An Improved Algorithm to Accelerate Regular Expression Evaluation

Michela Becchi
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-4306
mbecchi@cse.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

ABSTRACT
Modern network intrusion detection systems need to perform regular expression matching at line rate in order to detect the occurrence of critical patterns in packet payloads. While deterministic finite automata (DFAs) allow this operation to be performed in linear time, they may exhibit prohibitive memory requirements. In [9], Kumar et al. propose Delayed Input DFAs (D^2FAs), which provide a trade-off between the memory requirements of the compressed DFA and the number of states visited for each character processed, which corresponds directly to the memory bandwidth required to evaluate regular expressions.

In this paper we introduce a general compression technique that results in at most 2N state traversals when processing a string of length N. In comparison to the D^2FA approach, our technique achieves comparable levels of compression, with lower provable bounds on memory bandwidth (or greater compression for a given bandwidth bound). Moreover, our proposed algorithm has lower complexity, is suitable for scenarios where a compressed DFA needs to be dynamically built or updated, and fosters locality in the traversal process. Finally, we also describe a novel alphabet reduction scheme for DFA-based structures that can yield further dramatic reductions in data structure size.

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

General Terms
Algorithms, Performance, Design, Security.

Keywords
Deep packet inspection, DFA, regular expressions.

1. INTRODUCTION
Signature-based deep packet inspection has taken root as a dominant security mechanism in networking devices and computer systems. Most popular network security software tools—including Snort [10][11] and Bro [12]—and devices—including the Cisco family of Security Appliances [13] and the Citrix Application Firewall [14]—use regular expressions to describe payload patterns. In addition, application-level signature analysis has been recently proposed as an accurate means to detect and track peer-to-peer traffic, enabling sophisticated packet prioritization mechanisms [17].

Regular expressions are more expressive than simple patterns of strings and therefore able to describe a wider variety of payload signatures, but their implementations demand far greater memory space and bandwidth. Consequently, there has been a considerable amount of recent work on implementing regular expressions for use in high-speed networking applications, particularly with representations based on deterministic finite automata (DFA).

DFAs have attractive properties that explain the attention they have received. Foremost, they have predictable and acceptable memory bandwidth requirements. In fact, the use of DFAs allows one single state transition, and one corresponding memory operation, for each input character processed. Moreover, it has long been established that, for any given regular expression, a DFA with a minimum number of states can be found [3]. Even so, DFAs corresponding to large sets of regular expressions containing complex patterns can be prohibitively large in terms of numbers of states and transitions. Two recent proposals have tackled this problem in different ways, both trading memory size for bandwidth.

First, since an explosion in states can occur when many rules are grouped together into a single DFA, Yu et al. [15] have proposed segregating rules into multiple groups and evaluating the corresponding DFAs concurrently. This solution decreases memory space requirements, sometimes dramatically, but increases memory bandwidth linearly with the number of concurrent DFAs. For example, using 10 DFAs in parallel requires a ten-fold increase in memory bandwidth. This characteristic renders the approach infeasible for large rule-sets that must be stored in off-chip memories.

The second approach leverages the observation that the memory space required to store a DFA is a function of the number of states and the number of transitions between states. While the number of states can be minimized as a matter of course, the space needed to encode transitions can be reduced well beyond that of a straight-forward representation. Kumar et al. [9] observe that many states in DFAs have similar sets of outgoing transitions. Substantial space savings in excess of 90% are achievable in current rule-sets when this redundancy is exploited. The proposed automaton, called a Delayed Input DFA (D^2FA), replaces redundant transitions common to a pair of states with a single default transition. However, as explained in detail later, the use of default transitions implies that multiple states may be traversed when processing a single input character. In fact, the D^{2}FA approach requires a heuristic construction algorithm to bound the length of *default transition chains* in order to keep the memory bandwidth feasible. The original D^{2}FA heuristic has three weaknesses: 1) it requires a user-provided parameter value to operate which can only be determined experimentally for a given rule-set, 2) it creates a data-structure whose worst-case paths may be traversed for each input character processed, and 3) it requires multiple passes over large support data structures during the construction phase.

In this paper, we propose an improved yet simplified algorithm for building default transitions that addresses these problems. Notably, our scheme results in at most 2N state traversals when processing an input string of length N, *independent of the maximum length of the default transition chain*. On practical data sets, the level of compression achieved is similar than the original D^{2}FA scheme, while providing a superior worst-case memory bandwidth bound. Moreover, when the D^{2}FA scheme was configured to guarantee the same worst-case memory bandwidth bound than our algorithm, it produced a compression level about a factor of 10 smaller.

Our approach is based on a simple observation: all regular expression evaluations begin at a single start state, and the vast majority of transitions between states lead back to the start state or its near neighbors. As will be seen, this simple observation explains the extraordinary redundancy among state transitions that is exploited in an oblivious manner by the D^{2}FA technique. Furthermore, by formalizing the notion of *state depth* to quantify a state's distance from the start state, it is possible to construct nearly optimal default paths with a far simpler algorithm. By leveraging depth directly during automaton construction, greater efficiency and simplicity are achieved.

In describing our algorithm, we emphasize a number of details that relate directly to its practical implementation. First, we show that the algorithm can be incorporated directly into DFA generation—that is, into the NFA to DFA subset construction operation—which eliminates the need to either first generate a perhaps unfeasibly large uncompressed DFA prior to compression or to maintain the large support data structures required for subsequent DFA compression. This both allows larger rule-sets to be supported and decreases the cost of supporting frequent rule-set updates.

Our discussion also encompasses data structure encoding details. Most notably, we describe a novel scheme for alphabet reduction that can be applied to any DFA-based automaton. By selectively assigning characters to classes based on their common use as edge labels, both the number of transitions and the number of bits needed to label all edges uniquely are dramatically reduced. This approach yields further data size reductions by factors ranging from 2 to 10 in real-world rule-sets.

