A Memory Efficient Regular Expression Matching by Compressing Deterministic Finite Automata

Utkarsha P. Pisolkar
PG Student, Dept. of Computer Engineering,
G.E.S's R.H. Sapat College of Engineering,
Management Studies & Research, Nashik
Savitribai Phule Pune University, India

Shivaji R. Lahane
Asst Professor, Dept. of Computer Engineering,
G.E.S's R.H. Sapat College of Engineering,
Management Studies & Research, Nashik
Savitribai Phule Pune University, India

ABSTRACT
Regular expressions are very meaningful and now-a-days broadly used to represent signatures of various attacks. The focal component of today's security systems like intrusion detection and prevention system is a signature based regular expression matching. Deterministic finite automaton is often used to represent regular expressions. In regular expression matching, storage space of Deterministic finite automata is very important concern. A massive amount of memory is essential to store transition function of Deterministic finite automata. The method described in this paper reduces size of Deterministic finite automata which is in regular expression format. The performance of the regular expression matching by compressing Deterministic finite automata is evaluated by using regular expression set.

General Terms
Pattern matching algorithms, Network Security, Theory of Computations.

Keywords
Regular expressions; security attacks; deterministic finite automata; intrusion detection and prevention.

1. INTRODUCTION
Signature based regular expression matching is the focal operation in intrusion detection and prevention system, traffic classification, content based filtering and monitoring system etc. It inspects the packet data and compares that data against database of attack signatures or database of patterns of interests. It involves examination of a given sequence of text string for the presence of pieces of some pattern. A finite automaton is used to represent database of signatures. A deterministic Finite Automaton (DFA) is commonly used to represent these signatures [1]. Pattern matching process takes place over this DFA by traversing it.

Regular expression based pattern matching is very widely used today because regular expressions are compact and easier to express variety of security attacks instead of easy patterns of string. When regular expressions are implemented through DFA, it requires only one memory access per byte but require large amount of memory space to store their transition tables [2]. Therefore, they have limited use in real applications.

This paper describes a method for memory space reduction of DFA generated from regular expressions. This method has four main phases as Regular expression to Non-Deterministic finite automata (NFA) conversion, NFA to DFA conversion, DFA compression and matching on DFA. These four phases converts regular expressions to compressed form of DFA. The signature based regular expression matching is performed on compressed DFA.

The remaining paper is organized as follows. The related work of some of the existing DFA compression methods and different algorithms used till today are discussed in Section 2. Section 3 highlights the regular expression matching by compressing DFA work in detail, and Section 4 presents results of Regular expression matching by compressing DFA system on dataset. Concluding remarks and future work given in Section 5.

2. RELATED WORK
Compressions of DFA have been done till today, mainly include approaches like reducing transitions, states, input alphabet sets and bits that represents the transitions. The total number of transitions of the state were reduced by approaches like D^2FA [3] and CD^2FA [4]. The D^2FA [3] is the Delayed Input DFA (D^2FA) approach, which has used default transitions to reduced space requirements between the states that have larger number of common transitions [3]. The CD^2FA [4] is the Content Addressed Delayed Input DFA used to increase the speed of D^2FA through laying up more information on the transitions edges and replacing state numbers with content label to leave out past default transitions.

The total number of states in DFA was reduced by the approaches like Hybrid DFA-NFA, HFA and XFA. The hybrid DFA-NFA [5] has reduced number of states by merging the advantages of both automata DFA and Non-deterministic finite automata (NFA) [5]. The nodes that have state explosion problem were represented by NFA and for lasting nodes a DFA was used. A History-based Finite Automaton (H-FA) has reduced the number of states by storing the transition history in a history buffer which is a small and fast cache [6]. The extended character set (XFA) [7] is another approach that has reduced number of states of DFA. The conditional transitions were removed with many automata transformations. For complex regular expressions XFA approach is not good because separate DFA state is required per every regular expression.

