Public Review for

An Improved DFA for Fast Regular Expression Matching

Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci,
Gianni Antichi, and Andrea Di Pietro

Traffic classification and application identification are important functions in network administration. Traditionally, this was always done by using the port number in the transport header, but with the emerging of new applications using non standard port numbers, there is a need for another more general method. And even for legacy applications, there is a need to control the traffic to check whether it contains any harmful code as a virus or a worm. The most common approach to counter these two problems is to rely on deep packet inspection. Each undesirable (or even desirable) event is characterized by some series of characters, then the payload of packets is browsed for these specific strings seen as fingerprints of events. This method is largely used by today intrusion detection systems. The main problem with it is in the cost of browsing the memory to interpret the content of the payload and to check whether it contains some specific fingerprints or not. This is made more complex by the large number of fingerprints that exist.

To reduce the memory usage and memory access rate, it is often proposed to structure the memory in a tree form where each character in the payload triggers a transition on the tree. The technique doing that, called DFA, is known to improve memory storage and memory speed. The authors in this paper pushes this memory structuring issue to another limit by adding more levels of aggregation of states and transitions so that the memory is not accessed for each character. A local fast memory is used as a cache to store neighboring states. This new idea has been welcomed by all reviewers and recognized to be original and important in this context. The validation part has been also appreciated even though some factors were not considered as the CPU cost and the parallel memory access that one can use. It is clear that the idea will profit from a validation in a more general setting. For now, the paper has the merit of presenting a novel idea in this area with a good coverage of the state of the art.

Public review written by
Chadi Barakat
INRIA Sophia-Antipolis, France

An Improved DFA for Fast Regular Expression Matching

Domenico Ficara
domenico.ficara@iet.unipi.it

Stefano Giordano
s.giordano@iet.unipi.it

Gregorio Procissi
g.procissi@iet.unipi.it

Fabio Vitucci
fabio.vitucci@iet.unipi.it

Gianni Antichi
gianni.antichi@iet.unipi.it

Andrea Di Pietro
andrea.dipietro@iet.unipi.it

Department of Information Engineering, University of Pisa
via G.Caruso 16, Pisa, ITALY

ABSTRACT

Modern network devices need to perform deep packet inspection at high speed for security and application-specific services. Finite Automata (FAs) are used to implement regular expressions matching, but they require a large amount of memory. Many recent works have proposed improvements to address this issue.

This paper presents a new representation for deterministic finite automata (orthogonal to previous solutions), called Delta Finite Automata (δFA), which considerably reduces states and transitions and requires a transition per character only, thus allowing fast matching. Moreover, a new state encoding scheme is proposed and the comprehensive algorithm is tested for use in the packet classification area.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Security

Keywords

DFA, Intrusion Prevention, Deep Packet Inspection, Regular Expressions, Packet Classification

1. INTRODUCTION

Many important services in current networks are based on payload inspection, in addition to headers processing. Intrusion Detection/Prevention Systems as well as traffic monitoring and layer-7 filtering require an accurate analysis of packet content in search of matching with a predefined data set of patterns. Such patterns characterize specific classes of applications, viruses or protocol definitions, and are continuously updated. Traditionally, the data sets were constituted of a number of signatures to be searched with string matching algorithms, but nowadays regular expressions are used, due to their increased expressiveness and ability to describe a wide variety of payload signatures [22]. They are adopted by well known tools, such as Snort [23] and Bro [3], and in firewalls and devices by different vendors such as Cisco[26].

Typically, finite automata are employed to implement regular expression matching. Nondeterministic FAs (NFAs) are representations which require more state transitions per character, thus having a time complexity for lookup of $O(m)$, where m is the number of states in the NFA; on the other hand, they are very space-efficient structures. Instead, Deterministic FAs (DFAs) require only one state traversal per character, but for the current regular expression sets they need an excessive amount of memory. For these reasons, such solutions do not seem to be proper for implementation in real deep packet inspection devices, which require to perform on line packet processing at high speeds. Therefore, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [14, 13, 5, 21].

This paper focuses in memory savings for DFAs, by introducing a novel compact representation scheme (named δFA) which is based on the observation that, since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only. The δ in δFA just emphasizes that it focuses on the differences between adjacent states. Reducing the redundancy of transitions appears to be very appealing, since the recent general trend in the proposals for compact and fast DFAs construction (see sec.2) suggests that the information should be moved towards edges rather than states. Our idea comes from D^{2}FA [14], which introduces default transitions (and a "path delay") for this purpose.

Unlike the other proposed algorithms, this scheme examines one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process. Moreover, it is ortoghonal to several previous algorithms (even the most recent XFAs [21, 20] and H-cFA [13]), thus allowing for higher compression rates. Finally, a new encoding scheme for states is proposed (which we will refer to as *Char-State compression*), which exploits the association of many states with a few input characters. Such a compression scheme can be efficiently integrated into the δFA algorithm, allowing a further memory reduction with a negligible increase in the state lookup time.

We also test the integration of δFA and *Char-State compression* in the packet classification area, by representing the classification rule set through regular expressions.

In summary, the main contributions of this paper are:

- a novel compact representation of DFA states (δFA) which allows for iterative reduction of the number of states and for faster string matching;
- a new state encoding scheme (*Char-State compression*) based on input characters;
- the application of both schemes to classification algorithms to increase search speed.

The remainder of the paper is organized as follows. In section 2 related works about pattern matching and DFAs are discussed. Sec.3 describes our algorithm, by starting from a motivating example and sec.4 proves the integration of our scheme with the previous ones. Then in sec.5 the encoding scheme for states is illustrated and in the subsequent section the integration with δFA is shown. Finally, sec.8 presents the experimental results, while sec.9 proves the applicability of δFA to packet classification.

2. RELATED WORK

Deep packet inspection consists of processing the entire packet payload and identifying a set of predefined patterns. Many algorithms of standard pattern matching have been proposed [1, 9, 27], and also several improvements to them. In [24] the authors apply two techniques to Aho-Corasick algorithm to reduce its memory consumption. In details, by borrowing an idea from Eatherton's Tree Bitmap [10], they use a bitmap to compress the space near the root of the state machine, where the nodes are very dense, while path compressed nodes and failure pointers are exploited for the remaining space, where the nodes become long sequential strings with only one next state each.