To our knowledge, the two primary contributions made in this paper—depth-driven default path construction and class-based alphabet reduction—represent the most efficient and practical proposals to date for regular expression evaluation in high-speed networking contexts.

The remainder of this paper is organized as follows. In section 2, we give an overview of the D^{2}FA technique by way of an example. In section 3, we present our algorithm for building default transitions and compare it with the original proposal in [9]. In section 4, we present a general coloring algorithm for alphabet reduction and further reduce the number of DFA transitions. In section 5, we discuss several encoding schemes which can be used to represent the compressed D^{2}FA. In section 6, we present an experimental evaluation on data-sets used in the Snort and Bro tools and also in the Cisco security appliance. We then relate our work to the state of art (section 7) and conclude (section 8).

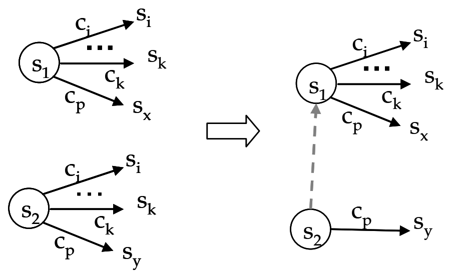

Figure 1: Example of transition reduction after introducing a default transition (in grey and dashed) from s_2 to s_1. Common transitions to s_i...s_k are deleted from s_2.

2. MOTIVATION

In this section, we describe the D^{2}FA approach and explain its compression algorithm. For a more detailed description, the interested reader can refer to [9].

The basic goal of the D^{2}FA technique is to reduce the amount of memory needed to represent all the state transitions in a DFA. This is achieved by exploiting the redundancy present in the DFA itself. To see how, consider a DFA with *N* states representing regular expressions over an alphabet Σ with cardinality |Σ| will contain N*|Σ| next state transitions. The authors of [9] observe that, in the case of practical rule-sets from commonly used network intrusion detection systems, many groups of states share sets of outgoing transitions. This redundancy can be exploited as follows. Suppose that states s_1 and s_2 transition to the same set of states $S=\{s_i,...,s_k\}$ for the same set of characters $C=\{c_i,...,c_k\}$. In this situation, the common transitions to s_1 and s_2 can be eliminated from one of the two states, say s_2, by introducing an unlabeled *default transition* from s_2 to s_1. State s_2 will then contain only |Σ|-|*S*| labeled transitions which are not in common with s_1. An example is shown in Figure 1.

During the string matching operation, the traversal of the compressed DFA will be performed according to the traditional Aho-Corasick algorithm [1], treating default transitions as failure pointers. In the example, if state s_2 is visited on input character *c*, all its outgoing labeled transitions are first considered. If a labeled transition for character *c* exists, it is taken and determines the next state. Otherwise, the default transition (which leads to state s_1) is followed, and state s_1 is inspected for character *c*. Notice that s_1 may or may not contain a labeled transition for character *c*. In the latter case, a default transition is followed again until a state containing a labeled transition for the current input character *c* is found. Thus, the number of state traversals involved in processing a character depends on the length of the default transition chains present in the D^{2}FA.

The heuristic proposed in [9] to build a D^{2}FA aims to maximize space reduction given a worst case time bound, the latter expressed in terms of the maximum number of states visited for each character processed. That is, the heuristic explores the tension between increasing the number of default transitions to reduce memory size and decreasing their number to reduce memory bandwidth.

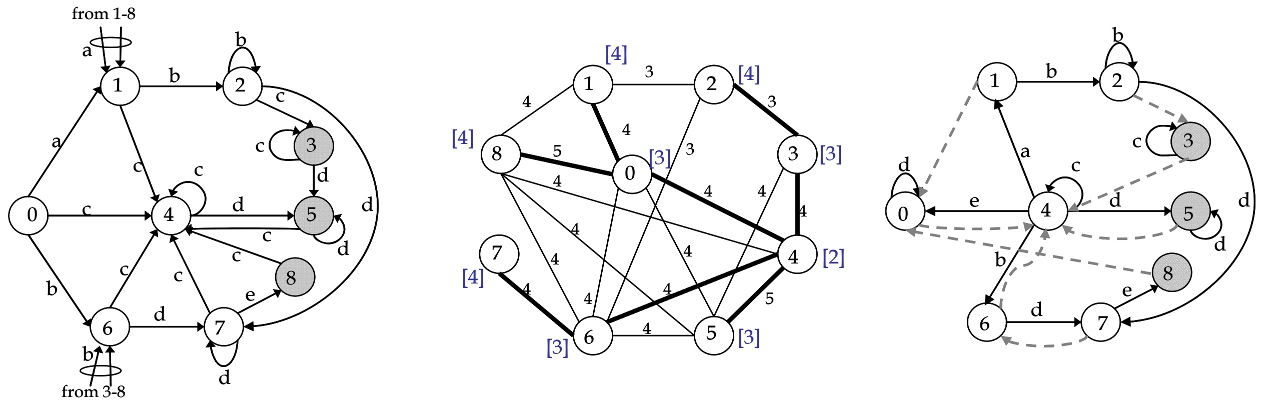
Figure 2: (a) DFA recognizing regular expressions: *ab+c+*, *cd+* and *bd+e* over alphabet {a,b,c,d,e}. Accepting states are represented in grey; transitions to state 0 are omitted. (b) Corresponding space reduction graph. For readability, only edges with weight greater than 3 are represented. Additionally, edges with weight 3 connecting state 2, which otherwise would be disconnected, are displayed. One possible maximum spanning tree with diameter bound 4 is highlighted in bold. The bracketed value at each state represents the corresponding distance parameter. (3) Resulting D^{2}FA (all the transitions are shown; default transitions are in grey and dashed).

As shown in [9], this tradeoff can be explored systematically as a maximum spanning tree problem on an undirected graph. If two states s_1 and s_2 have k labeled transitions in common, then introducing a default transition between the two will eliminate k labeled transitions. Therefore, the exploration space, also called a *space reduction graph*, consists of an undirected weighted graph having a vertex for each DFA state, and an edge connecting every two vertices sharing at least two outgoing transitions. The edge weights indicate the number of transitions that the endpoints have in common.

