REduce: Removing Redundancy from Regular Expression Matching in Network Security

Victor C. Valgenti and Min Sik Kim
Petabi, Inc.
Irvine, CA 92612
USA
Email: vvalgenti@petabi.com

Sung-II Oh
Infnis Networks, Inc.
Gangnam-gu, Seoul 135-824
Korea
Email: sungiloh@infnis.com

Inbok Lee
Korea Aerospace University
Goyang, Gyeonggi 412-791
Korea
Email: inboklee@kau.ac.kr

Abstract—Regular expressions have become a fixture in network security systems such as Network Intrusion Detection, Spam email filtering, and Antivirus. Unfortunately, regular expressions require considerably more resources in matching over fixed binary or character strings. Much research has focused on improving matching architectures or hardware support to create more efficient regular expression matching. This research, however, investigated whether or not the regular expression set itself contained any lever that might make for creating more efficient automata prior to moving such automata to any specific matching architecture or hardware. We found that typical Non-deterministic Finite Automata (NFA) construction methodologies create redundant paths in the NFA when used with the complex rule-sets employed in network security. This stems directly from the fact that creating optimized NFA is a hard problem. As such, we created REduce, a tool that uses shared prefixes among regular expressions as a heuristic to eliminate redundant paths among shared prefixes within constructed NFA. The end result is smaller matching automata (between 4–50% depending on the rule-set) and a 4–900% improvement in throughput due to reductions in active state. More importantly, REduce only targets NFA construction, thus the generated NFA can be converted to any specific matching architecture or hardware for cumulative improvement.

I. INTRODUCTION

Regular expressions perform a vital role in network security. The greater expressiveness of regular expressions over fixed strings allows network security administrators to create signatures that can ignore innocuous traffic while capturing attacking traffic. Unfortunately, regular expressions require significantly more resources during matching. Much research has sought to make regular expression matching more efficient through the use of more effective architectures [1]–[7], or improved hardware support [8]–[13], or both [14]. This work recognizes that the set of regular expressions itself is a strong determinant of the overall matching capabilities of any regular expression matcher. Armed with that understanding we sought to identify heuristic features within the regular expression sets that might be exploited for improved matching efficiency.

The inspiration for this research came from noting the large amount of redundancy in regular expression sets used in network security. For example, the two regular expressions listed below, culled from SpamAssassin [15], are nearly identical. Such redundancy is rampant throughout the rule-sets we have examined. This stems from the nature of attacks (i.e. targeting “HTTP”) or from rules targeting multiple forms of the same attack. Further, rule-sets, in practice, tend to grow in an unbounded fashion as new regular expressions are added without removing or refining old expressions. These factors lead to many regular expressions sharing common sub-patterns.

[^\]\]+ rDNS=\S*mail
[^\]\]+ rDNS=\S*mta

These redundancies are not trivially optimized away during the Non-deterministic Finite Automata (NFA) construction phase as minimizing NFA is a hard problem [16]. This difficulty stems from ambiguity in the regular expressions caused by alternation and repetition. The above two regular expressions are likely to create two redundant paths in a matching automata due to the repetition attached to the character class which serves as the beginning of the regular expressions (the ‘+’ symbol indicates 1 or many repetition). As these redundancies add up, the matching automata becomes bloated with many redundant paths. These redundant paths cause much more work during matching as they must be converted into either a large number of states for Deterministic Finite Automata (DFA) or require multiple active states during matching for NFA (must track all potential paths through the NFA).

In this work we introduce REduce, a method that uses common prefixes shared by regular expressions as a heuristic for removing redundant paths in NFA. REduce is comprised of two parts: a regular expression comparison engine and an NFA construction engine. The regular expression comparison engine allows for time-efficient comparison of regular expressions. The NFA construction engine uses common prefixes shared among regular expressions to partition and then piece-meal construct NFA without redundant paths. REduce creates NFA that are from 4% to 50% smaller (depending on the rule-set) than un-optimized NFA and with throughput from 4% to more than 900% faster. More importantly, REduce produces NFA that can be used by any architecture or hardware. Thus, REduce can be used independent of final matching architecture or hardware to achieve cumulative improvements. We demonstrate this with Hybrid Finite Automata (HFA) [5]. Ultimately, REduce creates heuristically compressed matching automata that can improve regular expression matching for Network Security at little cost in resources.
Fig. 1. NFA, DFA, and PCRE throughput vs total regular expressions.

II. BACKGROUND

Regular expressions are an important aspect of Network Intrusion Detection Systems (NIDS) like Snort [17] and Bro [18], as well as antivirus like ClamAV [19] and even email filtering like SpamAssassin [15]. Regular expressions provide a wide context within which to describe dynamic patterns, such as those occurring in polymorphic worms or customized attacks [20], or to better filter matches [17]. Unfortunately, regular expressions have greater complexity than string matching due to ambiguous characters, alternation, and repetition.

