Application of Boyer-Moore and Aho-Corasick Algorithm in Network Intrusion Detection System

Kezia Suhendra / 135150631
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
113515063@std.stei.itb.ac.id

***Abstract*—Nowadays, everyone is using networks to do almost all of their works that caused in the increasing number of hackers and intruders who want to intrude the computer's network system. Many hackers and intruders have made tons of successful attempts in bringing down company networks and web services. All of these are showing that security is a big issue for all networks in today's enterprise environment. A lot of methods have been used to defend the networks such as encryption, VPN, firewall, etc. Intrusion detection method can be considered as a new method to detect the anomaly or suspicious activity both at the network and host levels. String matching is the bottleneck of performance for the network intrusion detection system. Thus, it needs to use the most effective string-matching algorithm to be implemented in the system. There are Boyer-Moore and Aho-Corasick algorithm that will be discussed in this paper. This paper discusses the effectiveness of each algorithm in network intrusion detection system.**

Keywords—network intrusion detection system, Boyer-Moore, Aho-Corasick.

I. Introduction

Intrusion detection is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or security standard practices.[1] Intrusion detection system can find out about the intrusion when there are data packets that contain anomalies related to Internet protocols. Intrusion detection system will find out about the log suspicious activity by finding suspicious patterns known as signatures or rules and alert the system about that activity. All the malicious code will be stored in the database and each of the incoming data is compared with the stored data. Intrusion detection system can be divided into two categories: signature-based and anomaly detection system. For instance, if it is observed that a particular TCP connection requests a connection to a large number of ports, then it can be assumed that there is someone who is trying to conduct a port scan of most of the computers of the network. [2]

The attacks not always come from outside the monitored network, they can also come from the inside of the monitored network. This attack can be called as trusted host attack. The network intrusion detection system uses string-matching algorithm to compare the payload of the network packet with the pattern entries of the intrusion detection rules. This is why string matching is the important part of every network intrusion detection system. In fact, the performance of all network intrusion detection systems depends almost entirely on the performance of the string-matching algorithm. That's why the network intrusion detection system needs the most effective string matching to run in the system. The quicker the execution time, the faster the intrusion will be found by the system.

There are many other things that network intrusion detection system can provide such as backing up firewalls, controlling file access, controlling the administrator's activities, protection against viruses, detecting unknown devices, detecting default configurations, and many more. Snort is an open source intrusion detection system available to the general public for free (open source). Snort is capable to track packet logging and analyze real time traffic on IP networks. Snort can perform all the basic functions that are needed in network intrusion detection system such as protocol analysis and content matching. It is a cross-platform, lightweight intrusion detection system, which can be deployed on a variety of platforms to monitor TCP/IP networks and detect suspicious activity. [3]

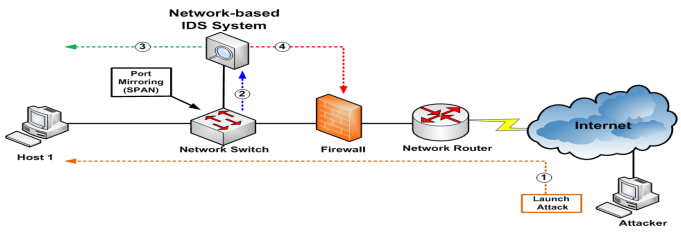

Figure 1 - NIDS Basics
(Source: http://s10.postimg.org/4467u4oc9/mod8_fig1.png)

II. Theories

A. Boyer-Moore Algorithm

In Boyer-Moore algorithm, the characters of pattern are matched starting from the last character of pattern to the first character of pattern (from right to left). Let the string has n characters and the pattern has m character(s). If there is a mismatch between the character of the pattern and the character of the string, the character in the string will be searched within the rest of the characters that haven't been matched. If there is no occurrence of the character in the pattern, then all of the characters of pattern will be shifted m character long towards the next character in the string. But, if there is an occurrence of

the character in the pattern, then the exact same character in the pattern will be shifted towards that certain character in the string. This algorithm will be more effective than ever if the alphabet character is large and slow when the alphabet is small.