This maximum spanning tree problem can be solved with Kruskal's algorithm [5] in O($n^2 logn$) time, n being the number of vertices in the space reduction graph. The algorithm analyzes the edges in decreasing order of weight, and connects the ones which do not generate loops (a partitioned data structure is used to speed up this check [8]). In the case of unconnected graphs, a forest of disjointed maximum spanning trees is returned.

After the operation of Kruskal's algorithm, the root of each tree can be selected so to minimize the length of the resulting chains of default transitions, which are then oriented accordingly. To this end, the node having the smallest maximum distance from any vertices within the same tree is chosen. However, the resulting worst case time bound can still be unacceptably large.

In order to limit the maximum default path length, the problem of determining a *maximum spanning tree forest with bounded diameter* is addressed. Since this problem is in general NP-hard, a heuristic is proposed. Specifically, the basic algorithm presented above is modified as follows. An edge under examination is selected only if its addition won't cause the violation of the pre-established diameter bound. In order to do this efficiently, a distance vector is maintained and updated at every edge insertion. Finally, a further refinement to this heuristic consists in prioritizing, among the edges with the same weight, the ones whose introduction will lead to a smaller increase in the current diameter bound.

An example of the operation of the algorithm is given in Figure 2. Figure 2(a) shows the original DFA (transitions leading to the initial state 0 are omitted for readability). The corresponding space reduction graph is represented in Figure 2(b), together with a maximum spanning tree obtained using the described heuristic with a diameter bound of 4 (that is, assuming a maximum default path length of 2). Notice that node 4, having a maximum distance from any vertices of 2, will be selected as the root of the tree and the default transitions will be oriented accordingly. The resulting D^{2}FA is represented in Figure 2(c), where default transitions are colored grey and dashed. It can be observed that the introduction of default transitions removes 33 labeled transitions, equal to the weight of the spanning tree.

Figure shows a maximum spanning forest which results from setting the diameter bound to 2. Notice that, in this case, the default transitions will be directed towards states 0 and 4 and only 28 labeled transitions will be saved. That is, a better worst case time bound is obtained at the cost of a reduced memory size reduction.

3. THE PROPOSAL

It can be observed that the compression algorithm in the D^{2}FA scheme is oblivious to the way a DFA is traversed, and operates only on number of outgoing transitions common to different states. We now take advantage of a simple fact – *DFA traversal always starts at a single initial state s_0* – in order to propose a more general compression algorithm which leads to a traversal time bound *independent of the maximum default transition path length*.

Before proceeding, we need to introduce a term. For each state s,

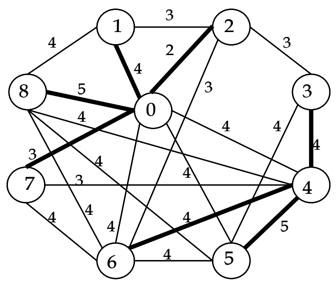
Figure 3: Possible forest of maximum spanning trees for DFA in Figure 2(a) when diameter bound 2 is used. Additional low weight edges connecting states 2 and 7 to the other vertices are displayed.

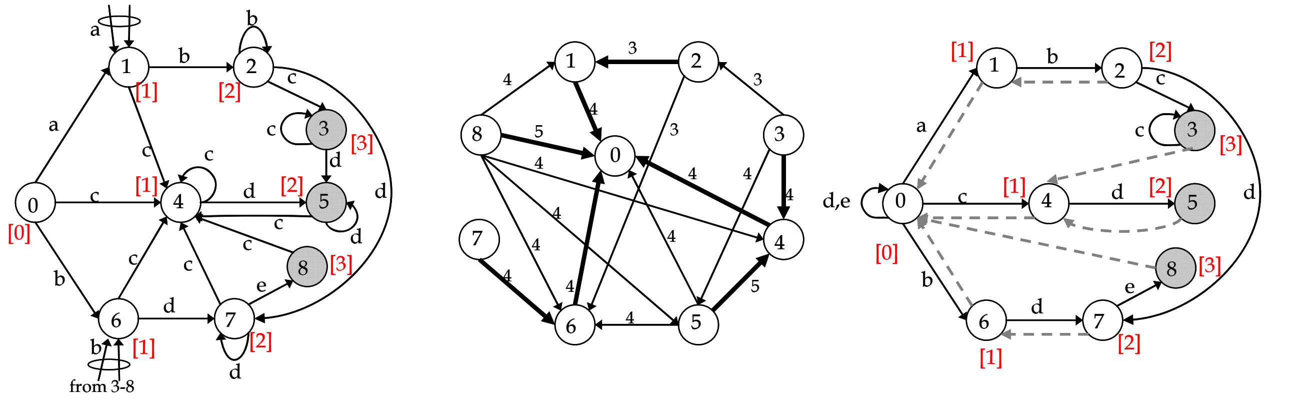

Figure 4: (a) DFA recognizing regular expressions: *ab+c+*, *cd+* and *bd+e* over alphabet {a,b,c,d,e}. Accepting states are represented in gray; transitions to state 0 are omitted. The bracketed value at each state represents its depth. (b) Corresponding (directed) space reduction graph. For readability, only edges with weight greater than 3 are represented. Additionally, edges with weight 3 connecting state 2, which otherwise would be disconnected, are displayed. Directed maximum spanning tree is highlighted in bold. (3) Resulting D^{2}FA (all the transitions are shown; default transitions are represented in grey and dashed).

we define its *depth* as the minimum number of states visited when moving from s_0 to s in the DFA. In other words, the initial state s_0 will have depth 0, the set of states S_1 directly reachable from s_0 will have depth 1, the set of states S_2 directly reachable from any of the S_1 (but not from s_0) will have depth 2, and so on. Clearly, the depth information for any DFA with n states can be constructed in $O(|\Sigma|n)$ time through an ordered visit of the DFA starting at state s_0. As an example, Figure (a) reports the depth information for the DFA considered earlier. Note that the depth of state 4 depends on it being reached directly from the initial state 0, even if transitions from other states to state 4 are also present in the DFA.

