Constrained Row-Based Bit-Parallel Search in Intrusion Detection

Ambika Shrestha Chitrakar and Slobodan Petrović
Norwegian Information Security Laboratory,
Norwegian University of Science and Technology (NTNU)
ambika.chitrakar2@ntnu.no, slobodan.petrovic@ntnu.no

Abstract

Most Intrusion Detection Systems (IDS) employ exact search for attack patterns in the analyzed traffic. Because of that, if an attacker introduces changes in the known attack pattern, the obtained new attack pattern becomes impossible to detect. To cope with this problem, an IDS can use approximate search instead of exact search. But then, false positives and false negatives can appear due to the fact that the type and/or the distribution of changes to the old attack traffic pattern is not taken into account. In this paper, we propose a new approximate search algorithm for IDS that introduces constraints on the numbers of individual change operations on the old attack traffic patterns. In such a way, we take into account a-priori knowledge about the type and/or distribution of changes. The experiments show that the false positive and false negative rates obtained with an IDS using approximate search with constraints are significantly reduced compared to a system without constraints. At the same time, the computational cost of introducing constraints is relatively small.

1 Introduction

Misuse-based intrusion detection systems (IDS) like Snort [1] use exact search to compare the content of network traffic with a set of known attack signatures. Because of that, the intrusions, whose signatures are not present in the misuse database cannot be detected. Approximate search, which allows certain number of errors (k) in the process of finding the occurrences of the search patterns in the search strings, if applied in an IDS, could ensure detecting new intrusions that are similar to the previously known ones. However, false positives and/or false negatives can then be produced since the type and/or the distribution of the alterations to the original attack traffic pattern (i.e. the total numbers of insertions, deletions, and substitutions) is not taken into account. By varying the value of k it is possible to influence the false positive and false negative rates to some extent but such a regulation is too imprecise and inefficient.

To reduce the numbers of false positives and false negatives when approximate search is used, we propose introducing certain constraints in the search process. If we know the probabilities of occurences of particular change/edit operations used to obtain the new attacks from the old ones, we can define appropriate constraints in approximate search

Presented at the Norwegian Information Security Conference 2016 (NISK-2016)

and adapt the search algorithm to the particular attack traffic generation process. The assumption about known probabilities of edit operations is justified by the fact that the attacker must limit the numbers of particular edit operations on the original attack pattern since too many individual operations can make behavior of the new attack pattern on the victim's system unpredictable (for example, the pattern could become harmless for the attacked system). The search constraint then limits the total number of occurences of each particular edit operation (usually deletions, insertions, and substitutions of symbols).

In this paper, we apply the ideas explained above in a constrained Row-Based Bit-Parallel aproximate search algorithm, CRBP-OpCount, which solves the approximate search problem with constraints on the maximum allowed numbers of edit operations. CRBP-OpCount simulates the Non-deterministic Finite Automaton (NFA) assigned to the search pattern, where the search allows up to k errors. It limits the total numbers of particular edit operations (deletions, insertions, and substitutions) to the values given in advance. This simulation is carried out in the Row-wise Bit-Parallel manner, first introduced by Wu and Manber in [9]. Unlike the algorithm described in [9], we limit the numbers of specific transitions in the NFA by assigning special counters to its states. This introduces some computational overhead, but with careful implementation and bearing in mind the fact that the maximum allowed numbers of edit operations cannot be too large, as explained above, efficient operation of an IDS implementing such an algorithm can be ensured and at the same time the false positive and false negative rates in the approximate search process can be kept under control.

We analyze the outputs of the CRBP-OpCount and the unconstrained approximate search algorithm (RBP) from [9] in order to compare their performances regarding efficiency and false positive/false negative rates. The inputs to the search algorithms (search patterns and search strings) in the experiments are generated from Snort rules.

The structure of the paper is the following. Section 2 reviews related work on approximate search. Section 3 explains the CRBP-OpCount algorithm. Experimental setup and the discussion of the experimental results are presented in Section 4 and the conclusion is given in Section 5.

2 Related Work

Modern fast exact search algorithms exploit so-called bit-parallelism, i.e. the possibility of encoding the status (active or inactive) of a Non-deterministic Finite Automaton (NFA) performing the search by means of a single bit of a computer word. Bit-parallelism has been extensively studied in the last 25 years (see, for example, [4, 7] etc.) The bit-parallelism technique was first proposed by Baeza-Yates and Gonnet [2] in the context of exact string matching. Wu and Manber [9] first showed how bit-parallelism could be used in (unconstrained) approximate search. They exploited bit-parallelism to simulate an NFA consisting of k rows, which is used in unconstrained approximate search.

