions used in the Snort and Bro NIDS, and in the Linux layer-7 application protocol classifier. Snort contains more than thousand and half expressions, although, they don't need to be matched simultaneously. An effective way to implement Snort rules is to identify the expressions for each header rule and then group the expressions corresponding to the overlapping rules (set of header rules a single packet can match). We use this approach. For Bro, we present results for the HTTP rules, which contain 648 regular expressions.

As the first step to construct compact DFA, we used the set splitting technique proposed in [10]. We created 10 sets of Cisco rules (Cisco, however uses slightly different grouping, about which we are not fully aware); there were a total of 180138 states, and each DFA had less than 64K states. We split the Linux expressions into 3 sets with a total 28889 states. For Snort rules, we present results for header rule "`tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS`"; it contains 22 expressions, which were split into four sets. Bro regular expressions were simple therefore we did not split them. Our representative regular expression groups are summarized in Table 1.

We applied CRO algorithm on these regular expression groups to create CD^{2}FAs. In Table 2, we report the outcome of the algorithm after every phase, and also report the number of trees in the D^{2}FA, total number of labeled transitions, and memory needed by the CD^{2}FA. We also report the size of the reduced alphabet. While reduction phase is most effective in reducing memory, alphabet reduction also reduces memory by nearly two times. It is clear that, memory reduction achieved by CD^{2}FA, constructed from the CRO algorithm, is between 2.5 to 20 times, when compared to a table compressed DFA. If we compare CD^{2}FA to uncompressed DFA (which is a fair comparison because a CD^{2}FA matches an uncompressed DFA in terms of throughput), the memory space reductions are much higher, between 5 to 60 times.

While CD^{2}FAs match uncompressed DFAs in terms of throughput, in a practical system with an on-chip cache, a CD^{2}FA may surpass a DFA by achieving higher cache hits due to its smaller memory footprint. In Figure 6, we report the throughput results, where we have performed a trace driven cache based memory model simulation using Dinero IV simulator [31]. In order to create near worst-case conditions for a cache, the input data stream contained a high concentration of matching patterns (around 10% matches), which resulted in very low spatial locality in automata traversal. Even under these conditions, we found that cache hit rates were moderately good (25-50%), enough to improve the throughput. Hit rates of CD^{2}FA were noticeably higher (>60%) as it had much smaller memory footprint. Hence its throughput is also much higher. Note that the throughput of a table compressed DFA is much lower as it requires more than one memory access per input character.

Table 2. CD^{2}FA constructed after each phase of the CRO algorithm. Last column is the ratio of memory size of a CD^{2}FA and that of a table compressed DFA (DFA$_{TC}$)

Dataset	Original DFA				CD^{2}FA										size of CD^{2}FA ÷ size of DFA$_{TC}$
					After creation phase			After reduction phase			After optimization phase and alphabet reduction				
	# of states	# distinct transitions	Memory (MB)		# of trees	# of transitions	Memory (MB)	# of trees	# of transitions	Memory (MB)	# of trees	# of transitions	Memory (MB)	Alphabet size	
			No compression	With table compression											
Cisco590	17713	1537238	9.07	6.23	4227	1099809	8.87	243	117743	0.80	243	62043	0.39	98	**0.062**
Cisco103	21050	1236587	10.77	9.56	4617	1205978	9.72	684	253239	1.87	684	122679	0.86	106	**0.089**
Cisco7	4260	312082	2.18	1.14	838	220705	1.76	194	59077	0.44	194	32842	0.23	126	**0.201**
Linux56	13953	590917	7.14	3.62	1741	459215	3.73	266	156485	1.17	266	85444	0.61	123	**0.168**
Linux10	13003	962299	6.65	3.35	3361	870623	7.27	994	382464	3.01	994	183237	1.48	118	**0.441**
Snort11	37167	441414	19.03	3.55	3024	806790	6.31	257	188913	1.28	257	65629	0.36	37	**0.101**

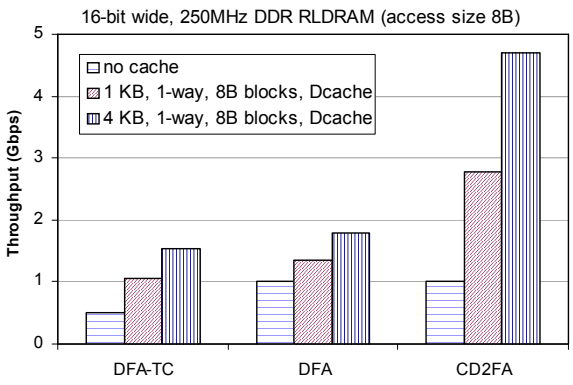


Figure 6. Throughput results on Cisco rules, without and with data cache. Table compressed DFA (DFA-TC), uncompressed DFA and CD^{2}FA are considered and the Input data stream results in a very high matching rate (~10%).