The algorithm proposed is based on the following lemma.

Lemma: If none of the default transitions in a D^{2}FA lead from a state with depth d_i to a state of depth d_j with $d_j \geq d_i$, then any string of length N will require at most 2N state traversals to be processed.

In other words, a 2N time bound is guaranteed on all D^{2}FA having only "backwards" transitions. In a sense, this can be thought of as a generalization of [1] to regular expressions.

The proof of the lemma is trivial. Each character processed causes exactly one labeled transition and zero or more default transitions to be taken. Let us suppose that, at a given point, a chain of k default transitions must be taken from a state s. Since default transitions are only directed towards states with smaller depth, state s must have depth $\geq k$. Thus, to get to state s, at least k labeled transitions (in excess to default transitions) must have been taken before. Therefore, the number of default transitions is always at least one less than the number of labeled transitions taken. Since a string of length N implies N labeled transitions to be followed, the total number of state traversals cannot be higher then 2N-1.

Notice that the presence of "backwards" labeled transitions does not affect the proof. In fact, this implies that, if a state of depth k is visited at a point, then *at least* (and not *exactly*) k labeled transitions must have been taken before. In other words, backwards labeled transitions contribute to make the average case better than the worst case.

3.1 Problem Formulation

The problem can be now formulated as an instance of maximum spanning tree on a *directed* graph. In fact, since the default transitions can be oriented only in the direction of decreasing depth, the space reduction graph consists in this case of directed edges. Notice that, once a maximum spanning tree (or forest) has been determined, no extra computation to determine the root and the orientation of the default transitions is needed.

Two similar algorithms to find the optimal solution to the problem have been proposed by Chu et al.[6] and Edmonds [7]. In both cases the maximum spanning tree is basically determined in two steps: edge selection and cycle resolution. First, each vertex selects its outgoing edge with maximum weight, which will be added to a set *E*'. If *E'* does not contain cycles, then its edges form a maximum spanning tree. Otherwise, each cycle is collapsed into a pseudo-node, and the weights of the edges exiting the pseudo-node are modified. The maximum weight edge exiting the pseudo-node is then selected and the previous edge exiting the same source vertex is excluded from *E'*. The basic idea is to eliminate each cycle by subtracting the minimum possible weight.

Note that the complexity of the algorithm resides in the cycle resolution phase. Fortunately, there is no need to perform this action in our instance of the problem. In fact, since edges in the space reduction graph are always directed towards decreasing depth, the graph does not contain any cycles. Therefore, any subset of edges belonging to it will be acyclic, too.

The complexity of the algorithm will depend only on the number of edges in the space reduction graph, that is, $O(n^2)$.

3.2 An example

An example of default transition construction with the proposed scheme is given in Figure , where the same DFA as in Figure 2 is used. In particular, Figure (b) represents the *directed* space reduction graph. Notice that, differently from Figure 2(b), there is no edge connecting nodes 4 and 6: in fact, the two states have the same depth. The root of the maximum spanning tree is now the initial state 0.

The corresponding D^2FA is represented in Figure (c). Notice that, even if the default transitions are different compared to Figure 2(c), 33 labeled transitions are again saved. In fact, the space reduction graph allowed several maximum spanning trees also in the undirected case, but the heuristic proposed in [9] did not privilege the one directed towards the initial state. On the other hand, the worst case traversal time bound has decreased. In fact, the diameter bound in Figure 2(c) is 4, leading to some default transition paths of length 2. This, in turn, translates into 3 state traversals for each character processed and to an overall O(3N) complexity for a string of length N.

To achieve the same time complexity of Figure (c) using the algorithm described in section II, a diameter bound of 2 has to be utilized. As shown in Figure , this leads to a lower memory saving (only 28 labeled transitions can be removed).

3.3 Algorithm

While the concept of a space reduction graph is useful to relate this problem to the one solved in [9] and to help find an optimal solution to it, a support graph data structure is not really needed to find the maximum spanning tree. In fact, the whole problem is reduced to having each state select the state with lower depth having the most number of outgoing transitions in common with it. In the case of ties, preference is given to the smaller depth. This limits the default transition path length and enforces locality during traversal.

```
procedure default_transition (DFA dfa=(n, δ(states, Σ)),
                                   modifies set default);
  list queue; set depth[n];
  for state s ∈ states ⇒ depth[s]=n; default[s]=s; rof
  depth[0]=0;queue.push(0);
  while (!queue.empty())⇒
    state s= queue.pop();
    int saving=0;
    for char c ∈ Σ⇒
      if (depth[δ(s,c)]=n) ⇒
        depth[δ(s,c)]= depth[s]+1; queue.push(δ(s,c));
      fi
    rof;
    for (state t ∈ states & depth[t]<depth[s]) ⇒
      int common:=# common transitions btw. s and t;
      if (common > 1 && (common>saving ||
          (common=saving && depth[t]<depth[default [s]])))
          saving:=common;
          default[s]=t;
      fi
    rof;
  end while;
end;
```

If the DFA traversal is performed in a breath-first fashion, both the default transitions and the depth computations can be done in a single pass, as shown in the pseudo-code above. The DFA is described in terms of the number of states *n* and of the function *δ(states,Σ)→states,* which associates a next state to each *(state, character)* pair. A queue is introduced to implement the breath-first traversal. Notice that when a state *s* is extracted from the queue, all the states with a smaller depth have already been processed, and therefore the depth vector will contain a correct value for them. States with a higher depth can be ignored (initializing their depth to *n* will therefore ensure correct operation). While not shown in the pseudo-code, the removal of the redundant labeled transitions can also be performed in the same pass.

