

number of partial flows D_m 's, and simply storing the partial flows until they can be merged can be quite memory-intensive.

The alternative we employ is simulating the DFA representing \mathcal{R} on the D_m 's. In order to do this for partial flows which are not a prefix of c , we need to know which state in the DFA to start the simulation from. Our key idea is very simple: to simulate the DFA on D_m 's with all potential beginning states for D_m in the DFA. This leads to a number of potential end states for each D_m . We merge the partial flows when possible, pruning the potential beginning states for D_m . Further, we utilize the so-called *equivalence classes* of states reached by simulating the DFA from different begin states.

We present three algorithms: a *sequential* algorithm, a *parallel* algorithm that collapses equivalent states into equivalence classes, and a *mixed* algorithm that tries to balance the tradeoffs of the first two.

4 The Sequential Algorithm

The algorithm maintains the information about the received partial flows in the form of a linked list R of objects $D_1, D_2, \dots, D_i, \dots, D_n$. Each $D_i = (S_i, E_i, L_i)$ describes a reassembled partial flow, and contains the following:

- (S_i, E_i) - the starting and ending offset of the reassembled data within the original data transmitted within the flow.
- L_i - a linked list of pairs (q_s, q_e) describing the starting and ending states of paths within the automaton representing the regular expression that can be traversed with the data corresponding to D_i .

At various stages of the algorithm we will attempt to find partial flows that either precede or succeed the newly arrived segment in the original data, and merge them into one list entry. If, as a result, we obtain two entries D_i and D_{i+1} in the list such that D_i precedes D_{i+1} in the original data, we will merge them into one entry as well.

4.1 Traversing DFA

As part of the algorithm, we need to traverse the automaton representing the regular expression with the data contained in the currently processed data segment d , beginning from a given state q_i within the automaton. The automaton traversal stops when an accepting state is reached, the end of the data is reached, or when there's no transition on the current data character from the current automaton state.

The return value of the traversal process is a pair of states (q_s, q_e) , designating the starting and ending states of the path traversed, as well as flags indicating whether the q_s is the starting state of the automaton, and whether q_e is an accepting state. The process can also return a null value if there is no useful path that can be traversed with the given input, which can happen in one of the two cases:

- we reach a state from which there is no transition with the next data character, or
- both the beginning and ending state of the traversal process is the starting state of the automaton.

4.2 Detecting Start of the Flow

The algorithm begins with R empty. The beginning of a flow is detected by inspecting the value of the SYN (synchronize) bit in the TCP header of the arriving packets, with 1 signifying the flow start. When processing the first packet of the flow, we distinguish between two types of regular expressions: those that start with the starting anchor '^' and require the first packet to match starting from the starting state of the automaton, and those that start with '.'*' and imply that the regular expression can be matched anywhere within the flow.

Thus the first data segment $d_1 = (s_1, e_1)$ of the flow is processed as follows:

Traverse the DFA beginning from the starting state of the automaton. If the regular expression starts with '^':

- If the traversal process returned null, we label the flow as "not matching", and no further processing is done.
- If the traversal process returned a pair of states (q_s, q_e) , with q_s marked as the starting state of the automaton, create a new entry $D_1 = (s_1, e_1, L_1)$ in R , where L_1 contains the pair (q_s, q_e) .

If the regular expression does not start with '^':

- If the traversal process returned null, create $D_1 = (s_1, e_1, < \text{empty list} >)$ in R .
- If the traversal process returned a pair of states (q_s, q_e) , with q_s marked as the starting state of the DFA, create a new entry $D_1 = (s_1, e_1, L_1)$ in R , where L_1 contains the pair (q_s, q_e) , and proceed to the next data segment.

4.3 Processing Subsequent Segments

Any other data segment $d_i = (s_i, e_i)$, $s_i > 1$, is processed as follows. For each object D_m in list R :

Duplicate handling:

- If d_i is fully contained in D_m , ignore d_i and proceed to the next segment.
- If D_m is fully contained in d_i , delete D_m from R .
- If d_i and D_m partially overlap, chop off the overlapping section of d_i by adjusting its (s_i, e_i) offsets accordingly. Formally, either $s_i = E_m + 1$ or $e_i = S_m - 1$ depending on whether S_m is smaller than s_i or otherwise.

Predecessor processing: Let $D_p = (S_p, E_p, L_p)$ be a predecessor of d_i , i.e. $E_p = s_i - 1$. If L_p is not empty, for each pair (q_s, q_e) in L_p :

- Traverse the DFA with d_i starting at q_e .

