A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation

MICHELA BECCHI, University of Missouri

PATRICK CROWLEY, Washington University in St. Louis

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. Kumar et al. [2006a] have proposed Delayed Input DFAs (D²FAs), which provide a trade-off between the memory requirements of the compressed DFA and the number of states visited for each character processed, which in turn affects the memory bandwidth required to evaluate regular expressions.

In this article we introduce *Amortized time–bandwidth overhead DFAs (A–DFAs)*, a general compression technique that results in at most $N(k + 1)/k$ state traversals when processing a string of length N , k being a positive integer. In comparison to the D²FA approach, our technique achieves comparable levels of compression with lower provable bounds on memory bandwidth (or greater compression for a given bandwidth bound). Moreover, the A-DFA algorithm has lower complexity, can be applied during DFA creation, and is suitable for scenarios where a compressed DFA needs to be dynamically built or updated. Finally, we show how to combine A-DFA with alphabet reduction and multistride DFAs, two techniques aimed at reducing the memory space and bandwidth requirement of DFAs, and discuss memory encoding schemes suitable for A-DFAs.

Categories and Subject Descriptors: C.2.2 [Computer Communication Networks]: General—Security and protection (e.g., firewalls)

General Terms: Algorithms, Design, Performance, Security

Additional Key Words and Phrases: Deep packet inspection, regular expressions, deterministic finite automata, memory compression

ACM Reference Format:

Becchi, M. and Crowley, P. 2013. A-DFA: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim.* 10, 1, Article 4 (April 2013), 26 pages.

DOI: <http://dx.doi.org/10.1145/2445572.2445576>

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 [Roesch 1999] and Bro [Paxson 1999]—and

This article is an extension of the conference paper “An improved algorithm to accelerate regular expression evaluation” presented at the *3rd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS’07)*.

Authors’ addresses: M. Becchi, Department of Electrical and Computer Engineering, University of Missouri; email: becchim@missouri.edu; P. Crowley, Department of Computer Science and Engineering, Washington University in St. Louis; email: pcrowley@wustl.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/04-ART4 \$15.00

DOI: <http://dx.doi.org/10.1145/2445572.2445576>

devices—including the Cisco family of Security Appliances [Cisco Systems 2011] and the Citrix Application Firewall [Citrix Systems 2007]—use regular expressions to describe payload patterns. In addition, application-level signature analysis has been proposed as an accurate means to detect and track peer-to-peer traffic, enabling sophisticated packet prioritization mechanisms [Sen et al. 2004].

Regular expressions are more expressive than simple patterns of strings and therefore able to describe a wider variety of payload signatures [Sommer and Paxson 2003], 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 (DFAs).

DFAs have attractive properties that explain the attention they have received. Foremost, they have deterministic memory bandwidth requirements. In fact, the use of DFAs allows one single state traversal, 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 constructed [Hopcroft 1971]. Even so, DFAs corresponding to sets of complex patterns can be prohibitively large in terms of numbers of states and transitions. In particular, specific subpatterns, namely constrained and unconstrained repetitions of wildcards and large character sets, may lead to state explosion during DFA generation [Becchi and Crowley 2007b; Yu et al. 2006].

Limiting hardware resource requirements is the main design issue when using DFAs to perform regular expression matching. In particular, two aspects have been studied in the literature. First, *compression mechanisms* aimed at minimizing the DFA memory footprint have been designed. Such schemes do not aim at avoiding state explosion. Instead, their goal is to allow an effective representation of feasible DFAs, that is, of DFAs that, having a limited number of states, can be generated on reasonable hardware. These DFA compression schemes typically leverage the transition redundancy characterizing practical DFAs. Second, *novel automata* to be used as an alternative to DFAs in case of state explosion have been proposed. Alphabet reduction [Becchi and Crowley 2007a; Brodie et al. 2006; Kong et al. 2008], run-length encoding [Brodie et al. 2006], default transition compression [Becchi and Crowley 2007a; Kumar et al. 2006a, 2006b], state merging [Becchi and Cadambi 2007] and δ FAs [Ficara et al. 2008] are mechanisms falling into the first category; multiple-DFAs [Brodie et al. 2006; Yu et al. 2006], hybrid-FAs [Becchi and Crowley 2007b, 2008a], history-based-FAs [Kumar et al. 2007a], and XFAs [Smith et al. 2008] fall into the second one.

In this article, we focus on the first problem. In particular, we elaborate on default transitions, an abstraction that has first been proposed in Kumar et al. [2006a] to target rule-sets from the networking domain. In particular, Kumar et al. [2006a] observe that many states in DFAs derived from practical datasets have similar sets of outgoing transitions. Substantial space savings in excess of 90% are achievable when this redundancy is exploited. The proposed automaton, called a Delayed Input DFA (D^2 FA), 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: (i) it requires a user-provided parameter value which can only be determined experimentally for a given rule-set, (ii) it creates a data structure whose worst-case paths may be traversed for each input character processed,

and (iii) it requires multiple passes over large support data structures during the construction phase.

In this work, we propose *Amortized time/bandwidth overhead DFAs (A-DFAs)*, 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. Moreover, the A-DFA construction algorithm can be generalized to result in an even lower memory bandwidth requirement: specifically, it will lead to a worst case of $N(k+1)/k$ state traversals to process a string of length N , k being a positive integer. Note that, for increasing values of k , the memory bandwidth requirement approaches that of an uncompressed DFA. On practical datasets, the level of compression achieved is similar to that of the original D²FA scheme, while the worst-case memory bandwidth bound is superior. Moreover, when D²FA and A-DFA are configured to achieve the same worst-case memory bandwidth bound, A-DFA yields a tenfold improvement in compression relative to D²FA.

The A-DFA approach is based on a simple observation: all regular expression evaluations begin at a single initial state and the vast majority of transitions between states are oriented backward, leading to the entry 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 D²FA technique. Furthermore, by formalizing the notion of *state depth* to quantify a state's distance from the initial 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 the A-DFA 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 transformation—which eliminates the need to either first generate a perhaps unfeasibly large uncompressed DFA or to maintain 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. Second, we show how to combine A-DFA with alphabet reduction and multistride DFAs, two techniques aimed at reducing the memory space and bandwidth requirement of DFAs. Third, we discuss memory encoding schemes suitable for A-DFAs, focusing on the ones allowing one memory access per state traversal.

To our knowledge, the combination of A-DFA and alphabet reduction represents the most efficient and practical DFA compression scheme proposed to date for regular expression evaluation in high-speed networking contexts. As anticipated, state explosion may take place when trying to compile datasets containing complex regular expressions into a single DFA. However, the most notable proposals that address the state explosion problem [Becchi and Crowley 2007b, 2008a; Brodie et al. 2006; Kumar et al. 2007a; Smith et al. 2008; Yu et al. 2006] use DFAs as building blocks. Therefore, A-DFA can be applied in combination with any of them.

The remainder of this article is organized as follows. In Section 2, we give background on the D²FA technique. In Section 3, we present A-DFA. In Section 4, we present a low-complexity alphabet reduction algorithm. In Section 5, we discuss several encoding schemes which can be used to represent an A-DFA. In Section 6, we show how to combine A-DFA with multistride DFAs. In Section 7, we present an experimental evaluation on datasets used in the Snort and Bro tools and also in the Cisco security appliance. We then relate our work to the state-of-the-art (Section 8) and conclude (Section 9).

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

2. BACKGROUND : DELAYED INPUT DFAS (D²FAS)

Our proposal is based on the concept of default transitions, an abstraction also used in automata designed for exact string match supporting token scanning in compilers and other parsers. Kumar et al. [2006a] have proposed D²FA, an algorithm to construct default transitions in automata accepting regular languages in order to meet the requirements of networking applications. Our scheme can be considered a refinement and a generalization of D²FA.