There are two methods in Boyer-Moore algorithm such as The Looking-Glass Technique and The Character-Jump Technique. Let the string named as s and the pattern named as w. The looking-glass technique does the comparison from the last character of w with one of the character in s. If it is a match, then the next comparison starts from the character on the left of the last character in w. W will be shifted until all of the characters in s has been compared. Meanwhile, let p as the character in the string s that mismatches with the character in w. The character-jump technique does an action when there is a mismatch between the character in s and in w. The action that will be taken based on the occurrence of p in w. There will be several types of action such as finding the exact character that match and align that character with p or shift all of the characters from the last comparison.

Boyer-Moore algorithm preprocesses the pattern and the string to build the last occurrence function. The last occurrence function will be represented in a table that consists of the index where the last occurrence of the character in string occurs in the pattern. If there is no such character in the pattern then fill the table with -1. Boyer-Moore worst case running time is O(nm + A) and the best case running time is O(n/m). Clearly, Boyer-Moore algorithm is faster than Brute Force algorithm.

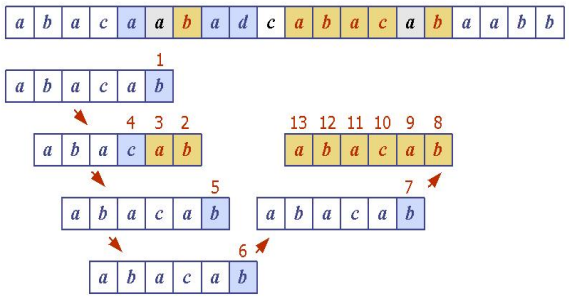

Figure 2.1 - Boyer-Moore String Example
(Source: https://koding4fun.files.wordpress.com/2010/05/complete_example.jpg)

Below is the implementation of Boyer-Moore algorithm in C#.

```
public static int bmMatch(string text, string pattern)
        {
            if (pattern.Length == 0)
            {
                return -1;
            }

            int[] charTable = makeCharTable(pattern);
            int[] jumpTable = makeJumpTable(pattern);

            for (int i = pattern.Length -
 1, j; i < text.Length;)
            {
                for (j = pattern.Length - 1;
pattern[j] == text[i]; --i, --j)
                {
                    if (j == 0)
                    {
                        return i;
                    }
                }
                i += Math.Max(jumpTable
[pattern.Length - 1 - j], charTable[text[i]]);
            }

            return -1;
        }

        private static int[] makeCharTable(string pattern)
        {
            const int ALPHABET_SIZE = Char.MaxValue + 1;
            int[] table = new int[ALPHABET_SIZE];
            for (int i = 0; i < table.Length; ++i)
            {
                table[i] = pattern.Length;
            }
            for (int i = 0; i < pattern.Length - 1; ++i)
            {
                table[pattern[i]] = pattern.Length - 1 - i;
            }
            return table;
        }

        private static int[] makeJumpTable(string pattern)
        {
            int[] table = new int[pattern.Length];
            int lastPrefixPosition = pattern.Length;
            for (int i = pattern.Length - 1; i >= 0; --i)
            {
                if (isPrefix(pattern, i + 1))
                {
                    lastPrefixPosition = i + 1;
                }
                table[pattern.Length - 1 - i] =
lastPrefixPosition - i + pattern.Length - 1;
            }
            for (int i = 0; i < pattern.Length - 1; ++i)
            {
                int slen = suffixLength(pattern, i);
                table[slen] = pattern.Length - 1 - i + slen;
            }
            return table;
        }

        private static bool isPrefix(string pattern, int p)
        {
            for (int i = p, j = 0; i < pattern.Length; ++i, ++j)

            {
                if (pattern[i] != pattern[j])
                {
                    return false;
                }
            }
            return true;
        }

        /**
         * Returns the maximum length of the substring ends at p
and is a suffix.
         */
        private static int suffixLength(string needle, int p)
        {
            int len = 0;
            for (int i = p, j = needle.Length - 1;
                    i >= 0 && needle[i] == needle[j]; --i, --
j)
            {
                len += 1;
            }
            return len;
        }
}
```

