*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2007 proceedings.*

# Memory-Efficient Regular Expression Search Using State Merging

Michela Becchi
Department of Computer Science and Engineering
Washington University, St Louis, MO
mbecchi@cse.wustl.edu

Srihari Cadambi
NEC Laboratories America
Princeton NJ
cadambi@nec-labs.com

*Abstract*— **Pattern matching is a crucial task in several critical network services such as intrusion detection and policy management. As the complexity of rule-sets increases, traditional string matching engines are being replaced by more sophisticated regular expression engines. To keep up with line rates, deal with denial of service attacks and provide predictable resource provisioning, the design of such engines must allow examining payload traffic at several gigabits per second and provide worst case speed guarantees. While regular expression matching using deterministic finite automata (DFA) is a well studied problem in theory, its implementation either in software or specialized hardware is complicated by prohibitive memory requirements. This is especially true for DFAs representing complex regular expressions present in practical rule-sets.**

**In this paper, we introduce a novel method to drastically reduce the DFA memory requirement and still provide worst-case speed guarantees. Specifically, we merge several "non-equivalent" states in a DFA by introducing labels on their input and output transitions. We then propose a data structure to represent the merged states and the transition labels. We show that, with very few assumptions about the original DFA, such a transformation results in significant compression in the DFA representation. We have implemented a state merging and transition labeling algorithm for DFAs, and show that for Snort and Bro security rule-sets, state merging results in memory reductions of an order of magnitude.**

## I. Introduction

In addition to examining structured information present in the header to classify a packet, many critical network services such as intrusion detection (IDS), policy management and identification of P2P traffic, require inspection of packet payloads. Also known as deep packet inspection, this provides better capability to classify packets based upon applications, content and state. Until recently, rule-sets for intrusion detection and other services primarily consisted of strings. However, current rule-sets like Snort [1], Bro [2], [3] and many others are replacing strings with the more powerful and expressive regular expressions.

The classical method to perform regular expression search is to use a deterministic finite automaton (DFA) [4], the focus of this paper. The main problem with DFAs is prohibitive memory usage. The number of states in a DFA scale poorly with the size and number of wildcards in the regular expressions they represent. As the number of wildcards in a regular expression grows, the number of DFA states increases sharply, exponentially in some cases. The presence of wildcards, one of the primary reasons why regular expressions are so expressive, also complicates merging multiple regular expressions. Two regular expressions with a moderate number of DFA states when considered individually may combine to form a composite DFA with a much larger state count. Since rule-sets typically consist of many regular expressions, it is beneficial to create a combined DFA since checking individual DFAs one-by-one imposes sequentiality in the processing, and decreases speed. This memory complexity makes software regular expression search engines extremely slow and not scalable to large rule-sets. It also makes hardware architectures difficult to design and implement.

Compounding this issue is the fact that critical network services such as intrusion detection must be performed online at high speeds. For a variety of reasons including router design, denial-of-service attacks and resource provisioning, routers must provide a worst-case speed guarantee. In the case of a DFA, this speed guarantee translates to an upper bound on the number of states visited for every input character in the payload traffic. Classical DFAs visit exactly one state per input character. However, due to memory limitations, many DFA generators such as Flex [5] build DFAs with fewer states, and rollback and revisit characters in the input multiple times. Such a strategy is unacceptable for critical, online network services.

In this paper, we address the memory problem for regular expression search, specifically for real rule-sets implemented using DFAs. We argue that by drastically reducing the memory requirement for DFAs, they become faster, more scalable and easier to implement in a software engine or as specialized hardware architectures. We propose a novel technique that allows non-equivalent states in a DFA to be merged using a scheme where the transitions in the DFA are labeled. By carefully labeling transitions, in effect, we are transferring information from the nodes to the edges of the graph representing the DFA. We propose a novel data structure to represent a DFA with merged states and labeled transitions, and show that this lossless compression method can achieve significant memory reductions in practice.

Unlike other DFA compaction approaches, we have no requirement on the transitions on which the two states reach their common destinations. A recent DFA compaction approach [6] (that does not do state merging, but instead removes transitions to common destinations) requires two states to not only have the same destinations, but also transition to those destinations using the same input characters. Another significant advantage of our scheme is that merging states creates more common destinations for other states. As an example, if states A and B transition to states C and D, they cannot be merged. However, if C and D were merged, then A and B have a common destination and could be merged. Thus, merging itself creates more opportunities for memory compaction.

In summary, the major contribution of our paper is the notion of merging distinct, non-equivalent states in a DFA using transition labeling. To this end, we make the following specific contributions:

- We describe a compact data structure that can represent a DFA with merged states and transition labels.
- We present a merging and labeling algorithm.
- We extend the bitmap data structure proposed for string matching [7] to DFAs, and introduce a modification using pointer indirection, which also reduces memory usage in its own right.
- We present an analysis of our scheme, and perform a systematic experimental study comparing state merging to previous compaction techniques.

The remainder of our paper is organized as follows. In Section II, we discuss related work. In Section III, we introduce the bitmap-based data structure for DFAs, and present a discussion of our proposed improvements. We also present a motivational example that is used throughout the paper. In Section IV, we motivate our proposed state merging scheme using the example. In Section V, we formally present state merging, and discuss and analyze our merging and labeling algorithm and the data structure for a merged state. In Section VI, we present experimental results, and conclude in Section VII.

## II. Related Work

Until recently, most intrusion detection and other rule-sets consisted of strings, not regular expressions. Classical software-based string matching algorithms include KMP, Boyer-Moore [8], Wu-Manber [9] and Aho-Corasick [10]. Among these, Aho-Corasick has the ability to handle multiple patterns and guarantees $O(n)$ search time for an input consisting of $n$ characters. The memory requirement of the Aho-Corasick algorithm is generally linear in the total size of the strings in the rule-set, but a real implementation for large rule-sets may require considerable storage. In [11], the authors proposed splitting the state machine so that the string matching problem is solved by converting the large database into many smaller state machines, each of which searches for a portion of the rules, and a portion of the bits of each rule.