Nowadays, state-of-the-art systems replace string sets with regular expressions, due to their superior expressive power and flexibility, as first shown in [22]. Typically, regular expressions are searched through DFAs, which have appealing features, such as one transition for each character, which means a fixed number of memory accesses. However, it has been proved that DFAs corresponding to a large set of regular expressions can blow up in space, and many recent works have been presented with the aim of reducing their memory footprint. In [28] the authors develop a grouping scheme that can strategically compile a set of regular expressions into several DFAs evaluated by different engines, resulting in a space decrease, while the required memory bandwidth linearly increases with the number of active engines.

In [14], Kumar et al. introduce the Delayed Input DFA (D^2FA), a new representation which reduces space requirements, by retrieving an idea illustrated in [2]. Since many states have similar sets of outgoing transitions, redundant transitions can be replaced with a single default one, this way obtaining a reduction of more than 95%. The drawback of this approach is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions.

To address this issue, Becchi and Crowley [6] introduce an improved yet simplified algorithm (we will call it BEC-CRO) which results in at most $2N$ state traversals when processing a string of length N. This work is based on the observation that all regular expression evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or its near neighbors. From this consideration and by leveraging, during automaton construction, the concept of state distance from the starting state, the algorithm achieves comparable levels of compression with respect to D^2FA, with lower provable bounds on memory bandwidth and greater simplicity.

Also, the work presented in [4] focuses on the memory problem of DFAs, by proposing a technique that allows non-equivalent states to be merged, thanks to a scheme where the transitions in the DFA are labeled. In particular, the authors merge states with common destinations regardless of the characters which lead those transitions (unlike D^2FA), creating opportunities for more merging and thus achieving higher memory reduction. Moreover the authors regain the idea of bitmaps for compression purposes.

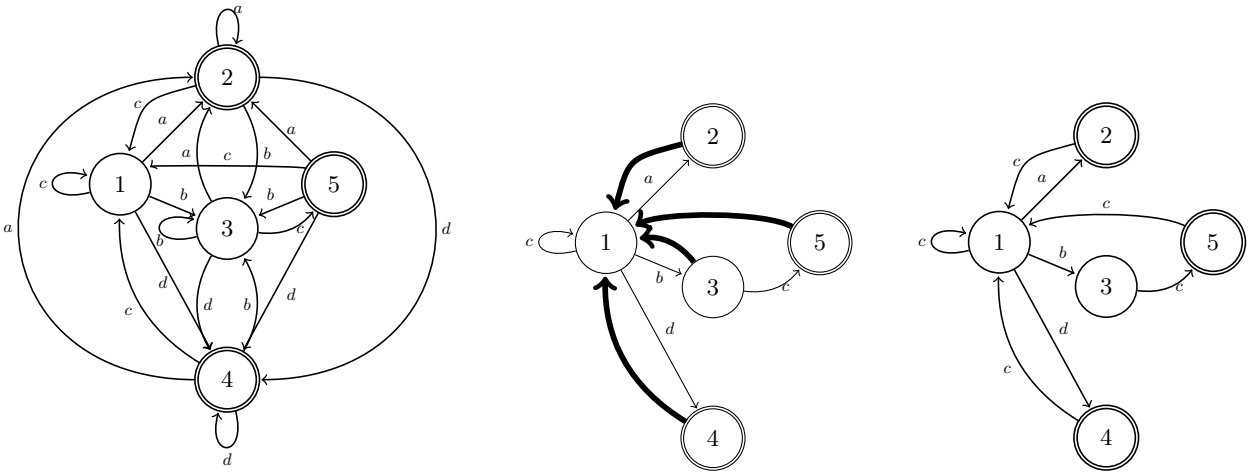
Run-Length-Encoding is used in [7] to compress the transition table of DFAs. The authors show how to increase the characters processed per state traversal and present heuristics to reduce the number of memory accesses. Their work is specifically focused on an FPGA implementation.

The work in [5] is based on the usual observation that DFAs are infeasible with large sets of regular expressions (especially for those which present wildcards) and that, as an alternative, NFAs alleviate the memory storage problem but lead to a potentially large memory bandwidth requirement. The reason is that multiple NFA states can be active in parallel and each input character can trigger multiple transitions. Therefore the authors propose a hybrid DFA-NFA solution bringing together the strengths of both automata: when constructing the automaton, any nodes that would contribute to state explosion retain an NFA encoding, while the others are transformed into DFA nodes. As shown by the experimental evaluation, the data structure presents a size nearly that of an NFA, but with the predictable and small memory bandwidth requirements of a DFA.

Kumar et al. [15] also showed how to increase the speed of D^2FAs by storing more information on the edges. This appears to be a general trend in the literature even if it has been proposed in different ways: in [15] transitions carry data on the next reachable nodes, in [4] edges have different labels, and even in [13] and [21, 20] transitions are no more simple pointers but a sort of "instructions".

In a further comprehensive work [13], Kumar et al. analyze three main limitations of the traditional DFAs. First, DFAs do not take advantage of the fact that normal data streams rarely match more than a few initial symbols of any signature; the authors propose to split signatures such that only one portion needs to remain active, while the remaining portions can be "put to sleep" (in an external memory) under normal conditions. Second, the DFAs are extremely inefficient in following multiple partially matching signatures and this yields the so-called *state blow-up*: a new improved Finite State Machine is proposed by the authors in order to solve this problem. The idea is to construct a machine which remembers more information, such as encountering a closure, by storing them in a small and fast cache which represents a sort of history buffer. This class of machines is called History-based Finite Automaton (H-FA) and shows a space reduction close to 95%. Third, DFAs are incapable of keeping track of the occurrencies of certain sub-expressions, thus resulting in a blow-up in the number of state: the authors introduce some extensions to address this issue in the History-based counting Finite Automata (H-cFA).

