NETWORK INTRUSION DETECTION SYSTEM USING STRING MATCHING

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIRMENTS FOR THE DEGREE OF

Bachelor of Technology

In

Computer Science & Engineering

By

SIDDHARTH SAHA

TELUGU PRAVEEN KUMAR

Department of Computer Science & Engineering

National Institute of Technology

Rourkela

2010

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled “**NETWORK INTRUSION DETECTION SYSTEM USING STRING MATCHING”** submitted by Mr. Siddharth Saha and Mr. Telugu Praveen Kumar in partial fulfillment of the requirements of the award of Bachelor of Technology Degree in Computer Science & Engineering at the National Institute of Technology, Rourkela is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university/institute for the award of any degree or diploma.

Date:

Prof. Sanjay Kumar Jena
Department of Computer Science & Engineering
National Institute of Technology
Rourkela – 769008.

ACKNOWLEDGEMENT

We wish to express our profound sense of sincere gratitude to our project mentor, Prof. Sanjay Kumar Jena, Department of Computer Science & Engineering, National Institute of Technology, Rourkela, without whose help, guidance and motivation this project could not have been completed. We would also like to thank Prof. S. Panigrahi, Department of Computer Science & Engineering, National Institute of Technology, Rourkela, for his extended and continued support and guidance throughout this project.

We also thank all the faculty of the department of Computer Science & Engineering, National Institute of Technology, Rourkela for their help and guidance on numerous occasions. The patient help of all our friends is also thankfully acknowledged.

Lastly, but not the least, we would like to thank for the authors of all the books, papers and websites that have been referred to.

Siddharth Saha
Roll No: 10606041
National Institute of Technology
Rourkela

Telugu Praveen Kumar
Roll No: 10606003
National Institute of Technology
Rourkela

CONTENTS

Page

ABSTRACT

Network intrusion detection system is a retrofit approach for providing a sense of security in existing computers and data networks, while allowing them to operate in their current open mode. The goal of a network intrusion detection system is to identify, preferably in real time, unauthorized use, misuse and abuse of computer systems by insiders as well as from outside perpetrators.

At the heart of every network intrusion detection system is packet inspection which employs nothing but string matching. This string matching is the bottleneck of performance for the whole network intrusion detection system. Thus, the need to increase the performance of string matching cannot be more exemplified.

In this project, we have studied some of the standard string matching algorithms and implemented them. We have then compared the performance of the various algorithms with varying input sizes. The main focus of the project was the Aho-Corasick algorithm. In addition to using the default implementation of suffix trees, we have used a dense hash set and a sparse hash set implementation- which are libraries from the Google code repository- and we show that the performance for these implementations are better. They give noticeable enhancement in performance when the input size increases.

Chapter 1

Introduction: Traditional Security and Network Intrusion

1.1 Introduction to Network Intrusion

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] To understand the meaning of intrusion detection, we can use an analogy [2] to the common "burglar alarm". Just like the burglar alarm, intrusion detection works on a computer system or network and is enabled to detect possible violations of security policies and raise an alarm to notify the proper authority.

A Network Intrusion detection System (NIDS) is an intrusion detection system that tries to detect malicious activity such as service attacks, port scans or even attempts to break into computers by monitoring network traffic.

1.2 Shortcomings of Traditional Security

In the case of traditional security techniques, one can identify the following shortcomings [3]-

a) **Firewall Evasion**

b) **Tunneling attacks:** Tunneling refers to the process of gaining unauthorized access to the network by encapsulating specific messages, which are blocked by the firewall, within messages of another type. The firewall generally implements rules of filtering of packets based on the network protocol that are allowed. Normally, the rules must require that the network be checked to ensure that the specific protocol is allowed. Invaders exploit this property to send malicious messages encapsulated in the type of message which the protocol allows.

   For example, if a firewall enables ports 25 and 80, it allows mail (SMTP) and Web (HTTP) traffic to pass into the internal network. Invaders exploit this property and perform the unauthorized activity within the allowed protocol, thus creating a tunnel through which the intruder can implement an attack. One instance of this type of tunneling can be the security holes in the firewalls which is exploited when implementing the Loki attack that enables the intruder to tunnel various commands into ICMP Echo requests to which an authorized reply is the ICMP Echo reply.

c) **Attacks due to incorrect firewall configuration:** It is a fact that most firewalls are configured and deployed by humans. And human beings are prone to error making. This knowledge is well-known to the intruders who try to take advantage of it. They try to find a security breach in the configuration of the firewall and exploit it.