The number of states and transitions were decreased by the Delta finite automaton [8] approach. The diversity between nearby states which have many familiar transitions was stored in this approach [8]. The size of input alphabet table was decreased by alphabet compression table [9] approach. The bunch of characters in an input alphabet set was transferred to a small bunch of clustered characters for some states which have analogous transitions in the automaton. For every partition, an isolated alphabet compression table was produced [9]. The number of bits that represents every state

were reduced by HEXA [10] (History based Encoding, eXecution and Addressing) approach [10]. This approach was accumulated several paths to every node in history. Some extra discerning data was added to the history so that every node had separate storage location [10].

3. A MEMORY EFFICIENT REGULAR EXPRESSION MATCHING BY COMPRESSING DETERMINISTIC FINITE AUTOMATA

The regular expression matching by compressing DFA reduces the size of DFA which is generated from regular expression as shown in Figure 1. It consists of four phases. The first phase takes regular expression set R as input and the NFA is constructed for each regular expression r ∈ R using Thomson algorithm [11]. The second phase converts NFA to DFA using subset construction algorithm [12]. These DFAs are then combined into one single DFA and at the end compression method is applied on this single DFA in thirds phase. Fourth phase applies the regular matching process on to compressed DFA.

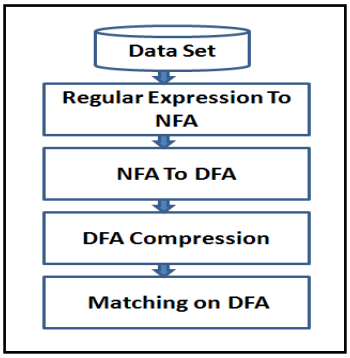

Fig 1: Block Diagram for Regular Expression Matching by Compressing DFA

3.1 Phase 1(Regular Expression to DFA Conversion)

Regular expression dataset R is input to the first phase. The Regular expression illustrates the pattern of the string. These regular expressions contains characters with symbols used in regular expressions such as closure (*) for zero and more occurrences, or (+) for one and more occurrences etc. Each regular expression r in R is converted into NFA using Thomson algorithm [11]. The Thomson algorithm consists of rules for conversion of each type of regular expression to each state in NFA [11]. The Figure 2 shows NFA for Regular expression "(a+b(ab)*)*".

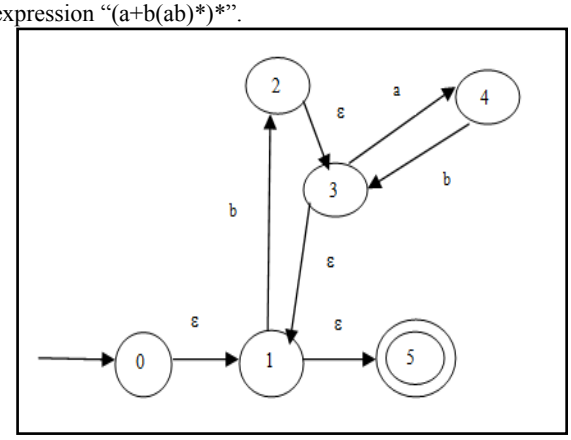

Fig 2: NFA for Regular Expression "(a+b(ab)*)*"

3.2 Phase 2 (NFA to DFA Conversion)

The set N of NFAs of given regular expressions is an input to the second phase. Each NFA nfa_i in set N is converted into corresponding DFA. The subset construction algorithm [12] is used for this conversion. Subset construction algorithm includes steps for converting each state in NFA to each state in DFA. The Figure 3 shows DFA for previously generated NFA in Figure 2.