The idea of adding some information to keep the transition history and, consequently, reduced the number of states, has been retrieved also in [21, 20], where another scheme, named extended FA (XFA), is proposed. In more details, XFA augments traditional finite automata with a finite scratch memory used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters and other instructions attached to edges and states). The experimental tests performed with a large class of NIDS signatures showed time complexity similar to DFAs and space complexity similar to or better than NFAs.



(a) The DFA (b) The D^{2}FA (c) The δFA

Figure 1: Automata recognizing (a^+),(b^+c) and (c^*d^+).

3. DELTA FINITE AUTOMATON: δFA

As above discussed, several works in the recent years have focused on memory reduction of DFAs to improve the speed of regular expression engines. Most of these methods trade size for number of memory accesses: at the cost of a few further memory references per input character, the DFAs can become significantly smaller and take advantage of the speed of smaller fast memories. The most important and cited example of such a technique is D^{2}FA [14], where an input character (hereafter simply "input char") can require a (configurable) number of additional steps through the automaton before reaching the right state.

3.1 A motivating example

In this section we introduce δFA, a D^{2}FA-inspired automaton that preserves the advantages of D^{2}FA and requires a single memory access per input char. To clarify the rationale behind δFA and the differences with D^{2}FA, we analyze the same example brought by Kumar et al. in [14]: fig.1(a) represents a DFA on the alphabet $\{a, b, c, d\}$ that recognizes the regular expressions (a^+),(b^+c) and (c^*d^+).

Figure 1(b) shows the D^{2}FA for the same regular expressions set. The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and a default transition to be taken for all input char for which a transition is not defined. When, for example, in fig.1(b) the D^{2}FA is in state 3 and the input is d, the default transition to state 1 is taken. State 1 "knows" which state to go to upon input d, therefore we jump to state 4. In this example, taking a default transition costs 1 more hop (1 more memory access) for a single input char. However, it may happen that also after taking a default transition, the destination state for the input char is not specified and another default transition must be taken, and so on. The works in [14] and [6] show how we can limit the number of hops in default paths and propose refined algorithms to define the best choice for default paths. In the example, the total number of transitions was reduced to 9 in the D^{2}FA (less than half of the equivalent DFA which has 20 edges), thus achieving a remarkable compression.

However, observing the graph in fig.1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state; in particular, adjacent states share the majority of the next-hop states associated with the same input chars. Then if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (because for each character they lead to the same set of states as 1). This means that state 2 can be described with a very small amount of bits. Instead, if we jump from state 1 to 3, and the next input char is c, the transition will not be the same as the one that c produces starting from 1; then state 3 will have to specify its transition for c.

The result of what we have just described is depicted in fig.1(c) (except for the local transition set), which is the δFA equivalent to the DFA in fig.1(a). We have 8 edges in the graph (as opposed to the 20 of a full DFA) and every input char requires *a single state traversal* (unlike D^{2}FA).

3.2 The main idea of δFA

As shown in the previous section, the target of δFA is to obtain a similar compression as D^{2}FA without giving up the *single state traversal per character* of DFA. The idea of δFA comes from the following observations:

- as shown in [6], most default transitions are directed to states closer to the initial state;
- a state is defined by its transition set and by a small value that represents the accepted rule (if it is an accepting state);
- in a DFA, most transitions for a given input char are directed to the same state.

By elaborating on the last observation, it becomes evident that most adjacent states share a large part of the same transitions. Therefore we can store only the differences between adjacent (or, better, "parent-child"[1]) states.

[1]here the terms *parent* and *child* refer to the depth of adjacent states

This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA shown in fig.1(c) only stores the transitions that *must* be defined for each state in the original DFA.

3.3 Construction

In alg.1 the pseudo-code for creating a δFA from a N-states DFA (for a character set of C elements) is shown. The algorithm works with the *transition table* $t[s, c]$ of the input DFA (i.e.: a $N \times C$ matrix that has a row per state and where the i-th item in a given row stores the state number to reach upon the reading of input char i). The final result is a "compressible" transition table $t_c[s, c]$ that stores, for each state, the transitions required by the δFA only. All the other cells of the $t_c[s, c]$ matrix are filled with the special LOCAL_TX symbol and can be simply eliminated by using a bitmap, as suggested in [24] and [4]. The details of our suggested implementation can be found in section 7.

Algorithm 1 Pseudo-code for the creation of the transition table t_c of a δFA from the transition table t of a DFA.

```
 1: for c ← 1, C do
 2:     t_c[1, c] ← t[1, c]
 3: end for
 4: for s ← 2, N do
 5:     for c ← 1, C do
 6:         t_c[s, c] ← EMPTY
 7:     end for
 8: end for
 9: for s_parent ← 1, N do
10:     for c ← 1, C do
11:         s_child ← t[s_parent, c]
12:         for y ← 1, C do
13:             if t[s_parent, y] ≠ t[s_child, y] then
14:                 t_c[s_child, y] ← t[s_child, y])
15:             else
16:                 if t_c[s_child, y] == EMPTY then
17:                     t_c[s_child, y] ← LOCAL_TX
18:                 end if
19:             end if
20:         end for
21:     end for
22: end for
```

The construction requires a step for each transition (C) of each pair of adjacent states ($N \times C$) in the input DFA, thus it costs $O(N \times C^2)$ in terms of time complexity. The space complexity is $O(N \times C)$ because the structure upon which the algorithm works is another $N \times C$ matrix. In details, the construction algorithms first initializes the t_c matrix with EMPTY symbols and then copies the first (root) state of the original DFA in the t_c. It acts as base for subsequently storing the differences between consecutive states.

Then, the algorithm observes the states in the original DFA one at a time. It refers to the observed state as *parent*. Then it checks the *child* states (i.e.: the states reached in 1 transition from parent state). If, for an input char c, the child state stores a different transition than the one associated with any of its parent nodes, we cannot exploit the knowledge we have from the previous state and this transition must be stored in the t_c table. On the other hand, when all of the states that lead to the child state for a given character share the same transition, then we can omit to store that transition. In alg.1 this is done by using the special symbol LOCAL_TX.