In order to improve the efficiency of IDS and ensure their efficient operation at high data rates, many attempts have been made to speed-up the IDS search algorithm. Bit-parallelism has been used (see, for example, [5]), but also bit-splitting architectures [8], dynamic linked lists [3] and other methods. All these algorithms focus on speeding-up the multi-pattern matching, but they still employ *exact* search, which means that they are vulnerable to known attack patterns that were subject to small changes. Thus, evasion of such an IDS is still relatively easy.

Approximate search in IDS has not been treated extensively in the literature. One of the attempts to increase the efficiency of IDS search algorithms by employing

unconstrained approximate search is described in [6]. But this approach only considers insertions of new strings in attack traffic without modifying the existing strings. This makes our constrained search algorithm approach to intrusion detection a novel one.

3 The CRBP-OpCount Algorithm

The row-wise bit-parallelism (RBP) algorithm [9] served as a starting point in the development of the CRBP-OpCount algorithm. To exploit bit-parallelism in solving the constrained approximate search problem, where the constraints are on the total numbers of elementary edit operations (insertions, deletions, and substitutions), we apply these constraints on each active state of the NFA assigned to the search pattern before making a transition to the next state. We illustrate this process on an example (see Fig. 1).

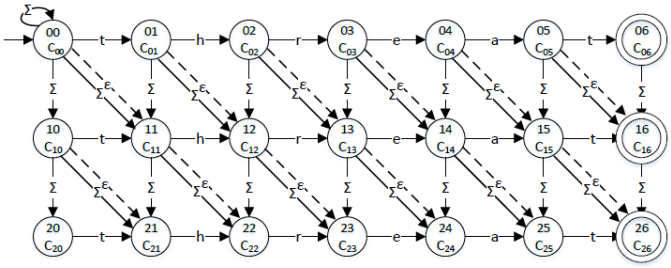

Fig. 1: NFA assigned to the search pattern "threat", allowing up to 2 character insertions, deletions, or substitutions (see text)

The NFA from Fig. 1 is assigned to the search pattern "threat" that allows up to 2 errors in constrained search. The 0^{th} row of the NFA corresponds to exact search, the 1^{st} row allows 1 error, and the 2^{nd} row allows 2 errors. The nodes R_{ij} in the NFA are called *states*, where R_{00} with a self loop is the initial state and the states with a double circle are terminal states. Each state contains a status bit D (that indicates whether the state is active or inactive, where active means that the state can be reached from other state(s) for a given input symbol) and a constraint buffer C, which contains triplets $[I,E,S]$ of elementary edit operations that are allowed for the transitions from the state R_{ij} (I insertions, E erasures/deletions, and S substitutions). The arrows in the NFA represent transitions: a horizontal arrow for an input symbol match, a vertical arrow for an insertion, a dashed-diagonal arrow for a deletion, and a solid-diagonal arrow for a substitution. For example, if there is a possibility of a substitution transition from the state R_{01} to the state R_{12}, the constraint value on the number of maximum permitted substitutions S is checked at the state R_{01} (the value S is contained in the constraint buffer $R_{01}.C$). If the constraint value S is greater than 0 then the substitution transition is allowed from R_{01} to R_{12}, otherwise the transition is not performed. In case of a successful substitution transition, the corresponding constraint value S is decreased by 1 for the state R_{12} and this state is made active (i.e. $R_{12}.D = 1$, $R_{12}.C.S = R_{01}.C.S - 1$). Similarly, the corresponding constraint values are checked for the deletion and insertion transitions. This process continues until all the input symbols of the search string are fed into the NFA. We say there is an *occurrence* of a search pattern in the search string at the position *pos*, if the NFA can reach one of the terminal states after receiving an input symbol.