B. *Suffix Trees*

A suffix tree is a compressed dynamic tree or prefix tree (trie) that contains all of the suffixes of the given text as their keys and positions in the text as their values. The suffix tree is an important data structure used for string-matching algorithm. Suffix tree stores string in a unique way so that common string will be stored only once. The suffix tree for string named as s with length n has exactly n leaves numbered 1 to n. Each edge of the tree will be labeled as a non-empty substring of s. There are no two edges that have string labels beginning with the same character and all the internal nodes have two children with the possible exception of the root.

The suffix tree of a string named as s with length n can be build in O(n) time. Meanwhile, the string searching operation of a substring with length m can be done in O(m) time after the suffix tree has been built. The greatest strength of suffix tree is

the ability to search efficiently with mismatches. The suffix tree can be implemented in many ways such as hash map, unsorted array, sorted array, and balanced search tree. Below is the example of suffix tree for string "BANANAS".

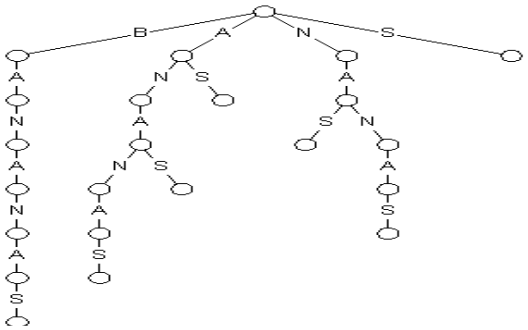

Figure 2.2 - Suffix Tree Example
(Source: http://marknelson.us/attachments/1996/suffix-trees/FIGURE1.gif)

From figure 2.2, can be seen that the tree contains all of "BANANAS" suffixes. All of the suffixes include the word "BANANAS" itself, "ANANAS", "NANAS", and so on until the suffix that left is the alphabet S.

C. Aho-Corasick Algorithm

Aho-Corasick algorithm is a string-matching algorithm that locates elements of a finite set of string within an input text. Basically, this algorithm is a kind of dictionary-matching algorithm because it matches all strings simultaneously. This algorithm can reduce the time taken for searching because it is capable to match all the strings at once. On the other hand, the other algorithm such as Boyer-Moore can match only one pattern at a time.

This algorithm constructs a dynamic tree or trie using suffix tree like structure for all of the patterns that need to be matched. The trie that had been constructed resembles a finite state machine with additional links between the various internal nodes representing a pattern to the node containing the longest proper suffix. These links make the transition between the failed string matches to other branches of the trie, which have a common prefix. It means there is no need to do a backtracking during the transition between string matches.

The trie will be needed during the run time for matching purposes. When the string dictionary is known in advance, the trie can be constructed offline and then the trie can be used for real time searching with network data. The suffix tree only need to be reconstructed when there is addition or change to the pattern in the set of rules. In case of the trie is known beforehand, the execution time will be linear based on the length of the input plus the number of matched entries.

The difference between this algorithm and Boyer-Moore algorithm in case of shifting is that in Boyer-Moore a single pattern is sliding along the text, meanwhile in this algorithm slides the trie while performing the shifting techniques. In this algorithm there are two types of shifting such as bad character shift and good prefix shift.

In bad character shift, if there is a mismatch occurs then the suffix tree is shifted to align to the next occurrence of the character in some other pattern that hasn't been checked in the tree. The length of the smallest suffix in the tree will shift the window if there is no match of a particular character in the pattern available. There is no backtracking in this algorithm.

On the other hand, in good prefix shift there is two kind of shift, the first one is allowing the window to be shifted to the next occurrence of a complete prefix that has already been encountered as a substring of some other pattern. The second one is shifting the window to the next occurrence of some prefix of the correctly matched text as the suffix of some other pattern in the tree.

