Scalable Regular Expression Matching on Data Streams

Anirban Majumder
Bell Labs Research India
manirban@alcatel-lucent.com

Rajeev Rastogi*
Yahoo! Labs, Bangalore
rrastogi@yahoo-inc.com

Sriram Vanama†
Indian Institute of Technology, Madras
vsriram@cs.iitm.ernet.in

ABSTRACT

Regular Expression (RE) matching has important applications in the areas of XML content distribution and network security. In this paper, we present the end-to-end design of a high performance RE matching system. Our system combines the processing efficiency of *Deterministic Finite Automata* (DFA) with the space efficiency of *Non-deterministic Finite Automata* (NFA) to scale to hundreds of REs. In experiments with real-life RE data on data streams, we found that a bulk of the DFA transitions are concentrated around a few DFA states. We exploit this fact to cache only the frequent core of each DFA in memory as opposed to the entire DFA (which may be exponential in size). Further, we cluster REs such that REs whose interactions cause an exponential increase in the number of states are assigned to separate groups – this helps to improve cache hits by controlling the overall DFA size.

To the best of our knowledge, ours is the first end-to-end system capable of matching REs at high speeds and in their full generality. Through a clever combination of RE grouping, and static and dynamic caching, it is able to perform RE matching at high speeds, even in the presence of limited memory. Through experiments with real-life data sets, we show that our RE matching system convincingly outperforms a state-of-the-art Network Intrusion Detection tool with support for efficient RE matching.

Categories and Subject Descriptors

H.3.5 [**Information Systems**]: Information Storage and Retrieval—*On-line Information Services*; H.2.m [**Information Systems**]: Database Management—*Miscellaneous*

General Terms

Algorithms, Design

*Work done while the author was at Bell Labs Research, India.

†Work done while visiting Bell Labs Research, India.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

Keywords

Data Streams, Deep Packet Inspection, Regular Expression Matching

1. INTRODUCTION

1.1 The RE Matching Problem

Informally, the RE matching problem seeks to detect all subsequences of an input stream that match any of a given set of *regular expressions* (REs). Stated more formally, let I be a (potentially infinite) stream of input symbols drawn from a finite alphabet Σ and let $R = \{r_1, \ldots, r_m\}$ be a set of REs over Σ. Then the RE matching problem is to detect all instances when a subsequence of I belongs to the language for an RE $r_i \in R$.

Alternately, suppose that we modify each RE r_i by prepending Σ^* to it; thus, each new r_i is now $\Sigma^* \cdot r_i$. Then, since Σ^* matches all sequences, our RE matching problem essentially boils down to detecting all prefixes of I that match any of the (new) r_i's in R. We address this problem in the paper.

1.2 Applications

Due to their expressive power, simplicity, and flexibility, REs are being extensively employed in content distribution and network security applications. For instance, consider XML content routing [15, 6]. Here, an overlay network of XML routers delivers XML documents of interest from producers to consumers. Each consumer specifies the documents he/she is interested in via XPath-based subscriptions. (XPath expressions incorporate a restricted form of REs.) Every router on the path between a producer and a consumer then aggregates the XPath subscriptions of all the consumers reachable from it, and only forwards incoming XML documents that match the (aggregated) subscriptions. Thus, documents are not blindly flooded in the network, but rather selectively forwarded to only those consumers who have expressed an interest in them. This helps to significantly reduce the load on network routers and links.

Network intrusion detection systems (NIDS) like Snort [10] perform *deep packet inspection* on an IP traffic stream to check if the packet payload (and header) matches a given set of signatures of well known security threats (e.g., viruses, worms). Traditionally, deep packet inspection has been limited to comparing packet contents to sets of strings. However, newer systems are replacing strings with REs because their increased expressiveness, ease of representation, and overall simplicity in terms of understanding. For instance, the Snort NIDS has evolved from no REs in its ruleset in April 2003 to 1131 (out of 4867 rules) using REs as of February 2006 [20]. Recently, Cisco has incorporated an RE-based content inspection capability into its Internetworking Operating Sys-

tem (IOS) [19], and layer-7 filters based on REs [14] are available for the Linux operating system as well.

In the RE matching applications described above, data (e.g., XML documents, IP packets) arrives at very high rates. Moreover, the number of RE patterns can be very large (e.g., Snort currently has over over 1500 RE-based rules [20], some of which are very complex). Thus, it is critical that our RE matching system have low processing and space overheads so that it can simultaneously meet the stringent throughput and memory requirements imposed by the above application domains.

1.3 Basic RE Matching Techniques

While flexible and expressive, pattern matching using REs is challenging in terms of both system throughput and memory usage. Traditionally, matching algorithms employ finite automata-based representations for REs. There are two types of automata: *Deterministic Finite Automaton* (DFA) and *Non-deterministic Finite Automaton* (NFA).

A DFA is a quintuple $M(S, s_0, \delta, \Sigma, F)$ where S is a finite set of states, $s_0 \in S$ is a special *start state* and $F \subseteq S$ is the set of *accepting* states, Σ is the *alphabet* (or symbol set) and $\delta : S \times \Sigma \rightarrow S$ is the *transition function*. An NFA N is defined in a similar way except that the transition function may return a set of states **i.e.** $\delta : S \times \Sigma \rightarrow 2^S$. We will use the notation $M(r)$ (or $N(r)$) to represent the DFA (or NFA respectively) corresponding to an RE r, and $|M(r)|$ (or $|N(r)|$) to denote the number of states in it. Furthermore, we will consider $r_1|\cdots|r_i$ to be the equivalent RE for an RE set $R_i = \{r_1, \ldots, r_i\}$, and the DFA for R_i $M(R_i) = M(r_1|\cdots|r_i)$ ($N(R_i)$ is defined similary). Now, any RE of length n can be efficiently converted into an equivalent NFA with $O(n)$ states [8]. Further, for every NFA, we can construct an equivalent DFA that essentially simulates the behavior of the NFA. The states of the DFA correspond to subsets of NFA states, and DFA state transitions are computed by applying the NFA state transitions to the NFA states corresponding to each DFA state. Thus, the DFA for a size n RE may contain as many as $O(2^n)$ states [8].

NFA-based solution. Here, the input REs in R are compiled into a single NFA by constructing an NFA for the combined RE $r_1|\ldots|r_m$. This is very efficient in terms of storage overhead. However, in an NFA, multiple states can be active at the same time, and thus each input stream symbol can cause multiple state transitions. In the worst case, all NFA states can be active simultaneously, and this can severely degrade the throughput of the pattern matching system. The experimental evaluation reported in [20] claims that for Snort rulesets a DFA-based approach is 50 to 700 times faster compared to a NFA-based solution. The NFA-based execution model is used by systems such as Snort [10] (for network-specific signature matching) and YFilter [5] (for filtering XML documents according to XPath queries).

DFA-based solution. One way to address the performance problems of the NFA-based approach is to combine the REs in R into a single DFA. Since a DFA has only one active state at any given time, it makes only one transition for every input symbol. However, as pointed out earlier, converting an NFA into a DFA may result in an exponential blow-up in the number of states. For illustrative purposes, we present below two common culprits responsible for DFA state space explosion in network packet scanning and XML routing applications [20, 7].

• *State explosion in single RE.* Many commonly found REs contain wildcards with length restrictions. For example, consider the RE $\Sigma^* a^+ \Sigma\{j\} b$ which matches any subsequence in which a preceding a is separated from b by exactly j characters. The problem here is that once a^+ in the RE has been matched, any subsequent symbol a in the input stream can be used to either match $\Sigma\{j\}$ or initiate a new different match with a^+. Thus, the DFA for the RE will need to keep track of all patterns of occurrence of a in the last j characters, and for this it will need $O(2^j)$ states.

As an example, the Snort rule `"AUTH\s[^\n]{100}"` detects an IMAP authentication overflow attempt where the input contains the string `AUTH` followed by a white space and then no return character in the next 100 bytes. A DFA for this rule will contain well over a million states! [20] reports that over 10% of the RE patterns in Snort contain wildcards with length restrictions that can lead to the above-mentioned exponential state blow-up.

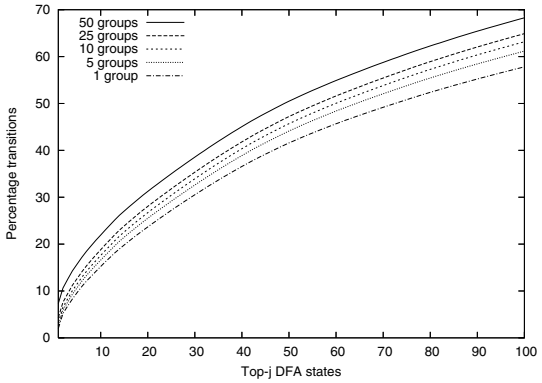

Figure 1: Distribution of DFA transitions over top-j states.

• *State explosion due to multiple-RE interactions.* When REs contain the unrestricted length wildcard term Σ^*, then the space requirements grow exponentially with the number of REs containing Σ^*. For example, suppose that $r_i = \Sigma^* x_i \Sigma^* y_i$ where $x_i, y_i \in \Sigma^+$. Then since Σ^* matches all the prefixes x_i, the DFA for REs $r_1, \ldots, r_m$ will need to keep track of which subset of the prefixes x_i have been matched for the various REs at any point in time. Since there are 2^m such subsets for the m prefixes, the number of DFA states required is at least $O(2^m)$.