3.3.1 Equivalent states

After the construction procedure shown in alg.1, since the number of transitions per state is significantly reduced, it may happen that some of the states have the same identical transition set. If we find j identical states, we can simply store one of them, delete the other $j - 1$ and substitute all the references to those with the single state we left. Notice that this operation creates again the opportunity for a new state-number reduction, because the substitution of state references makes it more probable for two or more states to share the same transition set. Hence we iterate the process until the number of duplicate states found is 0.

3.4 Lookup

Algorithm 2 Pseudo-code for the lookup in a δFA. The current state is s and the input char is c.

```
procedure Lookup(s, c)
1: read(s)
2: for i ← 1, C do
3:     if t_c[s, i] ≠ LOCAL_TX then
4:         t_loc[i] ← t_c[s, i]
5:     end if
6: end for
7: s_next ← t_loc[c]
8: return s_next
```

The lookup in a δFA is computed as shown in alg.2. First, the current state must be read with its whole transition set (step 1). Then it is used to update the local transition set t_{loc}: for each transition defined in the set read from the state, we update the corresponding entry in the local storage. Finally the next state s_{next} is computed by simply observing the proper entry in the local storage t_{loc}. While the need to read the whole transition set may imply more than 1 memory access, we show in sec.6 how to solve this issue by means of a compression technique we propose. The lookup algorithm requires a maximum of C elementary operations (such as shifts and logic AND or popcounts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases, the computational delay is negligible with respect to the memory access latency.

In fig.3 we show the transitions taken by the δFA in fig.1(c) on the input string *abc*: a circle represents a state and its internals include a bitmap (as in [24] to indicate which transitions are specified) and the transition set. The bitmap and the transition set have been defined during construction. It is worth noticing that the "duplicate" definition of transitions for character *c*. We have to specify the *c*-transition for state 2 even if it is the same as the one defined in state 1, because state 2 can be reached also from state 3 which has a different next state for *c*. We start ($t = 0$) in state 1 that has a fully-specified transition set. This is copied into the local transition set (below). Then we read the input char *a*

and move ($t = 1$) to state 2 that specifies a single transition toward state 1 on input char c. This is also an accepting state (underlined in figure). Then we read b and move to state 3. Note that the transition to be taken now is not specified within state 2 but it is in our local transition set. Again state 3 has a single transition specified, that this time changes the corresponding one in the local transition set. As we read c we move to state 5 which is again accepting.

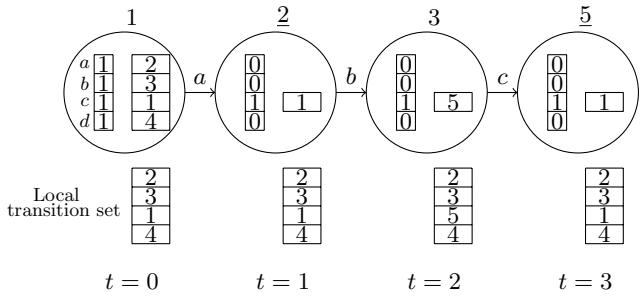


Figure 3: δFA internals: a lookup example.

4. APPLICATION TO H-CFA AND XFA

One of the main advantage of our δFA is that it is orthogonal to many other schemes. Indeed, very recently, two major DFA compressed techniques have been proposed, namely H-cFA [13] and XFA [21, 20]. Both these schemes address, in a very similar way, the issue of state blow-up in DFA for multiple regular expressions, thus candidating to be adopted in platforms which provide a limited amount of memory, as network processors, FPGAs or ASICs. The idea behind XFAs and H-cFA is to trace the traversal of some certain states that corresponds to closures by means of a small scratch-memory. Normally those states would lead to state blow-up; in XFAs and H-cFA flags and counters are shown to significantly reduce the number of states.

Since our main concern is to show the wide extent of the possible applications for our δFA, we report in fig.2(a) a simple example (again taken from a previous paper [13]). In the example, the aim is to recognize the regular expressions *.*ab[^a]*c* and *.*def*, and labels include also conditions and operations that operate on a flag (set/reset with +/-1) and a counter n (for more details refer to [13]). A DFA would need 20 states and a total of 120 transitions, the corresponding H-cFA (fig.2(a)) uses 6 states and 38 transitions, while the δFA representation of the H-cFA (fig.2(b)) requires only 18 transitions. Specifically, the application of δFA to H-cFA and XFA (which is tested in sec.8) is obtained by storing the "instructions" specified in the edge labels only once per state. Moreover edges are considered different also when their specified "instructions" are different.

5. COMPRESSING CHAR-STATE PAIRS

In a δFA, the size of each state is not fixed because an arbitrary number of transitions can be present, and therefore state pointers are required, which generally are standard memory addresses. They constitute a significant part of the memory occupation associated with the DFA data structure, so we propose here a compression technique which remarkably reduces the number of bits required for each pointer. Such an algorithm is fully compatible with δFA and most of the other solutions for DFA compression already shown in section 2. Our algorithm (hereafter referred to as *char-state compression* or simply *C-S*) is based on a heuristic which is verified by several standard rule sets: in most cases, the edges reaching a given state are labelled with the same character. Table 1 shows, for different available data sets (see section 8 for more details on sets) the percentage of nodes which are reached only by transitions corresponding to a single character over the total number of nodes.

Data set	p_{1char} (%)	r_{comp} (%)	η_{acc}	T_S (KB)
Snort34	96	59	1.52	27
Cisco30	89	67	1.62	7
Cisco50	83	61	1.52	13
Cisco100	78	59	1.58	36
Bro217	96	80	1.13	11

Table 1: Percentage of states reached by edges with the same one label (p_{1char}), *C-S* compression (r_{comp}), average number of scratchpad accesses per lookup (η_{acc}) and indirection-table size (T_S).