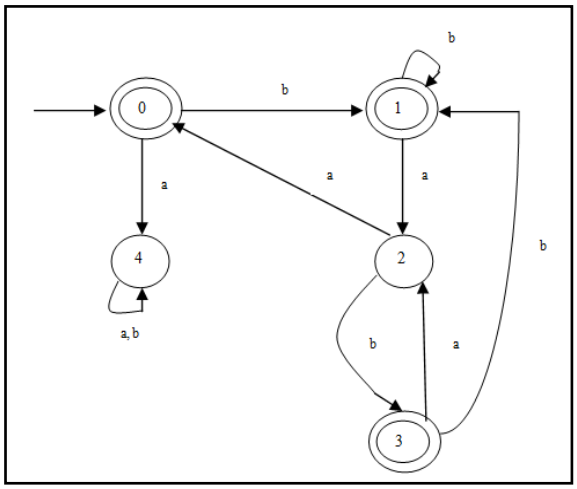

Fig 3: NFA to DFA of Regular Expression "(a+b(ab)*)*"

3.3 Phase 3 (DFA Compression)

The set D of DFAs of given NFAs is an input to the third phase. All DFAs in set D are combined into one single DFA. The Aho-Corasick algorithm [2] is used while combining the all DFAs. The Aho-Corasick algorithm has important property that it does not depend on precise input therefore it is not vulnerable to various security threats. Therefore Aho-Corasick algorithm is used in this work. At initial stage, the algorithm creates the finite machine from available set of keywords, and then traverses the finite machine using input text string for pattern matching purposes. The actions of pattern matching are determined by three functions. If a character is match, goto( ) function is executed with output( ) function otherwise it executes failure( ) function [2]. The transitions between DFA states are controlled by goto function and failure function. The output function is used for printing string if pattern matches [2]. The final single DFA, which is an output of Aho-Corasick algorithm, is used as an input for DFA compression approach. The compression approach reduces the size of DFA by using CompactDFA algorithm [1]. This algorithm decreases all transition rules to no more than one rule so that each state has only one transition rule. This converts all transitions to exacting state to a single rule. It has three steps as Grouping the States, Construction of Common Suffix Tree and Encoding [1].

3.3.1 Grouping the States

In the first step, set of states of single DFA is an input. Common suffix and longest common suffix values are calculated for DFA states [1]. If state s in DFA has one and more than one incoming transitions then Common Suffix value is calculated for that state. The label of the state is the common suffix value for it the lacking its final symbol [1].The extended length common suffix of a state is the longest common suffix for that state to which state s has an outgoing edge [1]. This step produces common suffix and longest common suffix values as an output.

3.3.2 Construction of Common Suffix Tree

In the second step, common suffix tree is constructed from which compressed rules are generated. Common suffix tree is constructed from a set of longest common suffix values. These values are become nodes of common suffix tree such that one value say lcs_i is a predecessor of another value lcs_j if and only if lcs_i is a suffix of lcs_j [1]. After that, for every internal node Connecting nodes are added to balance the tree such that every node in the tree should have number children equal to the power of two [1]. After that states are linked to the nodes in common suffix tree in contacted with its longest common suffix value such that a state is directly attached to the node if the node is leaf and if not then state is attached to one of the connecting node by balancing between all its attaching states. This step makes the common suffix tree as an output.

3.3.3 Encoding

In the third step, common suffix tree is an input and as an output it generates compressed rules for DFA. Encoding step encodes the common suffix tree, nodes and states. The size of the code is computed. The size of code is a number of bits necessary to predetermine the common suffix tree. The edges are determined by listing on every sibling edges [1]. The sibling edges are those edges that initiate from the similar node. Every edge is determined by its binary ordinal number and code size of $\Gamma \log n + 1 \rceil$ bits, where n is the number of sibling edges [1]. The nodes are determined by combination of codes of edges on pathway between root node and that node. The states are determined with the help of its position in common suffix tree. The compression rules are generated for those states that have the similar next state in DFA. This next state should not be the root of the tree. The compression rules are created in following way [1],

If state s_i has more than one incoming transitions then compressed rule for it includes three fields as code of the node common suffix of s_i is set as current state field, label on incoming link of s_i is set as symbol filed and code of node s_i is set as next state field.