   For example, statistical data collected in 1999 by ICSA (later renamed TruSecure) (http://www.trusecure.com) shows that up to 70% of all firewalls are vulnerable due to an incorrect configuration.

d) **Attacks by trusted hosts and networks:** As most organizations deploying security mechanisms use encryption for protecting files and external network connections, so the intruder's interest will lie on such locations where the encryption/protection of data transmission is missing or very minimal. This is generally the case where the data is stored and/or transmitted to trusted hosts and networks. Even if a VPN request connection is made between trusted hosts and networks and the main network in question, attacks by intruders can be very efficient. Furthermore, the probability of successful attacks in this type of attacks can be very high as, in most cases, not even minimum encryption is used for increasing performance.

e) **Attacks based on bypassing the firewall**

f) **Attacks by source address spoofing:** Address spoofing is a method which is used to hide the real address of the sender of a network packet, particularly the intruder. However, this can also be used to bypass the firewall and gain unauthorized access to a network or computer. Contemporary firewalls have in-built mechanisms to avoid this fraud. They are, practically, not deceived by this kind of address spoofing. But the principle of address substitution, in itself, remains an urgent issue that needs to be addressed.

   For instance, an intruder can mask her own address with the address of a trusted network address or with the address of a trusted host in the network and send packets containing malicious data which may adversely affect the network computers and data. This method is different from the attack by trusted host and network problem since in this case only the address of the trusted host is used rather than masquerading as the trusted host in the former case.

g) **Attacking the firewall itself:** As the firewall software and hardware are built by humans, they themselves are prone to attack from the intruders. A successful attack on the firewall can lead to very serious consequences as once successfully attacked, intruders can freely access the resources of the protected network without the risk of being detected and traced. Moreover, an intruder can also tweak with the configuration and rules of the firewall to allow other kind of intruders to attack the network.

h) **Attacks on the firewall authentication system:** In this case, the authentication system of the firewall itself is attacked by the intruder. Once the authentication system is attacked, the firewall will not even register a security violation because it will interpret the intruder as an authorized user. Example- CISCO PIX Firewall Vulnerability.

1.3 Conclusion

From the above discussion, we see the meaning of network intrusion. In particular we discussed in good detail which one encounters while using the traditional security. Usage of traditional security measures can have serious repercussions in the modern day intricacy of the web and the ubiquitousness and penetration of internet which make all the networks, and computers associated to it, more prone to such attacks. In this information age, when the value of data and information is critical, theft and misuse of data and information is the top priority of all organizations. Thus, the need of modern network intrusion detection system has taken prime importance.

Chapter 2

Network Intrusion Detection Systems

2.1 Network Intrusion Detection System

A network intrusion detection system is an intrusion detection system (1.1) that tries to detect malicious activity such as denial of service attack, port scans or attempts to crack into computers by monitoring network traffic. [4]

A network intrusion detection system reads all incoming packets and tries to find suspicious patterns known as signatures or rules. These rules are decided by a network administrator while the configuration and deployment of the network intrusion detection system based on the security and network policies of the organization. For instance, if it is observed that a particular TCP connection requests 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 all/most of the computers of the network. [4]

A network intrusion detection system is not limited to inspecting the incoming network traffic only. Patterns and outgoing intrusion can also be found from the outgoing or local traffic as well. Some attacks might also come from the inside of the monitored network, as in trusted host attack. (1.2)

At the heart of every modern network intrusion detection system there is a string matching algorithm. The network intrusion detection system uses the string matching algorithm to compare the payload of the network packet and/or flow against the pattern entries of the intrusion detection rules, which are a part of every network having a network intrusion detection system.

String matching needs significant memory and time requirements. In fact, the performance of all network intrusion detection systems depends almost entirely on the performance of the string matching algorithm. For example, the string matching routines in Snort account for more than 70% of the total execution time and 80% of the instructions executed in real time traces. [8]

2.2 Uses of a network intrusion detection system

As we have discussed in chapter 1 of this thesis, traditional security is not adequate for contemporary networks. With increased usage and more cases of intrusion taking place now-a-days than ever before, we need something to enhance the security. This is where a network intrusion detection system comes into the picture.