The spyware rule `"User-Agent\x3A[^\r\n]*ZC-Bridge"` in Snort is an example of a rule containing an unrestricted length wildcard term. It looks for an occurrence of the sub-pattern `ZC-Bridge` only if `User-Agent\x3A` has been previously detected and no carriage return or new line character occurred in between.

The key point to note here is that due to the exponential growth in the number of states, *DFA construction may simply be infeasible for most commonly found RE sets*.

1.4 Overview of Our Approach

Our RE matching strategy leverages the best of the NFA and DFA based approaches described above. It combines the space efficiency of the NFA solution with the processing efficiency of the DFA solution to deliver high throughput for hundreds of REs with limited memory usage.

Our approach relies on the following key observation: *For many real-world data streams and REs, a bulk of the DFA transitions are concentrated around a few DFA states.* The plots in Figure 1 validate this observation. They show the results of a simple experiment that we conducted using a subset of 100 Snort rules con-

taining wildcards with length restrictions upto 512. We matched the rules against a real-world TCP network packet trace (of size 16MB) obtained from the MIT Lincoln Laboratory [9]. We considered partitionings of the rules into k groups for $k = 1, 5, 10, 25$, and 50. For each partitioning of the rules into k groups, we constructed k (partial) DFAs for the k groups by lazily expanding DFA states when the k DFAs are run on the traffic stream from the MIT site. Now, for a given j, consider the top-j most frequently visited states during a run. (Note that the number of times a state is visited is approximately equal to the frequency of transitions out of the state.) In the graph in Figure 1, we vary j (from 0 to 100) along the horizontal axis, and plot on the vertical axis (for each j) the percentage of transitions out of the top-j states. The average number of states visited in the lazy DFAs was around 1000. Therefore, the top-100 states constitute nearly 10% of the total DFA states.

From the plots, we observe two interesting trends. First, most transitions involve only a few DFA states. Thus, for a majority of the time, the execution of the DFA is confined within a small fraction of the entire DFA. For example, as can be seen from the figure, nearly 50% of the execution time is spent in only 5% (50 states) of the entire DFA. The top-100 states (10% of the DFA) account for almost 70% of all transitions. So, although the total number of DFA states is huge (exponential in number), only a tiny fraction of the total possible states is actually visited, and further, a big chunk of the visits center around only a small portion of the visited states.

Second, as the number of groups k is increased, the transitions become more and more concentrated among a fewer set of states; thus, the locality of the transitions becomes more skewed. The explanation for this is that, with more groups, each group contains fewer REs, and this leads to much smaller overall DFA sizes due to reduced inter-RE interactions.

Our approach exploits the above two trends to perform fast RE matching (like the DFA approach) while incurring a small memory overhead (similar to the NFA approach). Essentially, it caches only the few most frequently visited DFA states (along with their outgoing transitions) in memory. As a result, a majority of the DFA state transitions can be carried out by performing only a single memory lookup. Only in the rare occasions when a particular transition for an input symbol is not in memory do multiple lookups need to be performed to construct the DFA transition as in the NFA approach. Thus, we are able to enjoy the throughput advantage of the DFA approach without having to store the entire DFA (which may be exponential in size).

In addition to caching frequent states, our approach also clusters the REs into k groups to further boost system throughput. This is because grouping, by shrinking the k individual DFAs, focuses the high frequency transitions among a fewer number of states in the k DFAs. Consequently, grouping allows more frequent states and transitions to be cached (in the same amount of memory), thus increasing cache hits and speeding up performance. There is a trade-off here, however, since grouping also causes the processing time to increase k-fold. This is because the k active DFA states will trigger k state transitions for each input stream symbol. Thus, it is not really desirable to have too many groups (e.g., with each RE in a separate group) since this would cause state transitions in too many automata (one per group).

1.5 Our Contributions

We make the following contributions in this paper.

- We present the end-to-end design of a high performance RE matching system that employs DFA state caching and RE grouping to scale to hundreds of REs. A key feature of our system is that it can handle general REs unlike existing proposals that restrict the RE syntax to specific expressions found in NIDS like Snort [10]. (See Section 2 for a detailed comparison with related work.)

- As mentioned earlier, grouping of REs involves trade-offs – on the one hand, it increases cache hits by packing more frequent transitions in the cache, but on the other hand, it increases the number of state transitions for each input symbol. To strike the right balance between the two, we employ an agglomerative clustering algorithm that greedily merges groups of REs to find a grouping that minimizes the overall processing overhead per input symbol. Our key contribution is a novel technique to estimate processing costs based on DFA samples – these samples are constructed using information about the probability distribution of symbols in the input stream. Our clustering algorithm assigns the REs whose interactions can result in state explosion to separate groups, thus controlling the overall DFA size and processing costs.

  This a major departure from existing approaches [16] for grouping REs that primarily use simple heuristics (e.g., cluster REs with a common prefix in the same group) or manual methods which may not work well in practice. Further, other proposals [20] construct the entire DFA for REs which may not be practical. In contrast, our methods do not compute the complete DFAs, but instead rely on DFA samples.

- We present a comprehensive caching strategy that employs both a static component as well as a dynamic one. We develop a computationally efficient scheme for determining static high frequency and densely connected DFA portions to store in the cache, and supplement this with a simple and efficient technique for dynamically replacing DFA states in the cache while always maintaining dense connectivity.

- Finally, through an extensive evaluation using real-life Snort RE data sets, we demonstrate the effectiveness of our approach compared to the Bro NIDS [16] that has support for efficient RE matching.

To the best of our knowledge, ours is the first end-to-end system capable of matching REs at high speeds and in their full generality. Through a clever combination of RE grouping, and static and dynamic caching, it is able to perform RE matching at high speeds, even in the presence of limited memory.

2. RELATED WORK

There is a vast body of research literature on optimizing the performance of string matching – some of the prominent algorithms include Aho-Corasick [1], Commentz-Walter [4], and Wu-Manber [17]. However, the RE matching problem, despite its importance, has received relatively little attention from the research community. Recent papers [16, 20, 13] have proposed RE matching algorithms that employ DFA representations to achieve low CPU costs and high performance. However, the algorithms can only handle very specific restricted forms of REs typically found in current packet payload scanning applications like Snort [10]. Consequently, for general REs found in many practical scenarios, the algorithms will either simply not work or their performance will be severely impacted, as explained below.

To cope with the exponential number of DFA states, the Bro NIDS [16, 18] builds the DFA "on-the-fly" during the actual matching. It groups RE patterns that match the same kind of traffic based on additional constraints like IP address ranges or ports contained

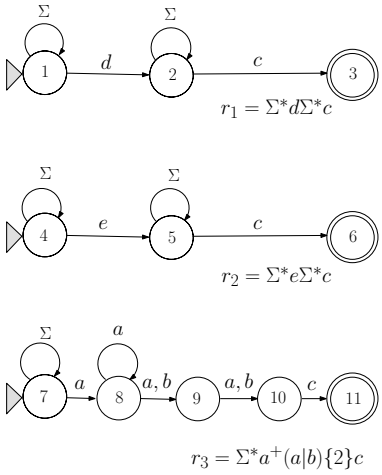

Figure 2: NFAs for r_1, r_2, and r_3.

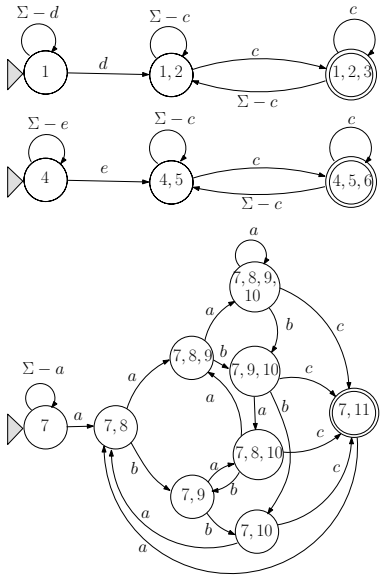

Figure 3: DFAs for r_1, r_2, and r_3.

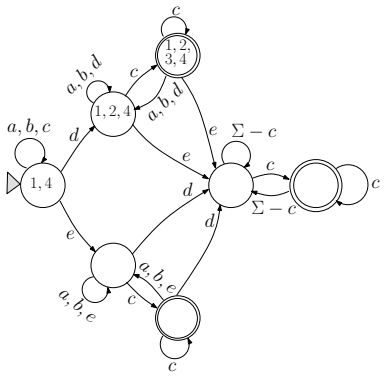

Figure 4: DFA for $R_1 = \{r_1, r_2\}$.

in intrusion detection patterns. Clearly, these ad-hoc grouping techniques to reduce DFA size will not work for general RE sets. Furthermore, Bro implements a memory-bounded dynamic cache to store DFA states, and expires old states on a least-recently-used (LRU) basis. However, such a dynamic cache has a significant cache maintenance overhead which can hurt the matching speed. In contrast, our approach implements a static cache as well in which it stores the frequent DFA states and transitions computed from DFA samples. States are evicted only from the dynamic cache, and that too using a strategy that takes into account both the last time of access as well as the frequency and connectivity to other states in the cache.