Similarly, this same algorithm can be combined with subset construction (i.e., the NFA-to-DFA transformation used to create an initial DFA) so to generate default transitions directly *during DFA creation*. In fact, it is enough to ensure that new DFA states are queued according to increasing depth, as is done above. The generation of an initial compressed DFA eliminates the need to first construct an uncompressed one; we consider this issue concretely in Section 3.4.2.

3.4 Discussion

The compression scheme proposed in this paper and the one proposed in [9] (and summarized in section II) can be compared from several perspectives. The goal of this section is to qualitatively summarize the most important points. An experimental evaluation of the two on practical rule-sets is presented in section VII.

3.4.1 Worst case time bound and memory reduction

As mentioned, while the original D^2FA scheme trades off worst-case bound on the processing time with memory size, the algorithm proposed here aims at achieving a constant 2N worst-case bound on the processing time. This is comparable with running the D^2FA algorithm with a minimal diameter bound of two.

As far as memory size reduction is concerned, our expectations are: i) better compression when compared to D^2FA with diameter bound equal to two, and ii) comparable compression when compared with D^2FA with higher diameter bound. We offer two reasons for these expectations.

The first is due to how regular expressions are used in this context.. Intuitively, they are characterized by a limited number of "forward" labeled transitions corresponding to the matching conditions in the described patterns. However, most transitions are "backwards": they correspond to mismatches, and they tend to return to the initial state and states closely connected to it. In the example of Figure 2(a), for instance, most transitions end at states 0, 1, 4 and 6. In the case of regular expressions with dot-star conditions, many transitions tend to fall back to the state the dot-star originates from (and to its close vicinity). Backwards default transitions will in general be able to replace backwards labeled transitions, which are the most numerous.

The second motivation is based on the nature of the problems addressed. Even if the directed-graph problem is more constrained than its undirected counterpart, at least when a high diameter bound is allowed, the algorithm proposed finds the optimal solution to it. On the other hand, the D^2FA scheme proposes a heuristic which can find suboptimal solutions. Especially in case of heavily connected space reduction graphs, the optimal solution to the constrained problem can be better than the suboptimal solution of the loosely constrained one.

3.4.2 Algorithmic complexity & practical details

As far as asymptotic complexity is concerned, the original D^2FA algorithm and the proposed one have $O(n^2 logn)$ and $O(n^2)$ time bounds, respectively.

The complexity of the first reduction algorithm is kept low though the use of support data structures (space reduction graph, d-heap and partition data structure). In practice, this fact has important implications which impact the implementation and the running time of the algorithm itself when large DFAs are taken into consideration.

Among the data structures listed above, the biggest and most problematic is the reduction graph. An adjacency list is an efficient graph edge representation; it allows fast navigation and requires about 17 bytes/edge when implemented as follows.

```
struct wgedge {
  vertex   l,r; // endpoints of the edge
  weight   wt;  // edge weight
  edge lnext; //link to next edge incident to l
  edge rnext;  //link to next edge incident to r
} *edges;
```

A fully connected graph with about 11,000 nodes will require 1GB of memory just for storing this data structure. A possible way around is to build partial graphs, including only edges of given weights (which is compliant with Kruskal's operation). This, however, leads to the need of several DFA scans, which negatively impact the overall running time.

By not needing this support data structure, or the others, our proposed algorithm is not affected by these issues. Even if this may not be problematic in networking applications where the update rate is low, our scheme may be preferable in more dynamic scenarios which may occur in the future (for instance, if signature generation is made automatic).

3.4.3 Additional aspects

One additional interesting aspect is that D^{2}FAs built with our proposed algorithm tend to foster locality during the traversal process. A probabilistic analysis of DFAs accepting the real-world regular expressions used below reveals that a small number of states accounts for a high percentage of the traversals. Intuitively, this can be explained by observing that mismatches are more likely to happen than matches, and that most transitions lead to a few states in the vicinity of the initial state. The probability of visiting a state with depth *k* is conditional upon the one of reaching the states leading to it, which must have depth at least *k-1*. Consequently, the probability of visiting a state tends to decrease as the depth increases.

Since the proposed algorithm tends to select states with low depth as targets of default transitions, it further accentuates the locality behavior of the DFA traversal operation. This suggests that the use of caches would positively affect the system throughput. The same is not true of a traditional D^{2}FA, where the direction of the default transitions is not controlled and can lead far away from the initial state.

4. REDUCING THE ALPHABET

In this section we present a means to reduce the size of the alphabet and further decrement the number of transitions needed to represent a DFA. This technique is orthogonal to the one presented to far: it can be applied on top of it or on the original DFA before the default transition computation is performed.

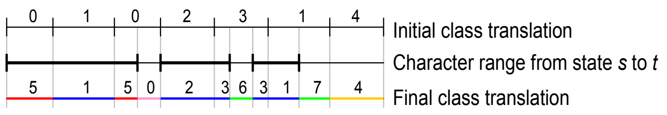
0 1 0 2 3 1 4
Initial class translation
Character range from state *s* to *t*
5 1 5 0 2 3 6 3 1 7 4
Final class translation

Figure 5: Example of the alphabet reduction algorithm when processing transitions leading from state s to *t*.