Common techniques for matching automata are to use Deterministic Finite Automata (DFA), Non-deterministic Finite Automata (NFA), or to match each regular expression ad-hoc using a library like Perl Compatible Regular Expressions (PCRE) [21] (which internally may use a DFA, an NFA, or neither depending on configuration). Further, it should be noted that even when DFA are used it is a common practice to first build an NFA, then convert the NFA to a DFA. Regardless, Fig. 1 illustrates some attributes common to regular expression matching techniques given the same set of rules and the same input all on a single core/thread. DFA maintain high throughput while the size of the DFA remains manageable. However, DFA grow exponentially as rules are added. In this case, the DFA starts small but is already 10GB when 200 rules have been added and exceeds available memory when attempting to add 400 rules (thus no data for DFA beyond 200 rules). NFA, on the other hand, grow linear in memory consumption as rules are added but do not maintain as high throughput as DFA since the NFA must maintain multiple active states that explore all possible traversals through the NFA given a particular input. Finally, PCRE are matched sequentially using the standard PCRE matching libraries. When matching only a single rule, PCRE performs admirably, however sequentially matching many rules is horribly inefficient in terms of time, though the amount of memory required is negligible.

Much research has sought to flatten the curves exhibited above to create a matching automata that will not degrade despite the number of rules and that can match at line rates. Hybrid Finite Automata [5] use small head DFA and tail NFA to gain most of the benefit of both types of automata without the significant downsides. XFA [4] and CD²FA [2], [3] add state to the automata to improve traversal and potentially shrink overall size while Ordered Binary Decision Diagrams [6] convert NFA to a different format for more efficient matching. Other research has sought to enhance matching through the use of specialized platforms such as: Field Programmable Gate Arrays (FPGA) [7], [8], Graphics Processing Units (GPU) [9], [11], Cell processors [10], Semiconductor architectures [13], or TCAM [12]. In our own prior research we bridged the gap between hardware and software by creating an architecture-friendly layout for the matching automata while exploiting parallelism in general purpose processors [14].

A problem that is not always addressed is that minimizing NFA is a hard problem [16]. There have been several papers exploring the impact on performance to regular expression matching in network security dependent on the regular expression set and the traffic involved [22]–[26]. The primary takeaway from this research is that the regular expression set itself, independent of and in addition to the matching architecture, can have significant impact on the efficiency of the matcher. Further, traffic that matches deeply with the regular expression set can cause excessive burden on the matching automata [22], [27], [28]. This can expose the matching automata to a potential Denial of Service attack that can be used to hide an attack or halt the system. Since the regular expression set can have such a significant impact on matching we wondered if levers existed within the regular expression sets that might enable more efficient matching.

Little research has examined the regular expression set for heuristic levers to improve matching. Yu et al. [1] proved that certain regular expressions could be re-written to reduce the number of states that would result from specific constructs without harming the semantics of the original regular expression. They managed to reduce the size of constructed DFA by modifying only those few rules with constructs that caused this large state expansion. Kosař et al. [29] demonstrated the use of NFA minimization schemes to reduce FPGA logic utilization for regular expressions mapped to FPGA, while Tie et al. [30] showed modest reduction in state in DFA by grouping regular expressions by similarity. However, we noted in our research that reducing the size of the matching automata is not necessarily of great benefit during matching. In fact, reduction in the number of states near the start of NFA provides for much greater improvements to matching efficiency than removing states distant from the start state.

A. Regular Expression Sets

We found that regular expression sets used in Network Security are complex and demonstrate a large degree of commonality. This stems from the nature of attacks to how well the set of regular expressions are maintained over time. As an example of how a regular expression set evolves over time we take a look at the default-enabled rules for the
Sourcefire Vulnerability Research Team (VRT) [31] Snort rule-set over the last 10 years. For the first four years Fig. 2 illustrates the wholesale addition of regular expressions, and rules, to the rule-set with none eliminated. In 2009 the size of the rule-set had become enough of a problem that efforts were made to simplify the set and many regular expressions were eliminated or refined. The following years represent the continual increase then culling of the rules in the rule-set. This example demonstrates the optimal process for maintaining a set of regular expressions for pattern matching. The set is periodically examined and revised; removing old or irrelevant rules and refining existing rules. Regardless, the quality of this revision process can greatly impact the quality of the rule-set potentially leaving behind numerous redundancies.