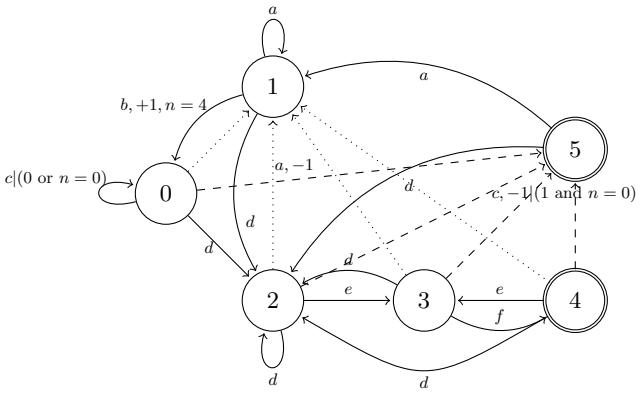
As a consequence, a consistent number of states in the DFA can be associated with a single character and can be referred to by using a "relative" address. More precisely, all the states reached by a transition labelled with character c will be given a "relative" identifier (hereafter simply *relative-id*); since the number of such states will be smaller than the number of total states, a relative-id will require a lower number of bits than an absolute address. In addition, as the next state is selected on the basis of the next input char, only its relative-id has to be included in the state transition set, thus requiring less memory space.

In a D^{2}FA, where a default transition accounts for several characters, we can simply store it as a relative-id with respect to the first character associated with it. The absolute address of the next state will be retrieved by using a small indirection table, which, as far as our experimental results show, will be small enough to be kept in local (or in a scratchpad) memory, thus allowing for fast lookup. It is clear that such a table will suffer from a certain degree of redundancy: some states will be associated with several relative-ids and their absolute address will be reported more than once. In the next subsection we then propose a method to cope with such a redundancy, in the case it leads to an excessive memory occupation.

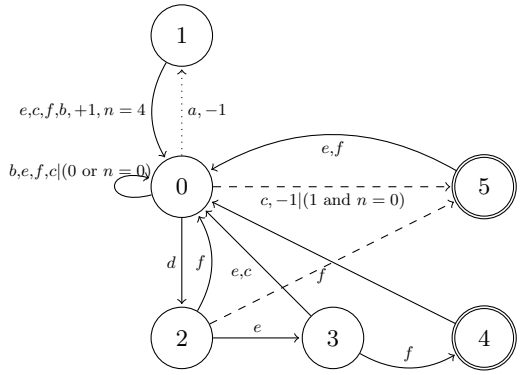
Figure 4 shows the distribution of the number of bits that may be used for a relative-id when applying our compression scheme to standard rule sets. As it can be noticed, next state pointers are represented in most cases with very few bits (less than five); even in the worst case, the number of bits is always below ten. In the second column of table 1, we show the compression rate achieved by *C-S* with respect to a naive implementation of DFA for the available data sets. As it appears from the table, the average compression is between 60% and 80%.

5.1 Indirection Table Compression

As claimed above, the implementation of *Char-State compression* requires a lookup in an indirection table which should be small enough to be kept in local memory. If several



(a) The H-cFA. Dashed and dotted edges have same labels, respectively $c, -1|(1 \text{ and } n = 0)$ and $a, -1$. Not all edges are shown to keep the figure readable. The real number of transitions is 38.



(b) The δH-cFA. Here all the 18 transitions are shown.

Figure 2: Automata recognizing *.*ab[^a]*c* and *.*def*

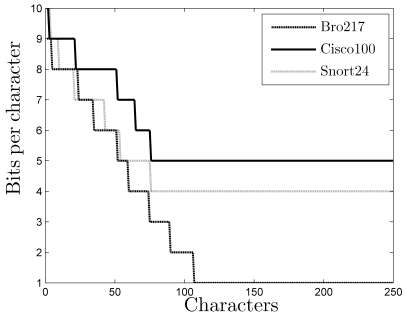


Figure 4: Distribution of the number of bits used for a relative identifier with our compression scheme for standard rule sets.

states with multiple relative-ids are present in such a table, this might be an issue. For this reason we present a lookup scheme which offers an adaptive trade-off between the average number of memory accesses and the overall memory occupation of the table.

The table that we use in our scheme encompasses two kinds of pointers: absolute pointers and local ones. When a state has a unique relative-id, its absolute address is written in the table; otherwise, if it has multiple relative-ids, for each one of them the table reports a pointer to a list of absolute addresses; such a pointer will require a consistently smaller number of bytes than the address itself. An absolute address is then never repeated in the table, thus preventing from excessive memory occupation. Such a scheme is somewhat self-adapting since, if few states have multiple identifiers, most of the translations will require only one memory access, while, if a consistent amount of redundancy is present, the translation will likely require a double indirection, but the memory occupation will be consistently reduced. Notice that the presence of different length elements in the table poses no severe issues: since the relative address is arbitrary, it is sufficient to assign lower addresses to nodes which are accessible with only one lookup and higher addresses to nodes requiring double indirection, and to keep a threshold value in the local memory. The results in terms of memory accesses and size of such a scheme applied to the available data sets are reported in fig.1.

6. C-S IN δFA

The *C-S* can be easily integrated within the δFA scheme and both algorithms can be cross-optimized. Indeed, *C-S* helps δFA by reducing the state size thus allowing the read of a whole transition set in a single memory access on average. On the other hand, *C-S* can take advantage of the same heuristic of δFA: successive states often present the same set of transitions. As a consequence, it is possible to parallelize the retrieval of the data structure corresponding to the next state and the translation of the relative address of the corresponding next-state in a sort of "speculative" approach. More precisely, let s and $s+1$ be two consecutive states and let us define A_s^c as the relative address of the next hop of the transition departing from state s and associated with the character c. According to the previously mentioned heuristic it is likely that $A_s^c = A_{s+1}^c$; since, according to our experimental data (see sec.8), 90% of the transitions do not change between two consecutive states, we can consider such an assumption to be verified with a probability of roughly 0.9. As a consequence, when character c is processed, it is possible to parallelize two memory accesses:

- retrieve the data structure corresponding to state $s+1$;
- retrieve the absolute address corresponding to A_{s+1}^c in the local indirection table.