Algorithm 1 applies the concepts introduced above. We now explain its details. The parameter P is a search pattern of length m, T is a search string of length n, and $con[I,E,S]$ is a constraint on the maximum number of permitted insertions (I), deletions (E), and

Algorithm 1 CRBP-OpCount

procedure CRBP-OPCOUNT(P, T, con)
Input parameters:
  $P = p_1 p_2 ... p_m$ - search pattern,
  $T = t_1 t_2 ... t_n$ - search string
  $con = [I, E, S]$ - constraints on total numbers of edit operations
k - maximum number of errors
  $k = I + E + S$
Preprocessing - defining bit masks
  For all characters c from the alphabet Σ, bit mask $B[c] \leftarrow 0^m$
  Then for all c in P:
  for $i \in 1...m$ **do**
    if $p_i = c$ **then** $B[c] \leftarrow B[c] \mid 0^{m-i}10^{i-1}$
Below, we count bits from the right (LSB= 1, MSB= m)
Preprocessing - initializing the NFA
  for $i \in 0...k$ **do**
    $R_i.D_j \leftarrow 1$, $j = 1, \ldots, i$
    for all j such that $R_i.D_j = 1$ **do**
      $R_i.C_j = \cup R_{i-1}.C_j$, with I, E, S reduced by 1 at a time
Processing of input symbols
  for $pos \in 1...$n **do**
    $R_0^{'}.D \leftarrow ((R_0.D{<<}1) \mid 0^{m-1}1) \& B[t_{pos}]$
    for all j such that $R_0^{'}.D_j = 1$ **do**
      $R_0^{'}.C_j = con$
    for $i \in 1...k$ **do**
      for all $j > 1$ **do**
        match.$D_j \leftarrow R_i.D_{j-1} \& B[t_{pos}]_j$
        if match.$D_j = 1$ **then** match.$C_j \leftarrow R_i.C_{j-1}$
        if $R_{i-1}.C_j.I > 0$ **then**
          ins.$D_j \leftarrow R_{i-1}.D_j$
          if ins.$D_j = 1$ **then**
            $newC \leftarrow R_{i-1}.C_j$
            $newC.I \leftarrow newC.I - 1$
            ins.$C_j \leftarrow newC$
        if $R_{i-1}.C_{j-1}.S > 0$ **then**
          sub.$D_j \leftarrow R_{i-1}.D_{j-1}$
          if sub.$D_j = 1$ **then**
            $newC \leftarrow R_{i-1}.C_{j-1}$
            $newC.S \leftarrow newC.S - 1$
            sub.$C_j \leftarrow newC$
        if $R_{i-1}^{'}.C_{j-1}.E > 0$ **then**
          del.$D_j \leftarrow R_{i-1}^{'}.D_{j-1}$
          if del.$D_j = 1$ **then**
            $newC \leftarrow R_{i-1}^{'}.C_{j-1}$
            $newC.E \leftarrow newC.E - 1$
            del.$C_j \leftarrow newC$
      $R_i^{'}.D \leftarrow$ match.$D \mid$ ins.$D \mid$ sub.$D \mid$ del.$D \mid 0^{m-1}1$
      $R_i^{'}.C \leftarrow$ match.$C \cup$ ins.$C \cup$ sub.$C \cup$ del.C
      if $R_i^{'}.D \& 10^{m-1} \neq 0^m$ **then** report an occurrence at pos

substitutions (S) to find the occurrences of a search pattern in the search string. Obviously, the total number of allowed errors k is equal to the sum of the constraints on I, E, and S.

The bit-mask $B[c] = b_m \dots b_1$ is computed for all the characters c of the search pattern P such that the j^{th} bit is set to 1 if the j^{th} character of the pattern P is the same as the character c. For characters other than the characters contained in the search pattern the bit masks $B[*]$ are equal to 0. This is a usual procedure for any bit-parallel search algorithm on strings, see for example [7].

Preprocessing is performed for all the rows R_i of the automaton, $i = 0 \dots k$. We set i starting bits to 1 for all R_i and to all the active bits of R_i, we assign a set of constraints C in such a way that the constraints on I, E, and S decrease by 1 at a time from the constraints from the previous row. Note that the constraint value of I, E, or S must be greater than 0 in order to assign it to a state. For R_1, the previous constraint array is always $con[I,E,S]$. Note that the set union removes the duplicate constraints. We illustrate preprocessing on an example. If $con[I,E,S] = [0,2,1]$ then preprocessing for each row will be as given below:

$R_1.C = \{[0,1,1],[0,2,0]\}$ for 1^{st} active state
$R_2.C = \{[0,0,1],[0,1,0]\}$ for 1^{st} and 2^{nd} active states
$R_3.C = \{[0,0,0]\}$ for 1^{st}, 2^{nd} and 3^{rd} active states.