If state has only one incoming transition the compressed rule for it includes three fields as code of the node s_j is set as current state field, label on incoming link of s_i is set as symbol field and code of node s_i is set as next state field, where s_j is the source of only edge to s_i .

If the state is root then the compressed rule for it includes three fields as * of code size is set as current state field, * is set as symbol field and code of root node is set as next state field.

This phase generates the set compressed rules for storing DFA in memory. The DFA is then stored into memory in the form of these compressed rules and pattern matching process uses these compressed rules. .

3.4 Phase 4 (Matching on DFA)

The compressed rules generated from regular expressions are used in matching phase. The state code is able to go with many compressed rules at a time. Therefore rule with longest prefix match is selected [1]. From the input data, which is to be inspected for presence of any security threat, one byte is selected at a time. This byte is matched with each compressed rule starting from root node rule. On every character, match process moves to next rule. When all characters in input text are over and if last state with which it has matched the rule is final state then input text contains security threats otherwise not. Therefore, total space required to store DFA and time required to match is depend on number of compression rules generated from phase 3.

4. EXPERIMENTAL RESULTS

The Snort data set [13] is used to evaluate the performance of a compressed DFA on simple patterns. The performance of uncompressed and compressed DFA of regular expression set and simple pattern set is tested on Intel Pentium Processor P6 100 with 2GB RAM.

The five different regular expression sets are used as RE20, RE12, RE6, RE5 and RE4. The RE20 set contains 20 regular expressions of length 7, RE12 set contains 12 regular expressions of length 4, RE6 set contains 6 regular expressions of length 5, RE5 set contains 5 regular expressions of length 16 and RE4 set contains 4 regular expressions of length 10.

Three different pattern sets are used as P100, P1000 and P5000. The P100 file contains 100 patterns, P1000 file contains 1000 patterns and P5000 file contains 5000 patterns. We have use two packet files as Pck500 and Pck5000. The Pck500 contains 500 packets and Pck5000 contains 5000 packets. These packet are randomly generated files through the programs. The storage space of DFA is measured in bytes and matching time is measured in milliseconds.

Table 1 shows results of DFA size on uncompressed and compressed DFA generated from regular expressions. Table 2 shows results of DFA size on uncompressed and compressed DFA generated from simple pattern set.

Table 1. DFA Size Comparison between uncompressed DFA and compressed DFA on regular expression set

Regular Expression Set	Length of Regular Expression	Size of uncompressed DFA	Size of compressed DFA
RE20	7	45378 byte	743 byte
RE12	4	34884 byte	1128 byte
RE6	5	26676 byte	484 byte
RE5	16	75924 byte	1164 byte
RE4	10	36936 byte	552 byte

Table 2. DFA Size Comparison between uncompressed DFA and compressed DFA on simple pattern set

Number of Patterns	Size of uncompressed DFA	Size of compressed DFA
100	20344 bytes	3368 bytes
1000	100652 bytes	9920 bytes
5000	202556 bytes	11394 bytes

Figure 4 shows results of pattern matching time on uncompressed and compressed DFA generated from regular expressions with pck500 input data file. Figure 5 shows results of pattern matching time size on uncompressed and compressed DFA generated from simple pattern set pck500 input data file.

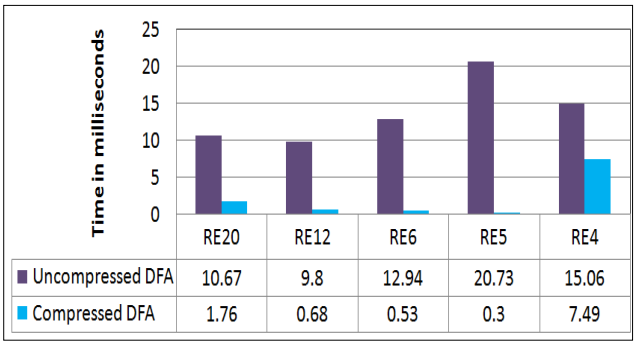

Fig 4: Comparison of pattern matching time on uncompressed and compressed DFA generated from regular expressions with pck500 input data file