In this work, we examined six separate sets of regular expressions employed in Network Security and research. First, we converted the ClamAV [19] rules from January 8, 2015 to regular expressions. ClamAV uses a signature format similar to the PCRE format making the conversion trivial. Next, we utilized the regular expressions as used in the evaluation of Ordered Binary Decision Diagrams (OBDD) [6] (derived from Snort rules) for a reference to rules used in academia. Third, a telecommunications company that offers Network Intrusion Detection services to its clients provided us with a set of rules that are currently used in their products. We labeled this set as “Proprietary”. The fourth rule-set is a completely random rule-set generated synthetically to offer a counter-point to the other sets that have a high degree of commonality. Generation was done in a similar fashion to that described by Becchi et al. [23]. The fifth rule-set reflects the default-enabled rules from the Sourcefire Vulnerability Research Team [31] Snort rule-set for registered users as of January 8, 2015. The final rule-set reflects the regular expressions pulled from SpamAssassin [15] as of January 8, 2015.

**TABLE I** illustrates some basic statistics for each rule-set concerning the use of alternation, counting, ambiguity, and repetition as these statistics provide some understanding about the complexity of the regular expressions. More alternation, counting, ambiguity and repetition imply greater complexity. By this measure, we can see that the ClamAV rule-set has almost zero alternation, limited counting and ambiguity, and a small amount of repetition. More than 95% of all the regular expressions in that set are fixed binary strings. As such, ClamAV represents the simplest and least complex set of rules further illustrated in Fig. 3 which shows the expected depth of traversal into regular expressions given uniformly random input. Ambiguous characters and repetition early in a regular expression makes for a longer curve in this figure and ClamAV shows a near vertical line at the least possible expected depth of traversal indicating that most of the regular expressions have zero ambiguity. The other rule-sets all show varying degrees of complexity both in Fig. 3 and in **TABLE I** with Snort and SpamAssassin coming in as the most complex, and the Random, OBDD and proprietary data sets closely following. We should note that the proprietary data set, though quite complex, uses fixed strings at the beginning of each regular expression to mitigate some of that complexity and thus explains the steeper curve in Fig. 3.

These data illustrate the “personality” of the targeted domains. ClamAV targets mostly fixed binary patterns and thus shows nearly a vertical line in Fig. 3. SpamAssassin targets strings expected in emails which tend to be small and with lots of ambiguity in order to catch variants. The proprietary set is designed for high-speed environments and thus demonstrates
TABLE II
GROUPING OF REGULAR EXPRESSIONS BY COMMON PREFIX.

<table>
<thead>
<tr>
<th>RE Set</th>
<th>% RE in Groups</th>
<th>Avg Group Size</th>
<th>Max Group Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClamAV</td>
<td>29.1%</td>
<td>1.27</td>
<td>219</td>
</tr>
<tr>
<td>OBDD</td>
<td>80.42%</td>
<td>3.36</td>
<td>338</td>
</tr>
<tr>
<td>Proprietary</td>
<td>66.7%</td>
<td>2.23</td>
<td>375</td>
</tr>
<tr>
<td>Random</td>
<td>0.0%</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Snort</td>
<td>60.74%</td>
<td>1.91</td>
<td>217</td>
</tr>
<tr>
<td>SpamAssassin</td>
<td>38.07%</td>
<td>1.41</td>
<td>69</td>
</tr>
</tbody>
</table>

Fig. 4. Example regular expression \(\backslash x2F[a-z]+\backslash x2epng/i\) as NFA.

a steep curve in Fig. 3, but still has a significant amount of repetition, alternation, counting and ambiguity implying that the regular expressions are still quite complex. The Random rules demonstrate greater complexity as a bi-product of the randomly assigned repetition, counting, and alternation anywhere within a regular expression. Snort rules are quite complex as Snort uses regular expressions as a final filter for removing false positives. The OBDD set, though derived from Snort, avoids some of the complexity of the Snort rules as more than half of those rules were harvested from non-regular expression aspects of the Snort rules.

While there is considerable variance between the regular expression sets, we found that all of the sets, other than the Random set, demonstrated significant commonality within each set. For example, in the Proprietary set the pattern ‘http’ occurs more than 1,000 times and ‘host’ more than 900. In the Snort set, the patterns ‘file’ and ‘object’ both occur around 150 times each. In SpamAssassin the pattern ‘inter’ occurs over 120 times and ‘Microsoft’ over fifty times. As we dig deeper into the sets we can see that there is considerable similarity between many regular expressions. In fact, if we group all of the regular expressions by shared prefixes (see Section IV-A) we see many shared prefixes as is illustrated in TABLE II.

III. RE COMPARE: COMPARING REGULAR EXPRESSIONS

Regular expressions may have textual representation but they cannot be accurately compared using string comparisons. For example, the two regular expressions below, taken from the Proprietary rule-set (truncated), are duplicates. In one the underscore is represented by its ASCII binary code \x5F, and in the other as the underscore character.