To reduce space usage, Yu et al. [20] propose two rewrite rules for specific REs found in NIDS like Snort and Bro. However, their rewrite techniques only apply to a small restrictive subset of REs, which does not even cover all the Snort rules. For instance, [20] cannot handle the following Snort rule whose DFA has an exponential size: `\s*filename\=\"[^\n]{100}\.(exe|lnk)"`. Another problem with the proposed rewriting rules is that they alter the semantics of the REs which is not too desirable. [20] also proposes a simple grouping algorithm for partitioning the input RE patterns into k groups such that patterns in a group do not interact with each other adversely to cause the DFA state space to blow up. However, the grouping scheme requires the DFA for every pair of REs to be constructed to measure inter-RE interactions, which may not always be feasible. In contrast, our grouping scheme relies on DFA samples as opposed to the actual DFAs, and aims to minimize the actual processing cost instead of just the interactions among REs.

Kumar et al.[13] use the rewriting and grouping techniques of [20] to compute DFAs for groups of REs. So their approach also suffers from all of the shortcomings of [20] mentioned above. Further, they reduce the space requirements of the computed DFAs by replacing several transitions (between states with identical outgoing transitions) with a single default transition. However, as we will see later, states, and not transitions, constitute the bulk of the space overhead. As a result, since the schemes of [13] do not reduce the number of states, their applicability is somewhat limited.

Other related work on RE matching includes the lazy DFA of Green et al.[7] and the RE-tree of Chan et al.[2]. The lazy DFA scheme, similar to [16], constructs the DFA lazily as stream symbols are processed. Further, it does not maintain a cache, but rather assumes that the completely expanded lazy DFA will be small enough to fit in main memory. This is clearly a strong assumption and may not always hold, especially for potentially infinite streams. Another drawback of the lazy DFA approach is that it combines all the REs into a single DFA – as a result, it does not realize the DFA size reductions that are possible due to grouping. The RE-tree is a hierarchical index structure for REs similar in spirit to other spatial index structures like the R-tree. A hierarchical approach like an RE-tree requires multiple passes over the input stream. This can be problematic for long (potentially infinite) streams since the stream prefix matching ($\Sigma^* r_i$) may be too big to accommodate in main memory. The hierarchical approach in RE-tree also suffers from the problem of false positives since upper level REs are more general, and thus may indicate a match even though there is none. In this case, successive passes are needed to match the REs in the child nodes, which could be wasteful.

Recent work of Muthukrishnan et al.[12] looks at the regular expression matching problem on out-of-order data streams. In their context, elements in the data stream define some order and the delivery mechanism of the underlying network might produce out-of-order arrival of data elements at the receiver. The problem is to efficiently match a set of regular expressions at the receiver, with respect to the ordered data stream, without performing explicit reordering of the stream elements. In order to perform accurate and efficient stream processing, they start the simulation of the DFAs from all potential beginning states (which, as they mention, could be all non-accepting states of the DFAs in the worst case) and progressively prune away redundant states as successive fragments of the stream elements arrive. In our work, we do not consider the out-of-order arrival of stream elements. Moreover, their techniques build the DFA fully, which may be impossible for many of the regular expressions that we consider in this paper, because of their exponential sizes.

3. RE MATCHING SYSTEM ARCHITECTURE

In this section, we describe the architectural details of our RE matching system and the high-level RE matching logic.

3.1 Grouping REs

We begin by clustering the set of input REs R into a small number (k) of groups $R_1, \ldots, R_k$ such that the total size of the DFAs for the R_i's is much smaller than the DFA for R.

EXAMPLE 1. *Consider the 3 REs $r_1 = \Sigma^* d \Sigma^* c$, $r_2 = \Sigma^* e \Sigma^* c$ and $r_3 = \Sigma^* a^+ (a|b)\{2\} c$. Figure 2 depicts the NFAs for the 3 REs. We construct the DFAs for the 3 REs in Figure 3, and for each DFA state, we show the corresponding NFA states. (In the DFA for r_3, we omit the transitions to the start state.) The (lazy) DFA when the 3 REs are combined together contains close to 40 states and is too big to depict pictorially. Now suppose that we partition R into two groups R_1 containing r_1 and r_2, and R_2 containing only r_3. The (lazy) DFA for R_1 is shown in Figure 4 and contains 7 states. Since the DFA for r_3 contains 9 states, we get that $M(R_1)$ and $M(R_2)$ (the DFAs for R_1 and R_2) contain a total of only 16 states, and are more than a factor of 2 smaller than $M(R)$ (the DFA for R) which contains close to 40 states.*

The goal of clustering is to reduce the processing cost per input symbol by decreasing the DFA sizes and increasing the cache hits. We present our clustering technique for minimizing CPU cost in Section 4.

3.2 Static Components

NFAs. For each RE r_i in R, we store the NFA $N(r_i)$ in memory. The NFAs are utilized to determine a DFA transition that is missing from the cache. Access to the NFA is expensive; therefore our cache management tries to reduce the number of misses.

DFA Cores. For each cluster R_i, we extract a small frequent core D_i from its DFA $M(R_i)$ and store this in memory. The core is a densely connected portion of the DFA which is likely to be visited more often for the given input stream data distribution. It includes the start state of $M(R_i)$, and also all the transitions between DFA states contained in the core.

With each state in the DFA core, we store the corresponding set of NFA states. The NFA states are used in conjunction with the NFAs to compute the DFA transitions (for the state) that are missing from the DFA core. With each core DFA state, we also store the transitions to other core states - so these transitions can be quickly determined using a single lookup.

There are two sources of memory usage in core DFAs: states and transitions. The space overhead for a DFA state includes the memory required to store the set of NFA states for the state. This can get fairly large, and in the worst case, it can be equal to the number of NFA states in each $N(R_i)$. Transitions, on the other hand, can be stored extremely succinctly by associating an integer identifier $s.id$ with each core state s, and using these short state identifiers for encoding each transition, for e.g., as "$\delta(s_1.id, \sigma) = s_2.id$". Furthermore, the number of transitions is linear with respect to the number of states because for each state there can be at most $|\Sigma|$ (256 for the ASCII character set) links to next states. Thus, the transitions take up less space compared to the memory required for storing the DFA core states. Therefore, we consider the number of states as the primary factor for determining the memory usage, and assume that the available memory has sufficient capacity to accommodate up to α DFA states.

Finally, we maintain appropriate indices for accessing the entry for a core DFA state using either its identifier or the set of NFA states corresponding to it. These indices can be used to determine if a state is present in the DFA core or not, and if present, gain access to the entry for the state (containing its NFA states and transitions).

We allocate α_d portion of the memory to store the DFA cores. In Section 5, we present a heuristic for computing a good set of cores $\mathcal{D}$ for the RE clusters such that (1) the number of core states is at most α_d, and (2) the probability of missing a transition when processing the input stream is minimized. Note that the second condition also minimizes the overall computation cost.

3.3 Run-time Components

Cache. We maintain a cache $\mathcal{C}$ of size α_c ($\approx \alpha - \alpha_d$) to store additional transitions that are not included in the DFA cores. Unlike the DFA cores which are static, the cache is dynamic and can thus adapt to changes in the input stream data distribution. However, there is a run-time overhead associated with cache maintenance for adding and deleting states and transitions from the cache. In fact, [16] reports that this additional overhead of maintaining the cache can slow down system performance by more than a factor of 2. Thus, we allocate the bulk of the memory to the DFA cores, and a much smaller portion to the cache $\mathcal{C}$.

Our caching algorithm, described in Section 6, prefers to cache states based on their recency and connectivity to other states in the cache. It maintains the invariant that for a transition $\delta(s_1, \sigma) = s_2$ to be cached, both states s_1 and s_2 must be present in the cache.

Current DFA states. For each of the k DFAs for the k RE clusters, we keep track of the currently active DFA state.

3.4 RE Matching Logic

Algorithm 1 describes our RE matching procedure. The variable cur_st_i maintains the currently active state for $M(R_i)$. Transitions that are already in the DFA cores or cache take a single memory lookup to resolve. However, for a missing transition $\delta(cur\_st_i, \sigma)$, we first obtain the set of NFA states corresponding to cur_st_i, and then look up the individual transitions in $\mathcal{N}$ for each NFA state on the symbol σ to obtain the NFA states for the next DFA state. This requires lookups equal to the number of NFA states in cur_st_i. We finally insert this state nxt_st_i and the transition into the cache. Observe that Procedure REPLACE may need to eject an existing state in the cache to make room for nxt_st_i and is described in further detail in Section 6. Also, note that a transition is only added to cache $\mathcal{C}$ if the two states involved in the transition are already in the cache - this is to preserve the cache invariant mentioned earlier. It is easy to see that, unlike the DFA cores, the cache $\mathcal{C}$ may contain only a subset (and not all) of the transitions between the DFA states in $\mathcal{C}$.