For each input symbol from the search string, we run the row-wise NFA simulation cycle following approximately the lines from [9] (see Fig. 1), but at the same time we apply the constraints on all the active states of each row.

Since the 0^{th} row of the automaton indicates exact search (with 0 errors), all the active states in this row are assigned the constraint $con[I,E,S]$. Rows, whose ordinal numbers are greater than 0 indicate the numbers of errors, which can be insertion, deletion, or substitution errors. For all the active states in each such row, we need to modify the constraint values depending on the type of error. For a symbol match (horizontal transition), the constraint value from the previous state is copied to the current active state. For an insertion error (vertical transition), the constraint set C of the state in the same column from the previous row, updated at processing of the previous input symbol is checked. If it has the allowed number of insertion errors (i.e. more than 0) then the insertion transition is allowed and we reduce the corresponding constraint value(s) by 1 and store the whole constraint C as a new constraint for the current active state. In the case of a substitution error (solid-diagonal transition), the constraint set C of the state in the previous column from the previous row, updated at processing of the previous input character is checked. If it has the allowed number of substitution errors (i.e. more than 0) then the substitution transition is allowed. The constraint value(s) for the substitution is reduced by 1 and stored as a new constraint C for the current active state. Finally, in the case of a deletion error (dashed-diagonal transition), the constraint set C of the state in the previous column from the previous row, updated at processing of the *current* input character is checked. If it has the allowed number of deletion errors (i.e. more than 0) then we allow the deletion transition, reduce the constraint value(s) by 1, and store the whole constraint C as a new constraint for the current active state. We change the status of a state to inactive if any of these transitions was not performed due to the constraints.

We execute the operations described above until all the input symbols from the search string are fed into the automaton. We say there is an occurrence of a search pattern in the search string if the last state bit of at least one row is equal to 1 (i.e. the corresponding state is active). For example, if the last bit of the row R_3 is active then we say there is an occurrence with 3 errors. In order to check the number of insertions, deletions, or

substitutions used until that occurrence, the constraint set C can be checked at the active state.

4 Experimental Work

Experimental setup

We performed an experiment in order to compare the false positive rates and time efficiency of the CRBP-OpCount algorithm and the unconstrained approximate RBP search algorithm from [9]. This has been done by simulating the algorithms in an Intel(R) Core(TM) i7-4600U CPU @ 2.10 GHz 2.70 GHz laptop with 8 GB installed memory (RAM) and by analyzing their outputs. In the experiment, we first created the search patterns and search strings based on selected Snort rules and then we fed them into the CRBP-OpCount and the unconstrained RBP algorithms. These two algorithms tried to find the occurrences of each search pattern in all the search strings and generated the outputs. Details about these procedures are explained below.

For the experiment, the Snort version 2.9.6 was installed together with the corresponding rules available to the registered users. There were 73 different categories of rules, where the categorization is based on the attack types. We selected only one of these categories, `backdoor.rules`, in which many rules contain one or more so-called "content" fields, whose presence indicates activation of the Snort exact search algorithm at processing the corresponding traffic. It is possible to use any other rule category if testing the rules from that category in Snort activates the same search algorithm. To demonstrate the capabilities of approximate search algorithms, with or without constraints, we extracted the search patterns of the "content" option keywords in the rules contained in the `backdoor.rules` file and used them to create search patterns and search strings for the experiment. For example, one of the rules present in the `backdoor.rules` file is given below:
alert tcp `$HOME_NET 16959 -> $EXTERNAL_NET any` (msg: "BACKDOOR subseven DEFCON8 2.1 access"; flow: from_server, established; content: "PWD"; classtype: trojan-activity; sid:107; rev:6;)

From this rule, the "content" field string "PWD" would be extracted for the experiment and then distorted by applying insertions, deletions, and substitutions of characters. It was then supposed that traffic containing a string distorted in this way was sent to the victim running Snort and the same rule applied to such traffic pattern would not trigger. The experiment would then test approximate search algorithms, with and without constraints, on the distorted search string instead of the original Snort exact search algorithm.