\x\x00p\x00_\x00o\x00a\x00m\x00e
\x\x00p\x00\x5F\x00o\x00a\x00m\x00e

This illustrates how regular expressions may have different orthography for the same character sequences. Further, order doesn’t matter in alternation or character classes as illustrated by ‘a(b|d)e’ which is the same as ‘a(c|d)b’ and ‘a[b|c]d’ which is the same as ‘a[c|d]b’. String comparison of regular expressions is only valid for trivial sets of regular expressions. In order to accurately compare regular expressions it is necessary to compile those regular expressions using a regular expression parser like the PCRE library [21]. Parsing a regular expression using PCRE will result in a sequence of ‘opcodes’ describing the regular expression. While these opcodes adequately handle ambiguity in individual characters or character classes, the order of the opcodes is still defined by the orthography. It is necessary to devise a consistent representation for regular expressions in order to make side-by-side comparisons between two or more regular expressions.

We recognized that the opcodes produced by PCRE describe a flattened parse tree of the regular expression. All we need to do is to convert that parse tree into a consistent data model that we can use for consistent comparison. For example, note the following regular expression taken from the VRT Snort rule-set [31]: \(\backslash x2F[a-z]+\backslash x2epng/i\). Fig. 4 visualizes this regular expression as an NFA (note references to the start in case of a non-match are assumed).

Fig. 5 illustrates an abbreviated parse tree of the example regular expression. The concatenators within the parse tree have already defined the regular expression as a sequence of potential steps or transitions (i.e. characters). These concatenators are highlighted in Fig. 5. In other words, the concatenators represent single steps in the sequence of the regular expression.

Observation 1. If two regular expressions share a common subsequence, then they also share a common subtree within the parse tree.

The significance of Observation 1 is two-fold. First, given a regular expression it is possible to flatten the parse tree of that regular expression into a linear sequence of these transitions using a consistent ordering for character classes and alternation. Secondly, these transitions may be decorated with repetition (like the ‘+’ in the example indicating one or many times) to preserve these constructs. Thus, the above regular
expression could be represented as: \([/]/, [A-Z, a-z]+, [ ]\), \([P, p], [N,n], [G,g]\).

In the case of alternation, the process is more complex as illustrated in Fig. 6. In order to handle a regular expression like \(\text{‘ab(cde[fgh]ij’)}\) the data model must allow for recursive exploration of subtrees. Further, these groups of alternation may have repetition ascribed to them as in the above example. This is solved by ensuring that subtrees are consistently ordered (lexicographical ordering suffices). Alternate branches are always evaluated in this order. Further, groups and finite steps in a regular expression are abstracted out to a single, interchangeable object. The only difference is that groups are recursively explored, while finite steps need no exploration. The numbers in Fig. 6 show the order of exploration while ‘A’ and ‘B’ mark the order in which branches are sorted so that comparison can be consistent. Each box in Fig. 6 represents a single step further into the regular expression. Thus, if the regular expression \(\text{‘ab(cde[fgh]ij’)}\) were compared to \(\text{‘ab(cde[fgh]kl’)}\), we would see that these regular expressions match out to step 5 in Fig. 6. Note that matching is only valid if it occurs across all alternate branches for a given step.

This structure is termed a transition sequence with each step either a finite transition (i.e. a character) or a transition group (a nested regular expression) and with possible repetition or other modifiers decorating any step. Each finite transition step is represented as a bit-set eliminating ambiguities due to orthography. Each transition group points to an ordered set of one or more transition sequences that are likewise composed of finite transitions or groups. Given these constraints it becomes possible to compare any two regular expressions side-by-side, step-by-step, and arrive at the correct result. Comparison can extend into nested subtrees even if only part of the subtree matches. For example, \(\text{‘ab(cd[f]g)’}\) would match Fig. 6 out to a depth of four steps (a and b, and then the first two steps of each branch). This data-model has a worst-case time complexity determined by the algorithm used for sorting alternate branches. However, this worst-case can only occur when the number of alternate branches nears the size of the input. Typical alternation extends to 3 or 4 alternate branches, and even then alternation is in the minority of regular expressions as evidenced back in TABLE I. Thus, the typical time complexity is defined by the PCRE conversion (which is linear) and the transition sequence construction which, without any alternation, is also linear. Finally, the time complexity of comparing two transitions sequences depends on the comparison algorithm used.