- If the traversal returns a pair (q_e, q_{e1}) , delete the pair (q_s, q_e) from L_p , store the pair (q_s, q_{e1}) in L_p and update $E_p = e_i$.
- If the traversal returns null, delete (q_s, q_e) from L_p . If this renders L_p empty, label the current flow as "not matching".

If L_p is empty:

- Traverse the automaton with d_i beginning at the automaton's start state.
- If the traversal returns a pair (q_s, q_e) , insert the pair (q_s, q_e) in L_p , and update $E_p = e_i$.
- If the traversal returns null, update $E_p = e_i$; L_p remains empty.

If there is no predecessor for d_i in R :

- Create a new entry $D_p = (S_p = s_i, E_p = e_i, L_p = \langle \text{empty list} \rangle)$ in R .
- Traverse the automaton with d_i starting at every non-accepting state, and insert all non-null pairs returned by the traversal process in L_p .

Successor Processing: At the end of predecessor processing part of the algorithm, we have either merged d_i in an existing D_p , or created a new D_p for the newly arrived segment. At this stage of the algorithm we check whether D_p has a successor in R .

If a successor $D_s = (S_s, E_s, L_s)$, such that $S_s = E_p + 1$, is found (else, proceed to the next arriving data segment):

- If both L_p and L_s are non-empty, update $S_s = S_p$, merge L_p into L_s and delete D_p from R .
- If L_s is empty, update $S_s = S_p$, merge L_p into L_s and delete D_p from R .
- If L_p is empty, update $S_s = S_p$ and delete D_p .

The merging procedure of the lists is as follows:

- For any pair of states (q_{sp}, q_{ep}) in L_p , if q_{ep} is an accepting state, copy (q_{sp}, q_{ep}) to L_s
- For each pair of states (q_{ss}, q_{es}) in L_s , not including those just copied from L_p :
If there is a pair (q_{sp}, q_{ep}) in L_p such that $q_{ep} = q_{ss}$, delete (q_{ss}, q_{es}) from L_s and insert (q_{sp}, q_{es}) to L_s .
Else delete (q_{ss}, q_{es}) from L_s .

Match detection: At any step of the algorithm, if a pair of states (q_s, q_e) such that q_s is the starting state of the automaton and q_e is an accepting state is found in any of the L lists, label the flow as matching the regular expression.

5 The Parallel Algorithm

In the algorithm above, if we find no predecessor for the newly arrived data segment, we traverse the automaton with the segment, starting at each non-accepting state. This can be a performance bottleneck since the automaton can have a large number of states. In addition, the traversal process can result in a large number of pairs (q_s, q_e) , and a significant

number of those pairs can be duplicates ($q_{s1} = q_{s2}$ and $q_{e1} = q_{e2}$) stored in the different lists, or pairs with different starting states but identical ending states ($q_{s1} \neq q_{s2}$ and $q_{e1} = q_{e2}$).

Definition 1. An *equivalence class* is a list of automaton state pairs that have different starting states but identical ending state, and is described as $Q = (l_s, q_e)$, where l_s is a list of starting states $(q_{s1}, q_{s2}, \dots, q_{sk})$.

Thus, we improve the sequential algorithm by storing automaton state equivalence classes instead of state pairs. This would entail several changes as shown below.

5.1 Data Structure

For each element D_i of the list R we maintain the following information: (S_i, E_i) - the starting and ending offset of the reassembled data within the original data transmitted within the flow; L_i - the list of equivalence classes, describing the starting and ending states of paths within the automaton representing the regular expression that can be traversed with the data corresponding to D_i .

5.2 Traversing DFA

Given a list of automaton states and a data segment d_i containing characters $x_1x_2\dots x_n$:

1. Attempt to make a transition from each of the states q_j with the first character x_1 . Store all pairs of states (q_j, q_k) , where $q_k = \delta(q_j, x_1)$, in a temporary list.
2. Find all pairs in the list with identical end states, delete them from the list and replace them with the corresponding equivalence class. As a result, we obtain a list of equivalence classes $Q_1 = (l_{s1}, q_{e1}), Q_2 = (l_{s2}, q_{e2}), \dots$, with $|l_{si}| \geq 1$.
3. For each Q_i , attempt to make a transition $\delta(q_{ei}, x_2)$ unless q_{ei} is a final accepting state. If such transition exists, update $Q_i = (l_{si}, \delta(q_{ei}, x_2))$. Repeat the equivalence class merging procedure.
4. Repeat steps (2) and (3) until one of the following:
 - No new transition can be made on the next x_i .
 - End of the data segment d_i is reached. Return the resulting list of equivalence classes.
 - An equivalence class Q_i is obtained such that one of the states in l_{si} is the start state of the automaton, and q_{ei} is a final accepting state. Label the flow as a match of the regular expression.