In [7], the authors propose two improvements to the Aho-Corasick algorithm. The first is the use of a bitmap data structure for each state. The second is to employ path compression where a "chain" of successive states all with a single outgoing transition are merged into a single state. As we will see, this is quite different from our state merging. Instead of "path-compressing" contiguous single-output states, we merge *any* two (or more) states in the DFA. If the states to be merged have common destinations, memory usage is reduced.

The authors in [12] propose using a TCAM to accelerate pattern matching. They consider a subset of all possible regular expressions. Their approach suffers from the drawbacks of TCAMs such as poor scalability and high power consumption, and may be unsuitable for large and complex rule-sets. [13] performed an analysis of regular expressions commonly used in networking, and proposed rewriting some rules to mitigate the memory blow-up. The rewriting is performed in a "safe" manner, which guarantees left-first matches. [13] also analyzes the complexity of DFAs for real regular expressions used in network-based IDS.

Current algorithms to improve regular expression search trade-off memory for speed. Perhaps the closest work to our own in the Delayed DFA (D2FA) presented in [6]. Unlike our approach, D2FA does not merge states or label transitions. Rather it identifies two (or more) states that transition to the same set of destinations on the same input characters. For example, if both states $S0$ and $S1$ transition to state $S2$ on character `a` and to state $S3$ on character `b`, then the `a` and `b` transitions of state $S1$ are removed and replaced by a single "default" transition to state $S0$. Upon reaching $S1$, if the input is $a$ or $b$, we take the default transition to $S0$ and then transition to the appropriate destination state. Thus, D2FA achieves memory compaction by removing duplicated transitions, but this happens at the expense of latency; states with a default transition require more than one transition per input character. There are two major differences between our scheme and D2FA. First, D2FA requires target states to have the same destinations *as well as* the same character to transition to those destinations. We do not have this restriction, and can merge states with common destinations, regardless of the characters on which they transition to those destinations. In other words, the states that D2FA targets are a subset of the states that we can merge. Second, in our case, merging states creates opportunities for more merging. By contrast, D2FA is a static technique.

In [14], the authors propose increasing the speed of regular expression search by expanding the alphabet. Specifically, they process two characters (bytes) for every state transition in the DFA. This produces an exponential increase in memory usage (since the cardinality of the alphabet is now squared). However, they propose several heuristics to mitigate the memory blow-up. This effort is targeted towards a hardware implementation, while our algorithm is more general.

## III. Bitmap-based Data Structures for DFAs

Using bitmaps is a natural first step towards achieving memory compaction for DFAs. Bitmaps have been used before for packet classification [15], [16] as well as for string matching using Aho-Corasick state machines [7].

In order to illustrate the use of bitmaps, as well as motivate our scheme, we use a simple example shown in Figure 1. The same example is used throughout the paper. It represents the

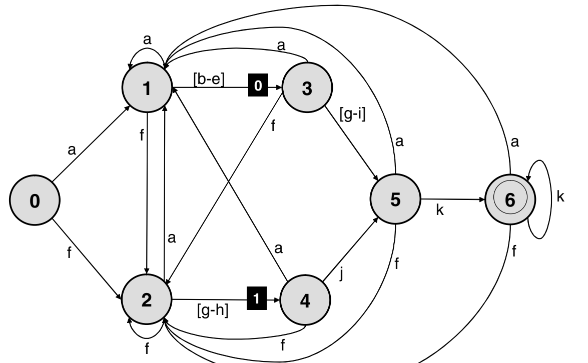


Fig. 1. DFA for regular expression `(a[b-e][g-i]|f[g-h]j)k+`. All transitions not shown lead to state 0. The labels (shown within black squares) appearing on transitions going into states 3 and 4 are discussed in Section IV.

```
struct DFA_state {
    RegExList *accepted_regex;
    DFA_state *next_state[256];
    DFA_state *failure;
}
```

Fig. 2. Basic, naive data structure representing a state in a DFA.

regular expression `(a[b-e][g-i]|f[g-h]j)k+`. Consistent with regular expression syntax, `[b-e]`, `[g-i]` and `[g-h]` indicate ranges of characters, i.e., any character within the range is permissible. This regular expression accepts all patterns that either start with an `a` followed by a single character from the range `[b-e]`, then by a single character in the range `[g-i]`, or with an `f` followed by a `g` or `h` followed by a `j`, and culminating with one or more `k`'s. Transitions leading to the starting state (state 0) are not shown in Figure 1 to avoid clutter.

The basic, naive data structure for representing the DFA is shown in Figure 2. Each state consists of a structure containing next state pointers for every character in the alphabet. Thus, for the ASCII alphabet consisting of 256 characters, this structure contains 256 next state pointers.