The basic goal of the D²FA scheme 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 size $|\Sigma|$. Such a DFA will contain $N \cdot |\Sigma|$ next state transitions. Kumar et al. [2006a] 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 $S = \{s_i, \dots, s_k\}$ for the same set of characters $C = \{c_i, \dots, c_k\}$. 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 the $|\Sigma| - |S|$ labeled transitions which are not in common with s_1 . This example is illustrated in Figure 1.

During the string matching operation, the traversal of the compressed DFA will be performed according to the traditional Aho-Corasick algorithm [Aho and Corasick 1975], treating default transitions as failure pointers. In the example, if character c is received while in state s_2 , all the 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 D²FA. Therefore, the introduction of default transitions introduces compression at the cost of increased memory bandwidth.

The problem of constructing default transitions so as to maximize compression and minimize the memory bandwidth overhead is reduced in Kumar et al. [2006a] to finding a maximum spanning forest with a given diameter bound DB in an undirected graph. The diameter bound represents twice the number of allowed extra memory accesses per character processed. Intuitively, the higher the DB , the higher the required memory bandwidth, and the less constrained the problem, the better the compression

Fig. 2. DFA accepting regular expressions: ab^+c^+ , cd^+ and bd^+e . Omitted transitions lead to the initial state 0. Accepting states are colored gray. The depth information is represented in square brackets.

achieved. The proposed algorithm exhibits $O(n^2 \log n)$ time complexity and $O(n^2)$ space complexity, n being the number of states in the DFA.

3. AMORTIZED BANDWIDTH-TIME OVERHEAD DFAS (A-DFAS)

In this section, we explain our A-DFA scheme in detail. In Section 3.1, we prove that A-DFA construction provides worst-case traversal guarantees (and, therefore, bounds on the memory bandwidth requirements). In Section 3.2, we illustrate a formulation of the A-DFA construction problem leading to the algorithm described in Section 3.3. In Section 3.4, we discuss various A-DFA properties.

3.1. Rationale

To explain our A-DFA scheme and prove its worst-case traversal guarantees, we need to introduce the notion of state depth. For each state s , 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 2 reports the depth information for the DFA accepting regular expression ab^+c^+ , cd^+ and bd^+e . 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.

The A-DFA construction algorithm is based on the following lemma.

LEMMA L1. *If none of the default transitions in a compressed DFA leads from a state with depth d_i to a state with 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 compressed DFA having only backward default transitions. In a sense, this can be thought of as a generalization of Aho and Corasick [1975] to regular expressions. Note that the lemma only prevents default transitions between states at the same depth.

The proof of Lemma L1 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 d 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 greater or equal to d . Thus, to get to state s , at least d 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 than $2N-1$.

Notice that the presence of backward labeled transitions does not affect the proof. In fact, this implies that if a state at depth d is visited at a point, then at least (and not exactly) d labeled transitions must have been taken before. In other words, backward labeled transitions contribute to improve the average case with respect to the worst case.

This formulation can be generalized by requiring that default transitions skip at least k depth levels, k being a *strictly positive* parameter. This generalization implies an even smaller amortized traversal complexity, which translates into a lower memory bandwidth requirement. In particular, the following lemma holds.

LEMMA L2. *If all default transitions in a compressed DFA lead from a state at depth d_i to a state at depth d_j with $d_j \leq d_i - k$ and k being a strictly positive integer, then any string of length N will require at most $N(k+1)/k$ state traversals to be processed.*

Note that our first formulation corresponds to setting the parameter k to 1. States with depth smaller than k will not have a default transition. For values of k greater than 1, the average number of state traversals per character becomes close to 1, at the cost of a decreased memory compression.

The proof of Lemma L2 is straightforward. The worst case in the traversal corresponds to the minimum number of labeled transitions that can be taken before following a default transition. Since each default transition must skip at least k depth levels, in the worst case a default transition will be taken every k labeled transitions. That means $k+1$ transitions will be needed to process k characters, leading to an amortized traversal complexity of $N(k+1)/k$ state traversals to process an input string of length N .

An even more generic formulation of the problem is given by the following lemma.

LEMMA L3. *Assume a support graph having a node for each DFA state, an edge with length -1 for each default transition, and an edge with length $1/k$ for each labeled transition in the compressed DFA. If the support graph contains no negative loops, then the amortized time to process a string of length N using the compressed DFA is $N(k+1)/k$.*

The proof of Lemma L3 follows. Since there are no negative loops in the support graph, the shortest loop L_0 has total length 0, and consists of k labeled and one default transitions. The worst case in the traversal corresponds to the string that takes the zero-length loop L_0 repeatedly. In fact, taking L_0 implies taking a default transition as frequently as possible for the given compressed DFA. Since taking L_0 implies taking $k+1$ transitions every k input characters, the worst-case traversal cost is $N(k+1)/k$.

Note that Lemma L3 does not require all default transitions to be backward directed and to skip k depth levels. Thus, being less constrained, this formulation can provide better memory compression (for the same k) than the formulation in Lemma L2. In Sections 3.2 and 3.3, we show that a compressed DFA with the properties of Lemma L2 can be computed using an $O(n^2)$ -time algorithm, n being the number of DFA states. Finding an optimal default transition compressed DFA using Lemma L3 is a harder problem which can be tackled through computationally intensive heuristics (rather

than through an optimal algorithm). For example, Lemma L3 can be used to refine a compressed DFA built according to Lemma L2, starting from the states characterized by a higher number of labeled transitions. In this work, we focus on a space- and time-efficient algorithm to compute a compressed DFA according to Lemma L2 (as mentioned, L1 is a special case of L2). We show that the nature of DFAs from practical regular expression sets makes such algorithm not only efficient, but also effective.

3.2. Problem Formulation

The goal of the A-DFA scheme is to build a default transition compressed DFA with the traversal properties either of Lemma L1 or of Lemma L2. In other words, we want to answer the following question: how do we set the default transitions in a DFA so to achieve maximum compression and ensure a given worst-case traversal cost? Lemmas L1 and L2 imply that the default transitions must be backward directed (and skip k depth levels).

The problem of building an A-DFA according to Lemmas L1 and L2 can be formulated as an instance of a maximum spanning tree on a *directed* graph, that we will call *space reduction graph*. Specifically, the space reduction graph will be built as follows. First, it will have a node for each state in the DFA. Second, it will have an edge connecting a pair of nodes if the corresponding DFA states have depth difference greater than or equal to k . Third, each edge will be oriented from the node associated to the state with larger depth to the node associated to the state with smaller depth. Finally, if s_X and s_Y are DFA states and n_X and n_Y the corresponding nodes in the space reduction graph, the edge connecting n_X and n_Y will have a weight equal to the number of common transitions between s_X and s_Y .

Every edge in the space reduction graph represents a possible default transition, and its weight represents the number of labeled transitions that can be removed from the DFA by adding that particular default transition. Since each DFA state can have only one outgoing default transition, selecting a set of viable default transitions for a DFA is equivalent to determining a spanning tree (or forest) in its space reduction graph. The spanning tree (or forest) with maximum weight is the one leading to the removal of the most labeled transitions in the DFA (that is, to the highest level of compression). Therefore, the optimal default transition selection corresponds to the maximum spanning tree in the space reduction graph. 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 required. This is in contrast to the D²FA scheme [Kumar et al. 2006a].

An example of default transition construction with the proposed scheme is given in Figure 3. In particular, Figure 3(a) represents the space reduction graph when the parameter k is set to 1. The root of the maximum spanning tree (represented in bold) is the initial state 0. The corresponding A-DFA is shown in Figure 3(b): 33 labeled transitions are saved. The interested reader can refer to Becchi and Crowley [2007a] to see the application of the D²FA algorithm [Kumar et al. 2006a] to the example of Figure 3. To obtain the same level of compression the diameter bound parameter must be set to 4, leading to a traversal complexity of $3N$.