5.3 Processing Data Segments

The procedure (both dealing with the first segment of the flow and the subsequent segments) is mostly identical to the sequential version of the algorithm, storing equivalence classes instead of pairs of states. The important difference

in the parallel version is in the predecessor handling part of the algorithm, when the segment d_i arrives out of order:

Predecessor Processing: If there is no predecessor for d_i in R , create a new entry $D_p = (S_p = s_i, E_p = e_i, L_p = \langle \text{empty list} \rangle)$ in R . Traverse the automaton using the modified traversal procedure, with d_i and the list of all non-accepting states as an input. If the flow is not declared “matching”, store the returned list of equivalence classes in L_p . A similar optimization can be applied for the case when a predecessor is found, but $|L_p|$ is large.

Successor Processing: At the end of predecessor processing part of the algorithm, we have either merged the newly arrived segment d_i in an existing partial flow D_p , or created a new D_p based on d_i . If a successor $D_s = (S_s, E_s, L_s)$, such that $S_s = E_p + 1$, is found in R , and $|L_p| > 0$ and $|L_s| > 0$, we merge the predecessor and the successor into one partial flow by updating $S_s = S_p$, merging L_p into L_s and deleting D_p from R . We omit the detailed description of the merge procedure of the L lists due to space constraints.

5.4 The Mixed Version

The parallel version of the algorithm significantly reduces the amount of states that needs to be maintained at each step of the algorithm. However, the structure that maintains the states - a list of equivalence class objects - is now more complex, and therefore the overhead of accessing and updating an equivalence class in the list is more significant. To achieve a better tradeoff, we have developed a simple hybrid that integrates both the sequential and the parallel versions of the algorithm. The mixed algorithm will still take advantage of the equivalence classes while improving the parallel algorithm’s overall performance.

- For any out of order data segment d_i , run the parallel version of the algorithm for k steps, processing k first characters in d_i and obtaining a list of equivalence classes.
- Run the sequential version of the algorithm with the remaining characters in d_i , starting from every equivalence class’ ending state q_e .

In this approach, we assume that running the parallel version of the algorithm for the first k input characters will yield a limited amount of equivalence classes, thus reducing the amount of states starting from which we apply the sequential version of the algorithm.

6 Experiments

The timing experiments described below were performed on a 2.8GHz processor server. Due to space constraints we do not present the full extant of experiments conducted.

6.1 Out-of-order DFA Traversal Time

In order to compare the three versions, we collected a set of data from our research center’s network connection sent in TCP packets with either the source or the destination port 80, with the total of 5,565 data segments. We simplified the study by supporting only a limited subset of regular expression language, and by simply replacing every occurrence of ‘.’ with a set of all supported characters. We tested our implementation on four regular expressions :

Regex 1: `^HTTP/1.[01].*[0-5][0-1][0-9]` - match an HTTP response message.

Regex 2: `^(OPTIONS|GET|HEAD|POST|PUT|DELETE|TRACE|CONNECT).*HTTP/1.[01]` - an HTTP request message.

Regex 3: `HTTP/1.[01].*User-Agent: Mozilla/[45].0` - messages generated by Mozilla versions 4.0 or 5.0.

Regex 4: `HTTP/1.[01]Host:.*google.co.uk` - messages with the Host header matching `google.co.uk`.

The DFA’s built for each of these regular expressions contained 109, 134, 214 and 212 states respectively (no DFA minimization was done). There were 451 data segments within the data set that matched the first regular expression, 454 that matched the second, 356 the third and 119 the forth.

In this experiment we compare the running time of the out-of-order traversal procedure of the three versions of the algorithm, when traversing the DFA for each of the regular expressions as if every data segment of the set had arrived out-of-order. The motivation behind the test is that the out-of-order traversal procedure is the bottleneck of the algorithm and the algorithm with the minimal out-of-order traversal time is the most efficient one. For the mixed version, we ran it with different values of k in order to find its optimal value. The results are presented in Tables 1.

The results demonstrate that the parallel version of the algorithm outperforms the sequential version by more than 50%, and that the mixed version is exceedingly faster than the sequential or the parallel for any value of k we used, with $k = 1$ yielding the best results for the two regular expressions with the starting anchor and a single ‘.’, and $k = 2$ or 3 for the two regular expressions that contained two ‘.’s. We observe that as we increase the value of k , the traversal time grows as well. Thus the optimal value of k roughly equals the number of ‘.’s within the regular expression being matched.