In order to roughly evaluate the efficiency of our implementation in terms of the state lookup time, we refer to a common underlying hardware architecture (described in section 7). It is pretty common [25] that the access to a local memory block to be than twice as faster than that of to an off-chip memory bank: as a consequence, even if a double indirection is required, the address translation will be ready when the data associated with the next state will be available. If, as it is likely, $A_s^c = A_{s+1}^c$, it will be possible to directly access the next state (say $s+2$) through the absolute pointer that

has just been retrieved. Otherwise, a further lookup to the local indirection table will be necessary.

Such a parallelization can remarkably reduce the mean time needed to examine a new character. As an approximate estimation of the performance improvement, let us suppose that our assumption (i.e. $A_s^c = A_{s+1}^c$) is verified with probability $p = 0.9$, that one access to on-chip memory takes $t_{on} = 4T$ and to an external memory $t_{off} = 10T$ [25], and that an address translations requires $n_{trans} = 1.5$ memory accesses (which is reasonable according to the fourth column of table 1). The mean delay will be then:

$$\overline{t_{par}} = (1-p)(t_{off} + n_{trans} \times t_{on}) + p \times t_{off} = 10.6T$$

This means that even with respect to the implementation of δFA the *C-S* scheme increases the lookup time by a limited 6%. On the contrary, the execution of the two tasks serially would required:

$$\overline{t_{ser}} = (t_{off} + n_{trans} \times t_{on}) = 16T$$

The parallelization of tasks results then in a rough 50% speed up gain.

7. IMPLEMENTATION

The implementation of δFA and *C-S* should be adapted to the particular architecture of the hardware platform. However, some general guidelines for an optimal deployment can be outlined. In the following we will make some general assumptions on the system architecture; such assumptions are satisfied by many network processing devices (e.g. the Intel IXP Network Processors [12]). In particular, we assume our system to be composed by:

- a standard 32 bit processor provided with a fairly small local memory (let us suppose a few KBs); we consider the access time to such a memory block to be of the same order of the execution time of an assembly level instruction (less than ten clock cycles);
- an on-chip fast access memory block (which we will refer to as scratchpad) with higher storage capacity (in the order of 100 KB) and with an access time of a few dozens of clock cycles;
- an off-chip large memory bank (which we will refer to as external memory) with a storage capacity of dozens of MBs and with an access time in the order of hundreds of clock cycles.

We consider both δFA and *Char-State compression* algorithms. As for the former, two main kinds of data structures are needed: a unique local transition set and a set of data structures representing each state (kept in the external memory). The local transition set is an array of 256 pointers (one per character) which refer to the external memory location of the data structure associated with the next state for that input char; since, as reported in table 3.b , the memory occupation of a δFA is generally smaller than 1 MB, it is possible to use a 20 bit-long offset with respect to a given memory address instead of an actual pointer, thus achieving a consistent compression.

A δFA state is, on the contrary, stored as a variable-length structure. In its most general form, it is composed by a 256 bit-long bitmap (specifying which valid transition are already stored in the local transition set and which ones are instead stored within the state) and a list of the pointers for the specified transitions, which, again, can be considered as 20 bit offset values.

If the number of specified transitions within a state is small enough, the use of a fixed size bitmap is not optimal: in these cases, it is possible to use a more compact structure, composed by a plain list of character-pointer couples. Note that this solution allows for memory saving when less than 32 transitions have to be updated in the local table.

Since in a state data structure a pointer is associated with a unique character, in order to integrate *Char-State compression* in this scheme it is sufficient to substitute each absolute pointer with a relative-id. The only additional structure consists of a character-length correspondence list, where the length of the relative-ids associated with each character is stored; such an information is necessary to parse the pointer lists in the node and in the local transition set. However, since the maximum length for the identifiers is generally lower than 16 bits (as it is evident from figure 4), 4 bits for each character are sufficient. The memory footprint of the character-length table is well compensated by the corresponding compression of the local transition set, composed by short relative identifiers (our experimental results show a compression of more than 50%). Furthermore, if a double indirection scheme for the translation of relative-ids is adopted, a table indicating the number of unique identifiers for each character (the threshold value we mentioned in section 5.1) will be necessary, in order to parse the indirection table. This last table (that will be at most as big as the compressed local transition table) can be kept in local memory, thus not affecting the performance of the algorithm.

8. EXPERIMENTAL RESULTS

This section shows a performance comparison among our algorithm and the original DFA, D^{2}FA and BEC-CRO. The experimental evaluation has been performed on some data sets of the Snort and Bro intrusion detection systems and Cisco security appliances [26]. In details, such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to obtain a reasonable amount of memory for DFAs and to observe different statistical properties. Such characteristics are summarized in table 2, where we list, for each data set, the number of rules, the ascii length range and the percentage of rules including "wildcards symbols" (i.e. *, +, ?). Moreover, the table shows the number of states and transitions and the amount of memory for a standard DFA which recognizes such data sets, as well as the percentage of duplicated states. The choice of such data sets aims to mimic the size (in terms of DFA states and regular expressions) of other sets used in literature [4, 14, 5, 6] in order to obtain fair comparisons.

Tables 3 illustrate the memory compression achieved by the different algorithms. We have implemented the code for our algorithm, while the code for D^{2}FA and BEC-CRO is the *regex-tool* [18] from Michela Becchi (for the D^{2}FA the code runs with different values of the diameter bound, namely the diameter of the equivalent maximum weight spanning tree found in the space reduction graph [14]; this parameter affects the structure size and the average number of state-traversals per character). By means of these tools, we build a standard DFA and then reduce states and transitions through the different algorithms. The compression in tab.3.a is simply expressed as the ratio between the num-

Dataset	# of regex	ASCII length range	% Regex w/ wildcards (*,+,?)	Original DFA	
				# of states	# of transitions
Snort24	24	6-70	83.33	13886	3554816
Cisco30	30	4-37	10	1574	402944
Cisco50	50	2-60	10	2828	723968
Cisco100	100	2-60	7	11040	2826240
Bro217	217	5-76	3.08	6533	1672448

Table 2: Characteristics of the rule sets used for evaluation.

(a) Transitions reduction (%). For δFA also the percentage of duplicate states is reported.