The basic idea is the following: in a DFA recognizing regular expressions over an alphabet Σ each state has potentially |Σ| outgoing transitions, one for each symbol in Σ. However, Σ can be partitioned into classes of symbols $C_1,...,C_k$ which are indistinguishable for the purposes of the DFA operation. Two symbols c_i and c_j will fall into the same class if they are treated the same way in all DFA states. In other words, given the transition function $\delta(states, \Sigma)\rightarrow states$, $\delta(s,c_i)= \delta(s,c_j)$ for each state *s* belonging to the DFA. Notice that it is not required that transitions on c_i and c_j lead to the same target from different source states.

```
procedure character_class (DFA dfa=(n, δ(states, Σ)),
                           modifies set class);
  int max_class=0; class ← 0;
  for state s ∈ states ⇒
    for state t ∈ states ⇒
      set char_covered[|Σ|] ← false;
      set class_covered[|Σ|] ← false;
      set remap[|Σ|] ←0;  int on_zero=0;
      for  (char c ∈ Σ-{'\0'} & δ(s,c)=t) ⇒
        char_covered[c] := true;
        if (class[c]=0) ⇒
          if (on_zero=0) ⇒ on_zero = ++max_class; fi
          class[c]=on_zero;
        else
          class_covered[class[c]]=true;
        fi
      rof
      for (char c∈ Σ) ⇒
        if (!char_covered [c] && class_covered[class[c]]) ⇒
          if (remap[class[c]]==0) ⇒
            remap[class[c]]= ++max_class;
          fi
          class[c]=remap[class[c]];
        fi;
      rof;
    rof;
  rof;
end;
```

Once the class translation $C(\Sigma)\rightarrow \{1..k\}$ has been computed, the alphabet is reduced from cardinality |Σ| to *k*. *k* next state transitions will therefore suffice at each state. An additional *alphabet translation table* encoding the symbol-to-class mapping is required to allow the string matching operation. In practical scenarios (ASCII alphabet) this table will contain 256 entries, with a maximum width of 1 byte (for heavily compressed alphabets 5-6 bits per character may suffice). This indexing table can be efficiently cached and its access can be pipelined with the real DFA query.

Intuition about the potential transition savings implied by alphabet reduction is given by the following observations. First, regular expressions defined over an alphabet Σ tend in practice to use only a

Table 1: Characteristics of the rule-sets

Data-set	# of RegEx	ASCII length range	% RegEx w/ wild-cards (*,+)	% RegEx w/ char ranges ≥ 5
Snort24	24	6..70	37.5	50
Snort34	34	15..99	38.2	32.4
Snort31	31	16..120	41.9	93.5
Cisco11	11	9..13	90.9	9.1
Cisco43	43	15..73	32.6	27.9
Cisco612	612	3..50	0	1.6
Bro217	217	5..76	1.4	13.4

subset of the symbols. The characters which do not appear in any patterns accepted by the DFA often translate into backwards transitions to the same state. Second, in practical cases there are groups of symbols naturally handled together. As an example, carriage return (CR) tends to be treated together with line feed (LF), and, when case is ignored, lowercase alphabetic characters [a-z] tend to appear with their uppercase counterparts [A-Z].

As in the previous section, we want to provide a low complexity algorithm to perform alphabet reduction which operates by scanning the DFA without needing support data structures whose sizes are quadratic in the number of DFA states.

The basic idea is to build the character translation information by doing cluster-division. Specifically, the algorithm initially assumes to have a unique character class, say 0. It then loops over the states and analyzes the outgoing transitions. For each state *s*, characters leading to the same target *t* do potentially belong to the same class, unless they led to different targets for some state *s'* previously processed.

The operation of the algorithm for each pair of states *(s,t)* is shown in Figure. Suppose that the character translation before processing states *(s,t)* is as in the first row, and that the range of characters transitioning from *s* to *t* is as in the second row (bold). The following actions must be taken, as indicated on the third row: i) a new character class must be opened for bold characters previously mapped to class 0 (red segments 1 and 2); ii) the same character class can be preserved for bold characters overlapping a previously defined character class (blue segments 2,5,6,8, and 9); iii) new character classes must be opened in case of non bold characters intersecting a character class partially covered by bold characters (green segments 7 and 10), iv) no action should be taken for non bold characters covering class 0 (pink segment 4) or any uncovered character class (yellow segment 11).

The resulting algorithm is presented in the pseudo-code above. Its complexity is $O(n^2)$ – more precisely $O(|\Sigma|n^2)$ – and it can be combined with our proposed algorithm for default transition generation and subset construction.

5. ENCODING

There are several ways to encode a DFA whose transitions have been compressed with the above techniques. In this section, we briefly describe the two most appealing alternatives, namely, bitmaps and content-addressing.

5.1 Bitmaps

A scheme also exploited in a related context [18] consists of associating a bitmap as large as the alphabet size to each DFA state. Bits corresponding to uncompressed labeled transitions present in the current state can be set to 1; the remaining bits are set to 0. Thus, a state traversal will consist of two accesses: the first (bitmap) to determine whether the default pointer or a labeled transition must be followed, and the second to actually retrieve the next state information.

The basic disadvantage of this scheme is that it requires several accesses for each state traversal. However, bitmaps allow a compact memory representation. First, state identifiers can be simply represented through their base address in memory: in practical cases 20-bit pointers are sufficient. Second, the length of the necessary bitmaps can substantially decrease after alphabet reduction. Third, other techniques proposed in the literature [19] allow bitmap compression. This is especially true for bitmaps having a restricted number of 1, as is the case of practical datasets (see section VI).

5.2 Content addressing

A second encoding technique, proposed in [16], consists in representing state identifiers with *content labels*, which are stored in memory as next state transitions. A state content label contains several fields: a state discriminator, the list of characters for which a labeled transition is defined, and an identifier for the default transition state. Since the size of a content label depends on the number of labeled transitions defined for the corresponding state, its use can be

Table 2: Comparison between the compression achieved through the D^{2}FA basic algorithm and through our scheme. Different values of the diameter bound (DB) are reported in case of D^{2}FA scheme (DB=∞: no bound is used)

Dataset	Original DFA				D^{2}FA algorithm						Our algorithm	
					Compression (as a function of the diameter bound)							
	# of states	*# of transitions*	*# distinct trans.*	*% duplicates*	*DB=2*	*DB=6*	*DB=10*	*DB=14*	*DB=∞*	*max def. length*	*Compression*	*max def. length*
Snort24	13886	3554816	36763	98.97	89.59	98.48	98.91	98.92	98.92	16	98.71	12
Snort34	13825	3539200	38573	98.91	89.33	98.48	98.85	98.86	98.86	16	98.69	10
Snort31	20052	5133312	54960	98.93	74.42	97.18	98.42	98.6	98.63	13	98.44	6
Cisco11	24011	6146816	156566	97.45	86.73	97.08	97.37	97.38	97.38	12	96.63	8
Cisco43	20320	5201920	48764	99.06	90.16	98.46	99	99.05	99.05	14	98.97	8
Cisco612	11309	2895104	14618	99.5	79.3	97.46	98.93	99.18	99.25	12	99.09	5
Bro217	6533	1672448	7221	99.57	76.49	97.9	99.07	99.4	99.41	9	99.33	9

effective only for those states which are highly compressed. The remaining states should have all their outgoing transitions represented in a traditional way.