The performance of this algorithm depends on the suffix tree's performance. Implementing the right hash set of the suffix tree can enhance the performance of this algorithm. The complexity for this algorithm is O(n) and clearly it is faster than Boyer-Moore's complexity.

Here is an example of string matching in Aho-Corasick algorithm, consider a dictionary consisting of the following words {a, ab, bab, bc, bca, c, caa}.

Table 1 - Aho-Corasick Data Structure
(Source: https://en.wikipedia.org/wiki/Aho–Corasick_algorithm)

Path	In Dictionary	Suffix Link	Dictionary Suffix Link
()	-		
(a)	+	()	
(ab)	+	(b)	
(b)	-	()	
(ba)	-	(a)	(a)
(bab)	+	(ab)	(ab)
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	-	(a)	(a)
(caa)	+	(a)	(a)

The black directed arc from node to node represents the name that is found by appending one character. So there is a black arc from (bc) to (bca). The blue directed arc from node to node represents the longest possible strict suffix of it in the graph. For example, for node (caa), its strict suffixes are (aa), (a) and (). There is a blue arc from (caa) to (a) because (a) is the longest strict suffixes that exists in the graph. The green arc from node to the next node in the dictionary suffix represents the way that can be reached by following blue arcs. For example, there is a green arc from (bca) to (a) because (a) is the first node in the dictionary that is reached when following the blue arcs to (ca) and then on to (a).

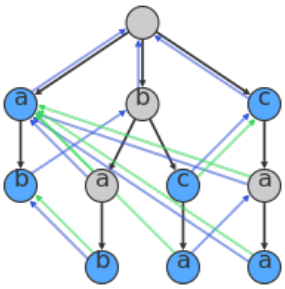

Figure 2.3 - Aho-Corasick Example
(Source: https://upload.wikimedia.org/wikipedia/commons/thumb/6/62/Ahocorasick.svg/220px-Ahocorasick.svg.png)

At each step, the current node is extended by finding its child, and if that doesn't exist, finding its suffix's child, and if that doesn't work, finding its suffix's suffix's child, and so on, finally ending in the root node if nothing's seen before. When the algorithm reaches a node, it outputs all the dictionary entries that end at the current character position in the input text. This is done by printing every node reached by following the dictionary suffix links, starting from that node, and continuing until it reaches a node with no dictionary suffix link. In addition, the node itself is printed, if it is a dictionary entry. [4]

Execution on input string abccab yields the following steps.

Node	Remaining String	Output: End Position	Transition	Output
()	abccab		start at root	
(a)	bccab	a:1	() to child (a)	Current node
(ab)	ccab	ab:2	(a) to child (ab)	Current node
(bc)	cab	bc:3, c:3	(ab) to suffix (b) to child (bc)	Current Node, Dictionary suffix node
(c)	ab	c:4	(bc) to suffix (c) to suffix () to child (c)	Current node
(ca)	b	a:5	(c) to child (ca)	Dictionary suffix node
(ab)		ab:6	(ca) to suffix (a) to child (ab)	Current node

Table 2 - Analysis of Input String abccab
(Source: https://en.wikipedia.org/wiki/Aho–Corasick_algorithm)