Dataset	D^{2}FA					BEC-CRO	δFA	
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		trans.	dup. states
Snort24	98.92	98.92	98.91	98.48	89.59	98.71	96.33	0
Cisco30	98.84	98.84	98.83	97.81	79.35	98.79	90.84	7.12
Cisco50	98.76	98.76	98.76	97.39	76.26	98.67	84.11	1.1
Cisco100	99.11	99.11	98.93	97.67	74.65	98.96	85.66	11.75
Bro217	99.41	99.40	99.07	97.90	76.49	99.33	93.82	11.99

(b) Memory compression (%).

Dataset	D^{2}FA					BEC-CRO	δFA + C-S
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		
Snort24	95.97	95.97	95.94	94.70	67.17	95.36	95.02
Cisco30	97.20	97.20	97.18	95.21	55.50	97.11	91.07
Cisco50	97.18	97.18	97.18	94.23	51.06	97.01	87.23
Cisco100	97.93	97.93	97.63	95.46	51.38	97.58	89.05
Bro217	98.37	98.34	95.88	95.69	53	98.23	92.79

Table 3: Compression of the different algorithms in terms of transitions and memory.

ber of deleted transitions and the original ones (previously reported in tab.2, while in tab.3.b it is expressed considering the overall memory saving, therefore taking into account the different state sizes and the additional structures as well. Note also, in the last column of tab.3.a, the limited but effective state-reduction due to the increased similarity of states obtained by the δFA (as described in sec.3.3.1). Although the main purpose of our work is to reduce the time complexity of regular expression matching, our algorithm achieves also a degree of compression comparable to that of D^{2}FA and BEC-CRO, as shown by tab.3. Moreover, we remark that our solution is orthogonal to these algorithms (see sec.4), thus allowing further reduction by combining them.

Figure 5 shows the average number of memory accesses (η_{acc}) required to perform pattern matching through the compared algorithms. It is worth noticing that, while the integration of *C-S* into δFA (as described in sec.6) reduces the average state size, thus allowing for reading a whole state in slightly more than 1(< 1.05) memory accesses, the other algorithms require more accesses, thus increasing the lookup time. We point out that the mean number of accesses for the integration of δFA and *C-S* is not included in the graph in that *C-S* requires accesses to a local scratchpad memory, while the accesses the figure refers to are generally directed to an external, slower memory block; therefore it is difficult to quantify the additional delay introduced by *C-S*. However, as already underlined in section 6, if an appropriate parallelization scheme is adopted, the mean delay contribution of *C-S* can be considered nearly negligible on most architectures.

Finally, table 4 reports the results we obtained by applying δFA and *C-S* to one of the most promising approach for regular expression matching: XFAs [21, 20] (thus obtaining a δXFA). The data set (courtesy of Randy Smith) is composed of single regular expressions with a number of closures that would lead to a state blow-up. The XFA representation limits the number of states (as shown in the table). By adopting δFA and *C-S* we can also reduce the number of transitions with respect to XFAs and hence achieve a further size reduction. In details, the reduction achieved is more than 90% (except for a single case) in terms of number of transitions, that corresponds to a rough 90% memory compression (last column in the table). The memory requirements, both for XFAs and δXFAs, are obtained by storing the "instructions" specified in the edge labels only once per state.

Dataset	# of states	# of trans. XFA	# of trans. δXFA	Compr. %
c2663-2	14	3584	318	92
s2442-6	12	3061	345	74.5
s820-10	23	5888	344	94.88
s9620-1	19	4869	366	92.70

Table 4: Number of transitions and memory compression by applying δFA+*C-S* to XFA.

Figure 6 resumes all the evaluations by mixing speed performance (in terms of memory accesses) and space require-

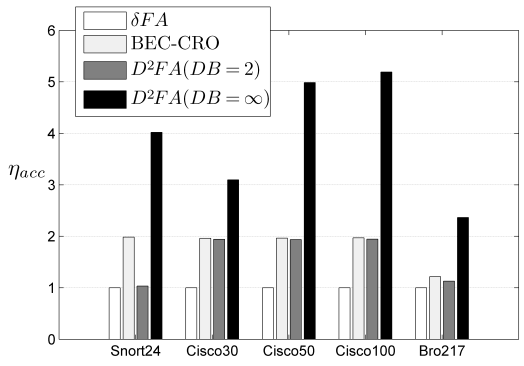


Figure 5: Mean number of memory accesses for δFA, BEC-CRO and D^{2}FA for different datasets.

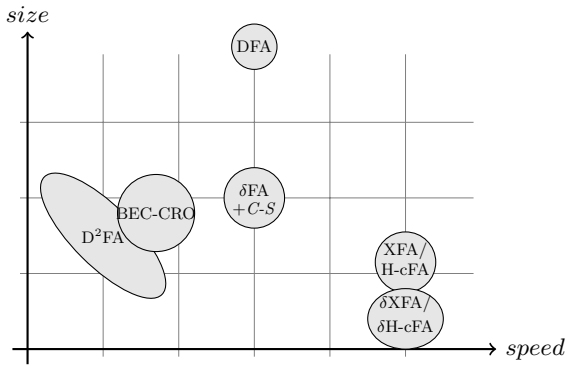


Figure 6: Comparison of speed performance and space requirements for the different algorithms.

ments in a qualitative graph (proportions are not to be considered real). It is evident that our solution almost achieves the compression of D^{2}FA and BEC-CRO, while it proves higher speed (as that of DFA). Moreover, by combining our scheme with other ones, a general performance increase is obtained, as shown by the integration with XFA or H-cFA.

9. PACKET CLASSIFICATION WITH δFA

In the last ten years, a large number of papers on packet classification have been published. However, such a topic still represents an open research field: its importance keeps growing, both at the edge and at the core of network, because it is necessary to provide QoS and security at high speeds. Moreover, all existing techniques and algorithms do not represent a final solution that can be applied to all scenarios: in most cases they trade memory occupation for search speed by yielding good results in either one or another context.

Our proposal is to use δFA to represent classification rule sets. While such rules generally specify single bits of the header, we need byte-wise rules to define regular expression; therefore rules are "exploded" to 8-bits boundaries. Each rule (defined by the usual 5-tuple $SrcIP$, $DestIP$, $SrcPort$, $DestPort$, and $L4-Protocol$) has then to be translated into a regular expression and an equivalent DFA has to be built. Each byte belonging to header fields composing the 5-tuples is taken in the DFA as a character.