Two similar algorithms to find the optimal solution to the directed maximum spanning tree problem have been proposed by Chu et al. [1965] and Edmonds [1967]. 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 pseudonode, and

Fig. 3. (a) Space reduction graph for DFA in Figure 2. The maximum spanning tree is highlighted in bold. The parameter k is set equal to 1; (b) resulting A-DFA. All transitions are represented. Default transitions are dashed and colored gray.

the weights of the edges exiting the pseudonode are modified. The maximum weight edge exiting the pseudonode 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 time complexity of the A-DFA algorithm depends only on the number of edges in the space reduction graph, that is, $O(n^2)$, n being the number of states in the DFA. Its space complexity is simply $O(n)$.

3.3. Algorithm

While the concept of a space reduction graph is useful to relate this problem to the one solved in Kumar et al. [2006a] 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 (and depth difference greater than or equal to k) having the most outgoing transitions in common with it. In the case of ties, preference is given to the smallest depth. This limits the default transition path length and enforces locality during traversal.

If the DFA traversal is performed in a breadth-first fashion, both default transitions and depth computations can be done in a single pass, as shown in the pseudocode that follows. The DFA is described in terms of the number of states n and of the function $\delta(states, \Sigma) \rightarrow states$, which associates a next state to each (state, character) pair. 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 greater depth can be ignored (initializing their depth to n will therefore ensure correct operation). While not shown in the pseudocode, 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., NFA-to-DFA transformation) so as 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 before. The generation of an initial compressed DFA eliminates the need to first construct an uncompressed one.

ALGORITHM 1: A-DFA construction

Input: DFA $dfa = (n, \delta(states, \Sigma))$; integer k
Output: set $default[n]$;
list $queue$; set $depth[n]$;
for (state $s \in states$) **do** $depth[s]=n$; $default[s]=s$; **end for**
 $depth[0]=0$; $queue.push(0)$;
while ($!queue.empty()$) **do**
 state $s = queue.pop()$;
 int $saving=1$;
 for (char $c \in \Sigma$) **do**
 if ($depth[\delta(s, c)] == n$) **then**
 $depth[\delta(s, c)] = depth[s] + 1$;
 $queue.push(\delta(s, c))$;
 end if
 end for
 for (state $t \in states \ \& \ depth[t] \leq depth[s] - k$) **do**
 int $common = common_transitions(s, t)$;
 if ($common > saving \ || \ (common == saving \ \& \ depth[t] < depth[default[s]])$) **then**
 $default[s] = t$;
 $saving = common$;
 end if
 end for
end while

3.4. Discussion

The compression scheme proposed in this article and the one proposed in Kumar et al. [2006a] can be compared from several perspectives. We want to qualitatively summarize the most important points.

3.4.1. Worst-Case Bound on Memory Size and Bandwidth. As mentioned, while the D²FA scheme trades off memory size with memory bandwidth, the A-DFA algorithm aims at achieving a constant $2N$ worst-case bound on the memory bandwidth. This is comparable with running the D²FA algorithm with a minimal diameter bound of two. Our expectations are: (i) better compression when compared to D²FA with diameter bound equal to two, and (ii) comparable compression when compared with D²FA with higher diameter bound. Two reasons motivate these expectations.

The first is due to the characteristics of DFAs accepting sets of regular expressions. Intuitively, they are characterized by a limited number of forward transitions corresponding to the matching conditions in the described patterns. However, most transitions are backward directed: 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, for instance, most transitions lead to states 0, 1, 4, and 6. In the case of regular expressions with dot-star terms, many transitions fall back to the state the dot-star originates from (and to its close vicinity). Backward-directed default transitions will in general be able to replace backward-labeled transitions, which are the most numerous. More generally, an A-DFA only prevents default transitions between two states with the same depth, but adds no restrictions otherwise.

The second motivation is based on the nature of the problem addressed. Even if the directed-graph problem is more constrained than its undirected counterpart, at least when a high diameter bound is allowed, the A-DFA algorithm finds the optimal solution to it. On the other hand, the D²FA scheme uses a heuristic that can find sub-optimal solutions. In the 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 and Practical Issues. As far as asymptotic complexity is concerned, the D²FA and the A-DFA algorithms have $O(n^2 \log n)$ and $O(n^2)$ time bounds, respectively. The complexity of D²FA is kept low through the use of support data structures (space reduction graph, d -heap, and “partition” data structure [Tarjan 1983]). In practice, this fact has important implications that impact the implementation and the running time of the algorithm itself when large DFAs are addressed.

Among the data structures listed earlier, the largest 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 lnxt, rnxt; /* link to next edge incident to l and 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 [Kruskal 1956]). This, however, leads to the need of several DFA scans which negatively impact the overall running time.

The A-DFA algorithm is not affected by these issues because it does not need this support data structure. 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 that may occur in the future (for instance, if signature generation is made automatic). Additionally, as we will show later, our algorithm is the only practical one when very large alphabets are used, as happens in the case of multistride automata.

3.4.3. Traversal Locality. One additional interesting aspect is that A-DFAs tend to foster locality during the traversal process. A probabilistic analysis of DFAs accepting the real-world regular expressions 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 (and of closure states). The probability of visiting a state with depth d depends of the probability of visiting states leading to it, which must have depth at least $d-1$. Consequently, the probability of visiting a state tends to decrease as the depth increases.

Since the A-DFA 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 necessarily true of a traditional D²FA, where the direction of the default transitions is not controlled and can lead far away from the initial state. The interested reader can find an extensive experimental evaluation of A-DFAs using a variety of cache configurations in Becchi et al. [2008].

3.4.4. Packet- versus Flow-Level Processing. While the original D²FA compression algorithm limits the number of state traversals on every single character processed, our proposed A-DFA algorithm amortizes the extra state traversals due to default transitions across the whole input text. In the worst case, an input character can trigger a number of default transitions equal to the maximum depth in the DFA divided by k .

This fact must be taken into account when sizing the input buffers to the regular expression matching engine. This holds particularly when the packet represents the unit of processing and input flows span multiple packets. It is possible to architect a denial-of-service attack by interleaving malicious flows so as to fill the input buffers. It must be observed that in order for this to happen, parallel flows must switch packets on a deep state. That is, multiple flows matching a large portion of some long patterns must coexist. This situation should obviously alarm the system even in the absence of a full match.

4. ALPHABET REDUCTION

We now present a mechanism to reduce the size of the alphabet and further decrease the number of transitions needed to represent a DFA. This technique is orthogonal to the one previously described: it can be applied to the original DFA before or after the default transition computation is performed.

4.1. The Idea

The basic idea is the following: in a DFA recognizing regular expressions over an alphabet Σ each state has potentially $|\Sigma|$ outgoing transitions, one for each symbol in Σ . However, Σ can be partitioned into classes of symbols $C_1, \dots, C_k$ 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.

Once the class translation $C(\Sigma) \rightarrow \{1..k\}$ has been computed, the alphabet is reduced from size $|\Sigma|$ to size 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 subset of the symbols. The characters that do not appear in any patterns accepted by the DFA often translate into backward 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*].

We describe next 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.

