computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

DEVELOPING A FASTER PATTERN MATCHING ALGORITHMS FOR INTRUSION DETECTION SYSTEM

Ibrahim Obeidat [1], Mazen AlZubi [2]

[1] The Hashemite University, Jordan, imsobeidat@hu.edu.jo
[2] Zayed University, United Arab Emirates, zoubicom@gmail.com

Paper history:
Received 25 February 2019
Received in revised form 18 July 2019
Accepted 10 September 2019
Available online 30 September 2019

Keywords:
Algorithms;
Pattern Matching;
Boyer-More;
Aho-Corasick;
Rabin Karp;
Knuth-Morris-Pratt;
IDS: Intrusion Detection System;
signature based malicious.

Abstract: Fast pattern matching algorithms mostly used by IDS, which are considered one of the important systems used to monitor and analyze host and network traffic. Their main function is to detect various types of malicious and malware files by examining incoming and outgoing data through the network. As the network speed growing, the malicious behavior and malware files are increasing; the pattern matching algorithms must be faster. In this research paper we are presenting a new method of pattern matching, which could be a platform for enhancement in the future. In this field, researchers spared no efforts to introduce fast algorithms for pattern matching. The Most popular algorithms are Boyer-Moore, Aho–Corasick, Naïve String search, Rabin Karp String Search and Knuth–Morris–Pratt. Based on studying these techniques we are developing algorithms that process the text data, using different algorithm technique and then we'll test the performance and compare the processing time with the fastest proven pattern matching algorithms available. Document the result and draw the overall conclusion.

1. INTRODUCTION

The rapid growth in information technology and specifically the hardware capabilities brings a new challenge to security. Intrusion detection system must perform to the optimal level to detect signature based malicious behavior. The technique behind the detection system is the algorithms being implemented. The algorithm is meant to process text. Pattern matching is to locate specific pattern in a raw data. The faster the compute system the more efficient the pattern matching needs to be. Therefore, there is a collective demand to bring a faster pattern-matching algorithm to keep pace with hardware rapid development, performance improvement and innovations. On the other hand, the accuracy and the efficiency need to be maintained using such algorithms. We are developing a new text processing technique and we'll measure its performance. The algorithm will index the text string and create an array of similar character indexes. The arrays of the characters will be responsible to assemble the pattern need to be matched or detected. We will conduct comparison with two of the fastest patterns matching algorithms to date. They are Aho-Corasick and Boyer-Moore. If the performance is less, then it might be considered as new algorithm where further research and enhancement could be done.

2. RELATED WORK

2.1 OVERVIEW

Pattern matching algorithms also called a string searching algorithms are class of strings algorithms that processes a large number of or text to find a place of patterns. String matching consists in finding one, or more generally, all the occurrences of a pattern in a text. The pattern and the text are both strings built over a finite alphabet. In several applications, texts need to be structured before searched. Even if no further information is known on their syntactic structure, it is possible and indeed extremely efficient to build a data structure that supports searches [1]. Because the data of

monitoring system updates constantly and the size of data expands day by day, the financial institutions have to spend much time in matching the database of account holders and clients' transaction. The earliest algorithm of matching single pattern is the algorithm of Brute-Force (the algorithm of BF), which is algorithm of matching in order and the efficiency is low. In 1977, Knuth D.E, Morris J.H and Pratt V.R proposed an algorithm of matching single pattern, the algorithm of KMP, which eliminate the problem of the comparison of backtracking. In the same year, Boyer and Moore proposed the algorithm of BM [2], which can skip by the rule of bad character and good postfix. The algorithm of BM performs efficiently in the algorithm of matching single pattern. In 1975, Aho and Corasick proposed the algorithm of AC [3], which uses the finite state machine to match strings and can match all the pattern strings by scanning text strings once. However, the algorithm of matching single pattern scans text strings once, and it only can match one pattern string. In 1993, Fan Jang-Jong proposed the algorithm of AC-BM [4], which uses the idea of the finite state machine of AC in preprocessing and uses the idea of skipping of BM in scanning. The algorithm highly improves the traditional the algorithm of AC in the part of scanning and matching.

We are developing an algorithm on that bases. Boyer Moore and Aho Corasick are the best efficient pattern matching algorithms. A lot of variations introduced and built based on the string search concepts and techniques implemented in these algorithms. Our main focus will be on these two algorithms hence they are the fastest pattern matching available.