We compare our solution with the most recent and high performance packet classification algorithms, by using the code provided by Haoyu Song at [17].

The Parallel Bit Vectors (BV) [16] is a decomposition-based algorithm, targeted to hardware implementation. It first performs the parallel lookups on each individual field (returning a bit vector where a bit representing a filter is set if the filter is matched on this field) and then shares these information (with a bitwise AND operation) to find the final matching. Instead, HiCuts [11] is a decision tree-based algorithm, based on a geometric view of the packet classification problem; the construction algorithm recursively cuts the space into smaller non overlapped sub-regions, one dimension per step, where a low cost linear search allows for finding the best matched filter. HyperCuts [19] is another decision tree structure, very similar to HiCuts; it introduces several additional refinements and especially allows cutting on multiple dimensions per step rather than only one. For HiCuts and HyperCuts we assume two values of the space factor (2 and 4), a parameter which regulates the trade-off between memory consumption and number of accesses.

The synthetic filter sets used for the decision structures construction and the packet traces used for the lookup evaluation are both generated by ClassBench [8]. The filters are real filter sets provided by Internet Service Providers and network equipment vendors. In particular, three filters of 100 rules have been chosen for the performance evaluation: an access control list, a NAT list for a firewall and a decision tree format list for IP Chain.

In this phase, the actual target of our algorithm is a fast lookup, and the results in tab.5 confirm our ideas: δFA requires for the lookup the least number of memory accesses in the worst case for all filter sets, for instance with a reduction up to 60% with respect to BV. Also, our algorithm shows good performance even in terms of mean number of accesses. We point out that, since in real implementation the cost of memory accesses overcomes the cost of actual processing, the speed of any operations depends almost exclusively on the number of memory accesses.

Unfortunately, at this stage very preliminary results also prove a very remarkable memory consumption for δFA if compared to the standard classification algorithms. This is mainly due to the wildcards in the rule sets, which lead to state blow-up in the finite automaton. Some recent improvements ([5], [13] and [21, 20]) proved to solve this issue, obtaining good results, and might be used to decrease memory consumption of δFA and to make our algorithm available for packet classification in real network devices.

Algorithm	ACL1_100		FW1_100		IPC1_100	
	Mem. ref.		Mem. ref.		Mem. ref.	
	mean	max	mean	max	mean	max
HyperCuts–2	12.97	23	12.65	29	6.96	16
Hypercuts–4	10.2	17	11.5	26	5.76	13
HiCuts–2	6.17	17	15.02	28	6.1	17
Hicuts–4	6.47	16	11.5	26	4.85	18
BV	24	31	22	28	26	33
δFA	9.78	13	8.69	13	12.2	13

Table 5: Comparison between packet classification algorithms in terms of memory accesses.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new compressed representation for deterministic finite automata, called Delta Finite Automata. The algorithm considerably reduces the number of states and transitions and it is based on the observation that most adjacent states share several common transitions, so it is convenient to store only the differences between them. Furthermore, it is orthogonal to previous solutions, this way allowing for higher compression rates. Another fundamental feature of the δFA is that it requires only a state transition per character (keeping the characteristic of standard DFAs), thus allowing a fast string matching.

A new encoding scheme for states has been also proposed (that we refer to as Char State), which exploits the association of many states with a few input chars. Such a compression scheme can be efficiently integrated into the δFA algorithm, allowing a further memory reduction with a negligible increase in the state lookup time.

Finally, the integration of both schemes has also been proposed for application in the field of packet classification, by representing the classification rule set through regular expressions. The experimental runs have shown good results in terms of lookup speed as well as the issue of excessive memory consumption, which we plan to address by exploiting the recent techniques presented in [13] and [21, 20].

Acknowledgements

We would like to thank Michela Becchi for her extensive support and for her useful *regex-tool*. We are grateful to Sailesh Kumar, William Eatherton and John Williams (Cisco) for having provided the regular expression set used in Cisco devices. We thank Randy Smith for his precious suggestion and the XFAs used in our tests. Finally we would like to express gratitude to Haoyu Song for his freely available packet classifier implementations that helped us in our work and to the anonymous reviewers for their insightful comments.

11. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison Wesley, 1985.

[3] *Bro: A system for Detecting Network Intruders in Real Time, http://bro-ids.org/.*

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

[5] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. of CoNEXT '07*, pages 1–12. ACM, 2007.

[6] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. of ANCS '07*, pages 145–154, 2007.

[7] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *Proc. of ISCA'06*, June 2006.

[8] *Classbench, A Packet Classification Benchmark, http://www.arl.wustl.edu/~det3/ClassBench/.*

[9] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of ICALP '79*, pages 118–132. Springer-Verlag.

[10] W. Eatherton, Z. Dittia, and G. Varghese. Tree bitmap: Hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communications Review*, 34, 2004.

[11] P. Gupta and N. McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.

[12] *Intel Network Processors, www.intel.com/design/network/products/npfamily/.*

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

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

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

[16] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214, 1998.

[17] *Haoyu Song, Evaluation of Packet Classification Algorithms, www.arl.wustl.edu/~hs1/PClassEval.html.*

[18] *Michela Becchi, regex tool, http://regex.wustl.edu/.*

[19] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. Technical report, 2003.

[20] R. Smith, C. Estan, and S. Jha. Xfa: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.

[21] R. Smith, C. Estan, and S. Jha. Xfas: Fast and compact signature matching. Technical report, University of Wisconsin, Madison, August 2007.

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

[23] *Snort: Lightweight Intrusion Detection for Networks, http://www.snort.org/.*

[24] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. of INFOCOM 2004*, pages 333–340.

[25] G. Varghese. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[26] J. W. Will Eatherton. *An encoded version of reg-ex database from cisco systems provided for research purposes.*

[27] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Dept.of Computer Science, University of Arizona.

[28] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. of ANCS '06*, pages 93–102.