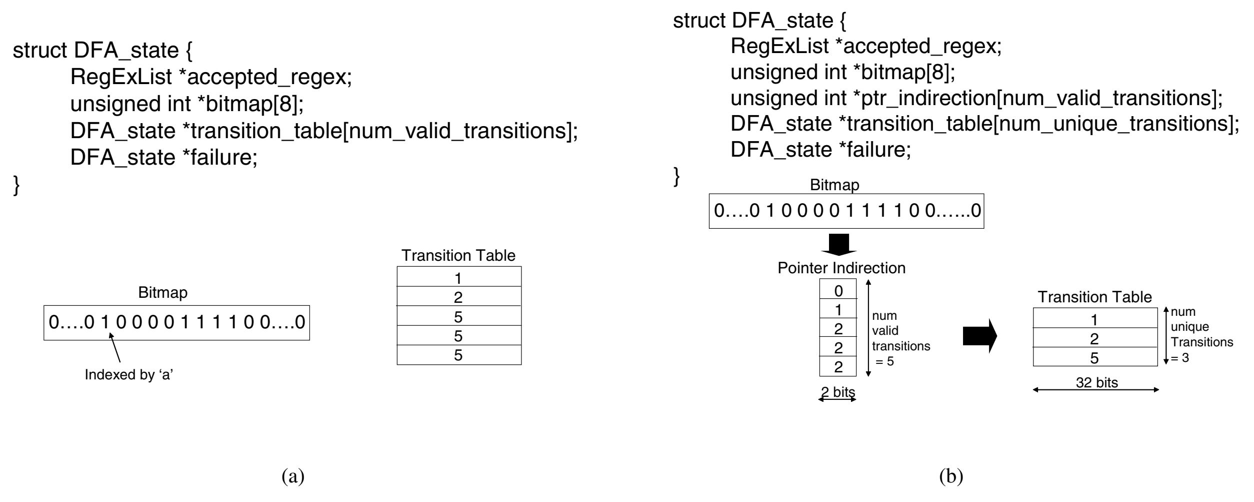
Figure 3(a) demonstrates a method of compacting the basic data structure using bitmaps [7], [15], [16]. In DFAs representing practical rule sets, a state seldom has valid outgoing transitions for all possible characters in the alphabet. Rather, a few characters transition to valid next states, while several transition to a default "failure" state. For many states, the failure state is the start (initial) state. Instead of maintaining explicit next state pointers, the bitmap-based data structure maintains pointers to valid next states in a *transition table*, and uses a bitmap indexed by the input character to generate an address into the transition table. A '1' in the $i$'th position of the bitmap indicates a valid next state transition for input character with ASCII value $i$; the address into the transition table is obtained by counting the number of 1's until bit $i$ in the bitmap. On the other hand, a '0' in the bitmap indicates that the next state pointer is not present in the transition table, and that we should default to the failure state (denoted by a single failure pointer). Bitmap-based data structures obtain considerable compression since the number of valid next states are usually much smaller than the cardinality of the alphabet. Figure 3(a) also shows the bitmap for State 3 from the example in Figure 1. Note that the number of entries in the transition table is equal to the number of valid (non-failure) outgoing pointers from State 1, which in this case is 5. The five 1's in the bitmap correspond to these valid outgoing transitions from State 3 (on characters `a, f, g, h` and `i`).

It may be observed that basic bitmap-based data structures do not take advantage of duplicate entries in the transition table. For example, if a state $S0$ transitions to state $S1$ on input character 'a' as well as input character 'b', the transition table contains $S1$ twice, in successive locations. This is because the bitmap will have 1's in both the 'a' position as well as the 'b' position. Thus, two distinct transition table addresses will be generated for 'a' and 'b', necessitating separate entries. This is also shown in the example in Figure 3(a), where the transition table contains State 5 repeated thrice.

A straightforward way of addressing this is to use a second bit for every location in the bitmap. The new bit indicates if the address into the transition table must be incremented. However, for alphabets with large cardinality requiring large bitmaps, such a strategy results in excessive memory usage.

A better scheme is to use one level of pointer indirection. Figure 3(b) shows this for State 3 from the example of Figure 1. A pointer indirection table is inserted between the bitmap and transition table. Now, the transition table contains only distinct next state entries, in this case, 3 entries. The bitmap generates an address into the pointer indirection table, which in turn contains a pointer into the transition table. In practical DFAs, the number of distinct entries in the transition table is typically smaller than the total number of entries. Thus, the width of the pointer indirection table need only be the logarithm of the number of distinct next states. In the example, each entry in the pointer table needs only 2 bits; thus, 16 such entries may be packed into a 32-bit memory word. Although this is not the primary contribution of our paper, we still present results on the use of pointer indirection with bitmaps. To the best of our knowledge, such a data structure has not been reported and studied in DFAs for network-based intrusion detection rule-sets before.

Bitmap-based data structures have two other disadvantages, especially when implemented in software. First, some computation (1's counting) is necessary in order to process the bitmap and obtain an address into the transition table. Second, fetching the bitmap could require several memory accesses. These issues may be resolved to a certain extent by breaking up a large bitmap into several smaller bitmaps, each annotated with some extra information. For example, a 256-wide bitmap, which requires 8 32-bit memory accesses, could be divided into 8 32-bit bitmaps. Each 32-bit bitmap now needs additional information indicating how may 1's are present in the bitmaps preceding it. Thus, at the expense of some memory, a large



```
struct DFA_state {
        RegExList *accepted_regex;
        unsigned int *bitmap[8];
        DFA_state *transition_table[num_valid_transitions];
        DFA_state *failure;
}
```

```
struct DFA_state {
        RegExList *accepted_regex;
        unsigned int *bitmap[8];
        unsigned int *ptr_indirection[num_valid_transitions];
        DFA_state *transition_table[num_unique_transitions];
        DFA_state *failure;
}
```

Fig. 3. (a) Rudimentary bitmap-based data structure. The lower part shows the bitmap and its transition table for State 3 from the example in Figure 1. (b) More compact bitmap-based data structure using pointer indirection.

fraction of memory accesses could be reduced. We do not detail these techniques in this paper, as they are orthogonal to our main contributions. Similar memory access reduction strategies could be applied to any bitmap-based technique, including the one in this paper.

## IV. State Merging: A Motivational Example

In this section, we introduce state merging with a motivational example. Consider states 3 and 4 in Figure 1. They they have common destinations in states 1, 2 and 5. However, they do not transition to state 5 on the same input character. As we will see, this is not a requirement for state merging.