4.2. The Algorithm

The basic idea is to build the character translation information by performing cluster-division. Specifically, we initially assume to have a single character class, say 0. We loop over the states and analyze 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 alphabet reduction algorithm is shown in Algorithm 2. The resulting alphabet translation table is represented through the parameter *class*, which maps each character of the initial alphabet Σ to the corresponding character class. The algorithm

ALGORITHM 2: Alphabet Reduction

Input: DFA $dfa = (n, \delta(states, \Sigma))$
Output: set $class[|\Sigma|]$;
int $alphabet_size = 0$;
for (**state** $s \in states$) **do**
 for (**state** $t \in states$) **do**
 set $char_covered[|\Sigma|] = \text{false}$;
 set $class_covered[|\Sigma|] = \text{false}$;
 set $remap[|\Sigma|] = 0$;
 for (**char** $c \in \Sigma \ \& \ \delta(s, c) = t$) **do**
 $char_covered[c] = \text{true}$;
 $class_covered[class[c]] = \text{true}$;
 end for
 for (**char** $c \in \Sigma$) **do**
 if ($!char_covered[c] \ \& \ class_covered[class[c]]$) **then**
 if ($remap[class[c]] == 0$) **then** $remap[class[c]] = ++alphabet_size$; **end if**
 $class[c] = remap[class[c]]$;
 end if
 end for
 end for
end for

operates by scanning the DFA. For each pair of states (s, t) , variables $char_covered$ and $class_covered$ store the information about the characters and character classes covered by transitions from s to t , while the variable $remap$ keeps information about necessary class remapping. On every iteration, a character is remapped to a new class if its class is covered by the transitions between the current state-pair but the character itself is not.

The time complexity of the algorithm is $O(n^2)$ —more precisely $O(|\Sigma|n^2)$, while its space complexity is again $O(n)$. The alphabet reduction algorithm can be combined with the A-DFA compression algorithm and performed during NFA-to-DFA transformation.

5. MEMORY ENCODING

An orthogonal design problem consists of selecting the concrete data structure used to perform regular expression matching; such a data structure must account for every bit needed to represent the DFA.

Given an alphabet Σ of size $|\Sigma|$, a naive solution represents a state s as a list of $|\Sigma|$ next state pointers. However, this approach does not leverage the fact that when default transitions are used, most states have only a restricted number of outgoing transitions.

Alternatively, we consider three techniques to encode default transition compressed DFAs: (i) *linear encoding*, (ii) *bitmapped encoding*, and (iii) *indirect addressing* (a slight variation of *content addressing* [Kumar et al. 2006b]). These encoding schemes access memory differently and thus differ in the number of memory accesses required for each state traversal. While each of these techniques can be tuned, for the sake of simplicity and comparison we assume the use of a 32-bit word-aligned memory layout and avoid DFA-specific optimizations.

5.1. Linear Encoding

If linear encoding is used, a state with l labeled transitions is encoded through $l+1$ 32-bit words with the first one representing the default, and the others the

remaining transitions. Each word has one bit indicating whether the transition is the last one within the state, 8 bits representing the input character upon which the transition must be taken (in case of labeled transitions), and the remaining bits devoted to the next state address. A state traversal starts from the first word, and involves going through the transitions in sequence until the one matching the input character is found or its absence is verified. When using linear addressing, a threshold t_Σ is used. States having more than t_Σ outgoing transitions are fully represented through $|\Sigma|$ pointers (and accommodated in a separate memory region) to allow fast access.

5.2. Bitmapped Encoding

Bitmaps [Varghese 2004] admit a reasonable upper bound on the number of memory accesses needed to process a character. Specifically, in this context a bitmap is an array of $|\Sigma|$ bits, each one indicating whether the corresponding transition exists or not. Each state is encoded through a bitmap and a sequence of $l+1$ memory words, each one representing a next state pointer. Upon state traversal, the bitmap is first analyzed. If it contains a 0 in the position of the input character, then only a direct access to the default transition is performed. Otherwise, a pop-count of the number of 1s preceding the current position is made, and this information is used to directly access the proper next state transition. Several techniques to compress bitmaps have been proposed [Varghese 2004]. As in the linear encoding case, states with a high number of outgoing transitions should be fully encoded.

5.3. Indirect Addressing

Indirect addressing can be used to further reduce memory bandwidth requirements. Specifically, this scheme allows a single memory access (and a hash computation) per state traversal. Indirect addressing associates to each state a unique identifier consisting of: (i) the list of characters upon which there exists an outgoing transition, and (ii) a set of bits, called a *state discriminator*. The discriminator is introduced to ensure that all state identifiers are different, even for states having labeled transitions on the same set of characters.

An indirection operation is required in order to translate a state identifier into a memory address. Specifically, the base memory address of each state is computed by applying a hash function to the state identifier. The interested reader can refer to Kumar et al. [2006b] for an explanation on how to map state identifiers to memory addresses in a manner that ensures good memory utilization.

When a state is traversed, the characters in the state identifier determine the outgoing transitions. The identifiers of the targets of the outgoing transitions are stored contiguously in memory. The order of the characters in the state identifier is used to perform direct memory accesses to the next state.

In our work paper, 8-bit discriminators are assumed. Two possible configurations are allowed: (i) 32-bit and (ii) 64-bit state identifiers. In the former case, states with more than 3 outgoing transitions must be fully represented. In the latter case, this limit is moved to 7.

An example of indirect addressing is shown in Figure 4. The identifiers of the states in Figure 4(a) and the corresponding memory layout are shown in Figure 4(b) and (c), respectively. State 1, for example, has outgoing transitions on characters a , b , and c . As a consequence, its identifier is built concatenating a , b , c , and an 8-bit discriminator (D_1). The base address of state 1 is computed by applying the hash function h to its identifier. Finally, for state 1, the memory contains the identifier of target state 2 (for transitions on characters a and b) and that of target state 3 (for transition on character c).

Fig. 4. An example of indirect addressing (focused only on labeled transitions). In particular: (a) portion of a DFA (only labeled transitions are shown); (b) state identifiers associated to states 1, 2, and 3; (c) memory layout. In the state identifiers, D_1 , D_2 , and D_3 are 8-bit discriminators. Each state identifier contains the characters of the corresponding outgoing transitions. The base memory address of each state is determined by hashing the corresponding identifier (through hash function h).

6. MULTISTRIDE DFAS

Default transition compression and alphabet reduction can be applied in combination with a technique aimed at reducing memory bandwidth requirements, namely multistride DFAs. The idea is to allow a DFA to process multiple input characters at the same time. Stride doubling has been proposed in Brodie et al. [2006] in the context of: (i) small DFAs consisting of 40–100 states, and (ii) a compressed alphabet of limited size. The proposal targets ASIC architectures where it is conceivable to have multiple DFA engines running in parallel. In this work, we aim at studying the feasibility of multistride DFAs in contexts where the underlying patterns are more complex and numerous.

6.1. Addressing Multistride DFAs

A DFA of stride k , or k -DFA, is meant to process k input characters at a time. Intuitively, the k -DFA can be constructed by tracing, from each state in the original DFA, all possible k -character combinations and the corresponding target states. The basic problem in building a k -DFA lies in the memory storage requirement. In fact, given a DFA defined on an alphabet Σ of size $|\Sigma|$, the noncompressed k -DFA will have $|\Sigma|^k$ outgoing transitions per state. Let us assume, for the sake of example, we have a 1K-state DFA defined on the ASCII alphabet. The full 2-DFA version will consist of about 65 million transitions, whereas the full 4-DFA will present about 4.3 trillion transitions. When compiling several regular expressions together, DFAs from practical datasets easily reach 100K states.

To make a k -DFA feasible, we take advantage of alphabet reduction and A-DFA. In particular, the use of alphabet reduction (also exploited in Brodie et al. [2006] using a different reduction algorithm) can be motivated by observing that when the stride increases, the automaton in effect uses only a small subset of the entire alphabet. The benefit of A-DFA can be explained as follows. Default transition compression acts by removing transition redundancy present in a DFA. In other words, it takes advantage of the fact that different states transition to the same target on the same input character. If the stride doubles, then the number of transitions in the DFA increases quadratically, but the number of states does not. Intuitively, the fraction of distinct transitions decreases. Therefore, transition redundancy tends to increase with the stride.

Fig. 5. DFA (a) and stride-2 DFA; (b) accepting regular expression ab^+cd^+ . Accepting states are colored gray. Omitted transitions lead to state 0. In the stride-2 DFA, outgoing transitions from state 0' are the same as for state 0, and the ones from state 1' are the same as for state 1. A wildcard in the character pair is represented through a dot (.).