A. Subsequences in Regular Expressions

This section will define what subsequences can be merged within a regular expression, assuming shared subsequence can be identified as per Section III, without impacting the intended semantics of the original regular expressions. First, assume that \(wuw \in L(RE_1)\) and \(xuvy \in L(RE_2)\). A naïve algorithm will make the union of \(RE_1 \cup RE_2\) as \((w | x)uw | srv | y\) which will accept the words \(wuv\) and \(xuv\) in addition to \(wuw\) and \(xuv\), creating a false positive since \(wuv \notin L(RE_1)\), \(xuv \notin L(RE_2)\), and \(wuv \notin L(RE_1)\), \(xuv \notin L(RE_2)\).

**Lemma 1.** If we merge regular expressions with a common prefix or a common suffix, the resulting language does not change.

**Proof.** We show an informal proof here. Assume that \(wuv \in L(RE_1)\) and \(wuv \in L(RE_2)\) where \(w, u,\) and \(v\) are words whose lengths are equal to or greater than one. If we combine these two regular expressions into one, then we can see that it will accept the compound words \(wu\) or \(wv\). Let \(\alpha\) be a Boolean variable whose value is TRUE if there is matching occurrence of \(w\) in the traffic and FALSE otherwise. For \(u\) and \(v\), we define two more variables \(\beta\) and \(\gamma\). Then, we can decompose the problem such that \(L(RE_1 \cup RE_2)\) accepts \((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)\). It follows that the language accepts \(\alpha \wedge (\beta \vee \gamma)\) by the distributive property. Similarly, if the two compound words were \(uwu\) and \(wuv\), then by the same logic, we get \((u | v)uw\).

In fact, this holds if the two words are in the pattern \(wux\) and \(wuv\), which would create \(w(u | v)x\).

B. Redundancy In Automata

When two or more regular expressions are added to the same automaton then, optimally, shared prefixes should likewise be shared in the generated NFA. This is trivially accomplished with simple regular expressions (regular expressions without repetition, counting, or alternation) or if there are only a few regular expressions. However, when the number and complexity of regular expressions grows, ensuring this minimal NFA becomes a hard problem requiring extra state to properly handle. This becomes costly both in terms of time and space. If no attempt to minimize states in the NFA is made, then numerous redundant paths may make for less efficient traversal. Thus, a time-efficient means of removing redundant paths in an NFA is desirable.

**Lemma 2.** If two or more regular expressions share a known prefix, then those regular expressions can be merged into a single NFA such that they all share the same prefix.

**Proof.** Given that \(wu \in L(RE_1)\) and \(uw \in L(RE_2)\) where \(w, u,\) and \(v\) are words whose lengths are equal to or greater than one, then it extends that \(w, u,\) and \(v\) all have automata representations. As such, it follows that an NFA can be created for each: \(NFA_w, NFA_u,\) and \(NFA_v\). If we modify the final state of \(NFA_w\) by merging it with the start states for \(NFA_v\)
and \(NFA_u\), then this will create a new NFA, \(NFA_{w/u}\), that contains exactly one path along the prefix \(w\).

A similar proof can be used for suffixes as well. Since RE-compare offers a time-efficient means of identifying prefixes, or suffixes, among two regular expressions it is possible to use this as a heuristic to remove redundant paths from the NFA. We have noted earlier that simply merging equivalent states in an NFA is not necessarily beneficial to practical use of said NFA. So we examined the impact of merging prefixes and suffixes, since by Lemma 1 merging suffixes and prefixes does not alter the semantics of the underlying regular expressions.

Merging suffixes has the advantage of removing more redundant states from NFA than merging prefixes. Suffixes will never be merged given a typical NFA construction algorithm thus all merged suffixes offer a direct reduction in the size of the NFA. However, suffixes are rarely encountered when matching against expected case traffic since non-attack traffic rarely intersects deeply enough with the rule-set to reach the regions within the NFA representing those suffixes. Merging suffixes reduces active state and the working set for only a small portion of any traversal of the NFA. On the other hand, though merging prefixes does not remove as many states from the NFA, since the only states merged are those not optimized by the normal NFA construction routines, each state merged close to the start of the NFA removes one redundant path deeper into the NFA. This has the immediate impact of reducing required active state during any traversal and reducing the effective size of the working set. As such, we have focused on prefixes in this work, though we should note that future work is examining the benefit of merging suffixes in addition to prefixes.

IV. REDUCE: REMOVING REDUNDANCY FROM NFA

The goal is to create an NFA that is optimized. If all prefixes are merged then, by Lemma 2, the NFA has no redundant paths among prefixes. This means that for any traversal only a single instance of a prefix must be traversed. We call this a prefix-minimal NFA.