Below is the implementation of Aho-Corasick algorithm in C#.

```
        public AhoCorasickTree(string[] keywords)
        {
            if (keywords == null) throw new ArgumentNullException("keywords"
);
            if (keywords.Length == 0) throw new ArgumentException("should co
ntain keywords");

            _rootNode = new AhoCorasickTreeNode();

            var length = keywords.Length;
            for (var i = 0; i < length; i++)
            {
                AddPatternToTree(keywords[i]);
            }

            SetFailures();
        }

        public bool Contains(string text)
        {
            var currentNode = _rootNode;
            var length = text.Length;
            for (var i = 0; i < length; i++)
            {
                while (true)
                {
                    var node = currentNode.GetNode(text[i]);
                    if (node == null)
                    {
                        currentNode = currentNode.Failure;
                        if (currentNode == _rootNode)
                        {
                            break;
                        }
                    }
                    else
                    {
                        if (node.IsFinished)
                        {
                            return true;
                        }

                        currentNode = node;
                        break;
                    }
                }
            }

            return false;
        }

        private void AddPatternToTree(string pattern)
        {
            var latestNode = _rootNode;
            var length = pattern.Length;
            for (var i = 0; i < length; i++)
            {
                latestNode = latestNode.GetNode(pattern[i])
                             ?? latestNode.AddNode(pattern[i]);
            }

            latestNode.IsFinished = true;
            latestNode.Results.Add(pattern);
        }

        private void SetFailures()
        {
            _rootNode.Failure = _rootNode;
            var queue = new Queue<AhoCorasickTreeNode>();
            queue.Enqueue(_rootNode);

            while (queue.Count > 0)
            {
                var currentNode = queue.Dequeue();
                foreach (var node in currentNode.Nodes)
                {
                    queue.Enqueue(node);
                }

                if (currentNode == _rootNode)
                {
                    continue;
                }

                var failure = currentNode.Parent.Failure;
                var key = currentNode.Key;
                while (failure.GetNode(key) == null && failure != _rootNode)

                {
                    failure = failure.Failure;
                }

                failure = failure.GetNode(key);
                if (failure == null || failure == currentNode)
                {
                    failure = _rootNode;
                }

                currentNode.Failure = failure;
                if (!currentNode.IsFinished)
                {
                    currentNode.IsFinished = failure.IsFinished;
                }

                if (currentNode.IsFinished && failure.IsFinished)
                {
                    currentNode.Results.AddRange(failure.Results);
                }
            }
        }

        private class AhoCorasickTreeNode
        {
            public readonly AhoCorasickTreeNode Parent;
            public AhoCorasickTreeNode Failure;
            public bool IsFinished;
            public List<string> Results;
            public readonly char Key;

            private int[] _buckets;
            private int _count;
            private Entry[] _entries;

            internal AhoCorasickTreeNode()
                : this(null, ' ')
            {
            }

            private AhoCorasickTreeNode(AhoCorasickTreeNode parent, char key
)
            {
                Key = key;
                Parent = parent;

                _buckets = new int[0];
                _entries = new Entry[0];
                Results = new List<string>();
```

```
        }

        public AhoCorasickTreeNode[] Nodes
        {
            get { return _entries.Select(x => x.Value).ToArray(); }
        }

        public AhoCorasickTreeNode AddNode(char key)
        {
            var node = new AhoCorasickTreeNode(this, key);

            var newSize = _count + 1;
            Resize(newSize);

            var targetBucket = key % newSize;
            _entries[_count].Key = key;
            _entries[_count].Value = node;
            _entries[_count].Next = _buckets[targetBucket];
            _buckets[targetBucket] = _count;
            _count++;

            return node;
        }
    }
```

III. The Application of Boyer-Moore and Aho-Corasick Algorithm

A. *Boyer-Moore Algorithm*

Boyer-Moore algorithm's performance is based on the case of the string matching. The worst-case example for Boyer-Moore is the pattern to match is defined as "baaaaa" and the string is defined as "aaaaaaaaa". The shift of the pattern can be seen in the picture below.

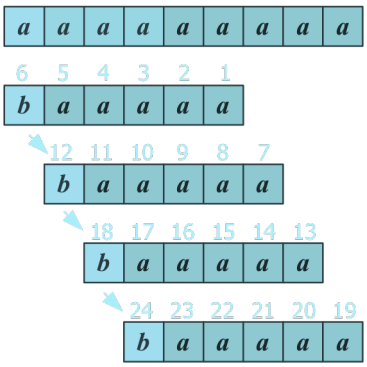

Figure 3 - Boyer-Moore Worst Case
(Source: http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Pencocokan-String-(2017).ppt)

Beside that kind of case, the performance of Boyer-Moore algorithm can be considered fast enough in string matching. In this algorithm, several arrays will be needed to store the string, the pattern to be matched, and the last occurrence of the character in the pattern. The shifting processes will depend on the type of pattern to be matched with the string. But in this case, we'll not use the kind of pattern for the worst-case scenario.