Fig. 6. Steps involved in building a k -DFA. Each alphabet translation table ($TxTable$) depends on the previous one.

In Figure 5(b) we show the stride-2 DFA corresponding to the DFA in Figure 5(a). Every transition in the 2-DFA is defined on a character pair. Note that, in several cases, one of the two symbols in the pair covers a large character range or is a wildcard. Also, notice that the original DFA is defined on alphabet $\{a, b, c, d, \text{remaining characters}\}$ of size 5. After alphabet reduction is performed, the 2-DFA is defined on alphabet $\{ab, bb, bc, cd, dd, da, [^d]a, d[^ad], \text{remaining character pairs}\}$ of size 9. This leads to two observations. First, the sizes of the compressed alphabets (5 and 9 in the example) are far smaller than those of their uncompressed counterparts (256 and 65,536, respectively). Second, stride doubling does not increase quadratically the size of the reduced alphabet (9 is less than 25).

Alphabet reduction and default transition compression are used not only to compress the k -DFA, but also to make building a k -DFA feasible with reasonable hardware resources. In particular, we find it necessary to concurrently implement stride doubling and default transition compression in order to avoid troublesome memory size requirements when constructing the automaton. Moreover, alphabet reduction is performed after each stride doubling, as shown in Figure 6. Note that the different alphabet translation tables ($TxTable_i$) can be combined to obtain the global alphabet translation $(\Sigma_1)^k \rightarrow \Sigma'_k$.

It must be observed that combining stride doubling with default transition compression is feasible since the A-DFA algorithm is based on a breadth-first traversal of the DFA (and does not require additional data structures). This would not be possible using the D²FA scheme, which requires a fully generated DFA and an additional data structure with size quadratic in the number of DFA states. We observed that performing default transition compression *after* stride doubling on medium-size DFAs is infeasible on reasonably equipped machines; we have used a Linux machine with a 4GB main memory. When processing a 2K-state DFA, we ran out of memory during creation

of the stride-4 version (with a reduced alphabet) whereas only 10% of the memory was used when A-DFA compression was combined with stride doubling.

One last problem to be addressed when generating a k -DFA is the preservation of correct accepting states. At each state traversal we want to know *which* regular expressions, if any, are matched. This information can either be associated with the states or with the transitions. We opt for the first alternative, because the second would make the number of possible transitions explode; on each character, we can have a nonmatching transition or a transition accepting any possible combination of rules. Note that our formulation differs from that in Brodie et al. [2006], whose goal is to determine whether a match occurs, but not to specify the matched patterns. Let us assume that, when doubling the stride, we transition from state s_x to state s_y moving through state s_w , which accepts rule j . Since s_y was originally nonaccepting, we must create a new state $s_y^{[j]}$ accepting j , and associate the transitions to it. The k -DFA will have more states than the original one. However, s_y and $s_y^{[j]}$ will have the same outgoing transitions. Therefore, $s_y^{[j]}$ will account for a single default transition connecting it to s_y (it can be shown that s_y has always depth smaller than $s_y^{[j]}$). If the rule-set contains r rules and the original DFA has n states, then in the worst case $n \sum_{k=1}^r C_r^k = n \sum_{k=1}^r \binom{r}{k}$ new accepting states consisting of a single default transition can be created.

6.2. Multistride DFA Generation Algorithm

The resulting DFA stride doubling algorithm, which combines stride doubling with default transition compression, is illustrated in the following pseudocode.

The original DFA dfa is fully represented through its transition function $\delta(state, \Sigma)$, whereas the computed dfa' is compressed and represented through its set of default and labeled transitions (*default tx* and *labeled tx*, correspondingly). A mapping between states and accepted rules is used in both cases (*rules* and *rules'*). Note that each state can accept zero, one, or more regular expressions.

The algorithm traverses the original DFA in a breadth-first fashion. For each state, we first compute the outgoing transitions (**for** (*char c1* $\in \Sigma$) loop), and then we perform default transition compression (**for** (*state t* $\in all$ & *depth[t]* < *depth[s]*) loop).

The *mapping* variable is used to determine when a new accepting state has to be created. Note that new accepting states are assigned a default transition upon creation (line *default tx[ns]=t2; labeled tx[ns]= $\emptyset$* ;) and are not included in the traversal (line **if** (*t2* $\in all$ & *t2* < *size(states)*)...).

The time complexity of the proposed stride doubling algorithm corresponds with that of default transition compression, that is, $O(n^2)$, n being the number of DFA states.

7. EXPERIMENTAL EVALUATION

We now show the results of an experimental evaluation conducted on practical datasets extracted from the Snort [Roesch 1999] and Bro [Paxson 1999] network intrusion detection systems and the Cisco security appliance [Cisco Systems 2011]. As highlighted in the Introduction, A-DFA is a compression mechanism for DFAs, and not a scheme to avoid state explosion during DFA generation. Therefore, our experimental evaluation is performed on sets of patterns that do not incur substantial state explosion when generating a DFA. Larger and more complex rule-sets require the use of alternative automata to limit state explosion, such as multiple DFAs [Yu et al. 2006], hybrid-FA [Becchi and Crowley 2007b, 2008a], history-based-FA [Kumar et al. 2007a], and XFAs

ALGORITHM 3: Stride doubling

Input: DFA $dfa = (n, \delta(states, \Sigma), rules(states))$
Output: A-DFA $adfa' = (default_tx, labeled_tx, rules'(states'))$
list $queue, all$; **set** $depth, tx$; **state** $ns = size(states)$;
map $mapping : (states, set) \rightarrow states'$;
for ($state\ s \in states$) **do** $mapping(s, rules(s)) = s$; $rules'(s) = rules(s)$; **end for**
 $queue.push(0)$; $all.push(0)$; $depth[0] = 0$;
while ($!queue.empty()$) **do**
 $state\ s = queue.pop()$;
 for ($char\ c1 \in \Sigma$) **do**
 $state\ t1 = \delta(s, c1)$;
 for ($char\ c2 \in \Sigma$) **do**
 $state\ t2 = \delta(t1, c2)$;
 if ($mapping(t2, rules(t1) \cup rules(t2)) = \lambda$) **then**
 $mapping(t2, rules(t1) \cup rules(t2)) = ns++$;
 $rules'(ns) = rules(t1) \cup rules(t2)$;
 $default_tx[ns] = t2$; $labeled_tx[ns] = \emptyset$;
 end if
 $t2 = mapping(t2, rules(t1) \cup rules(t2))$;
 $tx(s, c1c2) = t2$;
 if ($!t2 \in all \ \& \ t2 < size(states)$) **then**
 $queue.push(t2)$; $all.push(t2)$;
 $depth[t2] = depth[s] + 1$;
 end if
 end for
 end for
 $int\ saving = 1$;
 for ($state\ t \in all \ \& \ depth[t] < depth[s]$) **do**
 if ($common_tx(s, t) > saving$) **then** $default_tx[s] = t$; $saving = common_tx(s, t)$; **end if**
 end for
 $labeled_tx[s] = uncommon_tx(s, default_tx[s])$;
end while

Table I. Characteristics of Considered Rule-Sets

Data set	# of RegEx	ASCII length range	% RegEx w/ wildcards and large char sets	% RegEx w/ char sets ≥ 5
<i>bro217</i>	217	5..76	1.4	13.4
<i>cisco11</i>	11	9..13	90.9	9.1
<i>cisco43</i>	43	15..73	32.6	27.9
<i>cisco612</i>	612	3..57	0	1.6
<i>snort25</i>	25	10..52	16	96
<i>snort50</i>	50	2..54	12	90
<i>snort24</i>	24	7..36	45.8	66.7

[Smith et al. 2008]. Since all these representations use DFAs as building blocks, the A-DFA technique can be applied in combination with any of them.