The patterns associated to the "content" keywords of the rules contained in the file `backdoor.rules` were extracted in a text file in such a way that the duplicate patterns were eliminated and each pattern occupied one line in the text file. Some of the extracted patterns contained a mix of text and hexadecimal numbers such as `BN|10 00 02 00|`, or `Remote|3A| You are connected to me.` Because of that, it was necessary to convert all the extracted text to hexadecimal numbers. We converted all the text that was not enclosed within the pipe (|) characters to the hexadecimal numbers since the hexadecimal numbers were enclosed within the pipe characters in the extracted patterns. Finally, spaces were removed from the extracted patterns. After completing all these pre-processing actions, we obtained total of 61 search patterns for the experiment.

Search strings in our experiment were the strings that contained errors introduced by applying individual edit operations on the previously extracted search patterns. We introduced four different sets of errors on all the search strings in order to get four

different sets of distorted search strings: i) 5% insertions, 5% substitutions; ii) 10% deletions, 5% substitutions; iii) 10% deletions, 10% substitutions, and iv) 10% deletions, and 20% substitutions. We created four different sets of search strings in order to test the search algorithms with different sets of errors. Here, every search pattern had at least one occurrence in the list of distorted search strings and each search string was created with different constraints $con[I,E,S]$ depending on the corresponding error sets. We also removed search strings shorter than or equal to 3 hexadecimal numbers. After removing all the short search strings, we obtained total of 52 search strings in each set.

The CRBP-OpCount and the unconstrained RBP algorithms take the search patterns and the search strings as inputs and try to find the occurrences of the search patterns in the list of search strings. The CRBP-OpCount handles constraints on the maximum numbers of individual edit operations and RBP handles only the maximum number of errors k (i.e., k = #insertions+#deletions+#substitutions) while performing the approximate search. The implementations of the CRBP-OpCount and the RBP algorithms were capable of processing hexadecimal numbers instead of characters after a little modification.

The outputs of the CRBP-OpCount and the RBP approximate search algorithms in our experiment include the numbers of occurrences of 61 different search patterns in the list of 52 search strings for four different sets of errors introduced in the search strings. We created four cases to group the outputs of the CRBP-OpCount and the RBP for the analysis:

- *CaseA* includes: distorted search strings generated by the first set of errors (5% insertions and 5% substitutions); CRBP-OpCount with 5% deletions and 5% substitutions; CRBP-Opcount with 5% deletions and 10% substitutions; RBP with 10% error tolerance level; RBP with 15% error tolerance level.

- *CaseB* includes: distorted search strings generated by the second set of errors (10% deletions and 5% substitutions); CRBP-OpCount with 10% insertions and 5% substitutions; CRBP-Opcount with 10% insertions and 10% substitutions; RBP with 15% error tolerance level; RBP with 20% error tolerance level.

- *CaseC* includes: distorted search strings generated by the third set of errors (10% deletions and 10% substitutions); CRBP-OpCount with 10% insertions and 10% substitutions; CRBP-Opcount with 10% insertions and 15% substitutions; RBP with 20% error tolerance level; RBP with 25% error tolerance level.

- *CaseD* includes: distorted search strings generated by the fourth set of errors (10% deletions and 20% substitutions); CRBP-OpCount with 10% insertions and 20% substitutions; CRBP-Opcount with 15% insertions and 20% substitutions; RBP with 30% error tolerance level; RBP with 35% error tolerance level.

In each case, the CRBP-OpCount and the RBP search algorithms were executed two times; one with the number of errors that were used to generate the distorted search strings, and other with small deviations. For example, in CaseA, the total number of errors introduced was 10%. Therefore, RBP with 10% and 15% error tolerance levels were used. Since the distorted search strings for CaseA were generated with 5% insertions and 5% substitutions, CRBP-OpCount with 5% deletions and 5% substitutions, and CRBP-OpCount with 5% deletions and 10% substitutions (small deviation from the actual constraints) error constraints were used. The constraints on edit operations and the error tolerances were chosen in such a way that the algorithms would generate at least one occurrence of each distorted search string.

Results and discussion

The output Tables 1, 2, 3, and 4 show the false positives and false negatives generated by the CRBP-OpCount and the RBP search algorithms for CaseA, CaseB, CaseC, and CaseD, respectively. The search patterns that are listed in these tables are the strings that had more than 1 occurrence or no match in one of the outputs of each case. Other search patterns generated only one occurrence, which was not useful to analyze the numbers of false positives and false negatives in the outputs. Therefore, they are not included in the tables. The "Total" in the Tables 1, 2, 3, and 4 is the total number of occurrences of the search patterns, "Search Patterns (N)" is the number of search patterns available in the tables, "False Negative (FN)" is the number of search patterns that were not matched in the search strings, and "False Positive (FP)" is the number of reported occurrences, which were not present in reality and it is given by $FP = Total + FN - N$.