EXAMPLE 2. *Figure 5 shows the static and run-time components for the 3 REs $r_1 = \Sigma^* d \Sigma^* c$, $r_2 = \Sigma^* e \Sigma^* c$ and $r_3 = \Sigma^* a^+ (a|b)\{2\} c$. REs r_1 and r_2 are grouped together in a single cluster R_1, and r_3 is kept in a separate cluster R_2. The DFA cores D_1 and D_2 for the two clusters are stored in memory. For the input stream dabaaba, transitions for the first DFA ($M(R_1)$) are carried out completely using the DFA core D_1 and the current state is $\{1, 2, 4\}$. However, for the second DFA ($M(R_2)$), only the initial transitions for the stream prefix "$daba$" are performed using the DFA core D_2. Subsequent transitions for the stream suffix "aba" are not contained in D_2, and so need to be carried out by looking up the NFA state transitions for the NFA states corresponding to each DFA state. For instance, after processing the stream prefix "$daba$", $M(R_2)$ is in state $\{7, 8, 10\}$. Now, when the next symbol*

Algorithm 1 RE-MATCH$(I, \mathcal{N}, \mathcal{D}, \mathcal{C})$

input : I: input stream, $\mathcal{N}$: input set of NFAs, $\mathcal{D}$: set of DFA cores, $\mathcal{C}$: cache.

for $i = 1, \ldots, k$ **do** cur_st_i = Start state for DFA core D_i;
while input stream I is not empty **do**
  Dequeue symbol σ at the head of the stream;
  for $i = 1, \ldots, k$ **do**
    if transition $\delta(cur\_st_i, \sigma)$ is in $\mathcal{D}$ or $\mathcal{C}$ **then**
      $nxt\_st_i = \delta(cur\_st_i, \sigma)$;
    else
      Determine nxt_st_i by looking up transitions in $\mathcal{N}$ for NFA states in cur_st_i on symbol σ;
      if nxt_st_i is not in $\mathcal{C}$ **then**
        REPLACE$(\mathcal{C}, nxt_st_i)$;
      end if
      if cur_st_i and nxt_st_i are both in $\mathcal{C}$ **then**
        Add transition "$\delta(cur\_st_i, \sigma) = nxt\_st_i''$ to $\mathcal{C}$;
      end if
    end if
    $cur\_st_i = nxt\_st_i$;
  end for
end while

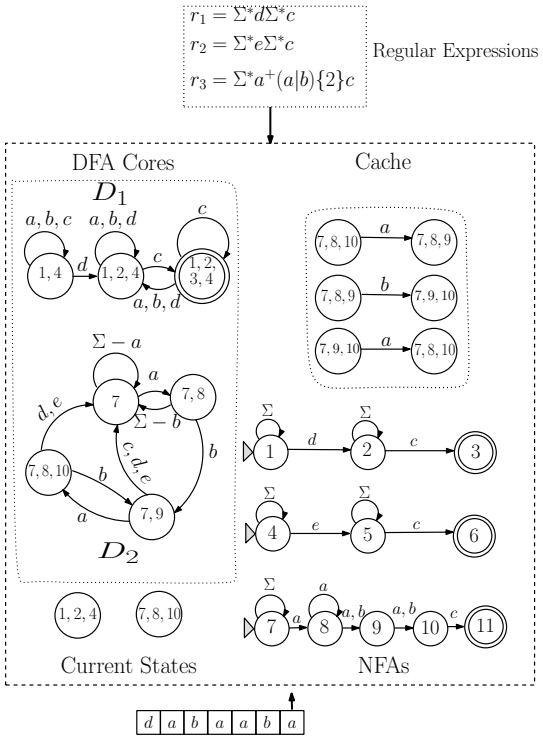

Figure 5: System model example.

a is encountered in the stream, the next DFA state $\{7, 8, 9\}$ *is determined from the NFA for* r_3 *by applying the transition function to NFA states 7, 8 and 10 for symbol a. The next 2 DFA transitions for symbols b and a from the DFA states* $\{7, 8, 9\}$ *and* $\{7, 9, 10\}$ *are resolved similarly, and finally, the current state is* $\{7, 8, 10\}$. *Observe that the last 3 transitions not present in* D_2 *are retained in the cache.*

3.5 Processing Cost Estimation

A number of steps like clustering REs and computing DFA cores require that we are able to estimate the processing cost per input symbol for a particular configuration of our system. For DFA cores D_i stored in memory for the RE clusters R_i, we formally derive this processing cost, and denote it by $cost(\{D_1, \ldots, D_k\})$. Suppose that p_i is the probability that a DFA transition from $M(R_i)$ belongs to D_i, and let n be the average number of NFA states per DFA state. Then, the average processing cost (in terms of the number of DFA or NFA state transition computations) is given by

$$cost(\{D_1, \ldots, D_k\}) = \sum_i (p_i + (1 - p_i) \cdot n) \qquad (1)$$

The first term in the above summation corresponds to the instances when the transition from $M(R_i)$ is in the DFA core (with probability p_i) and requires only a single lookup. The second term accounts for the case when the transition is not in the DFA core (with probability $1 - p_i$) and n NFA lookups are required (on average) to determine the transition.

Note that for simplicity, we do not consider the effects of the cache $\mathcal{C}$ when modeling processing costs; rather, we only focus on the DFA core sets. This, however, does not create a problem because the DFA cores will typically be much bigger than the cache. Of course, one way to also include the effects of the cache is to simply assign the complete memory α to DFA cores in our calculations, instead of only α_d.

To estimate costs using Equation (1), we still need to resolve the hit probability p_i for each DFA core D_i and the average number of NFA states n per DFA state. Computing accurate values for these parameters is clearly non-trivial because it may be impossible to construct the entire DFAs due to their exponential size. So instead of accurate values, we use DFA samples to compute estimates for these two quantities, as explained below.

Algorithm 2 constructs a DFA sample for a cluster R_i of REs. It considers a t-length sample of the input stream obtained by drawing successive symbols from Σ according to a probability distribution $\mathbb{P}$ over the alphabet Σ. It then lazily expands the DFA for R_i using the t-length stream sample as input to derive the DFA sample $\tilde{M}(R_i)$. With each transition in $\tilde{M}(R_i)$, we associate a weight which is the number of times the transition is traversed during the t-length sample.

The probability distribution $\mathbb{P}$ can be easily learnt from past segments of the input stream I. To handle correlations, we compute conditional probabilities for the occurrence of symbols from Σ in the input stream given a preceding subsequence of some fixed length. Thus, we compute $\mathbb{P}(\sigma/x)$ from past stream segments for each symbol $\sigma \in \Sigma$ and l-length sequence x. We then use this to pick the next symbol σ depending on the last l generated symbols. This Markov chain approach has been shown to model correlations well without incurring very high space overheads [11].

Now, consider a DFA core D_i which is a subset of the DFA sample $\tilde{M}(R_i)$. Further, let $w(D_i)$ be the total weight of D_i's transitions in the DFA sample. Then, we estimate the hit probability p_i for DFA core D_i as $w(D_i)/w(\tilde{M}(R_i)) = w(D_i)/t$. For n, we simply use the average number of NFA states per DFA state across the DFA samples $\tilde{M}(R_1), \ldots, \tilde{M}(R_k)$. In our experiments, we found that choosing t to be equal to $O(\alpha)$ (of the order of the memory size) yields fairly good estimates for p_i and n.

We can use the cost for a given configuration of DFA cores (as defined in Equation (1)) to compute $cost(\{R_1, \ldots, R_k\})$, the minimum processing cost for the set of RE clusters $R_1, \ldots, R_k$. This

Algorithm 2 DFA-SAMPLE($R_i, \mathbb{P}, t$)

input : R_i: RE cluster, $\mathbb{P}$: probability distribution over the alphabet, t: length of the sample.
output : DFA sample $\tilde{M}(R_i)$.

$cur\_st = \tilde{M}(R_i) =$ Start state of $N(R_i)$;
for $j = 1$ to t **do**
  Pick next symbol σ from Σ according to the distribution $\mathbb{P}$;
  Determine nxt_st by looking up transitions in $N(R_i)$ for NFA states corresponding to cur_st on symbol σ;
  if transition $\delta(cur\_st, \sigma) = nxt\_st$ is not in $\tilde{M}(R_i)$ **then**
    Add the transition to $\tilde{M}(R_i)$ and set its weight to 1;
  else
    Increment weight of the transition in $\tilde{M}(R_i)$ by 1;
  end if
  $cur\_st = nxt\_st$;
end for
return $\tilde{M}(R_i)$;

is essentially the minimum cost over all possible DFA core configurations $D_1, \ldots, D_k$ satisfying (1) D_i is a subset of $\tilde{M}(R_i)$, and (2) $\sum_i |D_i| \leq \alpha_d$. More formally, $cost(\{R_1, \ldots, R_k\}) = \min\{cost(\{D_1, \ldots, D_k\}) : D_i \subseteq \tilde{M}(R_i)$ and $\sum_i |D_i| \leq \alpha_d\}$. We use this processing cost estimate for REs to cluster them in Section 4, and propose techniques for computing the best DFA core configuration that leads to the lowest processing cost in Section 5.

4. RE CLUSTERING

As we saw earlier, combining all the REs into a single DFA can lead to high processing costs. This is because the explosion in the DFA state space results in an increased number of cache misses (low p_is) and expensive NFA lookups (large n) to compute the DFA transitions. Now, the other extreme of maintaining a separate DFA for each input RE alleviates this state explosion problem to a large extent, but requires m DFA state transitions to be computed for each input stream symbol. Our RE clustering algorithm looks to find a middle ground between these two extremes - it groups the input REs into a small number of clusters $R_1, \ldots, R_k$ such that the total size of the DFAs for the k clusters is much smaller compared to the DFA for R. This way, it is able to keep the processing costs low by balancing the number of DFA transition computations with the number of cache misses.