The first task when we merge two states is to label their input transitions. This is required because merged states have a single data structure. When this data structure is accessed during DFA traversal, we must know how the state was reached, and which portion of the data structure to access. The only constraint for labeling is that all input transitions of a state in the original DFA have the same label. Figure 1 shows an assignment of label 0 to all input transitions of state 3, and label 1 to all input transitions of state 4. This is sufficient to distinguish the two states after they are merged. All other transition labels are unnecessary at this point, and are hence not computed.

Figure 4 shows the DFA obtained after merging states 3 and 4. The merged state is represented as 3_4. Note that the output transitions of 3_4 are represented with trailing labels. For example, the transition `[g-i]/0, j/1` indicates that the same next state, in this case state 5, is reached from state 3_4 upon receiving input characters `g, h, i` with label 0 or input character `j` with label 1. On the other hand, the transitions `a/0,1` and `f/0,1` are taken on characters `a` and `f` irrespective of the label with which state 3_4 was reached.

In Figure 4, we observe that states 1 and 2 now have a common destination state 3_4. This was not the case in the original example of Figure 1. Thus, the process of merging

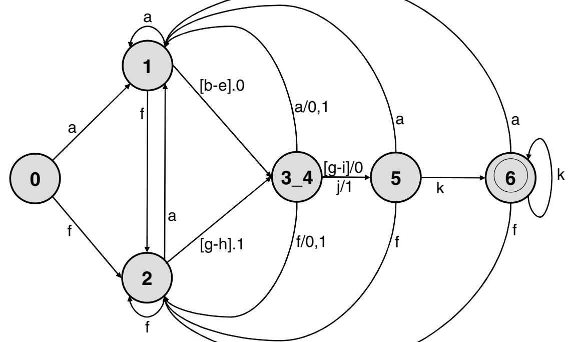


Fig. 4. DFA after merging states 3 and 2 from the example of Figure 1. Each transition arc is denoted by the character on which the transition occurs followed by the transition label.

has created an opportunity for further merging. Again, labels 0 and 1 suffice to ensure that states 1 and 2 can be distinguished after merging, i.e., *these labels can be reused*. If no merged state in the DFA contains more than 2 original states, labels 0 and 1 are sufficient to label the entire DFA. We generalize this property in Section V.

Following merging of states 1 and 2, we obtain the DFA shown in Figure 5. We now have transitions of the type `a.0/0,1`, where the label following the dot pertains to the destination state, and the labels following the “/” pertain to the source state of the transition. We refer to such labels as the destination labels and source labels respectively. The transition `a.0/0,1` from state 3_4 to state 1_2 means that (i) the transition carries with it a label `0` that tells its destination state, 1_2 that the transition is meant for underlying original state 1, and (ii) the transition is taken when its source state 3_4 receives labels `0` or `1`. Similarly, `[b-e].0/0` means that the transition is taken when its source state 1_2 receives inputs characters `b, c, d, e` and was reached with label `0`, and

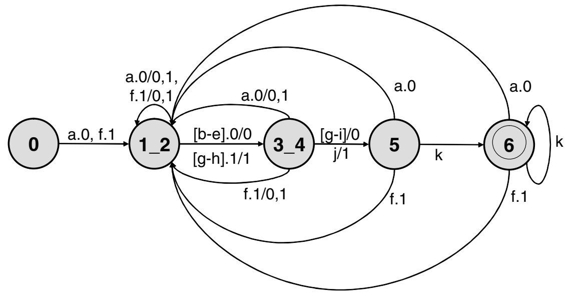


Fig. 5. DFA after merging states 1 and 2 from the example of Figure 4.

it carries with it label 0 to indicate to its destination state 3_4 that the transition was meant for underlying original state 3.

The transition labels thus ensure no loss of information when states are merged. However, we obtain compression by using a modified data structure to represent the merged DFA. The data structure for merged state 1_2 is shown in Figure 6. The two bitmaps are the same as those in the individual data structures for the states prior to merging. The appropriate bitmap is chosen based on the destination label of the input transition. The pointer indirection table indexes the transition table (where next state pointers are stored), and also provides the destination labels for the outgoing transitions.

Two important observations may be made. First, the data structure *does not need to store the source labels*; they are implicit. For example, the transition `[b-e].0/0` is only taken when source state 1_2 receives label 0 : this is ensured by having separate bitmaps (and pointer indirection tables) for each source label. Second, common destinations need only be represented once in the transition table. For merged state 1_2, 1_2 itself and 3_4 are destinations common to several transitions. They are represented only once in the combined transition table to obtain compression.

Note that an alternative implementation could have outgoing labels in the transition table, associated with the next state pointers. However, this necessitates duplicating states in the transition table. For example, merged state 1_2 would have had two separate entries 3_4 with label 0, and 3_4 with label 1, requiring two entries in the wide transition table. With the labels in the indirection table, duplication is less expensive since the table is not as wide. For the rest of this paper, we assume that the labels are in the pointer indirection table.

State merging also requires that we update the data structures of all states affected by the merger. For example, when we created state 1_2, we must label transitions into the data structure of 3_4 since there are some transitions from 3_4 to the newly merged state 1_2.

The simple example illustrates the benefits of state merging. In more complex DFAs corresponding to real networking rule-sets, opportunities for merging are plentiful, as we demonstrate in Section VI. Also, as emphasized earlier, merging states creates opportunities for further merging by creating more shared destinations.

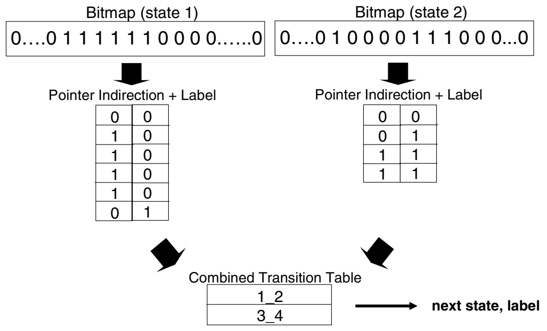


Fig. 6. Merged data structure for state 1_2 of Figure 5.