REduce accomplishes this feat by adding regular expressions to the REduce NFA by longest common prefix. In other words, the regular expressions with the longest common prefix are merged into an NFA, then the next, and so on until all regular expressions have been merged into separate, longest-matching, NFA. These merged NFA are then iteratively merged together, also by longest prefix match, until a single NFA remains. The end result is the REduce NFA. At any point within the algorithm, the longest prefix is known and thus it is possible to ensure that regular expressions do not diverge until that point. The primary difficulty is performing this in a time-efficient manner as comparing each regular expression to all others at any given point can prove costly, even if the comparison itself can be done efficiently. We have adopted a multi-tiered partitioning approach to build the NFA in pieces in order to reduce the NFA construction time. This also has the effect of reducing the memory cost to no larger than that of the final NFA. The REduce NFA construction process follows these steps:

1) Grouping: Partition the set of regular expressions into groups based on the first few steps in the transition sequence. This reduces the number of regular expressions that must be compared to each other.

2) Subgrouping: Identify longest prefix matches and assign them to subgroups.

3) Merging: Build subgroup NFA, then merge all subgroup NFA into group NFA, then merge all group NFA into the final NFA.

A. Partitioning Regular Expressions by Minimum Prefix Length

Our first-tier grouping is based on this simple fact.

Observation 2. If two regular expressions have a common prefix of length \(k\), then they have also a common prefix of length \(j\) where \(0 < j < k\).

We exploit this property by setting \(j\) to a value that will describe our minimum shared length. Using this minimum shared length we create a key of the first \(j\) steps in the transition sequence for each regular expression and then push that regular expression into a map of regular expression groups. As regular expressions are added to the map, they will automatically be clustered to the proper group through the key. The key is not random and directly reflects the regular expression meaning that there are never incorrectly grouped regular expressions.

This scheme places all regular expressions into a group in a single pass. Due to the merging algorithm, as outlined in Section IV-C, the value for the minimum shared length has no impact on the resultant NFA though it will affect the size of the groups created. For this reason we chose a value of 4 for the minimum shared length which created the grouping as illustrated in TABLE II. Smaller values created fewer larger groups, while larger values created more smaller groups. In either case, the difference was only a few seconds in processing with the value of 4 demonstrating the best construction time.

B. Longest Common Prefixes

Once a group has been created, that group is further divided into subgroups that define the longest common prefixes. To determine longest common prefixes it is necessary to compare all the regular expressions in the group to each other. The grouping stage should limit the number of regular expressions in the group, though theoretically the entire set could fall in one group. Regardless, we adopt a simple clustering algorithm that will cluster the set of regular expressions into subgroups of longest common prefixes, or as singletons (a single regular expression) in \(\frac{n(n-1)}{2}\) comparisons where \(n\) is the number of regular expressions in the group. Algorithm 1 provides pseudo code for how the subgroups are formed. After one pass, all of the regular expressions for the group will either be clustered with their longest prefix match, or they will be a singleton.
In most cases, there will be at most one singleton, though it is possible for multiple singletons in a single group. This is actually a desirable result for it favors longer matches and forces shorter matches into either their next longest prefix match, or a singleton where the next longest match will be found during merging.

C. Building the REduce NFA

All of the grouping has set the stage for building the REduce NFA. Algorithm 2 illustrates the basic concept of merging employed. Essentially, all of the regular expressions in a subgroups are merged to create a subgroup NFA. Then, subgroups are merged together by the longest common prefix between two subgroups. Once all of the subgroups have been merged together there will remain only a single subgroup left, which contains the merged NFA for all subgroups in that group and is thus the group NFA. This is repeated for all groups. As can be seen, this is a costly process which motivated much of the partitioning and subgrouping done earlier. We note, for regular expressions that do not cluster to any group, they are represented as a group of one regular expression and are simply converted into a group NFA directly.

Once the group NFA have been built, then the groups are merged. At this point, we know that none of the groups share a common prefix longer than, or equal to, the minimum shared length. Thus, we can use a hybrid of our grouping and merging techniques to merge groups. We repeat the grouping algorithm after decreasing the minimum shared length by one. All groups that hash to the same key are merged into a single group and the process continues on the modified set of groups after decreasing the minimum shared length by one again. This continues until the minimum shared length becomes one. At that point, groups are merged as normal, but rather than continuing the merging algorithm, all group NFA are directly added to the final REduce NFA. This is because after the grouping at length 1 is complete there can be no more divergence. Thus, the groups at that point represent the overall NFA and all that remains is to tie the groups to the start of the REduce NFA. The NFA can now be treated as any other NFA and converted to other formats like a Hybrid Finite Automata [5], a DFA, or translated to hardware.