We need to be careful about how we group REs since assigning the wrong REs to the same cluster can lead to large DFA sizes and low hit probabilities p_i. For instance, in Example 1, we saw that with REs r_1 and r_2 in one group, and RE r_3 in a separate group, the DFAs have 16 total states as shown in Figures 3 and 4. However, if we assign REs r_2 and r_3 to the same group with r_1 by itself in a separate group, then the DFAs for the two groups would have a total of 21 states. This is because the DFA for the group $\{r_2, r_3\}$ has 18 states (due to replication of the DFA for r_3 twice, once when the symbol e has occurred in the stream and the other when it hasn't).

Our goal is to group the REs into k clusters $R_1, \ldots, R_k$ such that the average processing cost per input symbol is minimum. Algorithm 3 shows our basic clustering technique. RE-CLUST follows the agglomerative style of clustering, merging a pair of clusters in each iteration of the while loop until there is only one remaining cluster. In each iteration, it merges the cluster pair that results in the smallest processing cost. The grouping with the smallest processing cost among all the considered groupings is finally returned.

Algorithm 3 RE-CLUST(R)

input : R: input set of REs.
output : Set of clusters with low processing cost.

$min\_clust = cur\_clust = \{\{r\} : r \in R\}$;
$min\_cost = cost(min\_clust)$;
while cur_clust contains more than one cluster **do**
  $min\_cost' = \infty$;
  for each cluster pair R_1, R_2 in cur_clust **do**
    $cur\_clust' = cur\_clust - \{R_1, R_2\} \cup \{R_1 \cup R_2\}$;
    /* Merge clusters R_1, R_2 */
    if $cost(cur\_clust') < min\_cost'$ **then**
      $min\_clust' = cur\_clust'$;
      $min\_cost' = cost(cur\_clust')$;
    end if
  end for
  if $min\_cost' < min\_cost$ **then**
    $min\_clust = min\_clust'$;
    $min\_cost = min\_cost'$;
  end if
  $cur\_clust = min\_clust'$;
end while
return min_clust;

5. FINDING DFA CORES

We now turn our attention to computing $cost(\{R_1, \ldots, R_k\})$, the minimum processing cost for the RE clusters $R_1, \ldots, R_k$. Our goal is to compute frequently visited DFA cores for the k clusters such that the processing cost is minimum. Thus, we are interested in computing DFA cores $D_1 \subseteq \tilde{M}(R_1), \ldots, D_k \subseteq \tilde{M}(R_k)$ such that $\sum_i |D_i| \leq \alpha_d$ and $cost(\{D_1, \ldots, D_k\})$ is minimum.

5.1 Problem Statement

Each DFA sample $\tilde{M}(R_i)$ can be viewed as a weighted directed graph $G_i = (V_i, E_i, w)$ where V_i and E_i are the set of vertices and edges respectively, and w is a weight function on the edges of G_i. Vertices in V_i correspond to states in $\tilde{M}(R_i)$. Further, there is a directed edge (u, v) in E_i from vertex u to v if and only if there is a corresponding transition in $\tilde{M}(R_i)$ between the states s_u and s_v for the two vertices. And finally, each edge $(u, v) \in E_i$ has an associated weight $w(u, v)$ which is equal to the sum of the weights of all the transitions from s_u to s_v in $\tilde{M}(R_i)$.

A DFA core D_i is thus a subgraph of G_i. It is completely characterized by its vertex set $U_i \subseteq V_i$, and includes all the transitions between the states corresponding to its vertices. This is because, as we mentioned earlier in Section 3, with each DFA state in the core, we need to store all the NFA states for it. Thus, on an average, the storage overhead per DFA state is n, which can be large. In contrast, transitions consume very little space since each transition can be completely specified using a pair of (integer) state identifiers. Further, a state can have at most $|\Sigma|$ outgoing transitions (for the ASCII character set, $|\Sigma| = 256$). Thus, the size of the DFA core D_i is dominated by the number of vertices in U_i, and $|D_i| \approx |U_i|$.

From Equation (1), it follows that $cost(\{D_i\})$ is minimized when the quantity $\sum_i p_i$ is maximum (because $n \geq 1$). Further, if $w(U_i)$ denotes the weight of the subgraph induced by U_i (that is, the sum of the weights of edges between vertices in U_i), then $\sum_i p_i = \sum_i w(U_i)/t$. Thus, finding DFA cores D_i with minimum cost is equivalent to finding vertex sets $U_i \subseteq V_i$ such that the sum of the weights of the induced components $\sum_i w(U_i)$ is maximum.

Now, to maximize $w(U_i)$, we place two additional requirements

on U_i. First, we require that each subgraph induced by U_i contains the start vertex s_i (corresponding to the start state in $\tilde{M}(R_i)$). The justification for this is that for most of the real-world REs that we used in our experiments, we found that the start state is visited very frequently, and thus transitions leaving/entering the start state have high frequency. Moreover, note that the induced subgraph for U_i includes all the transitions between vertices in U_i. Thus, to maximize the weight $w(U_i)$ of this induced subgraph, we need to select vertices that are densely connected. This is the rationale for our second requirement that the subgraph induced by U_i is strongly connected[1].

Thus, our problem of computing DFA cores can be formally stated as follows.

DFA Core Computation Problem: *Given directed weighted graphs $G_i = (V_i, E_i, w)$, start vertices $s_i \in V_i$, and memory restriction α_d, find vertex sets $U_i \subseteq V_i$ containing s_i such that (1) $\sum_i |U_i| \leq \alpha_d$, (2) the subgraph induced by U_i is strongly connected, and (3) $\sum_i w(U_i)$ is maximum.*

5.2 Hardness Result

The following theorem indicates the intractability of computing DFA cores.

THEOREM 1. *The DFA Core Computation Problem is NP-hard.*

The proof is based on a reduction from the *Directed Steiner Tree* (DST) problem which is known to be NP-hard [3] and is stated as follows: Given a directed graph $G = (V, E)$, a root vertex $s \in V$, and a set of vertices $U \subseteq V$, find the minimum subgraph of G that connects s to all the vertices in U. In the reduction, we first derive the graph $G' = (V', E', w)$ from G where $V' = V$, $E' = E \cup \{(u, s) : u \in U\}$ and w assigns a weight of 1 to all edges $(u, s), u \in U$ and 0 to the remaining edges. We then show that the DST problem is equivalent to the problem of finding the minimum strongly connected subgraph containing s and having weight $|U|$ in graph G', which is equivalent to our DFA Core Computation Problem.

5.3 Heuristic for Single RE Cluster

In light of the NP-Hardness result of the previous subsection, we resort to a heuristic solution to compute good DFA cores with large weights. Initially, we focus on computing a single DFA core for the directed graph $G = (V, E, w)$ for a single RE cluster. We then build upon our single-core solution to compute k DFA cores for multiple RE clusters in the next subsection.

We are looking for a vertex set $\hat{V}$ that contains the start state s and that induces a strongly connected subgraph with large weight. Our strategy to identify such vertices is to consider maximum-weight walks of varying lengths j starting and ending at the start state s. The intuition here is that any heavy strongly connected subgraph of G will contain such high weight walks, and so considering these walks will help to identify the vertices in the subgraph. Of the various walks (with different lengths), we choose the walk with the maximum density. Here, we define the density of a walk of length j containing vertices W_j as $w(W_j)/|W_j|$. Thus, the walk with the highest density contributes the maximum weight per vertex when added to the subgraph. We contract the maximum density walk into the start vertex s and repeat the above step of enumerating walks to find the next maximum density walk in the modified graph. We continue this until we have identified and collapsed enough vertices to reach the space limit of α_d.

[1]A *strongly connected component* of a directed graph is a subgraph such that for every node pair $\{u, v\}$ there is a directed path from u to v and *vice versa*.

Algorithm 4 DFACORESINGLE(G, s, α_d)

input : $G = (V, E, w)$: directed weighted graph, s: start vertex, α_d: max number of vertices.
output : DFA core with large weight and size $\leq \alpha_d$.

$l = 0$; $S_0 = \{s\}$; $W = \emptyset$;
repeat
  $l = l + 1$;
  $S_l := S_{l-1} \cup W$;
  Contract vertices in W into the start vertex s in G;
  $W =$ MAXDENSITYWALK(G, s);
until $W = \emptyset$ or $|S_l \cup W| > \alpha_d$;
return S_l;

Algorithm 4 depicts our heuristic for computing a desirable DFA core (with large weight) that fits in memory. It repeatedly invokes Algorithm 5 to compute successive walks W with high density and adds them to the DFA core. It contracts the vertices in W into s by deleting the vertices from G and replacing all edges (u, v), $u \in W$ ($v \in W$) in G by (s, v) ((u, s)). The algorithm terminates once the core reaches the memory limit of α_d or Algorithm 5 returns $\emptyset$ implying that there are no more cycles in G and so G is no longer strongly connected. Note that the final core returned by Algorithm 4 is strongly connected. This is because every walk W starts and ends at s and so the set of vertices in W is strongly connected. Further, all the walks contain the common vertex s. Thus, since the union of overlapping strongly connected components is also strongly connected, we get that the computed core is strongly connected.