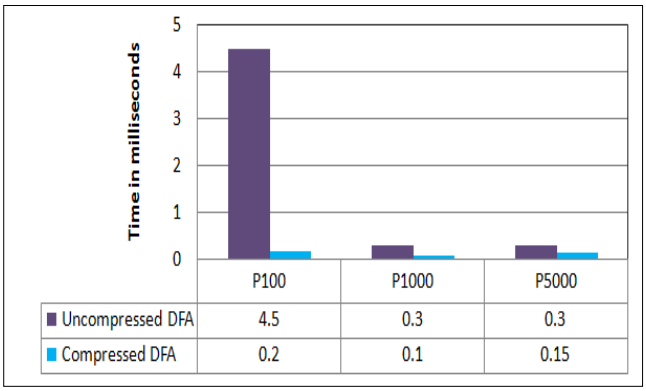

Fig 5: Comparison of pattern matching time on uncompressed and compressed DFA generated from simple pattern set with pck500 input data file

5. CONCLUSION AND FUTURE WORK

Regular expressions are broadly used to represent signatures of security attacks. DFA is easy way to express regular expressions. Memory space required to store DFA is very large. To address this problem, this paper has described the method which reduced the size of DFA generated from regular expression. The regular expression matching by compressing DFA method has converted regular expressions into DFA of minimum size. The DFA is stored into memory in the form of compressed rules. The compressed DFA of regular expressions is used at the end in regular expression matching process. As a future work, one may consider the regular expression which represents security attacks in special symbols for building and compressing deterministic finite automata.

6. ACKNOWLEDGMENT

We are glad to express our sentiments of gratitude to all who rendered their valuable guidance to us. We would like to express our appreciation and thanks to our Principal, Prof. Dr. P. C. Kulkarni. We are also thankful to our Head of Department, Computer Engineering, Prof. N. V. Alone. We thank the anonymous reviewers for their comments.

7. REFERENCES

[1] AnatBremler-Barr, D.Hay, Y. Koral, "CompactDFA:Scalable pattern matching Using Longest Prefix Match Solutions," in IEEE/ACM Transaction on networking,vol-22,No.2,April 2014.

[2] A.V. Aho and M.J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." Communications of the ACM, 18(6):333–340, 1975.

[3] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection", in Proc. of ACM SIGCOMM , pages 339-350. ACM, 2006.

[4] S. Kumar, J. Turner, J. Williams, "Advanced algorithms for fast and scalable deep packet inspection", in Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS), pages 81-92. ACM, 2006.

[5] M. Becchi, P. Crowley, "A hybrid finite automaton for practical deep packet inspection", in Proc. Conf. Emerging Netw. Exp. Technol.(CoNEXT), pages 1-12, 2007.

[6] S. Kumar, B. Chandrasekaran, J. Turner, G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia", in Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS), pages 155-164. ACM, 2007.

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

[8] D.Ficara, S.Giordano, G. Procissi, F.Vitucci, G.Antichi, A.D. Pietro, "An Improved DFA for Fast Regular Expression Matching" ACM SIGCOMM Computer Communication Review, Volume 38, Number 5, October 2008.

[9] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in Proc. Int. Conf. Security Privacy Commun. Netw. (Securecomm), 2008.

[10] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "HEXA: Compact data structures for faster packet processing", in Proc. IEEE Conf. Comput. Commun. (INFOCOM), 2009.

[11] Ken Thompson, "Programming techniques: regular expression search algorithm", in Communications of the ACM, Pages 419-422 , Volume 11 Issue 6, June 1968

[12] John E. Hopcroft and Jeffrey D. Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley Publishing, Reading Massachusetts, 1979.

[13] "SNORT," 2010 [Online]. Available: http://www.snort.org