In our experiments, Snort rules have been filtered according to the header (tcp, \$HOME_NET, any, \$EXTERNAL_NET, any). Regular expressions have been partitioned into seven sets so as to obtain both DFAs with reasonable size and sets of DFAs with different characteristics in terms of number of wildcards, frequency of character sets, and so on. The basic characteristics of the resulting datasets are summarized in Table I.

Table II. Comparison between the Compression Achieved through the D²FA and the A-DFA Schemes

Data set	Original DFA		D ² FA						A-DFA	
	# states	% distinct trans	Compression					Max def. path length	Compression	Max def. path length
			DB = 2	DB = 6	DB = 10	DB = 14	DB = ∞			
<i>bro217</i>	6,533	0.4	21.7	1.9	0.8	0.6	0.6	8	0.7	5
<i>cisco11</i>	24,007	2.6	13.4	2.9	2.6	2.6	2.6	14	3.4	8
<i>cisco43</i>	19,134	0.9	13.0	1.4	0.9	0.9	0.9	14	1.0	8
<i>cisco613</i>	11,312	0.5	19.5	2.4	1.1	0.8	0.8	12	0.9	5
<i>snort25</i>	38,499	0.9	20.8	2.5	1.5	1.5	1.5	12	1.5	6
<i>snort50</i>	15,176	1.2	24.0	4.3	1.9	1.5	1.4	15	1.9	8
<i>snort24</i>	10,819	2.9	16.8	3.1	3.0	3	3.0	21	3.5	16

Different values of the diameter bound (*DB*) are reported in case of D²FA scheme (*DB* = ∞: no bound is used). For A-DFA, parameter *k* is set to 1. Compression is expressed as the ratio between the number of final and original transitions, in percent terms.

7.1. Compression

Our first goal is to quantify the memory compression achieved through the A-DFA scheme and compare it to that of the D²FA scheme. To this end, we implemented the D²FA algorithm and ran it on the considered rule-sets with multiple diameter bound values. In one experiment, the diameter was left unbounded and the maximum default path length was measured. The results are shown in Table II, where the compression is expressed as the percentage ratio between the number of final and original transitions. The smaller the reported value, the higher the compression achieved. In the third column we show the percentage ratio of distinct transitions, which bounds the amount of compression ideally achievable.

Note that the A-DFA algorithm achieves a degree of compression notably higher than the counterpart D²FA with diameter bound equal to two, which has the same worst-case bound on bandwidth. Moreover, the compression is comparable to that of D²FA 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 D²FA. Finally, it's worth observing how the achieved compression is close to the ideal value reported in column four.

Next, we evaluate the behavior of the A-DFA algorithm for increasing values of parameter *k*, leading to memory bandwidth requirements below two accesses per state traversal. Specifically, in Table III we show the compression reported when *k* varies from 1 to 5, resulting in an average number of state traversals per input character of 2, 1.5, 1.33, 1.25, and 1.2. We can observe that, although the compression decreases when *k* increases, the size of the A-DFA remains below 10% that of the original DFA for *k* less than or equal to 3. Additionally, the compression achieved for *k* equal to 5 (leading to 1.2 state traversals per input character) is comparable to that of D²FA with diameter bound equal to 2. These two results hold across all datasets but *cisco11* and *snort24*.

Two factors, as shown in Table I, make *cisco11* and *snort24* exhibit a worse behavior. First, both rule-sets contain a high number of wildcard repetitions. The resulting DFAs tend to consist of independent sub-DFAs, with a consequent decrease in the degree of connection of the space reduction graph. Second, the patterns in these sets are short (this is especially true for *cisco11*). Therefore, the depth of most states is contained and skipping depth levels when creating default transitions has a more obvious impact. In general, increasing the *k* parameter won't have a detrimental effect on compression for long patterns and highly connected space reduction graphs.

Table III. Compression Achieved Using Different Settings for Parameter k

Data set	D ² FA	A-DFA				
	$DB = 2$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
<i>bro217</i>	21.7	0.7	2.4	8.3	15.4	22.9
<i>cisco11</i>	13.3	3.4	18.1	30.9	43.3	53.4
<i>cisco43</i>	13.0	1.0	3.0	7.3	11.9	17.2
<i>cisco613</i>	19.5	0.9	1.8	5.5	11.6	19.6
<i>snort25</i>	20.8	1.5	2.1	2.7	3.4	4.2
<i>snort50</i>	23.9	1.9	5.2	9.0	13.3	17.3
<i>snort24</i>	16.8	3.5	13.8	24.4	33.4	42.4

Again, the percent ratio between the number of final and original transitions is reported.

Table IV. Effect of Alphabet Reduction

Data set	DFA characteristics			D ² FA, $DB = 2$			D ² FA, $DB = \infty$			A-DFA, $k = 1$		
	# of states	$ \Sigma $	dist. trans.	Compress.		transitions	Compress.		transitions	Compress.		transitions
				BAR	AAR		BAR	AAR		BAR	AAR	
<i>bro217</i>	6,533	111	6,931	21.7	9.6	160,832	0.59	0.6	9,526	0.7	0.7	10,957
<i>cisco11</i>	24,007	38	42,389	13.4	2.3	139,629	2.62	0.8	46,686	3.4	0.9	55,985
<i>cisco43</i>	19,134	65	33,122	13.0	3.7	180,831	0.92	0.7	33,796	1.0	0.7	35,474
<i>cisco613</i>	11,312	115	12,336	19.5	9.0	260,757	0.75	0.7	19,330	0.9	0.8	23,155
<i>snort25</i>	38,499	49	45,126	20.8	4.4	434,331	1.46	0.9	84,359	1.5	0.9	86,013
<i>snort50</i>	15,176	63	26,076	23.9	6.1	238,390	1.43	0.9	33,669	1.8	1.0	39,815
<i>snort24</i>	10,819	54	23,505	16.8	3.7	103,463	2.95	0.9	24,641	3.5	1.0	27,765

The compression before (BAR) and after (AAR) alphabet reduction and the final number of labeled transitions are reported. A-DFA (with parameter k set to 1) is compared to D²FA with diameter bound equal to 2 and without bound.

7.2. Alphabet Reduction

We want to evaluate the combined effect of default transition compression and alphabet reduction. In Table IV we report the data for D²FA (diameter bound equal to two and no bound set) and for A-DFA with parameter k set to one. The value in the fourth column represents the number of distinct transitions after alphabet reduction has been performed. For each compression scheme, we show the level of compression before and after alphabet reduction. Additionally, we show the number of labeled transitions when both default transition compression and alphabet reduction have been applied.

The following observations can be made. First, alphabet reduction implies further compression on all datasets and over all algorithms and their parameterizations. Second, when combining alphabet reduction with A-DFA, at least 99% of the transitions are removed on all datasets. Each A-DFA is left with an average of two or three labeled transitions per state. Third, the compression of A-DFA gets closer to that of the unbounded D²FA and remains significantly better than D²FA with diameter bound equal to two.

Note that alphabet reduction is particularly effective on *cisco11* and *snort24* datasets, on which default transition-based schemes achieve a smaller level of compression. In fact, the presence of wildcards limits the number of regular expressions compiled into the corresponding DFAs. The limited number of patterns and their short length lead to small alphabets.

7.3. Memory Footprint

The theory behind the default compression algorithms in consideration, namely D²FA and A-DFA, provides an upper bound to the number of state traversals necessary to

Fig. 7. Memory footprint for A-DFA with parameter k set to 1 and using different memory encoding schemes. The last set of data – *average* – represents the average memory footprint across the different datasets in consideration.

process an input text of a given length. As pointed out in Section 5, the cost of each state traversal, that is, the number of memory accesses it implies and the width of such memory accesses, depends on the encoding scheme utilized. In this section we evaluate the use of linear encoding and indirect addressing to represent compressed DFAs. We omit bitmapped addressing, since, as shown in Becchi et al. [2008], the overhead due to bitmaps generally leads to higher memory space requirements and is not justified by a reasonable gain in memory bandwidth requirements.