8. CONCLUDING REMARKS

In this paper we introduce the Content Addressed Delayed Input DFA (CD^{2}FA), which provides compact representation of regular expressions. A CD^{2}FA is built upon the recently proposed delayed input DFA (D^{2}FA), whose state numbers are replaced with content labels. The content labels compactly contain information which are sufficient for the CD^{2}FA to avoid any default traversal, thus avoiding unnecessary memory accesses and hence achieving higher throughput. While a CD^{2}FA requires number of memory accesses equal to those required by an uncompressed DFA, in systems with a small data cache, CD^{2}FA surpasses uncompressed DFAs in throughput, due to their small memory footprint and high cache hit rate. We find that with a modest 1 KB data cache, CD^{2}FA achieves two times higher throughput as compared to an uncompressed DFA, and at the same time requires only 10% of the memory required by a table compressed DFA. Consequently, CD^{2}FAs can implement regular expressions much more economically and improve throughput and scalability in the number of rules.

9. ACKNOWLEDGMENTS

We are grateful to Will Eatherton for providing us the regular expression sets used in Cisco security appliances. This work was supported by the NSF Grants CNS-0325298 and URP grant from Cisco Systems.

REFERENCES

[1] R. Sommer, V. Paxson, “Enhancing Byte-Level Network Intrusion Detection Signatures with Context,” ACM conf. on Computer and Communication Security, 2003, pp. 262--271.

[2] J. E. Hopcroft and J. D. Ullman, “Introduction to Automata Theory, Languages, and Computation,” Addison Wesley, 1979.

[3] S. Kumar et al, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection", in ACM SIGCOMM'06, Pisa, Italy, September 12-15, 2006.

[4] Bro, http://www.icir.org/vern/bro-info.html

[5] M. Roesch, “Snort: Lightweight intrusion detection for networks,” Systems Administration Conference (LISA), November 1999.

[6] S. Antonatos, et al, “Generating realistic workloads for network intrusion detection systems,” In ACM Workshop on Software and Performance, 2004.

[7] A. V. Aho, M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” Comm. of ACM, 18(6):333–340, 1975.

[8] B. Commentz-Walter, “A string matching algorithm fast on the average,” Proceedings of ICALP, pages 118–132, July 1979.

[9] S. Wu, U. Manber,” A fast algorithm for multi-pattern searching,” Tech. R. TR-94-17, Univ of Arizona, 1994.

[10] Fang Yu, et al., “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection”, UCB tech. report, EECS-2005-8.

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” IEEE Infocom 2004, pages 333--340.

[12] L. Tan, and T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” ISCA'05.

[13] I. Sourdis et al, “Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching,” FCCM, 2004, pp. 258–267.

[14] S. Yusuf and W. Luk, “Bitwise Optimised CAM for Network Intrusion Detection Systems,” IEEE FPL 2005.

[15] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” IEEE FCCM, Rohnert Park, CA, April 2001.

[16] C. R. Clark and D. E. Schimmel, “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” In 13th FCCM conference.

[17] J. Moscola, et al, “Implementation of a content-scanning module for an internet firewall,” IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, USA, April 2003.

[18] Phillip Hall, "On representatives of subsets," J. London Math. Soc.,vol. 10, pp. 26--30, 1936.

[19] Will Eatherton, John Williams, “An encoded version of reg-ex database from cisco systems provided for research purposes”.

[20] TippingPoint X505, www.tippingpoint.com/products_ips.html

[21] Cisco IOS IPS Deployment Guide, www.cisco.com

[22] Tarari RegEx, www. tarari.com/PDF/RegEx_FACT_SHEET.pdf

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

[24] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” Proc. of the American Mathematical Society, 7:48-50, 1956.

[25] N.J. Larsson, “Structures of string matching and data compression,” PhD thesis, Lund University, 1999 .

[26] S. Dharmapurikar, et al, “Deep Packet Inspection using Parallel Bloom Filters,” IEEE Hot Interconnects 12, August 2003.

[27] Z. K. Baker, V. K. Prasanna, “Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs,” in Field Programmable Logic and Applications, Aug. 2004, pp. 311–321.

[28] Y. H. Cho, W. H. Mangione-Smith, “Deep Packet Filter with Dedicated Logic and Read Only Memories,” Field Programmable Logic and Applications, Aug. 2004, pp. 125–134.

[29] M. Gokhale, et al., “Granidt: Towards Gigabit Rate Network Intrusion Detection Technology,” Field Programmable Logic and Applications, Sept. 2002, pp. 404–413.

[30] J. Levandoski, E. Sommer, and M. Strait, “Application Layer Packet Classifier for Linux”. http://l7-filter.sourceforge.net/.

[31] M. Hill and J. Elder, “DineroIV tracedriven uniprocessor cache simulator,” http://www.cs.wisc.edu/markhill/DineroIV, 1998.

[32] SafeXcel Content Inspection Engine, regex acceleration IP.

[33] Network Services Processor, OCTEON CN30XX Family.

[34] D. Fotakis, et. al, “Space efficient hash tables with worst case constant access time,” In STACS, 2003.