2.2 BOYER-MOORE

BM algorithm (Boyer-Moore) [2] is a kind of matching algorithm based on postfix matching backwards from right side to left side is a distinctive characteristic of BM algorithm. When there is a character that fails to match in text string T, $T[i] = c$, it will skip in the text string to speed up the pattern string moving. If c is a string in P, finding out the rightmost c in P and then moving P to make $T[i]$ align it; if c isn't a string in P, moving P to make $P[0]$ align $T[i+1]$.The fundamental of BM algorithm: matching one character at a certain distance in T, and then determining whether skip to the right side and the distance of skipping according to the character, which is matched at present, whether appears in P or appears in which location. If it doesn't skip, it will compare with P from the right to the left in the present location. Otherwise it will skip to the next location of P according to the skipping distance that has been computed in advance the algorithm also includes two parts, preprocessing and scanning. In the part of preprocessing, it just considers the pattern string p and the set of strings Σ. We introduce a function of shift (c). It shows the situation that every character c appears in pattern string P, and the distance that the pattern string moves. The function value of shift is also saved in an array the following is a method of computing the array of shift. If c appears in the pattern string P, find out the location of c in the rightmost of P, and the location is index. The value of function is the distance between the c of rightmost and the rightmost of P, it is m-1-index; if c doesn't appear in the pattern string P, the value is the length of P. For example, $\Sigma=\{a,b,c,d\}$ and the pattern string $P = abacab$, we can compute it the Shift(a)=1, Shift(b)=0, Shift(c)=2, Shift(d)=6. And the time complexity of shift is $O(m+|\Sigma|)$. Although the average efficiency of BM algorithm is high, it doesn't performance well in the worst case. The complexity of the worst case is $O(mn)$, such as, a text string $=aaa\ldots a$, a pattern string $P=baa\ldots a$. BM algorithm is designed as a kind of algorithm that matches the single pattern string in text. Among the algorithms of matching single-pattern, BM algorithm is proved to performance best. However, when there are various kinds of key words to match in the filtering and matching of content, the BM algorithm has to match every kind of pattern. The time complexity of the BM algorithm is $O(n)$ when matching the single pattern, but it is $O(kn)$ when matching the multiple patterns.

2.3 AHO-CORASICK ALGORITHM

AC algorithm (Aho-Corasick) [3] is a kind of algorithm that based on finite state machine. It preprocesses the set of pattern strings to form a state machine in a tree before start matching. It just scans the text string T once, and then it can find out the all patterns that match with T in P. The algorithm also includes two parts, preprocessing and scanning. It generates three functions: goto, failure function: and output function. The following is the matching process of AC algorithm: starting from state zero, picking up a character from the text string, and then going to the next state with goto function and the failure function. When the output function of some state is not a null, it means that it successes to find

out the pattern string. It structures the finite state machine in the part of preprocessing, and the time complexity of structuring transition function is $O(M)$, M is the total length of all patterns. The time complexity of structuring the failure function is $O(M)$, too. In the part of scanning, it scans every character of the text T on the basis of the finite state machine that has structured before. Every character just has one transition function, and the time complexity of scanning is $O(n)$. So, the total time complexity of the algorithm that based on the finite state machine is $O(M+n)$. The time complexity is related to the length of text and pattern but is not related to the content of text and pattern. It means that the average time of scanning in the best case or the worst case is the same, $O(M+n)$.

The disadvantage of AC algorithm is that the demand of space is large. Too many matching patterns will occupy plenty of space; even maybe make the system crash. So, AC algorithm can satisfy the demand and performances well in the case of a few patterns.

However, it doesn't skip when it scans the text. It inputs the text in order, which means it can't skip the unnecessary comparison. Obviously, AC algorithm is not the best matching algorithm impractical process of matching.

2.4 RABIN-KARP ALGORITHM

Rabin-Karp Algorithm is the simplest string searching algorithm. This algorithm was developed by Michael O. Rabin and Richard M. Karp in 1987. This algorithm uses the hash function to discover the potential pattern in the input text. For the length of text n and pattern p of mutual length m, its average and best-case running time is $O(n+m)$ in space $O(p)$, and also the worst-case time is $O(nm)$ in space $O(m)$ [5]. It is used to discover the hash value of the certain pattern substring and then it discovers the hash value of all possible m length substring of the input text. If the hash value of the pattern and text substring match than it returns the value otherwise next substring value is matched to calculate the string of length m.