The use of content labels has the benefit of allowing one memory access per state traversal. In fact, the analysis of the state identifier determines which state—the current one or the default transition's target—must be analyzed to retrieve the next state information. The mapping between the content label and the effective memory address of the corresponding state is performed through a hash function.

In [16], a content-addressed D^2FA (CD^2FA) is proposed by the use of recursive content labels combined with a D^2FA having mostly diameter bounds of 2 to require just one 64-bit wide memory access per character processed. The content labels are allowed to be 64 bits wide, making the scheme effective when a great percentage of the states have less than 5 labeled transitions. For data-sets where this condition does not hold (for instance, because of the frequent presence of larger character ranges) this scheme may not to be effective.

6. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of the proposed algorithm on practical data-sets from the Snort and Bro intrusion detection systems and the Cisco security appliance. Snort rules have been filtered according to the headers ($HOME_NET, any, $EXTERNAL_NET, $HTTP_PORTS/any) and ($HOME_NET, any, 25, $HTTP_PORTS/any). In the experiments which follow, rules have been grouped so to obtain DFAs with reasonable size and, in parallel, have datasets with different characteristics in terms of number of wildcards, frequency of character ranges and so on. The basic characteristics of the datasets are summarized in Table 1.

Our first goal is to compare the memory compression achieved through our scheme to that of D^2FA. To this end, we implemented the D^2FA algorithm [16] and ran it on these rule-sets with multiple diameter bound values. In one experiment, the diameter was left unbounded and the maximum default length was measured. The results are shown in Table 2, where the compression is expressed as the ratio between the number of deleted transitions and the original ones. Note that our algorithm achieves a degree of compression notably higher than the counterpart D^2FA with diameter bound equal to two which has the same worst-case bound on bandwidth. Moreover, the compression is comparable to that of D^2FA with no diameter bound, which, as pointed out and as the maximum default length values show, exhibits the worst performance in terms of throughput. Even if the amortized time complexity of our algorithm is 2N independent of the maximum default path length, it is interesting to note that this parameter is kept lower than that of D^2FA.

Table 3: Comparison between number of transitions with our scheme and D^2FA.

Dataset	Original DFA		Transitions after compression		
	# of states	*distinct trans..*	*Our scheme*	*D^2FA*	
				DB=2	*DB=∞*
Snort24	13886	36763	46005	369879	38491
Snort34	13825	38573	46298	377613	40225
Snort31	20052	54960	80004	1313003	70389
Cisco11	24011	156566	207303	815415	161135
Cisco43	20320	48764	53463	511953	49570
Cisco612	11309	14618	26218	599318	21630
Bro217	6533	7221	11247	393130	9816

To clarify the significance of these compression, Table compares the number of transitions obtained using our scheme to that of the D^2FA with diameter bounds of 2 and infinity. As can be seen, our algorithm yields in most scenarios a factor of 10 or more fewer edges than the D^2FA. The advantage is greatest in cases like Bro217 and Cisco612 where the space reduction graphs are heavily connected and hence orienting and eliminating some edges does not greatly restrict the exploration space.

In Table we represent the result of performing alphabet reduction on the given DFAs. The achieved alphabet size and the compression both in relative and in absolute terms are shown. The following observations can be made. First, alphabet reduction implies further compression on all the datasets and over all the algorithms and their parameterizations. Second, the degree of compression achieved by our algorithm is higher than 99% in all cases. Third, the performance of our algorithm gets closer to that of the unbounded D^2FA and remains significantly better than D^2FA with diameter bound equal to 2.

Finally, we consider encoding the compressed DFA produced by our algorithm through content addressing and comparing the results to that of CD^2FA[16]. To this end, we implemented the D^2FA generation algorithm which is described in [16]. To perform alphabet

Table 4: Effect of alphabet reduction. The degree of compression and the number of transitions before (BAR) and after (AAR) alphabet reduction are displayed. Our algorithm is compared to D^2FA with diameter bound equal to 2 and without bound.

Dataset	# of nodes	alpha bet size	D^2FA algorithm, DB=2			D^2FA algorithm, DB=∞			Our algorithm		
			compression %		*transitions*	*compression %*		*transitions*	*Compression %*		*transitions*
			BAR	*AAR*	*after AR*	*BAR*	*AAR*	*after AR*	*BAR*	*AAR*	*after AR*
Snort24	13886	46	89.59	97.87	75752	98.92	99.49	18095	98.71	99.4	21504
Snort34	13825	51	89.33	97.63	84046	98.86	99.47	18856	98.69	99.43	20342
Snort31	20052	53	74.42	94.48	283339	98.63	99.21	40347	98.44	99.13	44819
Cisco11	24011	38	86.73	97.74	138922	97.38	99.24	46689	96.63	99.09	55955
Cisco43	20320	65	90.16	97.09	151161	99.05	99.31	36037	98.97	99.27	37784
Cisco612	11309	115	79.3	90.46	276110	99.25	99.33	19316	99.09	99.2	23139
Bro217	6533	111	76.49	89.59	174035	99.41	99.43	9526	99.33	99.34	10957

Table 5: Comparison of number of transitions between CD^2FA and our algorithm with content addressing before (BAR) and after (AAR) alphabet reduction