V. EVALUATION

For evaluation we look at the improvement using a REduce NFA over a normal NFA for matching with an NFA, and for matching with a Hybrid Finite Automata (HFA) [5]. Our goal is to demonstrate that REduce improves the matching automata, and that improvement can work cumulatively with other approaches like HFA. The results shown here represent the cumulative improvements. We used the 6 sets of regular expressions as described in Section II-A. For evaluation traffic we first generated a packet capture of one million TCP packets, 1514 bytes long, containing uniformly random data (termed Rand). In addition to that, we create one capture per rule-set with one million 1514 byte TCP packets where the content of each packet matches one of the regular expressions in the target rule-set (termed Matching). The Matching packet capture was generated by randomly selecting a regular expression from the target rule-set and creating sufficient data to build a TCP packet 1514 bytes long that will match that particular regular expression and repeating this process 1 million times. The Rand packet capture represents a best-case scenario where the data is unlikely to match very deeply with the matching automata and thus should be processed efficiently. The Matching packet capture represents a worst-case scenario where every packet matches deeply with the matching automata and should greatly strain its ability to match efficiently. The techniques for building these packet captures are derived from those provided by Becchi et al. [23] and our own research [24].

Our matcher was built in C and our Automata Constructor (can build an NFA, or an HFA given an NFA) is comprised of a few thousand lines of C++ code. REDuce and REcompare
ClamAV OBDD Proprietary Random Snort SpamAssassin

Fig. 7. Average change in construction time due to REduce NFA (<1 = speedup; >1 = slowdown).

were also written in C++. For each set of regular expressions we create a normal NFA and then match against both the Rand and Matching packet captures. We create a REduce NFA and do the same. Each test is executed 10 times. The statistics noted here represent the average improvement of the matcher employing a REduce NFA versus the same scenario with a non-REduce NFA. The entire test was then repeated using HFA, one created with a non-REduce NFA and the other created with a REduce NFA. We examined construction time, reduction in size of the constructed automaton, and matching efficiency as a measure of throughput and reduction in active state.

A. Construction Time

Construction time is defined as: \( \frac{\text{REduce build time}}{\text{Non-REduce build time}} \). Intuitively, we expect REduce to require more time in construction due to the extra work in grouping and merging required during NFA construction. For the most part, this is the case as is illustrated in Fig. 7 with REduce requiring up to twice as much time to construct an NFA over typical NFA construction. However, when constructing an HFA, which first constructs an NFA and then converts a portion of that NFA to a head DFA and ties the two together, we see that for the OBDD, Random, Snort, and SpamAssassin rule-sets that the construction time is actually shorter for REduce. This stems from the fact that these rule-sets are the most complex yet REduce has created a prefix-minimal NFA. This means that there are few redundant paths added to the head DFA which makes the head DFA easier to build and capable of accommodating deeper traversals. This further removes possible redundant transitions from the head-DFA to tail NFA. All of this serves to make for a shorter build time for the HFA in these instances. It should be noted that even when REduce requires more time to create the NFA the scale is in tens of seconds (43 seconds for the proprietary rule set and 137 seconds for ClamAV).

B. Size Reduction

Size Reduction is defined as: \( \frac{\text{REduce size}}{\text{Non-REduce size}} \). For every state eliminated by REduce all transitions connected to that state are also eliminated, both of which serves to reduce the overall size of the NFA. A smaller NFA is desirable though a smaller working set is even more desirable during matching as it improves the efficiency of fast memory (caches). ClamAV saw the least reduction in size primarily due to the fact that most of the regular expressions in ClamAV were fixed binary strings and thus did not offer much room for improvement as Thompson’s algorithm [32] is able to optimize away most redundancy without help. Snort and OBDD (which is derived from Snort rules) demonstrate the best reduction in size due to the heavy use of repetition and alternation in the regular expressions. This is also true, to a lesser degree, with SpamAssassin and the Random rule-set. In fact, the primary reason for the large reductions in the OBDD, Random, Snort, and SpamAssassin rule-sets as HFA is a large reduction in the number of transitions. The merging of branches early in the NFA for these rule-sets reduces their complexity and eliminates redundant transitions from the head DFA to tail NFA which greatly aids in size reduction as well as a faster build time as discussed earlier.

C. Performance

For performance we examined two metrics. First, we looked at the average active states during traversal defined as: \( \frac{\text{REduce Avg Active State}}{\text{Non-REduce Avg Active State}} \). Our expectation is to see small improvement with the Rand packet capture and large improvement with the Matching packet capture. The reason for this is that random traffic will already have a smaller working set, potentially small enough to already fit in cache memory. Thus, making the working set even smaller will have only minimal impact. Conversely, we expect significant improvement in efficiency when processing the Matching packet capture as the removal of redundant paths will reduce active state.