As mentioned in Section 5, when considering linear addressing a threshold t_Σ on the number of outgoing transitions per state must be defined. States with more than t_Σ transitions will be represented naively using as many words as the size of the reduced alphabet. In different experiments we set t_Σ so to allow compressed states to fit 16B, 32B, and 64B blocks, 32B and 64B being standard access units for DRAM as well as standard cache-line sizes. For indirect addressing, we consider 32-bit and 64-bit state identifiers. In this case, states with more than 3 (32-bit) and 7 (64-bit) outgoing transitions are only alphabet reduced.

The memory footprints using A-DFA with parameter k set to 1 and the considered encoding schemes are reported in Figure 7. The rightmost dataset shows the average values across the analyzed rule-sets. We observe the following. First, except for *snort50*, indirect addressing is more effective when short (32-bit) identifiers are used. Second, linear addressing is in general preferable in that it allows encoding more states in a compressed fashion. With the exception of *snort50*, a 16B block is enough to guarantee good results, whereas doubling the block size decreases the space requirement only slightly while doubling the memory bandwidth requirements.

We have observed similar trends for different values of parameter k and when using the D²FA scheme. We now focus on linear encoding with a 32B block, since it exhibits good results across all datasets (including *snort50*), and has therefore the best average behavior. In particular, in Figure 8 and Figure 9 we show the normalized memory footprint as a function of the average memory bandwidth. Again, we considered A-DFA with k set to 5, 4, 3, 2, and 1 and D²FA with diameter bound of 2, 6, 10, and 14 (leading to 1.2, 1.25, 1.33, 1.5, 2, 2, 4, 6, and 8 state traversals per input character, respectively). For readability, the datasets have been split into two graphs.

We can observe that, when compared on the same memory bandwidth, A-DFA performs much better than D²FA. Second, the memory footprint of A-DFA requiring 2 memory accesses per character is almost the same as D²FA with higher memory

Fig. 8. Normalized memory footprint of A-DFAs and D2FAs characterized by different memory bandwidth requirements (Snort datasets from Table I). The normalized memory footprint represents the percent ratio between the compressed memory storage requirement and those of a naïve uncompressed representation using 256 pointers per state. Note that the x-axis is in $\log_2$ scale.

Fig. 9. Normalized memory footprint of A-DFAs and D2FAs characterized by different memory bandwidth requirements (Bro and Cisco datasets from Table I). The normalized memory footprint represents the percent ratio between the compressed memory storage requirement and those of a naïve uncompressed representation using 256 pointers per state. Note that the x-axis is in $\log_2$ scale.

bandwidth requirements. Third, even when only an average of 1.2 memory accesses per character are required, A-DFA always takes less than 13% the space of a naïve representation (and in some instances less than 5% of it). Therefore, given a storage budget, tuning the parameter k can be an effective way to optimize the bandwidth requirements.

7.4. Multistride DFAs

We want to analyze the combination of alphabet reduction, A-DFA, and stride doubling. First, we study the effect of increasing the dataset size. To this end, our first set of experiments is conducted by drawing an increasing number of patterns from Snort. All considered regular expressions contain character ranges and eight of them present dot-star subpatterns (however, six of these share a common prefix up to the dot-star term). The characteristics of the k -DFAs obtained with this dataset as well as

Table V. Alphabet Size, Number of States, and Average Number of Transitions per State on k -DFAs of Different Size

# RegEx	DFA			2-DFA			4-DFA		
	$ \Sigma $	# states	tx/state	$ \Sigma $	# states	tx/state	$ \Sigma $	# state s	tx/state
1	25	48	1.8	53	50	3	93	54	4.4
4	34	253	1.6	145	268	3	395	311	7.9
7	39	351	1.5	213	387	3	669	519	9.1
10	41	638	1.5	267	707	3.3	1,088	977	13.2
13	42	762	1.5	325	879	3.5	1,649	1,334	17.8
16	44	1,184	1.5	412	1,401	3.9	2,317	2,210	27.3
19	44	1,396	1.4	441	1,661	3.6	2,389	2,656	24.1
22	44	1,940	1.4	470	2,351	5.3	2,962	3,876	33.1

The number of considered regular expressions is progressively increased from 1 to 22.

Fig. 10. Distribution of the number of outgoing (labeled) transitions per state for the DFA corresponding to the last dataset in Table V. Strides 1, 2, and 4 are considered.

the corresponding alphabet size are detailed in Table V. As can be seen, strides of one, two, and four have been tested. The following observations can be made.

First, the need for handling acceptance conditions makes the number of states in the k -DFAs increase with the stride. However, as mentioned earlier, the additional states account only for a default transition. Second, the alphabet size grows faster on multistride automata than on basic automata. However, doubling the stride does not cause the alphabet size to increase quadratically, even with respect to the compressed base alphabet. Interestingly enough, for stride two, the average number of outgoing transitions per state stays below 4 up to 1.5K states, and is limited to 5.3 for the 22-regular expression dataset. The same number gets to 33.1 in case of stride four. Note that a naïve representation would imply 65,000 (for stride two) or 4.29 billion (for stride four) outgoing transitions per state.

To complete our analysis, in Figure 10 we illustrate the distribution of outgoing transitions per state for the 22-regular expression set considered before. Almost 90% of the states in the basic DFA have at most one and 9.5% of them have between two and seven transitions. For stride two, about 80% of the states have one or fewer outgoing transitions; between 16% and 17% of the states present more than one but fewer than 16 outgoing transitions, and a handful of states have a number of transitions comparable to the alphabet size. The distribution is similar in the case of stride four, with the exception that the percentage of zero-transition states in the 4-DFA increases sensibly

Table VI. Characteristics of DFA and Stride-2 DFA for Large Datasets

Data-set	DFA				2-DFA			
	states	$ \Sigma $	transitions	trans/state	states	$ \Sigma $	transitions	trans/state
<i>bro217</i>	6,533	111	10,957	1.68	30,311	3,576	263,979	8.71
<i>cisco11</i>	24,007	38	55,191	2.30	41,757	331	931,574	22.31
<i>cisco43</i>	19,134	65	35,447	1.85	32,465	878	243,994	7.52
<i>cisco613</i>	11,312	113	23,155	2.05	64,427	6,800	330,235	5.13
<i>snort25</i>	38,499	49	86,013	2.23	40,460	476	294,612	7.28
<i>snort50</i>	15,176	63	39,520	2.60	21,944	931	314,824	14.35
<i>snort24</i>	10,819	54	27,783	2.57	14,723	688	573,801	38.97

and the fraction of states with a large number of outgoing transitions increases slightly, too. These distributions suggest that, with a proper encoding, multistride automata can be effective on medium-size DFAs. Moreover, it may be beneficial to distinguish the 80%–90% of the states having a limited number of outgoing transitions from the other ones, and to adopt two different state representations.

Finally, in Table VI we show the characteristics of the compressed stride-one and stride-two DFAs for the larger pattern sets considered previously. Again, we can observe that the alphabet size does not increase quadratically, and the average number of transitions per state is limited, especially if compared with the uncompressed value of 65,536. In all cases, between 70% and 75% of the states have at most one outgoing transition and 10% have from two to seven. In some cases, however, the size of the stride-two automaton is substantially larger than that of the stride-one DFA. Therefore, stride doubling on large datasets may be practical only if the available memory justifies it.

8. RELATED WORK

The prior work in the area of regular expression matching at line rate can be classified into two implementation categories: memory based and logic based. Within the former, the finite automaton is stored in memory; within the latter, it is stored in (combinatorial and sequential) logic. Because of the need for reprogrammability, logic-based implementations typically target Field Programmable Gate Arrays [Becchi and Crowley 2008b; Clark and Schimmel 2003; Mitra et al. 2007; Sidhu and Prasanna 2001]. Logic-based solutions, which typically rely on Nondeterministic Finite Automata (NFAs), allow peak performances on single packet flows with relatively simple logic, but are not scalable to large numbers of flows; memory-based designs offer scalability in the number of flows at the cost of algorithmic and design complexity.