## V. State Merging in DFAs

Two or more states can be merged into a single state by introducing labels on their transitions. The merged states are represented using a data structure containing the individual bitmaps, a combined transition table consisting of the union of their individual transition tables, the updated pointer indirection table for each original state and a structure to store the labels. In this section, we describe state merging and labeling in more detail, and show that the DFA obtained after state merging is equivalent to the original DFA. We also present an algorithm to perform merging and labeling.

We use the terms "original states" and "merged states" to refer to the states of the DFA prior to merging and after merging respectively. Note that after merging, some original states may be unaffected and remain in the DFA as is.

### A. Labels

For every transition connecting two merged states, we define *source labels* and *destination labels*. A transition, represented by $c.l_d/l_0, l_1..$, thus has three attributes:

- a character $c$ upon which the transition is taken;
- a single destination label $l_d$ that indicates to the destination state which underlying original state this transition is meant for;
- one or more source labels $l_0$, $l_1$... that indicate to the source state upon which label to take this transition.

Each time a transition $c.l_d/l_0, l_1...$ is taken, label $l_d$ is produced and stored. Transition $c.l_d/l_0, l_1...$ will be taken if the current input character is 'c' and the stored label is any of $l_0, l_1...$. If either the source or destination states are not merged, those labels are absent from the transition. Clearly, labels cause an overhead in terms of memory since they need to be stored. Before presenting the merging and labeling algorithm, we show that the number of required *distinct* labels is bounded and small, and therefore their introduction only marginally affects memory usage.

**Lemma 1:** An original state of the DFA has the same destination label on all its input transitions.

This follows from the definition above: a destination label identifies the underlying original state. Every original state is thus assigned a single destination label on its input transitions.

**Lemma 2:** The total number of distinct destination labels required is equal to the maximum number of original states contained in any merged state.

The lemma implies that if there are at most $i$ original states in any merged state in a DFA, the entire DFA needs exactly $i$ destination labels (which are reused across different states throughout the DFA). The proof is straightforward. If we have $i$ states in a merged state, we need $i$ destination labels on the input transitions of the state to distinguish between the $i$ underlying original states. Say we have $i+1$ labels in the DFA, represented by $l_1$, $l_2...l_{i+1}$. Consider now any merged state $S$ that has the destination label $l_{i+1}$ on its input transitions. At least one of the $i+1$ labels is not used by $S$ since $S$ contains at most $i$ original states merged into it. Suppose the unused label is $l_k$. We contend that label $l_{i+1}$ can be reassigned to $l_k$. To see this, suppose $l_{i+1}$ indicates original state $s$ ($s$ is now merged into $S$). We simply re-label all input transitions of $s$ from $l_{i+1}$ to $l_k$. This will not affect input transitions to any other states because every transition connects exactly two states in a DFA, thereby maintaining the property that all original states use exactly one label for their input transitions.

## B. Legality of State Merging

We show that state merging does not affect the language accepted by the DFA. All DFAs have an initial, "start" state, and one or more "accept" states. It is essential to note that state merging does not modify the start or accept states.

**Lemma 3:** Let $D$ be a DFA and $D'$ the DFA obtained after merging one or more state pairs of $D$. A pattern is accepted by $D'$ if and only if the pattern is accepted by $D$.

Let $S$ be the start state, and $A_i$ an accept state of $D$ and $D'$. For a pattern to be accepted by a DFA, there is a path from $S$ to $A_i$. To prove the lemma therefore, we show that there exists a path between $S$ and $A_i$ in $D'$ if and only if the same path exists between $S$ and $A_i$ in $D$.

We prove sufficiency first. Let $p$ be a path from $S$ to $A_i$ in $D$. If none of the states on $p$ are affected by merging, the lemma clearly holds. Let us assume that a state $s$ on $p$ has been merged with other states to create a merged state $s_m$. Let the transition leading into $s$ on path $p$ be taken on character $c_i$, and the transition leading out of $s$ on path $p$ be taken on character $c_o$. For simplicity, we also represent the transitions by $c_i$ and $c_o$. After merging, $c_i$ becomes $c_i.l$ and $c_o$ becomes $c_o/l$. Note that since merging neither creates nor removes transitions, there is a one-to-one mapping between the transitions in $D$ and $D'$. When transition $c_i.l$ is taken in $D'$, label $l$ is produced and stored while traversing the merged state. Now since the next character in the pattern is $c_o$, transition $c_o/l$ is taken as the stored label is $l$. Therefore the same path $p$ will be traversed in $D'$.

To prove necessity, let $p'$ be a path from $S$ to $A_i$ in $D'$. If none of the states on $p'$ are affected by merging, the property clearly holds. Assume a merged state $s_m$ on $p'$. Let the input transition to $s_m$ on $p'$ be $c_i.l$ and the output transition from $s_m$ on $p'$ be $c_o/l$. If we "unmerge" $D'$ and remove the transition labels, $c_i.l$ becomes $c_i$ in the original DFA $D$, and $c_o/l$ becomes $c_o$ (due to the one-to-one mapping between transitions in $D$ and $D'$). Thus, the same path exists in $D$.

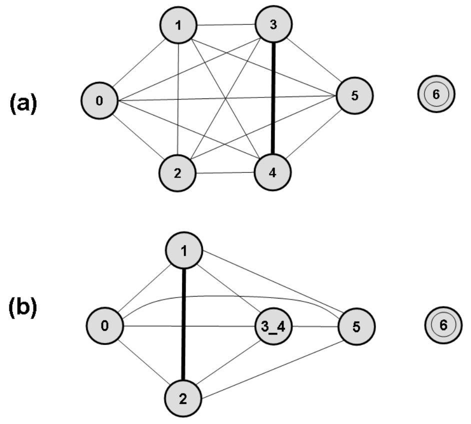


Fig. 7. Weight graphs for the DFA in (a) Figure 1 and (b) Figure 4. Bold edges have weight 3, others have weight 2.