The following are the basic needs that a network intrusion detection system can fulfill [3]-

a) **Backing up firewalls:** In many cases, intruders try to and penetrate firewalls to gain unauthorized access to corporate networks. This is done by attacking the firewall itself and breaking it down by tweaking its rules and signatures. In this case, the network intrusion detection system can decrease the risk of such attacks by temporarily backing up firewalls. The network intrusion detection system of this type filters packets based on their IP packet header. This enables the network administrator to deploy network intrusion detection systems with functionality comparable to that of very advanced firewalls. Further, this type of network intrusion detection system can also be used while the general firewall is down for maintenance or when the firewall software is being updated or for any other reason.

b) **Controlling file access:** Generally functions of controlling file access are done to specialized systems, such as Secret Net, which are intended specifically for protecting network information from unauthorized access. However, protection of some critically important files such as database files and password files cannot be done by such systems. Moreover, such systems are mainly developed for the Windows and NetWare platforms. So such systems fail in UNIX environments which are used for network applications in many organizations. So in such types of cases a network intrusion detection system comes to the rescue of network administrators. Mainly host based network intrusion detection systems are used in such cases which are based both on log-file analysis (Real Secure Server Sensor) and IDSs analyzing system calls (Cisco IDS Host Server).

c) **Controlling the administrator's activities:** Network intrusion detection systems can act as an additional control tool which can check unauthorized configuration changes by the hosts that have been granted administrative privileges.

d) **Protection against viruses:** There has been an alarming increase in the number of viruses and worms that now invade the internet and affect numerous computers everyday. Worm epidemics like the Red Code, Blue Code, Nimada etc. has demonstrated the danger of underestimating the dangers of such malicious programs.

Network intrusion detection systems can be helpful in these cases as well. Though they cannot replace the traditional antiviral software yet they can act as an additional barrier for viruses, worms and Trojan horses.

e) **Detecting unknown devices:** A network intrusion detection system can help in identifying the address of unknown/external hosts within the protected network segments. It can also detect increased traffic and special kind of activities from specific workstations which were not involved in such kind of activities before. Such activities can be a hint to malicious activities from the hosts and the network administrator must be informed about this.

f) **Analyzing the efficiency of firewall settings:** Firewalls are essential for protecting the corporate network from unwanted network activities. But a firewall can work desirably only when it is configured correctly. Incorrect configuration and inefficient testing of a firewall can wreck havoc on the network. Installing a network intrusion detection system before and after the firewall allows one to test the efficiency of the firewall by comparing the number of attacks before and after the firewall. In addition to this, it can also act as a backup for the firewall.

g) **Analyzing the information flows:** Situations where the communication specialists have no trustworthy information on the protocols used in the protected network segments. A network intrusion detection system can control all the protocols and services used in the corporate network, as well as the frequency of their use. This enables to make a scheme of information flow and the network map. [5] This creation of information flows and the network map is an essential requirement for successfully creating an information-security infrastructure.

h) **Analyzing data from the network equipment:** Log files from routers and other network equipment can serve as an additional source of information on the various attacks that a data network can be prone to. However, most organizations do not analyze these collected information because it is a time overhead for the organization and the tools available for such analyses (such as netForensics) are rather costly.

A network intrusion detection system can be configured to do this work. The task of collecting such log-file information and analyzing logged security events can be delegated to the intrusion detection system, which in this case, serves as a Syslog server. It can centralize such tasks of collecting log-file information and detect attacks and misuse of the network. It also prevents unauthorized modifications of the events logged. Moreover, the events logged are immediately sent to another server so that the intruder can't remove any traces after completing her operation.

i) **Collecting proof and handling incidents:** Network intrusion detection systems can, and should, be used for collecting proof of unauthorized activity in the network either by trusted hosts or outside perpetrators. This is accomplished by the network intrusion detection system with the following functionalities-

    I. Logging the events that take place during an attack in a database or an external file which can be analyzed later.

    II. Imitating applications which do not exist in order to deceive the intruder. This kind of functionality is also known as the deception mode [3] of working of the network intrusion detection system.

    III. The network intrusion detection system, in addition to logging events, also help in enhanced analysis of those log files which were created either by the system or the application software or database and web servers.