6.2 Size of the Equivalence Classes

To investigate the convergence rate of the number of equivalence classes we need to maintain on each step of the parallel DFA traversal procedure for an out-of-order packet, we collected this statistics while matching the data segment set with each of the four regular expressions.

Figure 1 shows the average number of equivalence classes at every step of the automaton traversal procedure. It

	Seq	Par	Mix $k = 1$	Mix $k = 2$	Mix $k = 3$	Mix $k = 4$	Mix $k = 10$	Mix $k = 100$	Mix $k = 200$
Regex 1	1:00	1:30	0:04	0:04	0:04	0:05	0:06	0:16	0:25
Regex 2	2:26	1:32	0:05	0:05	0:06	0:06	0:07	0:20	0:30
Regex 3	19:18	9:23	0:19	0:17	0:17	0:18	0:21	1:30	2:44
Regex 4	20:00	9:25	0:19	0:16	0:17	0:17	0:21	1:30	2:52

Table 1. Out-of-order DFA traversal time of the different versions of the algorithm (in minutes and seconds).

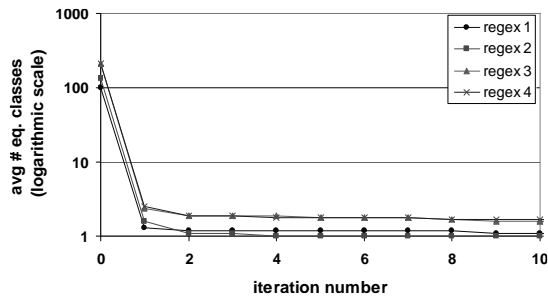


Figure 1. Convergence rate for the average number of equivalence classes.

can be seen that the number drops from hundreds to one or two, with the convergence rate for regular expressions starting with ‘.’ being slightly slower. Again, we can see that the average number of equivalence classes roughly equivalent to a number of ‘.’s within a regular expression.

It is important to note that since each partial flow contains at least one equivalence class, the number of equivalence classes that needs to be maintained at each step of the algorithm can be thought of as corresponding to the number of partial flows that the algorithm maintains at each step. We can therefore see from the experiment above that the average number of maintained partial flows is very low.

7 Related Work

In [2], authors studied various networking protocols and applications in depth to determine suitable signatures for them. They did not solve the problem of matching signatures across segments. We have used their application signatures in this study. Snort [3] is an intrusion-detection application that has a compiled list of several regular expression signatures to match attacks and intrusions. Snort systems use Perl Compatible Regular Expressions (pcre) [7] for regular expression matching which is performed on reassembled packet streams. In networking community, there is significant amount of work on matching regular expression signatures to IP packet streams, using specialized hardware like FPGAs [5, 6, 4], although with full TCP reassembly as well. [1] offers added focus on matching multiple regular expressions by grouping multiple regular expressions to eliminate common states. We are not aware of any Snort systems or specialized hardware solutions in network-

ing that match regular expression signatures in presence of out-of-order packets, without reassembly.

8 Conclusion

The problem of matching a regular expression to a data stream in presence of data quality problems is well-motivated in managing IP networks where signatures need to be matched against contents of flows to detect intrusions, worms or viruses, applications and protocols. Prior work either matched regular expressions against individual packets, missing matches across multiple segments, or reassembled the entire flow to match using standard methods, which is highly resource-intensive. We have proposed streaming algorithms that can match regular expressions across segments, even in presence of out-of-order packets and duplicates, by optimizing the state maintained on partial flows. Our experimental study with real data shows that the algorithms are successful in limiting the memory used and are efficient.

References

- [1] F. Yu, Z. Chen, Y. Diano, T. Lakshman, R. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. Tech. Report No. UCB/EICS-2005-8, October 2005.
- [2] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable, In-network Identification of P2P Traffic Using Application Signatures. In *Proc. WWW Conference, NY*, May 2004.
- [3] “Snort Network Intrusion Detection System.” <http://www.Snort.org>
- [4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2003.
- [5] R. Sidhu and V. Prasanna. Fast Regular Expression Matching using FPGAs In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, April 2001.
- [6] M. Attig and J. Lockwood. SIFT: Snort Intrusion Filter for TCP In *Symposium on High Performance Interconnects (HotI)*, August 2005.
- [7] PCRE - Perl Compatible Regular Expressions, www.pcre.org.
- [8] Application Layer Packet Classifier for Linux <http://l7-filter.sourceforge.net>