Table 1: Selected outputs for CaseA (CRBP-OpCount with 5% deletions (E) and 5% substitutions (S), CRBP-OpCount with 5% deletions (E) and 10% substitutions (S), RBP with 10% errors, and RBP with 15% errors in the search strings (FP - False Positives; FN - False Negatives))

SN	**Search Patterns**	**CRBP-OpCount**		**RBP 10%**	**RBP 15%**
		E=5%, S=5%	E=5%, S=10%		
1	424e10000200	2	2	2	2
2	436f6e6e65637465642e	1	1	2	4
3	686f7374	4	4	2	2
4	70494e67	2	2	1	1
5	7068417365	2	2	1	1
6	72303074	3	3	3	3
7	72657774	3	3	1	1
8	424e000200	2	2	2	2
9	636f6e6e6563746564	3	3	4	4
10	636c69656e74	2	2	2	2
Total		**26**	**26**	**21**	**23**
Search patterns (N)		**10**			
FN		**0**	**0**	**0**	**0**
FP = Total + FN - N		**16**	**16**	**11**	**13**

Table 2: Selected outputs for CaseB (CRBP-OpCount with 10% insertions (I) and 5% substitutions (S), CRBP-OpCount with 10% insertions (I) and 10% substitutions (S), RBP with 15% errors, and RBP with 20% errors in the search strings (FP - False Positives; FN - False Negatives))

SN	**Search Patterns**	**CRBP-OpCount**		**RBP 15%**	**RBP 20%**
		I=10%, S=5%	I=10%, S=10%		
1	436f6e6e656374652e	1	1	5	5
2	0B000000070000436f6e656326	1	1		1
3	686f7374	1	1	2	2
4	72303074	2	2	3	3
5	76657273696e3A474f4c322e76	1	1		1
6	424e000200	2	2	2	2
7	636f6e6e65637464	3	3	5	5
8	636c696574	2	2	2	2
Total Matches		**13**	**13**	**19**	**21**
Search patterns (N)		**8**			
FN		**0**	**0**	**2**	**0**
FP = Total Matches + FN - N		**5**	**5**	**13**	**13**

Table 3: Selected outputs for CaseC (CRBP-OpCount with 10% insertions (I) and 10% substitutions (S), CRBP-OpCount with 15% insertions (I) and 10% substitutions, RBP with 20% errors, and RBP with 25% errors in the search strings (FP - False Positives; FN - False Negatives))

SN	**Search Patterns**	**CRBP-OpCount**		**RBP 20%**	**RBP 25%**
		I=10%, S=10%	I=10%, S=15%		
1	4e65744261	1	1	1	3
2	424e100061	1	1		2
3	436f6e6e6563746561	1	1	5	5
4	0B000000076100436f6e656174	1	1		1
5	32000000060061022447269767 36100	1	1		1
6	52656d6f74653A31756172656 36f6e3163746564746d6531	1	1		1
7	686f7374	1	1	1	2
8	4654504f4e	1	1	1	2
9	7068417365	1	1	1	3
10	72303074	2	2	3	3
11	7768303042	1	1	1	2
12	7665727369513A474f4c325131	1	1		1
13	424e000200	2	2	2	2
14	636f6e6e65637451	3	3	5	5
15	636c696551	2	2	2	2
Total		**20**	**20**	**23**	**35**
Search patterns (N)		**15**			
FN		**0**	**0**	**4**	**0**
FP = Total + FN - N		**5**	**5**	**12**	**20**

Table 4: Selected outputs for CaseD (CRBP-OpCount with 10% insertions (I) and 20% substitutions (S), CRBP-OpCount with 15% insertions (I) and 20% substitutions (S), RBP with 30% errors, and RBP with 35% errors in the search strings (FP - False Positives; FN - False Negatives))