Algorithm 5 MAXDENSITYWALK(G, s)

input : $G = (V, E, w)$: directed weighted graph, s: start vertex.
output : A maximum density walk starting and ending at s.

$F_0(s) = 0$; $W_0(s) = \{s\}$;
for all $v \in V - \{s\}$ **do** $F_0(v) = -\infty$; $W_0(v) = \{v\}$;
for $j = 1$ to $2 \cdot |V|$ **do**
  for all $v \in V$ **do**
    Let u be the vertex such that $(u, v) \in E$ and $F_{j-1}(u) + w(u, v)$ is maximum;
    $F_j(v) = F_{j-1}(u) + w(u, v)$;
    $W_j(v) = W_{j-1}(u) \cup \{v\}$;
  end for
end for
if $\exists j$ such that $F_j(s) \geq 0$ **then**
  $r = \arg\max_{j:F_j(s)\geq 0} \frac{w(W_j(s))}{|W_j(s)|}$;
  return $W_r(s)$;
else
  return $\emptyset$;
end if

Algorithm 5 computes a high density walk starting and ending at start vertex s. It first enumerates maximum weight walks of length $j = 1, \ldots, 2 \cdot |V|$. We consider walks of length up to $2 \cdot |V|$ since this allows our walks to cover every edge (u, v) in G (at most $|V|$ hops from s to u and another $|V|$ hops from v to s). Variable $F_j(v)$ keeps track of the maximum weight of a j-length walk from s to v, and is computed using the following recursion:

$$F_j(v) = \max_{(u,v)\in E} \{F_{j-1}(u) + w(u, v)\} \quad (2)$$

Further, variable $W_j(v)$ tracks the vertices that lie on the maximum weight walk. Note that the value of $F_j(v)$ will be less than 0 if and only if there is no walk of length j from s to v. Thus, walks of length j starting and ending at the start vertex s will have $F_j(s) \geq 0$, and our algorithm returns the walk with the maximum density $w(W_j(s))/|W_j(s)|$ from among these.

Time complexity. Algorithm 5 has a time complexity of $O(|V|^2)$ (assuming that each vertex has a constant number of outgoing edges). Algorithm 4 invokes Algorithm 5 at most $|V|$ times, and so the worst-case time complexity of our DFA core computation algorithm is $O(|V|^3)$.

5.4 Heuristic for Multiple RE Clusters

We next turn our attention to computing k DFA cores for the graphs G_i corresponding to the DFA samples $\tilde{M}(R_i)$ for the k RE clusters. Thus, we are looking for vertex sets $\tilde{V}_i \subseteq V_i$ that induce strongly connected graphs with large weights while fitting within the memory limit of α_d.

Our solution for multiple cores builds upon the single-core solution of the previous section. Algorithm 4 computes a sequence $S_1, S_2, \ldots$ of strongly connected DFA cores with large weights and of increasing size for a single graph G. Let $S_1^i, S_2^i, \ldots, S_{j_i}^i$ denote this DFA core sequence when Algorithm 4 is invoked with graph G_i as an input parameter. Also, let (n_l^i, w_l^i) denote the size, weight pair $(|S_l^i|, w(S_l^i))$ for $l = 1, 2, \ldots, j_i$. Then our multiple core computation problem is to select a single DFA core $S_{l_i}^i$ for each i, $1 \leq l_i \leq j_i$, such that (1) $\sum_i n_{l_i}^i \leq \alpha_d$, and (2) $\sum_i w_{l_i}^i$ is maximum. The k DFA cores $S_{l_1}^1, \ldots, S_{l_k}^k$ then constitute our desired cores with large weight and satisfying the space constraints.

We can compute the optimal DFA cores $S_{l_i}^i$ using dynamic programming. Consider i DFA cores for graphs $G_1, \ldots, G_i$ and a weight w. Let $S_{l'_1}^1, \ldots, S_{l'_i}^i$ be the i DFA cores with the minimum number of vertices and total weight $\geq w$. We will use $S_i(w) = \cup_i S_{l'_i}^i$ to denote the set of vertices in these cores, and $N_i(w) = |S_i(w)|$ to denote the number of vertices in the cores. Then for an arbitrary w, $N_1(w)$ and $S_1(w)$ are equal to n_r^1 and S_r^1, respectively, where r is the smallest index value such that $w_r^1 \geq w$. For $i > 1$, $N_i(w)$ can be computed using the following simple recursion.

$$N_i(w) = \min_l \left\{ N_{i-1}(w - w_l^i) + n_l^i \right\} \tag{3}$$

Further, if $1 \leq r \leq j_{i-1}$ is the value for index l for the which the RHS is minimum, (that is, $N_i(w) = N_{i-1}(w - w_r^i) + n_r^i$), then $S_i(w) = S_{i-1}(w - w_r^i) \cup S_r^i$. The optimal set of DFA cores with the maximum weight for graphs $G_1, \ldots, G_k$ is then $S_k(w)$ where w is the maximum weight for which $N_k(w) \leq \alpha_d$ (so that the space constraints are not violated).

Now suppose that W_{max} denotes the maximum weight of any DFA core S_l^i computed by Algorithm 4. Note that $n_l^i \leq \alpha_d$ because Algorithm 4 does not generate DFA cores with more than α_d states. Thus, we get that for the optimal solution OPT, $W_{max} \leq w(OPT) \leq k \cdot W_{max}$. Thus, we need to compute $N_i(w)$ and $S_i(w)$ values for $1 \leq i \leq k$ and $1 \leq w \leq k \cdot W_{max}$. The total running time of our algorithm is therefore $O(k^2 \cdot W_{max} \cdot j_{max})$, where $j_{max} = \max_i\{j_i\}$ is the maximum number of input DFA cores S_l^i for a graph G_i. Due to the dependence on W_{max}, our algorithm has pseudo-polynomial time complexity. We show how to convert this pseudo-polynomial time algorithm into a *Fully-Polynomial Time Approximation Scheme*[2] (FPTAS) below.

[2]A *Fully Polynomial-Time Approximation Scheme* (FPTAS) is an approximation algorithm which (1) for a given $\epsilon > 0$, returns a solution whose cost is within $(1 \pm \epsilon)$ of the optimal cost, and (2) has a running time that is polynomial in the input size and $1/\epsilon$.

Algorithm 6 DFACORESMULTIPLE($\{G_i\}, \{s_i\}, \alpha_d, \epsilon$)

input: G_i: directed weighted graph for RE cluster R_i, s_i: start vertex in G_i, α_d: memory limit, ϵ: accuracy parameter.
output: k DFA cores with large weight and size $\leq \alpha_d$.

for $i = 1, \ldots, k$ **do** Invoke DFACORESINGLE(G_i, s_i, α_d);
Let S_l^i denote the S_l values, $l = 1, 2, \ldots, j_i$ generated during the execution of DFACORESINGLE(G_i, s_i, α_d);
Let (n_l^i, w_l^i) denote the size, weight pair $(|S_l^i|, w(S_l^i))$ for $l = 1, 2, \ldots, j_i$;
Let $W_{max} = \max_i\{w_{j_i}^i\}$;
Let $\delta := \frac{\epsilon \cdot W_{max}}{k}$;
Let $w_{max} = \lfloor \frac{W_{max}}{\delta} \rfloor$;
Set $w_l^i = \lfloor \frac{w_l^i}{\delta} \rfloor$ for all weights w_l^i;
for w=1 to $k \cdot w_{max}$ **do**
  Let (n_r^1, w_r^1) be the size, weight pair with the minimum value of $w_r^1 \geq w$;
  $N_1(w) = n_r^1$; $S_1(w) = S_r^1$;
end for
for $i = 2$ to k **do**
  for $w = 1$ to $k \cdot w_{max}$ **do**
    Let $r = \arg\min_l\{N_{i-1}(w - w_l^i) + n_l^i\}$;
    $N_i(w) = N_{i-1}(w - w_r^i) + n_r^i$;
    $S_i(w) = S_{i-1}(w - w_r^i) \cup S_r^i$;
  end for
end for
Let w be the maximum weight such that $N_k(w) \leq \alpha$;
return $S_k(w)$;

Algorithm 6 describes our approximation scheme for computing a set of DFA cores with size at most α_d and weight at least $(1 - \epsilon) \cdot w(OPT)$ for a given $\epsilon > 0$. It essentially runs the dynamic programming algorithm described above to find the DFA cores with the maximum weight, but it uses smaller weight values $\lfloor \frac{w_l^i}{\delta} \rfloor$ for each DFA core S_l^i and a smaller max weight value $w_{max} = \lfloor \frac{W_{max}}{\delta} \rfloor$, where $\delta = \frac{\epsilon W_{max}}{k}$.