2.5 KNUTH-MORRIS-PRATT ALGORITHM

The Knuth–Morris–Pratt were developed a linear time string searching algorithm by analysis of the brute force algorithm or naïve algorithm. The algorithm was developed in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris and they published it jointly in 1977.The Knuth-Morris-Pratt algorithm moderates the total number of comparisons of the pattern against the input string [6]. A matching time of $O(n)$ is accomplished by evading associations with essentials of 'S' that have earlier been involved in the comparison with some of the specific element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' certainly not occurs [7].

Components of KMP algorithm Include 1. The prefix function Π for a pattern summarizes the knowledge regarding however the pattern matches in contradiction of shifts of itself. This information may be accustomed avoid unusable shifts of the pattern "p". In other words, this succeeds avoiding backtracking on the string "S".2. The KMP Matcher With string "S", pattern "p" and pre-fix function "Π" as inputs, the prevalence of "p" in "S" is found and the algorithm yields the variety of shifts of "p" after which the existence is found. 3. Running - time analysis: The time period for computing the prefix function is $O(m)$ and time period of matching function is $O(n)$.

3. PROPOSED SOLUTION

3.1 METHODOLOGY

Our approach was studying multiple string-matching algorithms techniques and come up with new idea. We are introducing a function that will process the data by creating arrays of indexes. We create array for each character and store its index, if the character occurred more than once within the text then it's index will be added in the character array that belongs to. Then we will try to find the pattern required to match based on the characters indexes search within the indexes array created. We'll compare the performance of our function against Knuth–Morris–Pratt algorithm and document the result.

3.2 CONCEPT

Knowing the techniques of the string search, like Boyer Moore, which use rules to shift the character, and Aho-Corasick, which create automata of characters and links, we are building something similar. The concept behind our function is very simple, and it is working as following:

Stage One:

Store All Characters in the String provided into a Temporary array in small Letters:

CharArray (M) = [], where M = Length(string)

Example: Suppose we have the following string "intrusion detection system", M=26

So, we can represent this string inside the array as shown in Fig. 1:

Figure 1 – String inside the CharArray

So,
CharArray (0) = "*i*", CharArray (1) = "*n*",
CharArray (2) = "*t*", CharArray (3) = "*r*",
CharArray (4) = "*u*", CharArray (5) = "*s*",
CharArray (6) = "*i*", CharArray (7) = "*o*",
CharArray (8) = "*n*". Until end of text.

And the pattern that we wanna to find inside this string is the word (system):

Pattern = "system", Where *N* = Length (Pattern)

Stage Two:

For Each Character in Pattern, Locate the index of this character in the text and save the indexes into Temporary Grid or Array as follow:

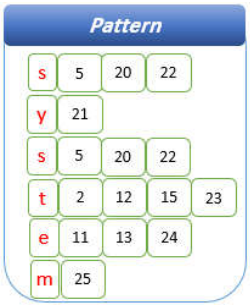


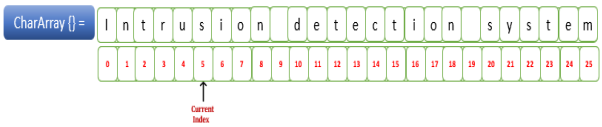
Figure 2 – Locate pattern letters indexes inside the string

Note: Please note that when the letter is repeated in the pattern no need to create a new row for the repeated letter because it indexes was calculated and saved with the array by the first time this letter found in the pattern.