SN	**Search Patterns**	**CRBP-OpCount**		**RBP 30%**	**RBP 35%**
		I=10%, S=20%	I=15%, S=20%		
1	4e65744271	1	1	3	3
2	424e100071	1	1	2	2
3	436f6e6e3563746535	1	1	5	5
4	0B00000035000043356e656335	1	1		1
5	686f7352	1	1	6	6
6	47697252	1	1	2	2
7	70494e6a	1	1	2	2
8	4654504f6a	1	1	2	2
9	72303040	2	2	6	6
10	72657740	1	1	4	4
11	7768303079	1	1	2	2
12	77616e79	1	1	3	3
13	766572736d6e3A476d4c322e6d	1	1		1
14	424e00026d	2	2	2	2
15	636f6e6d65637464	3	4	5	5
16	636c69656d	2	2	2	2
Total		**21**	**22**	**46**	**48**
Search patterns (N)		**16**			
FN		**0**	**0**	**2**	**0**
FP = Total + FN - N		**5**	**6**	**32**	**32**

From the Tables 2, 3, and 4, we can see that the RBP algorithm can generate false negatives (no occurrence of search patterns) if the error tolerance level (k) is not chosen

properly. The false negatives can be reduced by increasing the k but it may result in a greater number of false positives than with lower k. This is also true for the CRBP-OpCount algorithm. The Tables 2, 3, and 4 also show that the CRBP-OpCount algorithm is able to generate lower number of false positives compared to the RBP. It is because the CRBP-OpCount search algorithm can limit the number of constraints on specific edit operations, which is not possible in the case of the RBP unconstrained approximate search algorithm.

Fig. 2 shows the plots for the false positives obtained by the CRBP-OpCount1, CRBP-OpCount2, RBP1, and RBP2 algorithms in each case. CRBP-OpCount1 is a CRBP-OpCount algorithm with constraints on 5% deletions and 5% substitutions in CaseA, 10% insertions and 5% substitutions in CaseB, 10% insertions and 10% substitutions in CaseC, and 10% insertions and 20% substitutions in CaseD. CRBP-OpCount2 is a CRBP-OpCount algorithm with constraints on 5% deletions and 10% substitutions in CaseA, 10% insertions and 10% substitutions in CaseB, 10% insertions and 15% substitutions in CaseC, and 15% insertions and 20% substitutions in CaseD. RBP1 is a RBP algorithm with 10% error tolerance level in CaseA, 15% error tolerance level in CaseB, 20% error tolerance level in CaseC, and 30% error tolerance level in CaseD. RBP2 is the RBP with 15% error tolerance level in CaseA, 20% error tolerance level in CaseB, 25% error tolerance level in CaseC, and 35% error tolerance level in CaseD.

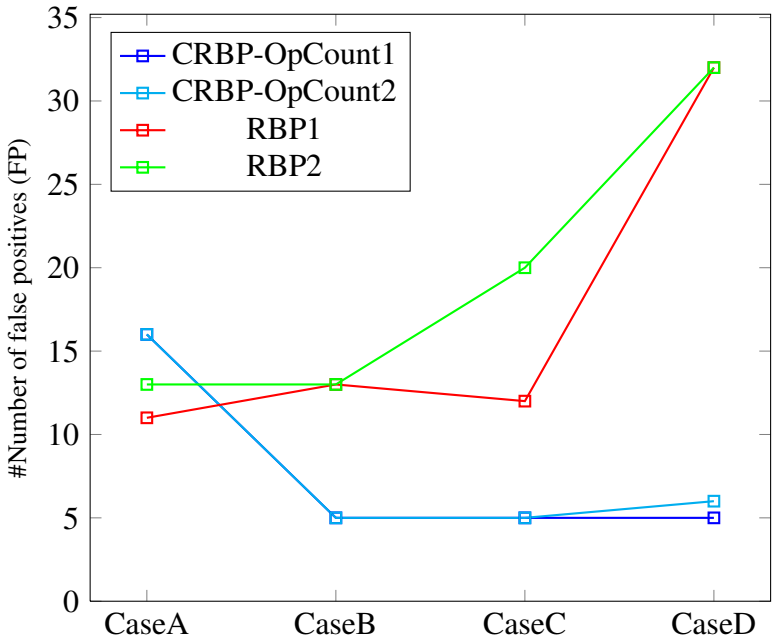

Fig. 2: Number of false positives obtained by the CRBP-OpCount1, CRBP-OpCount2, RBP1, and RBP2 approximate search algorithms in CaseA, CaseB, CaseC, and CaseD.