We now show that the solution returned by Algorithm 6 has weight which is at least $(1 - \epsilon)$ times the optimal. Let the optimum solution OPT consist of DFA cores $S_{l_i}^i$ and let our solution $\hat{OPT}$ contain DFA cores $S_{l'_i}^i$. Then we can write,

$$\begin{aligned} w(\hat{OPT}) &\geq \sum_{i=1}^{k} \delta \lfloor \frac{w_{l'_i}^i}{\delta} \rfloor \\ &\geq \sum_{i=1}^{k} \delta \lfloor \frac{w_{l_i}^i}{\delta} \rfloor \\ &\geq \sum_{i=1}^{k} (w_{l_i}^i - \delta) \\ &\geq w(OPT) - k \cdot \delta \end{aligned}$$

Rewriting the RHS based on the fact that $k \cdot \delta = \epsilon \cdot W_{max}$ and $w(OPT) \geq W_{max}$, we get $w(\hat{OPT}) \geq (1 - \epsilon) \cdot w(OPT)$. The running time of our approximation algorithm is $O(k^2 \cdot w_{max} \cdot j_{max}) = O(\frac{1}{\epsilon} \cdot k^3 \cdot j_{max})$.

6. DYNAMIC CACHING ALGORITHM

In addition to the DFA cores which are static, our RE matching system also maintains a dynamic cache $\mathcal{C}$ to store additional transitions that are accessed frequently, but not captured in the DFA cores. A key benefit here is that the cached transitions can be dynamically adapted to the latest stream data distribution. However, adding/deleting transitions from the cache can be expensive, and so we would like to avoid frequent updates to the cache. In this section, we present a novel scheme for caching DFA states and (a subset of the) transitions between them so that cache hit rates are maximized while keeping the number of cache modifications low.

In our experiments with real-world REs (e.g., Snort rules), we found that the DFA start state and states in its neighborhood typically have a large number of incoming transitions with high frequency. Further, the most frequent and densely connected regions of the DFA are generally centered around the start state. Consequently, our caching logic gives preference to DFA states that are closer to the start state. This way, it is able to cache more transitions with high frequencies and thus achieve higher cache hit rates. This also leads to fewer cache evictions since proximity to the start state is a relatively stable property compared to the most recent access time or the access frequency. Of course, it is possible that some states that are close to the start state are visited relatively infrequently. To make sure that such states are not retained in the cache indefinitely, we evict states from the cache if they have not been accessed in the past τ seconds. Here, a reasonable value for the eviction time threshold parameter τ is the average time interval between two successive visits to the α^{th} most frequently visited state in the DFA samples.

To implement our caching policy, we associate two attributes $\mathcal{C}[s].dist$ and $\mathcal{C}[s].time$ with each DFA state s in the cache $\mathcal{C}$. The variable $\mathcal{C}[s].dist$ attempts to capture the distance (in hops) of the state s from the start state of the DFA while $\mathcal{C}[s].time$ captures the time when the state s was last visited. Intuitively, our caching algorithm prefers to cache states s with smaller values of $\mathcal{C}[s].dist$ since these correspond to densely connected states that are closer to the start state. On the other hand, it evicts states with large values of $cur\_time() - \mathcal{C}[s].time$ (greater than τ) since these are essentially states that have not been visited in a long time.

Now, since our RE matching system does not expand the complete DFAs for RE clusters, we need to compute distances for new states on the fly as the states are generated. For states s contained in the DFA core $\mathcal{D}$, we can statically compute the distances a priori and store it as the attribute $\mathcal{D}[s].dist$ for DFA state s in $\mathcal{D}$. However, for DFA state s in cache $\mathcal{C}$, $\mathcal{C}[s].dist$ will need to be calculated at run-time when s is generated, and further will need to be updated if a shorter path from the start state to s is found. The following code fragment describes the computation of $\mathcal{C}[s].dist$, and is executed at the end of the **for** loop in Algorithm 1 (in Section 3.4) right after the statement $cur\_st_i = nxt\_st_i$ that computes the new active state. In the code, variable cur_dist_i keeps track of the distance for cur_st_i, and is initially set to 0.

$cur\_dist_i = cur\_dist_i + 1$;
if cur_st_i is in $\mathcal{D}$ **then**
    $cur\_dist_i = \min\{cur\_dist_i, \mathcal{D}[cur\_st_i].dist\}$;
endif
if cur_st_i is in $\mathcal{C}$ **then**
    $cur\_dist_i = \min\{cur\_dist_i, \mathcal{C}[cur\_st_i].dist\}$;
    $\mathcal{C}[cur\_st_i].dist = cur\_dist_i$;
    $\mathcal{C}[cur\_st_i].time = cur\_time()$;
endif

Algorithm 7 REPLACE($\mathcal{C}, nxt\_st_i, cur\_dist_i + 1$)

input: $\mathcal{C}$: cache, nxt_st_i: next state to be added to $\mathcal{C}$, $cur\_dist_i + 1$: distance of nxt_st_i from start state.

if there is room in $\mathcal{C}$ **then**
    Add nxt_st_i to $\mathcal{C}$;
else if $\mathcal{C}$ contains a state s' (with the smallest $\mathcal{C}[s'].time$) such that $cur\_time() - \mathcal{C}[s'].time > \tau$ **then**
    Remove s' and all transitions involving s' from $\mathcal{C}$;
    Add nxt_st_i to $\mathcal{C}$;
else if $\mathcal{C}$ contains a state s' (with the highest $\mathcal{C}[s'].dist$) such that $\mathcal{C}[s'].dist > cur\_dist_i + 1$ **then**
    Remove s' and all transitions involving s' from $\mathcal{C}$;
    Add nxt_st_i to $\mathcal{C}$;
end if
if nxt_st_i is added to $\mathcal{C}$ **then**
    $\mathcal{C}[nxt\_st_i].dist = cur\_dist_i + 1$;
    $\mathcal{C}[nxt\_st_i].time = cur\_time()$;
end if

Thus, the distance for the "new" active state is the minimum of the distance of the "old" active state plus 1 and the distance for the new state stored in $\mathcal{D}$ or $\mathcal{C}$. Further, the distance for the new state in the cache $\mathcal{C}$ is updated if the new distance is shorter, implying that a shorter path to the state has been found. Observe that $\mathcal{C}[s].dist$ stored in the cache may not be the exact distance of the state s from the start state, but rather our best estimate of this distance.

Algorithm 7 describes our cache replacement procedure which is invoked (by Algorithm 1) when the next active state nxt_st_i at distance $cur\_dist_i + 1$ is not contained in the cache $\mathcal{C}$. If there is no room in the cache, the procedure first checks to see if the state s' in the cache with the smallest timestamp is more than τ seconds old - if so, it replaces s' with nxt_st_i in the cache. If all states are fresh, then it checks to see if the state s' with the maximum distance in the cache has a distance greater than the distance for nxt_st_i (which as we saw earlier, is at least $cur\_dist_i + 1$). If this is the case, then it replaces s' with nxt_st_i in the cache. Thus, the REPLACE procedure prefers to cache states that are closer to the start state in order to ensure dense connectivity among states in the cache and fewer cache misses. Note that it is possible for Algorithm 7 to not add nxt_st_i to $\mathcal{C}$ if all the states in $\mathcal{C}$ have been recently accessed and are closer to the start state than nxt_st_i.

Our cache invariant requires that we only cache transitions between states that are already present in $\mathcal{C}$. To enforce this invariant, we keep track of all the cached transitions into and out of each state – this way we can delete the transitions when the state is evicted from the cache. Also, Algorithm 7 requires that we are able to quickly (in constant time) determine the state in the cache with the smallest timestamp or the largest distance. To find the state with the smallest timestamp, we use a list to order states according to their timestamp values. Each time a state is visited, it is moved to the head of the list; thus, the tail of the list yields the state with the smallest timestamp value in constant time. Similarly, we maintain multiple lists of states in the cache for the different distance values, and then use the list for the highest distance value to retrieve the state with the largest distance.

7. EXPERIMENTAL STUDY

In this section, we quantitatively assess the performance of our RE matching system. To gauge the efficacy of our approach, we compare its performance with the Bro[18] Intrusion Detection System. The Bro system utilizes the lazy DFA technique along with

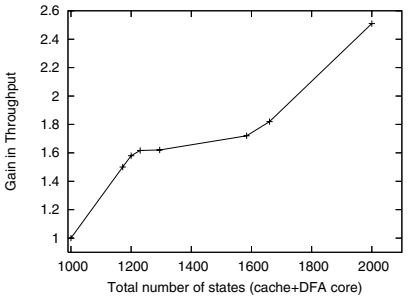

Figure 6: Throughput improvement over Bro NIDS.

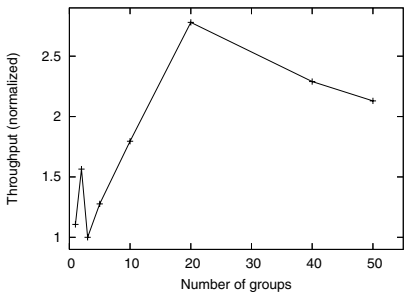

Figure 7: Sensitivity of throughput to number of groups ($\alpha_d = 1000$).

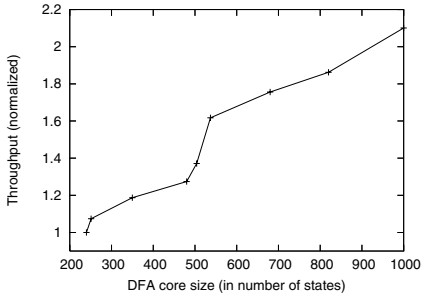

Figure 8: Effect of DFA core size on throughput.

an LRU queue to limit memory consumption. Our experimental results indicate that our approach offers a significant performance improvement (by factors ranging from 1.5 to 2.5) over Bro.