In this work, we focus on memory-based implementations. In particular, we study memory-efficient representations of DFAs. Beside the D²FA technique, several DFA compression schemes have been proposed in the literature.

Brodie et al. [2006] compress state transition tables using *run length encoding*, another technique that leverages the transition commonality characterizing DFAs from practical rule-sets. The efficacy of this scheme has been verified on small DFAs consisting of 40–100 states and representing a few exact-match patterns or single regular expressions. Our A-DFA scheme has been tested on small DFAs as well as on DFAs consisting of tens of thousands of states. More importantly, in Brodie et al. [2006] a heuristic to lay out states in memory so as to achieve good compression through run-length encoding is proposed. On the other hand, A-DFA uses an algorithm leading to an optimal solution for the given compression problem.

Becchi and Cadambi [2007] introduce *state merging*, a technique that achieves compression by combining states that share common targets, independent of the characters triggering the state transitions. The proposal uses bitmapped encoding together with pointer indirection to lay out the resulting data structure, and can be combined with default transition compression. The state merging algorithm relies on a space reduction graph and has a $O(n^3 \log n)$ time complexity. Since linear addressing performs generally better than bitmapped encoding, it is not clear that the added complexity of state merging is justified with respect to using an alphabet-reduced A-DFA coupled with linear addressing.

Kumar et al. [2006b] propose CD^2FA , a variation of D^2FA exploiting indirect addressing to achieve one memory access per character processed. A comparison between the compression achieved through A-DFA and CD^2FA is presented in Becchi and Crowley [2007a], and points out that CD^2FA shows limitations when the size of the underlying alphabet and the number of *root* states increase. The A-DFA scheme is especially preferable when combined with linear addressing and when caches or wide memory access units are used.

Ficara et al. [2008] propose δFA , a scheme that exploits similarities between adjacent states in order to allow one state traversal per input character. The compression achieved is less than that of A-DFA. Moreover, the lookup operation is more expensive in that it involves processing not only the transition on the current input character, but also the remaining ones. On each state traversal, the difference between the current and the next state must be computed and the result must be stored. Therefore, this scheme is likely to perform worse than both CD^2FA and A-DFA.

Kong et al. [2008] use multiple alphabet translation tables in order to obtain smaller alphabets and achieve higher compression when only alphabet reduction is performed. A heuristic that clusters states using similar alphabets is proposed. This method exhibits the overhead of storing multiple translation tables, which may not be tolerated in settings where fast memory is a scarce resource. Moreover, this technique does not generally lead to better results than A-DFA combined with basic alphabet reduction, which, as shown, leads to an average of two outgoing transitions per states.

DFAs accepting exact-match patterns can be efficiently compressed using Bit-split automata [Tan and Sherwood 2005] and hashing schemes [Kumar et al. 2007; Xu et al. 2011]. However, these techniques have been designed to address exact-match patterns, and not generic regular expressions. In fact, they cannot be directly applied to DFAs containing states with self-loops. The A-DFA scheme aims to address regular expressions, and not only exact-match patterns.

Finally, as mentioned in the Introduction, several proposals [Becchi and Crowley 2007b; Kumar et al. 2007a; Smith et al. 2008; Yu et al. 2006] have focused on alternative automata to avoid DFA state explosion, which occurs in the presence of repetitions of large character sets. Since all these representations use DFAs as building blocks, the A-DFA technique can be applied in combination with any of them.

9. CONCLUSION

We have shown that on practical datasets, the combination of alphabet reduction and A-DFA leaves 80–90% of the states with a single labeled outgoing transition, plus a default transition, and a negligible number of states with more than eight outgoing transitions. Even when small blocks of 16–32 bytes are used, the simple linear encoding scheme is sufficient to allow a single memory access per state traversal. Additionally, the traversal overhead of A-DFA is minimal, namely two state traversals per character processed in the worst case. We believe that the optimality of these results and the simplicity of the related algorithms make (transition-based) DFA

compression a well-covered problem. In our opinion, further study in the area should focus on different directions, such as the state explosion problem or the optimization of handling multiple flows.

REFERENCES

- Aho, A. V. and Corasick, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Comm. ACM* 18, 6, 333–340.
- Becchi, M. and Cadambi, S. 2007. Memory-Efficient regular expression search using state merging. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM)*. 1064–1072.
- Becchi, M. and Crowley, P. 2007a. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.
- Becchi, M. and Crowley, P. 2007b. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.
- Becchi, M. and Crowley, P. 2008a. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.
- Becchi, M. and Crowley, P. 2008b. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.
- Becchi, M., Franklin, M., and Crowley, P. 2008. A workload for evaluating deep packet inspection architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.
- Brodie, B., Taylor, D. E., and Cytron, R. K. 2006. A scalable architecture for high-throughput regular-expression pattern matching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
- Chu, Y. J. and Liu, T. H. 1965. On the shortest arborescence of a directed graph. *Science Sinica*. 14, 1396–1400.
- Cisco Systems. 2008. Silicon packet processor in the CRS-1 router. <http://www.cisco.com/en/US/products/ps5763/index.html>.
- Cisco Systems. 2011. Cisco adaptive security appliance. <http://www.cisco.com>.
- Citrix Systems. 2007. Citrix application firewall. <http://www.citrix.com>.
- Clark, C. R. and Schimmel, D. E. 2003. Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*.
- Edmonds, J. 1967. Optimum branchings. *J. Res. Nat. Bureau Standards* 71B, 233–240.
- Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., and Di Petro, A. 2008. An improved dfa for fast regular expression matching. *ACM SIGCOMM Comput. Comm. Rev.* 38, 5.
- Hopcroft, J. 1971. An nlogn algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computation*, J. Kohavi Ed., Academic Press, New York, 189–196.
- Kong, S., Smith, R., and Estan, C. 2008. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*.
- Kruskal, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.* 7, 1, 48–50.
- Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., and Turner, J. 2006a. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*.
- Kumar, S., Turner, J., and Williams, J. 2006b. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*.
- Kumar, S., Chandrasekaran, B., Turner, J., and Vargese, G. 2007a. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*. 155–164.
- Kumar, S., Turner, J., Crowley, P., and Mitzenmacher, M. 2007b. HEXA: Compact data structures for faster packet processing. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*.
- Mitra, A., Walid, N., and Bhuyan, L. 2007. Compiling pcre to fpga for accelerating snort nids. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*.

- Paxson, V. 1999. Bro: A system for detecting network intruders in real-time. *Comput. Netw.* 31, 23–24, 2435–2463.
- Roesch, M. 1999. SNORT: Lightweight intrusion detection for networks. In *Proceedings of the 13th System Administration Conference*.
- Sen, S., Spatscheck, O., and Wang, D. 2004. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*.
- Sidhu, R. and Prasanna, V. K. 2001. Fast regular expression matching using fpgas. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- Smith, R., Estan, C., Jha, S., and Kong, S. 2008. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*. 207–218.
- Sommer, R. and Paxson, V. 2003. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications and Security (CCS)*.
- Tan, L. and Sherwood, T. 2005. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
- Tarjan, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA.
- Varghese, G. 2004. *Network Algorithms: An Interdisciplinary Approach to Designing Fast Networked Devices* 1st Ed. Morgan Kaufmann, San Fransisco, CA.
- Xu, Y., Ma, L., Liu, Z., and Chao, J. 2011. A multi-dimensional progressive perfect hashing for high-speed string matching. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.
- Yu, F., Chen, Z., Diao, Y., Lakshman, T. V., and Katz, R. H. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

Received March 2012; accepted November 2012