This algorithm can be considered fast enough because it matches the pattern with the string from the last character to the first character of the pattern or in other words it does the string matching from right to left. If there is a mismatch occurs in the process then the pattern will be shifted right away.

The result of the Boyer-Moore algorithm in string matching can be seen in the table and graph below. The algorithm is run with inputs of various sizes and the pattern is kept at a constant size, 40.

Table 3 - Boyer-Moore Result

Input Size	Execution Time (in seconds)
20000	1.43
60000	3.89
100000	7.13

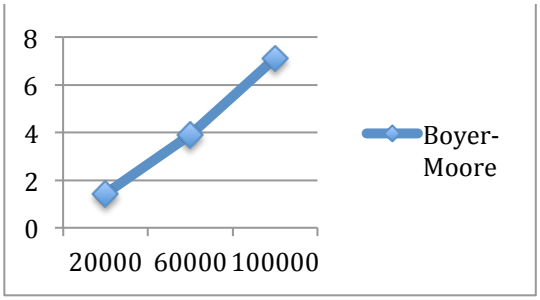

Diagram 1 - Boyer-Moore Result

B. *Aho-Corasick Algorithm*

Aho-Corasick algorithm's performance is based on the suffix tree implementation. Thus, the right structure should be chosen for the suffix tree implementation in order to gain the best result. Hash set is one of the structures that can be implemented in suffix tree. Both sparse hash set and dense hash set will be used as the structure for the suffix tree and we'll see which one is the better one to be implemented in suffix tree for this algorithm.

Sparse hash set is basically a hash table, which uses sparse table to implement the underlying array. This data structures stores only one element at one position in the hash set. If the position it is going to store is already occupied then it must search for the new position, which is still unassigned because each position is only allowed for one element. This kind of collision problem can be resolved by the quadratic internal probe, which is employed during the hashing operation.

Counting the key of the value so it can be inserted in the array exactly in the key position does the insertion in the hash set. The key can be counted using the modulus function, the integer i modulus by the size of the hash set. If the position had been occupied before then search for an available position.

If the table is full then the table will grow in size and all the data in the table will be rehashed and inserted into the new table using the same method. On the other hand, if the table is too empty then the table will shrink in size and all the data will be rehashed and inserted into the new table using the same method. When there is an element that needs to be deleted, then the value of the element will be replaced with the default-deleted values.

The main difference between sparse hash set and dense hash set is a sparse hash set employs sparse table as its underlying array, meanwhile dense hash set employs a simple array. From this difference, we can see that dense hash set will require more space than sparse hash set but the time requirement for dense hash set will be faster for various

operations rather than sparse hash set. This can happen because in sparse hash set, the memory management will take some time, which is not the case in dense hash set.

Despite the fact that dense hash set is faster than sparse hash set, it will be difficult to distinguish between the unassigned positions and the deleted positions. In a sparse table, this can be distinguished because the deleted values will be replaced by a bitmap and default values. Thus, dense hash set requires two default values for the unassigned and deleted positions. When a new dense hash set is made, all of the positions in the array will be initialized with the default values for unassigned positions.

The results that have been obtained with both sparse hash set and dense hash set implementations of the suffix tree in Aho-Corasick algorithm can be seen in the table below. The Aho-Corasick algorithm is run with inputs of various sizes and the pattern is kept at a constant size, 40.