Stage Three

Now we will start the algorithm search as follow:

One of the features of our algorithm is it go's directly to the first character in the pattern as shown below, it creates 2 indexes, first one is called **(Current Index)** this index holds the index of the first character inside the pattern. Second index called **(Next Index)** and its value = Current_Index + 1



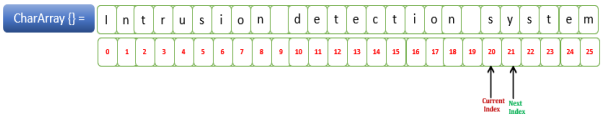
Current Index = 5, First character in the pattern

Figure 3 – First Character Search in the pattern

Explanation:

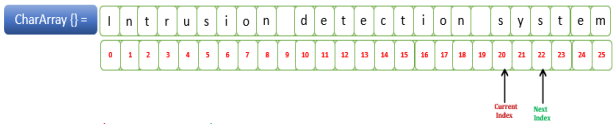
Logically, when searching for a pattern, this pattern has a special signature, when we analyzed this signature, we find that the letters or numbers that make up this pattern will be in order, for example if we have this pattern: (system), then for example if the letter (s) starts at cell number 25 so the next letter will be exactly in cell number 26 then next letter in cell number 27 and so on.

Now, return back to algorithm Since Current_Index in position 5 and hold the letter (*s*), and Next_Index in position 6 and hold the letter (*i*) which is not apart from our pattern then the algorithm cancel the current search and go's directly to next position index in letter (*s*) array which is (20).



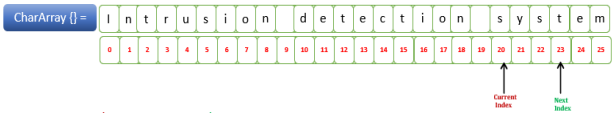
Current Index = 20, Next Index = 21

Since Current_Index in position 20 and hold the letter (*s*), and Next_Index in position 21 and hold the letter (*y*) which is apart from our pattern then Current_Index stay at position 20 and Next_Index will be incremented by 1, so in our case Next_Index = 22.



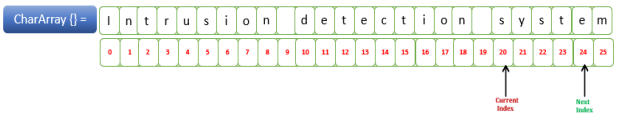
Current Index = 20, Next Index = 22

Since Next_Index in position 22 and hold the letter (*s*) which is apart from our pattern and letter (*s*) is the next letter in the pattern, then Current_Index stay at position 20 and Next_Index will be incremented by 1, so Next_Index = 23.



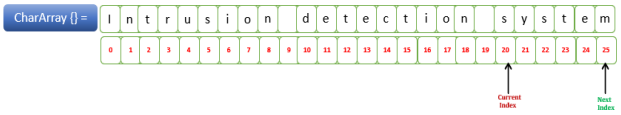
Current Index = 20, Next Index = 23

Since Next_Index in position 23 and hold the letter (*t*) which is apart from our pattern and letter (*t*) is the next letter in the pattern, then Current_Index stay at position 20 and Next_Index will be incremented by 1, so Next_Index = 24.



Current Index = 20, Next Index = 24

Since Next_Index in position 24 and hold the letter (*e*) which is apart from our pattern and letter (*e*) is the next letter in the pattern, then Current_Index stay at position 20 and Next_Index will be incremented by 1, so Next_Index = 25.



Current Index = 20, Next Index = 25

Since Next_Index in position 25 and hold the letter (*m*) which is apart from our pattern and letter (*m*) is the next letter in the pattern, then Current_Index stay at position 20 and Next_Index will be incremented by 1, so Next_Index = 25.

Since the Length of our pattern is 6 and we found our pattern inside the string given, so our algorithm exits the search and return the value (1) indicates that the pattern is found.

Some tips and features for proposed algorithm:

- At any time if the Next_Index will not be matching next character in the pattern then, the algorithm will move the Current_Index into the next position of the letter (S) which is position (22) and set Next_Index = 23 and starts searching again.
- Also, for saving the time of searching, if the remaining characters in the text > length(pattern) then the algorithm will exit the search and return the value of (0) indicates that the pattern given is not found.
- Another feature for our algorithm, it helps to locate each character index in a given text means that we can use it in many cases like (count each character inside the text, replace the characters ...).

In the below figure we create the algorithm in graphical user interface and save 10 patterns in algorithm database and enter a text with length of (452) Letters we run the program to find the patterns, it takes (2.419) seconds to find the (10) patterns inside the given text. And if you want to know where these patterns are located inside the given text just double click on the pattern in the yellow grid and the pattern will be highlighted in the given text.

Another feature for this algorithm it stores all occurrence of the pattern inside the given text not only the first occurrence.