Fig. 9 Illustrates the actual results for active states during traversal with all rule-sets showing a decrease in the amount
of active state. An active state represents a path that must be traversed by the NFA and there can be any number of active states depending on the input and the NFA. As expected, when matching deeply, the number of active states is greatly diminished. This is direct evidence that the number of redundant shared paths has been reduced in the matching automata. More surprising, however, is that this trend is also mirrored for the random traffic as well indicating that the working sets for most of the rule-sets are large even for random traffic. Other than ClamAV, and the HFA version of the Proprietary regular expression set, REduce resulted in a significant decrease in active state. ClamAV, however, demonstrated only a tiny improvement of about 4% across the board. This is, as stated earlier, due to the fact that most of the ClamAV regular expressions are fixed binary strings. In fact, this holds for the Proprietary rule-set as well when matching against random traffic. In that set the beginning of most of the regular expressions is a fixed sequence of characters without any repetition or alternation. Thus, during HFA construction most of these initial states are pushed into the head DFA and matching against random traffic need not ever leave the head DFA resulting in only a single active state.

Fig. 10 shows that reductions in active state typically translate to commensurate improvements in throughput. Throughput is defined in terms of speed-up for easier readability:

\[
\text{Throughput} = \frac{\text{Non-REduce Throughput}}{\text{REduce Throughput}}
\]

Though it is difficult to see in Fig. 10 all rule-sets saw a speedup, even if small. ClamAV saw the least improvement overall (about 4–6%) for the same reasons it saw minimal reductions in active state. The Random rule-set, which we expected to perform the worst due to limited commonality within the rule-set, demonstrated a 60% improvement when matching as an NFA and a 9–14% improvement when matching as an HFA. This demonstrates that rule complexity can prove more important than commonality as complex rules almost guarantee redundant paths near the start of an NFA which increase active state and decrease throughput. For Snort and the Proprietary rule-set (as HFA), as expected, we see a little improvement when matching against random traffic (between 6–20%) implying that the matching automata working set is small enough without REduce to operate at peak efficiency. More interesting is the 2 to 4 times speed-up for random traffic for the remaining rule-sets. This extended beyond expectations and illustrated that REduce can improve best-case performance in addition to worst-case performance. Lastly, aligned with our expectations, we note a significant speed-up for matching traffic.

For completeness we examined the performance of REduce when matching against non-synthetic traffic from various public and private sources. For private sources we used traffic collected at Washington State University from three separate sources: a local router within the Computer Science Department, a Local Area Network gaming event, as well as a computer Wargame event. For public data we used the following packet captures: DEFCON 11 and 17 capture the flag [33], the US Army Information Technology Operations Center CDX data sets [34], and the Mid-Atlantic Collegiate Cyber Defense Competition [35]. We repeated the same tests on samples from the listed traffic sources. Our expectation was that performance against this traffic should closely resemble that for random traffic as most of the traffic is innocuous. While the overall trends mirrored those for the random traffic shown in Fig. 9 and Fig. 10, the performance was actually between 8–29% better for the majority of the rule-sets. This demonstrates that even a small number of matching packets (less than 1%) is sufficient to impact matching efficiency. Even more surprising was that the Snort rule-set running with REduce saw an improvement of around 150% against this traffic indicating that the traffic intersected more closely with the Snort regular expressions.

VI. DISCUSSION

REduce does not work equally with all regular expression sets. Simple rule-sets, like ClamAV, are adequately handled through normal NFA construction and need not utilize REduce for improvements. However, even in these instances REduce
results in a small improvement to performance at the cost of increased NFA construction time. Conversely, regular expression sets that benefited from REduce saw strong improvement under REduce whether matching as a simple NFA or as an HFA. Though REduce can increase the construction time of NFA, the total time involved is still measured in a scale of tens of seconds rather than minutes. Thus the cost of REduce in terms of time is not sufficient to deter its use. Finally, the improvement from using REduce translated through to the matching architecture; HFA in this case. Since the REduce NFA is a prefix-minimal NFA it should always create an NFA that will work as well, or better, regardless translation to new hardware or architecture simply because the NFA has no redundancies in any shared prefixes. The only instance where this might fail is if the matching architecture, or hardware, depends on redundancy within the NFA. However, the benefit from REduce will likely vary across different matching architectures and hardware. Examining other matching architectures or hardware will only explore this variance thus we targeted only HFA in this work.

VII. CONCLUSION

In this paper we have demonstrated that eliminating redundant paths during NFA construction can improve the efficiency of the matching automata. Not only do the techniques outlined here shrink the size of the matching automata, they create a prefix-minimal NFA. This, in turn, requires less active state during traversal and will increase throughput. More importantly, this is accomplished by only modifying the construction of the initial NFA used for the matching automata. Thus, REduce can be used in conjunction with other matching architectures, or even hardware, to arrive at cumulative improvements for matching of regular expressions for network security at a reasonable investment in resources.

REFERENCES