The plots for the execution time taken by the CRBP-OpCount1, CRBP-OpCount2, RBP1, and RBP2 algorithms are provided in Fig. 3. These are the execution times taken by each algorithm for matching 61 search patterns with 52 distorted search strings. It shows that there is not much difference in the execution time of the CRBP-OpCount and the RBP search algorithms for a lower number of errors. However, the execution time has been increased for the larger numbers of errors. The rate of increase in the execution time is higher in the case of the CRBP-OpCount algorithm than the RBP unconstrained approximate search algorithm.

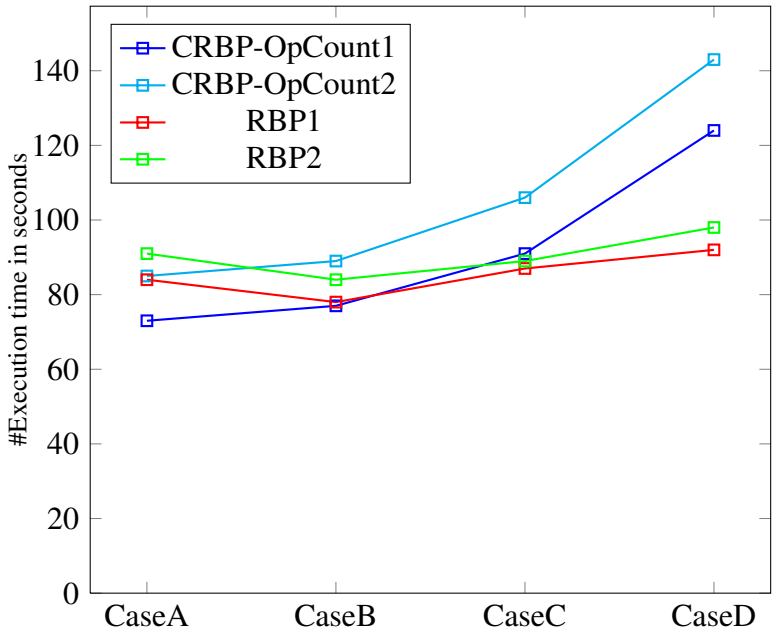

Fig. 3: Execution times taken by the CRBP-OpCount1, CRBP-OpCount2, RBP1, and RBP2 algorithms for matching 61 search patterns and 52 distorted search strings.

5 Conclusion

In this paper, we propose a constrained approximate search algorithm for application in misuse-based Intrusion Detection Systems (IDS). The algorithm introduces constraints on the individual numbers of change/edit operations in order to take into account a-priori knowledge about the habits of the attackers and the algorithms that they use when they transform the known attack patterns in order to pass unnoticed by the IDS. The use of a-priori knowledge about attackers is justified by the fact that it is not possible to apply just any change on the known attack traffic pattern in order to obtain a new attack. The attacker must limit the number of edit operations in this process otherwise the produced traffic might behave in an unpredictable way (for example, it might not be harmful at all). With the constraints incorporated in the approximate search algorithm, we managed to reduce the numbers of false positives and false negatives generated by the IDS significantly compared to the case when an approximate search algorithm without constraints was used on the same input data. In addition, by using bit-parallelism inherent to commonly used computers in the constrained approximate search algorithm together with a set of counters of allowed individual edit operations assigned to each state bit of the simulated NFA, we managed to keep the execution time needed to detect intrusions at a level low enough for efficient operation of the IDS, especially for the lower error rates.

References

[1] Snort. https://snort.org/. [Accessed: jan 2016].

[2] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, October 1992.

[3] Nguyen Le Dang, Dac-Nhuong Le, and Vinh Trong Le. A new multiple-pattern matching algorithm for the network intrusion detection system. *IACSIT International Journal of Engineering and Technology*, 8(2):94–100, April 2016.

[4] Simone Faro and Thierry Lecroq. Twenty years of bit-parallelism in string matching. In J. Holub, B. W. Watson, and J. Žďárek, editors, *Festschrift for Bořivoj Melichar*, pages 72–101. 2012.

[5] Sung il Oh, Min Sik Kim, and Inbok Lee. An efficient bit-parallel algorithm for ids. In *In Proc. of RACS 2013*, pages 43–44, 2013.

[6] J. Kuri and G. Navarro. Fast multipattern search algorithms for intrusion detection. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 169–180, 2000.

[7] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, USA, 2002.

[8] Lin Tan and Timothy Sherwood. Architectures for bit-split string scanning in intrusion detection. *IEEE Micro*, 26(1):110–117, 2006.

[9] Sun Wu and Udi Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, October 1992.