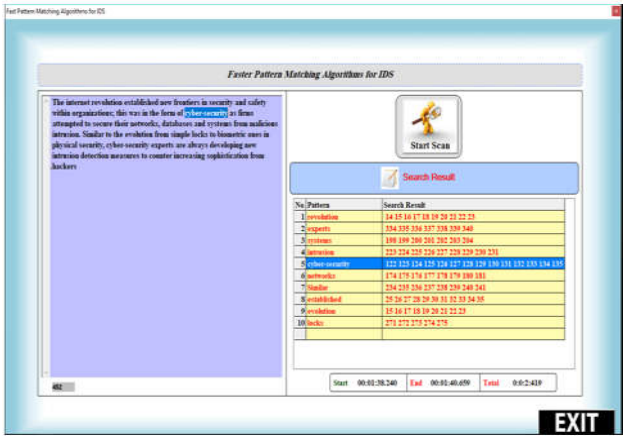


Figure 4 – Results for the algorithm for 10 Patterns

3.3 TIME COMPLEXITY

We tested the new algorithm by matching 5, 10, 15 and 20 Patterns and we conclude that our algorithm has two cases, case one: best case ($O(n)$), case two: worst case ($O(n+m)$) the following Graph shows the result of time complexity:

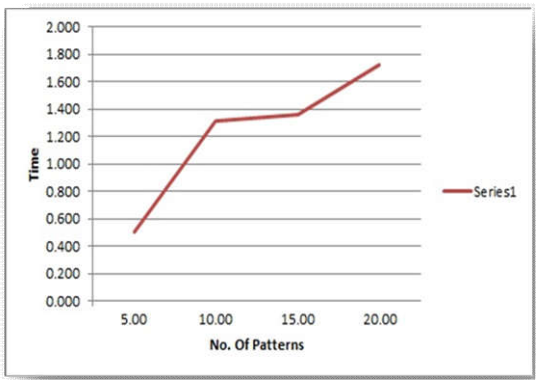


Figure 5 – Time Complexity for our Algorithm

3.4 COMPARISON BETWEEN AC AND BM AND OUR ALGORITHM

The following graph shows the time complexity for AC and Booyer-More Algorithms:

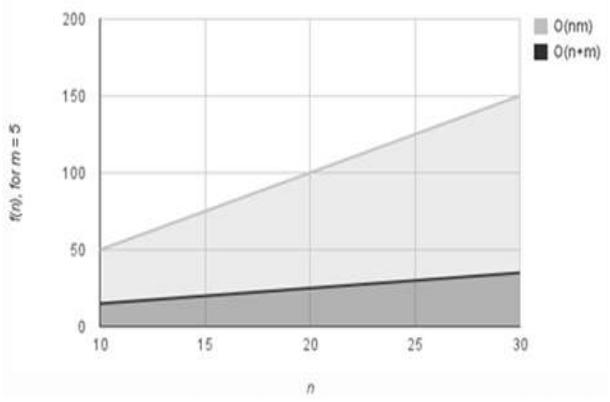


Figure 6 – Time Complexity for BM Algorithm [6]

Boyer-Moor its worst-case complexity is $O(n+m)$. The thing is that in natural language search Boyer-Moore does pretty Well.

4. CONCLUSION

There are Numerous basic string processing techniques, a few of them are called varieties, they are presented from well-known matching algorithms, after numerous tests and comparisons, we conclude that the performance of our algorithm is superior than the Aho-Corasick and Booyer-More algorithms. Within the future we are going do improvements and enhancing on our algorithm to come up with a better matching technique and faster than the current one.

5. REFERENCES

[1] M. Crochemore, T. Lecroq, "Pattern-matching and text-compression algorithms," *ACM Computing Surveys (CSUR)*, vol. 28, issue 1, pp. 39-41, March 1996.

[2] R.S. Boyer, J.S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, issue 10, pp. 762-772, 1977.

[3] A. Aho, M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, issue 6, pp. 333-340, 1975.

[4] J. Fan, K. Su, "An efficient algorithm for matching multiple patterns," *IEEE Transactionson on Knowledge and Data Engineering*, vol. 5, issue 2, pp. 339-351, 1993.

[5] S.K. Shivaji, Prabhudeva S., "Plagiarism detection by using Karp-Rabin and string matching algorithm together," *International Journal of Computer Applications*, vol. 116, issue 23, pp. 1–5, 2015.