## C. Merging and Labeling Algorithm

We now describe our merging and labeling algorithm. The goal of the algorithm is to merge states in such a way that the total memory requirement is minimized. We assume that the maximum number of labels is fixed, and associate each state pair with a weight. The weight provides a measure of the potential memory reduction achieved by merging the two states. A merge operation on two states $s_A$ and $s_B$ can cause the weight to change for other states, specifically the states connected to $s_A$ and $s_B$ and the states that share common targets with $s_A$ or $s_B$. It is difficult to formulate and analyze an optimum algorithm since the DFA changes with each operation. Our algorithm therefore proceeds iteratively, selecting the best choice at each iteration.

In order to keep track of the weights and efficiently access and modify them, we make use of a graph (informally referred to as the "weight graph") and a heap. The graph has a node for each state in the DFA, and an edge between every pair of states which can potentially be merged. The weights of the edges are stored in a heap. Figure 7 shows the weight graphs during the first two merging steps for the example from Figure 1. For purposes of illustration, in the figure the weights are assumed to be the number of next states that are common to the two states, rather than the memory savings due to the merge operation. Figure 7(a) shows the weight graph for the original DFA, and Figure 7(b) the weight graph after merging states 3 and 4.

The following terminology is useful in understanding the pseudocode of the algorithm (Figure 8). $D$ is the DFA being processed, assumed to be a global variable. $G$ is the weight graph corresponding to $D$. $e$ is an edge in $G$, and $h$ is a heap storing weight graph edges ordered according to edge weights. For two DFA states $s$ and $t$, $metric(s,t)$ is a measure of the memory savings when $s$ and $t$ are merged. $max\_labels$ is the

**Algorithm V.1:** MERGE_STATES($DFA\ D$)

```
weight graph G; heap h; weight graph edge e; state s;
init($G, h$)
$e \leftarrow$ deletemax($h$)
while ($e \neq NULL$ and $weight(e) > 0$)
    do  $s \leftarrow$ merge($G.left(e), G.right(e)$)
        update($G, h, s$)
        $e \leftarrow$ deletemax($h$)
```

**Algorithm V.2:** INIT($graph\ G, heap\ h$)

```
state pair s,t; edge e;
for each (s,t in D with s $\neq$ t)
  if (num_sub-states($s$) + num_sub-states($t$) <= $max\_labels$)
    do  $e \leftarrow G.$connect($s, t$)
        $e.weight \leftarrow$ metric($s, t$)
        $h.insert(e, e.weight)$
```

**Algorithm V.3:** UPDATE($graph\ G, heap\ h, state\ s$)

```
state t,u;
for each (t connected to s in G)
  if (num_sub-states($s$) + num_sub-states($t$) > $max\_labels$)
    do  $h.delete(G.edge(s, t))$
        $G.delete(G.edge(s, t))$
    else  $h.changekey(G.edge(s, t),$ metric$(s, t))$
for each (t with transition to s in D)
  for each (u connected to t in G)
    $h.changekey(G.edge(t, u),$ metric$(t, u))$
```

Fig. 8. DFA Merging and Labeling Algorithm.

maximum number of labels allowed. For states $s$ and $t$ with memory savings greater than 0, and whose sub-states do not exceed $max\_labels$ in number, there is an edge $e$ in weight graph $G$, the weight of $e$ being equal to $metric(s, t)$.

Algorithm V.1 shows the core of our procedure, which is a loop that terminates when merging can no longer produce memory savings, or if more than the maximum allowed labels are required. Algorithm V.2 constructs the weight graph $G$ and stores the edges in heap $h$. It considers all pairs of states in the given DFA. If the total number of sub-states in the state pair under consideration is less than the number of labels, it evaluates $metric(s, t)$ and assigns it to the weight of the edge connecting states $s$ and $t$ in the weight graph. Recall that $metric(s, t)$ is an exact measure of the memory savings possible when states $s$ and $t$ are merged.

When two states are merged, data structures of other states in the DFA, as well as weights of edges in the weight graph, must be updated. Algorithm V.3 shows a procedure for doing this. The first part updates the weights of all edges connected to the newly merged state. It also removes those edges from the heap and weight graph if the total number of sub-states of their state pair exceeds the number of labels. The final part of Algorithm V.3 recalculates the metric corresponding to state pairs where one state is connected to the merged state. This is required because states connected to a merged state will have fewer destinations, changing their metric.

### D. Analysis

We now present a complexity analysis of the above algorithms. The original DFA is assumed to have $n$ states.

First, we analyze the complexity of each iteration of Algorithm V.1. There is one call to the weight graph initialization function of Algorithm V.2. For each iteration, there is one call to the update function of Algorithm V.3. All heap operations take logarithmic time in the number of heap elements. Data structure initialization visits every state pair in the DFA, a maximum of $n^2$ state pairs. Inserting these elements into the heap therefore leads to $O(n^2 logn)$ complexity. Next, a single update operation in Algorithm V.3 has the same $O(n^2 logn)$ complexity if the states required to be updated are connected to all the states in the graph. In practice however, very few states need to be updated each time. Also, due to merging, the number of states in the graphs decreases with each iteration.

The maximum possible iterations occurs when we merge only two states per iteration, and continue until all merged states contain $max\_labels$ original states. In other words, we start with $n$ original states, and end up with $n/max\_labels$ states, meaning we can have up to $(n - n/max\_labels)$ iterations in Algorithm V.1. Since each iteration has complexity $O(n^2 logn)$, this leads to an overall complexity of $O(1 - 1/max\_labels)n^3 logn)$.