7.1 Testbed and Methodology

Data Sets.

In our experiments, we used the real-world TCP network packet trace obtained from the MIT Lincoln Laboratory [9]. This data set contains the TCP payload captured during a particular week[3]. As for Regular expressions, we used a subset of around 250 Snort [10] rules in which some of the rules had wild-cards with length restriction up to 512, and the remainder of the rules had unrestricted length wild-card terms.

Implementation details..

In the RE matching schemes, we implemented NFAs without ϵ-transitions. We observed that this leads to DFAs with states containing 100-150 NFA states. We had our own implementation of the Bro system to ensure a fair comparison between our matching scheme and Bro. Specifically, we do not penalize Bro for any additional processing it might be doing on top of regexp matching. To implement Bro, we combined all the REs into a single group and lazily built the DFA states at runtime. To bound the number of states in the DFA, we used an LRU cache whose size was varied in the experiments. In the implementation of our RE matching system, we collected DFA samples of size $t = 30\alpha_d$ where α_d is the memory assigned to the DFA cores[4]. To compute the probability distribution of stream symbols, we modeled it as a second order Markov process, i.e. we computed $\mathbb{P}(\sigma/x)$ from past stream segments for each stream symbol σ and for every possible sequence x of length 2. We set the accuracy parameter ϵ in the DFACORE-MULTIPLE algorithm to be $\frac{1}{2}$. Unless stated otherwise, the cache size was fixed at $\alpha_c = 1000$ states. All our experiments were carried out on a dual core, 2.2 GHz Pentium 4 machine with 1GB of RAM and running Ubuntu Linux v 7.04. We used the number of bytes processed per second as our evaluation metric. Throughout the experiments, the overall memory footprint was 500MB.

7.2 Results

7.2.1 Comparison with Bro

We now compare the performance of our system with the Bro NIDS. In this set of experiments, we varied the number of states in the DFA cores from 150 to 1000 states and measured the gain in throughput achieved over Bro by assigning an equal amount of memory (size of DFA core+Cache size) to it. Figure 6 shows the result of the experiment. In this experiment, the observed performance gain varied from 1.5 to 2.5. For small values of the DFA core size (with 200 states), the core hit rate was only 78% which finally escalated to 94% for larger core sizes (with 1000 states). The cache management overhead was observed to be almost 10 times more than the DFA cores. The small core hit rate and high overhead of cache maintenance accounted for the minor improvement of our system over Bro for small size of the DFA cores. However, as more states were added to the DFA cores, the small processing burden of static cores combined with the increased core hit rate resulted in a performance gain as high as 250%.

For 95% of the simulation time, the start state and the states in its immediate neighborhood were found to be active. This observation justifies our distance based caching strategy. The number of groups generated by our clustering algorithm varied widely (depending on the size of the DFA cores) from being as small as 2 for $\alpha_d = 600$ DFA states to a moderate value of 20 with $\alpha_d = 1000$ DFA states. The clustering algorithm was found to merge REs with common prefixes as well as other seemingly unrelated REs.

7.2.2 Effect of Grouping on Throughput

In our second set of experiments, we demonstrate the sensitivity of system throughput to the number of groups of REs. In this experiment, we fixed the DFA core size to 1000 states and evaluated the throughput of our system on some sampled steps of the clustering algorithm. The optimal grouping produced by RE-CLUST consisted of 20 groups of REs. However, for the purpose of this experiment, we continued the execution of RE-CLUST until it had merged all the REs into a single cluster. Figure 7 shows the result of this experiment. On the vertical axis, we plot the (normalized) throughput, and along the horizontal axis, we vary the number of groups. At an early stage of clustering, the throughput is low because of the large number of groups. The throughput monotonically increases as the number of clusters decreases, and finally with 20 groups, the throughput attains its maximum value. Beyond the optimal clustering step, the throughput keeps on decreasing and reaches its minimum at 3 clusters.

However, the next step of clustering results in an increase in throughput. This apparent anomaly can be explained as follows. Out of the three groups created by RE-CLUST in the previous step, two of them are similar with respect to the DFA samples. Therefore, in the next step, RE-CLUST merges these two clusters, and achieves a $\frac{3}{2}$ factor gain in performance. This experiment clearly

[3]This data set was first made available in May 1998.

[4]measured in number of states.

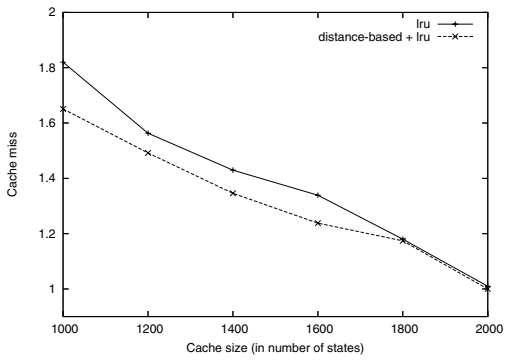

Figure 9: Comparison of caching strategies.

demonstrates that inappropriate clustering can slow down the performance of an Intrusion Detection System by more than 250%.

7.2.3 *Effect of DFA core sizes*

In the following set of experiments, we evaluate the performance of our system when the memory assigned to the DFA cores is varied. For this experiment, we first computed the optimal grouping of the regular expressions by setting α_d to be 1000. As can be seen from Figure 7, it produces 20 groups of regular expressions. Then we varied the amount of memory assigned to the static components, extracted the DFA cores that fit in the available space, and measured the throughput of the system. Figure 8 shows the result of this experiment. As a result of the increase in core sizes from 200 to 1000, the hit rate of the DFA cores grows from 75% to 87%. As can be observed from the plot, the improvement in hit rate is accompanied by a proportional increase in throughput.

7.2.4 *Effect of Caching*

In the fourth and final set of experiments, we empirically evaluate our caching strategy. In this experiment, we fixed the DFA core size at 600 states and varied the size of the cache from 1000 to 2000 states. For a fixed size of the core and variable cache size, we measured the number of cache misses. We repeated the same set of experiments, replacing our caching strategy by LRU. Figure 9 shows the result of this experiment. As can be observed from the plot, our distance-based caching strategy (combined with LRU) outperforms LRU by a factor of $\approx 10\%$. Simulation of the NFAs was observed to be at least twice as expensive as the cache management overhead. Therefore, a 10% difference in cache misses resulted in convincing savings in terms of processing overhead. However, beyond a certain cache size, the improvement was observed to be insubstantial. This is because with more states being added to the cache, the result is fewer cache evictions due to the concentration property of DFA states. Therefore, in the absence of cache eviction, both the strategies performed equally well.

8. CONCLUSION

In this paper, we presented the design of a high performance RE matching system that employs DFA state caching (both static and dynamic) and RE grouping. In order to combine the space efficiency of *Non-deterministic Finite Automata (NFA)* and processing efficiency of *Deterministic Finite Automata (DFA)*, we designed a novel agglomerative clustering technique which optimizes the trade-offs among overall DFA size and processing cost. To estimate the processing cost, our novel contribution is the concept of DFA samples, which are constructed using information on the probability distribution of symbols in the input stream. In order to reduce the processing burden, we devised efficient techniques for extracting high frequency and densely connected DFA portions to be stored in a static cache, and supplemented it with a dynamic caching strategy that maintains dense connectivity among DFA states. Our experimental results indicate that our system significantly outperforms a state-of-the-art Network Intrusion Detection System.

9. REFERENCES

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

[2] C. Chan, M. Garofalakis, et al. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003. ISSN 1066-8888.

[3] M. Charikar, C. Chekuri, et al. Approximation algorithms for directed steiner problems. In *SODA*. 1998.

[4] B. Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, pp. 118–132. Springer-Verlag, London, UK, 1979. ISBN 3-540-09510-1.

[5] Y. Diao, P. Fischer, et al. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*. 2002.

[6] Y. Diao, S. Rizvi, et al. Towards an internet-scale XML dissemination service. In *VLDB*, pp. 972–986. 2004.

[7] T. Green, A. Gupta, et al. Processing XML streams with deterministic automata and stream indexes. *ACM TODS*, 29, 2004.

[8] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company Inc.

[9] http://www.ll.mit.edu/IST/ideval/data/data_index.html. Lincoln laboratory, mit.

[10] http://www.snort.org. Snort network intrusion detection system.

[11] H. Jagadish, O. Kapitskaia, et al. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal*, 9(3):214–230, 2000.

[12] T. Johnson, S. Muthukrishnan, et al. Monitoring regular expressions on out-of-order streams. *ICDE*, 2007.

[13] S. Kumar, S. Dharmapurikar, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.*, 36(4):339–350, 2006. ISSN 0146-4833.

[14] J. Levandoski, E.Sommer, et al. Application layer packet classifier for linux. http://l7-filter.sourceforge.net/.

[15] A. Snoeren, K. Conley, et al. Mesh-based content routing using XML. In *SOSP*. 2001.

[16] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS*. 2003.

[17] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Tech. Rep. TR-94-17, 1994.

[18] www.bro ids.org. Bro network intrusion detection system.

[19] www.cisco.com. Cisco ios ips deployment guide.

[20] F. Yu, Z. Chen, et al. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, pp. 93–102. ACM Press, 2006. ISBN 1-59593-580-0.