[6] S. Wahlstrom, "Evaluation of string searching algorithms," *Proceedings of the IDT Mini-Conference on Interesting Results in Computer Science and Engineering*, 2013, pp. 1-6.

[7] G. Behera, "Novel pattern matching algorithm in genome sequence analysis," *International Journal of Computer Science and Information Technologies*, vol. 5, issue 4, pp. 5450–5457, 2014.

[8] E. Silva de Moura, G. Navarro, N. Ziviani, R. Baeza-Yates, "Fast and flexible word searching on compressed text," *ACM Transactions on Information Systems*, vol. 18, issue 2, pp. 113-139, 2000.

[9] S. Hasib, M. Motwani, A. Saxena, "Importance of Aho-Corasick string matching algorithm in real world applications," *International Journal of Computer Science and Information Technologies*, vol. 4, issue 3, pp. 467-469, 2013.

[10] M. Alicherry, M. Muthuprasanna,V. Kumar, "High speed pattern matching for network IDS/IPS," *Proceedings of the 2006 IEEE International Conference on Network Protocols*, 12-15 Nov. 2006, pp. 187-196.

[11] A. Corasick, The Life n Photo, 2008, [Online]. Available at: https://cskane.wordpress.com/2008/07/23/aho-corasick-trie

[12] K. Namjoshi, G. Narlikar, "Robust and fast pattern matching for intrusion detection," *Proceedings of the 2010 IEEE International Conference INFOCOM*, 14-19 March 2010, pp. 1-10.

[13] P. Pandiselvam, T. Marimuthu, R. Lawrance, "A comparative study on string matching algorithms of biological sequences," 2013, [Online]. Available at: https://arxiv.org/ftp/arxiv/papers/1401/1401.7416.pdf

[14] M. Kharbutli, M. Aldwairi, A. Mughrabi, "Function and data parallelization of Wu-Manber pattern matching for intrusion detection systems," *Network Protocols and Algorithms*, vol. 4, no. 3, pp. 46-61, 2012.

[15] M. Dubiner', Z. Galilt, E. Magen, "Faster tree pattern matching," *Journal of the ACM*, vol. 41, issue 2, pp. 205-213, March 1994.

[16] S. Doria, G. Landau, "Construction of Aho Corasick automaton in linear time for integer alphabets," 2012, [Online]. Available at: http://cs.haifa.ac.il/~landau/gadi/shiri.pdf

[17] R. Bhukya, and D. V. L. N. Somayajulu, "Exact multiple pattern matching algorithm using DNA sequence and pattern pair," *International Journal of Computer Applications*, vol. 17, issue 8, pp. 32-38, 2011.

[18] A. Ziad, M. Aqel, I. Emary, "Multiple skip multiple pattern matching algorithm (MSMPMA)," *IAENG International Journal of Computer Science*, vol. 34, no. 2, pp. 14-20, 2007.

[19] H. Gharaee, S. Seifi, and N. Monsefan, "A survey of pattern matching algorithm in intrusion detection system," *Proceedings of the 7th IEEE International Symposium on Telecommunications (IST)*, 2014, pp. 946-953.

[20] C.-H. Lin, et al., "Accelerating pattern matching using a novel parallel algorithm on GPUs," *IEEE Transactions on Computers*, vol. 62, issue 10, pp. 1906-1916, 2013.

[21] R. M. Karp, M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, issue 2, pp. 249-260, 1987.

[22] R. Ehtesham, M. El-Kharashi, F. Gebali, "A fast string search algorithm for deep packet classification," *Computer Communications*, vol. 27, issue 15, pp. 1524-1538, 2004.

[23] L. Colussi, "Correctness and efficiency of the pattern matching algorithms," *Information and Computation*, vol. 95, issue 2, pp. 225-251, Dec. 1991.

[24] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, "Speeding up two string matching algorithms," *Algorithmica*, vol. 12, no. 4-5, pp. 247-267, 1994.

Ibrahim M. Obeidat, *Ph.D. in Computer science, Associate Professor in of computer science at the Computer science department, the Hashemite University. Current research interests include Computer Networks, Cyber Security.*

Mazen I. Alzoubi*, Master Degree of Cyber Security at Zayed University. Current interests include: Computer Networks, Cyber Security.*