### E. Rule Database Updates

Updates to rule databases occur infrequently enough to not warrant processing them at LPM-like rates of several thousand per second. One strategy to handle updates is to create two new DFAs, one to hold the new regular expressions to be added, and the other to hold regular expressions to be removed. All three DFAs are looked up in parallel. Another strategy is to use a shadow copy of the merged DFA data structures. The shadow copies are updated offline, after which they switch positions with the online DFAs. Since state merging and labeling is compatible with any of these existing techniques, we do not describe them further in this paper.

## VI. Experimental Results

Now we present experimental results on Bro [2] and Snort [1] security rule-sets. We evaluate the benefits of our scheme over the naive implementation, bitmaps and bitmaps with pointer indirection (both described in Section III). The results show the reduction in the number of states and distinct transitions, as well as the final memory savings.

### A. Experimental Setup and Memory Representation

Our simulation infrastructure consists of a DFA generator and a DFA merging module. The latter is parameterized in the maximum number of sub-states a merged state can consist

| Data-set | | Characteristics | | | | Before Merging | | | After Merging | |
|---|---|---|---|---|---|---|---|---|---|---|
| IDS | source file | RegEx | Single wildcards | .* | .{x,y} | States | Transitions | Distinct Trans. | States | Distinct Trans. |
| Bro v0.8 | ex-web | 233 | 3 | - | 1 | 3,287 | 841,472 | 181,783 | 450 | 11,181 |
| Bro v0.9 | sig-addendum | 32 | 15 | - | 1 | 3,187 | 815,872 | 79,561 | 2,251 | 41,754 |
| SNORT 07/2006 | policy | 5 | - | 2 | - | 154 | 39,424 | 818 | 19 | 64 |
| SNORT 07/2006 | p2p | 6 | - | 6 | - | 210 | 53,760 | 1,003 | 27 | 89 |
| SNORT 07/2006 | web-php | 15 | 1 | 1 | - | 1,086 | 278,016 | 14,464 | 435 | 4,169 |
| SNORT 07/2006 | spyware/http | 7 | 1 | 10 | - | 4,984 | 1,275,904 | 39,016 | 534 | 2,053 |
| SNORT 07/2006 | spyware/port25 | 20 | 1 | 21 | - | 7,000 | 1,792,000 | 64,844 | 322 | 1,686 |
| SNORT 07/2006 | backdoor | 13 | - | 7 | - | 7,183 | 1,838,848 | 55,444 | 388 | 1,696 |

TABLE I
SUMMARY OF CHARACTERISTICS OF THE DATASETS USED AND OF THE CORRESPONDING DFAS.

of (i.e., the maximum number of labels allowed) and in the metric guiding the merging choices. Since the basic goal of merging is memory reduction, the metric selected gives a direct measure of the effective memory reduction due to each merger. We assume 256-bit wide bitmaps (note that existing bitmap compression techniques could be used on top of this, but we do not employ them in this paper). The width of the transition table is set to 32 bits, amenable to a software implementation. Labels add $log_2(max\_labels)$ to the width of the pointer indirection table. Apart from this, the width of the indirection table is a function of the number of distinct outgoing transitions of the corresponding state. This results in a few different widths for the pointer indirection table entries; blocks with different widths are assumed to be laid out in different memory regions aligned to 32-bit boundaries.

## B. Rule Databases

As mentioned, the datasets used consist of subsets of rules from Bro [2] and Snort [1] NIDS. We considered two distinct Bro rule-sets, one from v0.8 and one from v0.9. In the case of Snort, we considered a snapshot from July 2006 and processed subsets of rules containing Perl compatible regular expressions (PCREs). The first six columns of Table I detail the sources of the regular expressions and their general characteristics. All datasets contain wildcards and a few present counting constraints. We pre-filtered large datasets based on headers. Specifically, *backdoor*, *spyware/http* and *spyware/port25* datasets contain rules filtered respectively according to the headers $HOME_NET any - $EXTERNAL_NET $HTTP_PORTS/any, $HOME_NET any - $EXTERNAL_NET $HTTP_PORTS/any, and $HOME_NET any - $EXTERNAL_NET 25/any.

Columns seven to nine in Table I summarize the characteristics of the resulting DFAs. The DFA sizes vary from 154 to 7183 states. We also characterize the DFAs in terms of number of distinct transition to next states they contain. Two transitions from the same state are considered to be distinct if they lead to two different target states (regardless of the symbol upon which the transitions are taken). The number of distinct next state transitions shown in the table is summed over all the states in the DFA. Note that the Snort rule-sets have lower percentages of distinct next state transitions than the BRO rule-sets. This is due to the large number of character ranges (both in the form [$c_1$-$c_2$] and \d, \D, \w,\W,\s,\S) and to the fact that Snort regular expressions are not case sensitive.

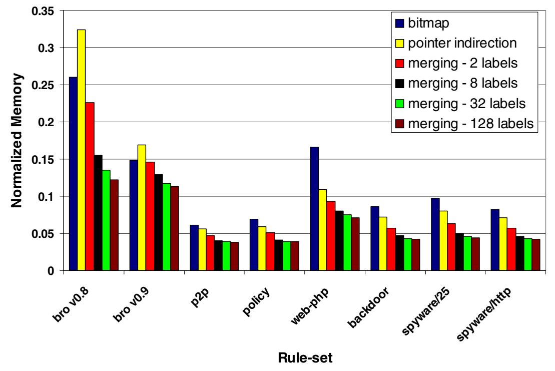


Fig. 9. Memory reduction with state merging, normalized to a naive implementation.