Dataset	**Alphabet**	**C^2DFA** *trees*	**C^2DFA** *# transitions* *BAR*	*AAR*	**Our algorithm** *# transitions* *BAR*	*AAR*
Snort24	46	879	252677	55584	46925	21934
Snort34	51	928	264652	62836	46298	20342
Snort31	53	5176	1357398	291089	84004	47060
Cisco11	38	1196	354868	75024	228033	61551
Cisco43	65	754	232548	83068	64156	45341
Cisco612	115	1712	270344	210163	28840	25453
Bro217	111	154	51090	28760	12372	12053

reduction, we use our algorithm since it is more effective and general than that proposed in [16]. Also, we assume 64-bit wide content labels. Because of that, even when using our scheme, we do not compress all those states which result in more than 5 labeled transitions and could therefore not be encoded with the defined labels. The results are represented in Table 5. It can be observed that the memory requirement of our scheme is better than that of CD^2FA, by a factor varying from 2 to 10. Thus, CD^2FA pays the better worst case time bound (one state traversal per character vs. the two of our scheme) in terms of greater memory requirement. In general, the CD^2FA scheme is bound to a precise state encoding, whereas our results are more general and broadly applicable.

7. RELATED WORK

Sommer and Paxson [20] were among the first to point out that the use of regular expressions can be substantially more effective than exact-match strings when specifying attack signatures.

In addition to the proposals already described, work in accelerating regular expressions has focused essentially on two distinct directions: FPGA-based implementations [21][22][23] and software-oriented approaches. The latter are amenable for deployment on general purpose processors or on small on-chip lookup engines coupled with off-chip memory banks [10][12][18][9][15][16]. The work presented in this paper locates itself in this second category.

In the context of FPGA implementations, Sidhu and Prasanna [21] showed that, if each state is encoded through a flip-flop, regular expression matching can be performed using a non deterministic automaton (NFA) in linear time, without encountering the state blowup issues related with the use of DFAs. Complexity is moved into the necessity to properly route signals on the FPGA.

Some advantages of software-based solutions are: their versatility, their limited cost of implementation, and the fact that they can be run at the higher clock rates associated with processors.

In order to achieve deterministic performance, a software-based solution must make use of DFAs, whose size can grow exponentially with the complexity of the regular expressions they recognize. Memory requirements, both in terms of occupancy and bandwidth, play an essential role in bounding the performance of these systems. As discussed, effective compression techniques [9][15][16] to handle this problem have been proposed. This work represents a refinement and, in a sense, a generalization of the compression scheme proposed in [9].

8. CONCLUSIONS

In summary, in this work we propose a compression technique for DFAs which ensures at most 2N state traversals when processing a string of length N. Experiments on practical data-sets from several network intrusion detection systems show a level of compression comparable to that of D^2FAs [9] with the advantage of a provably better worst case bound on the processing time.

In addition to the strong quantitative performance results, the proposed scheme has substantial qualitative benefits, including greater generality, simplicity and lower complexity. In contrast to related work, the algorithm can be used also in scenarios where a DFA needs to be built dynamically or updated often. Finally, by fostering locality during traversal, the scheme is amenable to implementation in processor-based networking systems, such as Cisco's Silicon Packet Processor [25] and Intel's IXP network processors [1], where caches or fast memories are closely coupled to each processor core.

9. ACKNOWLEDGEMENTS

This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794, and by gifts from Intel and Cisco Systems.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Communication of ACM, 1975.

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

[3] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton," in Theory of Machines and Computation, J. Kohavi, Ed. New York: Academic, 1971, pp. 189--196.

[4] R. Prim, "Shortest connection networks and some generalizations," Bell System Technical Journal,, 1957.

[5] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," Proc. of the American Mathematical Society, Vol. 7, No. 1. (Feb., 1956), pp. 48-50.

[6] Y. J. Chu and T. H. Liu, ``On the shortest arborescence of a directed graph", Science Sinica, v.14, 1965, pp.1396-1400.

[7] J. Edmonds, ``Optimum branchings", J. Research of the National Bureau of Standards, 71B, 1967, pp.233-240.

[8] R. E. Tarjan, "Data Structures and Network Algorithms," SIAM 1983

[9] S. Kumar et alt., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in ACM SIGCOMM, Sept 2006.

[10] M. Roesch, "SNORT: Lightweight Intrusion Detection for Networks," in 13th System Administration Conf., Nov 1999.

[11] SNORT: http://www.snort.org

[12] Bro: A System for Detecting Network Intruders in Real-Time. http://www.icir.org/vern/bro-info.html

[13] Cisco Systems. Cisco Adaptive Security Appliance. http://www.cisco.com. 2007.

[14] Citrix Systems. Citrix Application Firewall. http://www.citrix.com. 2007.

[15] F. Yu et alt., “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection”, in ANCS 2006

[16] S. Kumar, J. Turner and J. Williams, “Advanced Algorithms for Fast and Scalable Deep Packet Inspection”, in ANCS 2006

[17] S. Sen et alt., “Accurate, Scalable In-network Identification of P2P Traffic Using Application Signatures”, in Proc. of the XIII Intl. WWW Conf., New York City, May 2004

[18] N. Tuck et alt., “Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection”, in IEEE Infocom, pp. 333.340, Mar 2004.

[19] G. Varghese, “Network Algorithms: An Interdisciplinary Approach to Designing Fast Networked Devices”, Morgan Kaufmann, 1st ed., 2004.

[20] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context.”, in ACM CCS 2003

[21] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs", in FCCM 2001.

[22] C. R. Clark and D. E. Schimmel, “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” In FPL 2003.

[23] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, “Implementation of a content-scanning module for an internet firewall,” in FCCM, Napa, CA, USA, April 2003.

[24] http://www.tensilica.com

[25] Cisco Systems. Silicon Packet Processor in the CRS-1 Router. http://www.cisco.com/en/US/products/ps5763/index.html

[26] M. Adiletta et al., “The Next Generation of Intel IXP Network Processors”, in Intel Tech. Journal, Vol. 6, Iss 3, 2002.