Table 4 - Aho-Corasick Result

Input Size	Execution Time (in seconds)	
	Sparse Hash Set	Dense Hash Set
20000	23.34	21.47
60000	63.09	58.76
100000	101.29	92.48

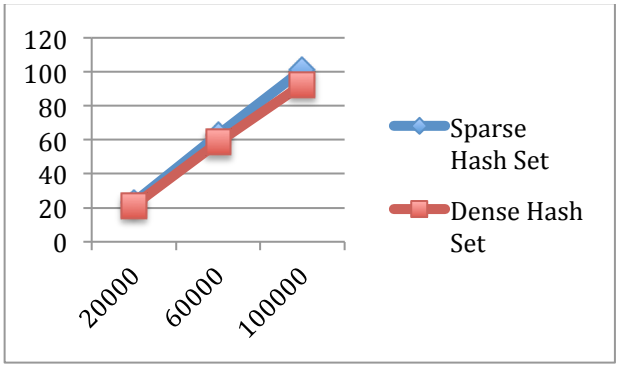

Diagram 2 - Aho-Corasick Result

From the table and graph for Aho-Corasick algorithm, can be seen that dense hash set in Aho-Corasick algorithm gives a better performance than sparse hash set in the Aho-Corasick algorithm. The more the input size increase, the better result the dense hash set will pronounce. But from both tables for Boyer-Moore and Aho-Corasick algorithm, can be seen that both kind of structures implementation for the suffix tree perform better than the Boyer-Moore algorithm in case of string matching.

IV. Conclusion

Security is needed in every company's networking system so that intruders or hackers from outside and inside the system can't intrude the system. In the network intrusion detection system, it needs to compare the pattern with the string as fast as possible so when there is an anomaly in the system's data packets, it can be detected early. Boyer-Moore is fast enough to be implemented in the network intrusion detection system.

Although Boyer-Moore can be categorized as a fast string matching algorithm, Aho-Corasick is faster than Boyer-Moore so it is better to implement Aho-Corasick in network intrusion detection system. But for a far better result, the suffix tree in Aho-Corasick should be implemented with dense hash set structure. Despite the fact that Aho-Corasik is the better solution for the network intrusion detection system, it's performance can be enhanced by implementing the suffix tree in Aho-Corasick using the right structure such as hash set. But you should be careful in choosing the right structure for hash set, not all of the hash set can enhance the algorithm performance.

Acknowledgment

First of all, I would like to thank God for His blessings so that I can finish this paper. I wish to express my sincere gratitude towards Dr. Ir. Rinaldi Munir MT., as my lecturer in this subject. I would also like to thank both of my parents who keep on praying for me and motivating me because without them I won't be here, studying in Bandung of Institute Technology.

References

[1] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf. Guide to Intrusion Detection and Prevention Systems (IDPS), NIST CSRC special publication SP 800-94, released 02/2007. Retrived on 15th May 2017.

[2] https://en.wikipedia.org/wiki/Intrusion_detection_system.Wikipedia. Intrusion Detection System. Accessed on 13th May 2017.

[3] http://ranger.uta.edu/~dliu/courses/cse6392-ids-spring2007/papers/USENIXLISA99-Snort.pdf. Martin Roesch, Snort-Lightweight Intrusion Detection for Networks, Stanford Telecommunications, Inc, 13th LISA conference, 1999, 229-223. Retrieved on 13th May 2017.

[4] https://en.wikipedia.org/wiki/Aho–Corasick_algorithm. Wikipedia. Aho-Corasick algorithm. Accessed on 15th May 2017.

[5] http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf. Biosequence Algorithms, Spring 2005. Lecture 4: Set Matching and Aho-Corasick Algorithm. Retrived on 15th May 2017.

[6] http://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/. Pattern Searching | Set 8 (Suffix Tree Introduction). Accessed on 14th May 2017.

[7] https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf. Suffix Trees. CMSC 423. Retrieved on 14th May 2017.

[8] https://www.lifewire.com/introduction-to-intrusion-detection-systems-ids-2486799. Introduction to Intrusion Detection System (IDS). Accessed on 13th May 2017.

[9] https://www.sans.org/security-resources/idfaq/what-is-network-based-intrusion-detection/2/3. Accessed on 13th May 2017.

[10] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Pencocokan-String-(2017).ppt. Munir, R. Pencocokan String. Retrieved on 14th May 2017.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Mei 2017

Kezia Suhendra 13515063