Table I shows the total number of states and distinct transitions after merging with 128 labels. The distinct transitions refer to the number of entries in the transition tables, which are used to computing the total memory requirement. The naive data structure represents all transitions in the DFA, whereas our data structure only represents distinct transitions. State merging reduces the number of states by an order of magnitude, and the number of transitions by two orders of magnitude.

## C. Memory requirement

Regardless of the reduction in the number of states and transitions, the bottom-line is memory reduction. Note that a 10x or a 100x reduction in the numbers of states and transitions will translate to a smaller reduction in memory usage because of overheads in the data structure, and the fact that some parts of merged states cannot be compacted. Figure 9 compares the memory requirement of the bitmap solution, of bitmaps with pointer indirection and of state merging with 2, 8, 32 and 128 labels (above which no further memory reduction is observed). We set the failure pointer of each state to its most frequently occurring target state. The data are normalized to the memory requirement of the naive data structure shown in Figure 2.

We observe that pointer indirection is better than the bitmap solution only if the additional data structure overhead is compensated by a reduction in the number of next state pointers. Because of the higher percentage of distinct transitions, pointer indirection alone is not beneficial for Bro databases. It can be noted that, despite the label overhead, state merging always reduces the memory requirement. Not only does it allow us to take advantage of common transitions *within a* state, but also *across* different states. The final memory requirement decreased by a factor of 10 for Bro and by a factor of 15-25 for Snort (over the naive implementation).

Compared to bitmaps, state merging on average halves the total memory requirement. Note that bitmaps present a constant overhead for the plain bitmap solution, the pointer indirection solution as well as the state merging technique. It is well known that bitmaps can be further compressed using several techniques [17]. Thus, if we ignore the overhead of the 256-bit wide bitmap in order to isolate the memory savings of state merging, we see that state merging produces a 5X memory reduction over bitmaps.

In real applications, complexity of a rule-set is indicative of the number of wildcards in its regular expressions, a characteristic that generally results in more common transitions both for a state and across different states. This is a favorable trend to techniques like state merging.

Finally, we note that state merging does not affect performance. Specifically, the number of memory accesses remains the same. To see this, consider the pointer indirection table that has the additional overhead of labels. Given that both the pointer indirection table entry as well as the number of bits required for the label are well below 32, the introduction of labels won't necessitate more memory accesses in a software implementation for typical DFAs. In a specialized hardware implementation, memory words may be carefully adjusted to ensure performance is unaffected.

A quantitative comparison between our scheme and D2FA [6] is difficult since the authors do not compute the actual memory savings, but report the reduced number of transitions. A possible implementation of D2FA could use bitmaps to filter out the default transitions, incurring the same bitmap overhead we experience. A D2FA implementation recently proposed in [18] uses ad-hoc state identifiers and assumes very wide memory accesses. This is therefore suited to a hardware implementation. We refer the reader to the qualitative comparison presented in Section II. Note that treating the default pointers as failure pointers, state merging can in fact be performed on top of a D2FA.

## VII. Conclusions

Regular expressions are increasingly used in networking security and policy management. A popular technique for performing regular expression search is the DFA. While DFAs offer $O(1)$ lookup time per input character, their memory requirements make them impractical for realistic rule-sets. Both software and hardware implementations of regular expression search engines that use DFAs are hampered by their prohibitive memory usage. Compounding this problem is the need to perform online payload inspection at high line speeds. Due to reasons pertaining to denial-of-service attacks, routers implementing these services must also provide a worst-case speed guarantee.

In this paper, we described a scheme to drastically reduce memory usage of DFAs while not affecting their speed, and still providing worst-case guarantees. We introduced the notion of merging non-equivalent states of a DFA using transition labeling. Significant memory savings are achieved when states with common destinations are merged and represented using our data structure. Unlike other DFA compaction approaches, we have no requirement on the transitions on which the two states reach their common destinations. Another advantage of our scheme is that merging states creates more common destinations for other states, thereby providing opportunities for more merging. We describe a polynomial-time state merging and labeling algorithm, and show that state merging does not affect the language accepted by the DFA.

We perform several experiments on real rule-sets from the Snort and Bro IDS, and show memory compactions ranging from 10X to 25X compared to the basic DFA data structure, and 5X compared to a bitmap-based data structure.

The results presented in this paper exemplify a software implementation. Future work encompasses tailoring the proposed scheme to a specialized hardware architecture.

## References

[1] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in *13th System Administration Conference*, pp. 229–238, Nov 1999.
[2] "A System for Detecting Network Intruders in Real Time (`http://www.icir.org/vern/bro-info.html`).".
[3] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," in *ACM Conference on Computer and Communication Security*, pp. 262–271, 2003.
[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Automata Theory, Languages and Compilation*. Addison Wesley, 3rd ed., 2004.
[5] V. Paxson, "Fast Lexical Analyzer Generator (`http://ftp.ee.lbl.gov/flex-2.5.4.tar.gz`),"
[6] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in *ACM Sigcomm*, Sept 2006.
[7] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," in *IEEE Infocom*, pp. 333–340, Mar 2004.
[8] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, pp. 762–772, 1977.
[9] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, 1994.
[10] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, pp. 333–340, 1975.
[11] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," in *ISCA*, 2005.
[12] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *ICNP*, pp. 174–183, 2004.
[13] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection."
[14] B. C. Brodie, R. K. Cytron, and D. E. Taylor, "A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching," in *ISCA*, pp. 191–202, 2006.
[15] T. V. Lakshman and D. Stidialis, "High Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," in *ACM Sigcomm*, Sept 1998.
[16] F. Baboescu and G. Varghese, "Scalable Packet Classification," *IEEE/ACM Transactions on Networking*, pp. 2–14, Feb 2005.
[17] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 1st ed., 2004.
[18] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *IEEE/ACM ANCS*, Dec 2006.