    IV. The network intrusion detection system can also help in getting information of the intruder like his DNS, MAC, NetBIOS and IP address.

j) **Performing inventory and creating a network map:** A map that shows as its theme primarily connections within a network, such as roads, subway lines, pipelines, or airport connections. [5] A network intrusion detection system can be used to create a network map which can be used to gather various information about network hosts and intruders. The information that could be collected are [3] -

    I. The role of the host and its DNS and NetBIOS names

    II. Network services

    III. Active service headers

    IV. Types and versions of operating systems and application software.

V. NetBIOS shares

VI. User and service accounts

VII. General parameters of the security policy- audit policy, user and password policy and so on.

k) **Detecting default configurations:** Most network administrators use the default network configurations for simplifying their tasks. But this also simplifies the task of an intruder because she knows the default network configurations. This makes the network more vulnerable and open to successful attacks. A network intrusion detection system can be configured to search the hosts where default configurations have been used and can also recommend corrective measures that can be taken.

2.3 Snort- An overview

Snort is a free and open source network intrusion prevention system and network intrusion detection system capable of performing packet logging and real time traffic analysis on IP networks. Snort was written by Martin Roesch and is now developed by Sourcefire. [6]

Snort performs all the basic functions of a network intrusion detection system which we discussed before. Mainly, Snort performs protocol analysis, content searching/matching and is commonly used to actively block or passively detect a variety of blocks and attacks, some of which are buffer overflows, port scans, web application attacks and operating system fingerprinting attacks.

Snort can also be combined with other free software to give a visual representation of visual data. It is a cross-platform, lightweight intrusion detection system which can deployed on a variety of platforms to monitor TCP/IP networks and detect suspicious activities. [7]

Snort was designed to fulfill the requirements of a prototypical lightweight network intrusion detection system. It has become a small, flexible, and highly capable system that is in use around the world on both large and small networks. It has attained its initial design goals and is a fully capable alternative to commercial intrusion detection systems in places where it is cost inefficient to install full featured commercial systems. [7]

2.4 Conclusion

In the previous chapter, we had seen that traditional security is inadequate for modern networks. For this purpose, network intrusion detection systems can into existence and all modern corporate organizations install appropriate network intrusion detection system to protect the data in their network. We also discussed the various uses of a network intrusion detection system. From the discussion, it can be inferred that a network intrusion detection system fills in the gap left by traditional security mechanisms like firewalls.

Lastly, we gave an overview of Snort, which is a very popular contemporary network intrusion detection and prevention software used by many organization because of its light weightiness and its ease of integration with other tools and it can suited to every organization’s needs.

Chapter 3

String Matching Algorithms

3.1 Introduction to string matching

String matching is an important subject in the wider domain of text processing. These algorithms are basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems. [9]

String matching generally consists of finding a substring (called a **pattern**) within another string (called the **text**). The pattern is generally denoted as,

$$x = x[0\,..\,m - 1]$$

whose length is m and the text is generally denoted as

$$y = y[0\,..\,n - 1]$$

whose length is n. Both the strings- pattern and text- are built over a finite set of characters which is called the **alphabet** and denoted by ∑ whose size is denoted by σ.

While searching the pattern within the text, at one time, we consider a subset of the text generally with the help of a **window** whose size is equal to m. Then the window is aligned with the pattern and they are matched for equality- either from the right or from the left- this specific work is known as an **attempt**. If they completely match, then either the algorithm ends or it continues to find any more occurrences of pattern in text. If they do not match, then the window is **shifted** to a new position. This is known as **sliding window mechanism**.

3.2 String Matching Algorithms

We have studied and implemented various algorithms for string matching. We discuss those algorithms which have been implemented and against whose results comparisons have been made.

3.2.1 Brute Force Algorithm

The brute force algorithm is a very trivial string matching algorithm. It consists in checking at each position from 0 to $n - m$ of the text by employing a window of size m whether it is the starting point of the pattern or not. This is done by comparing every character in the pattern with the corresponding character in the text. This comparison can be done in any order-either from left to right or from right to left or in any other order. If all the characters match, then it is said to be a **match**. If not, then the window is shifted exactly one position to the right.

The brute force algorithm requires no preprocessing of the pattern. Also it does not require any extra space in addition to the memory required to store the pattern and the text. The expected number of character comparisons in brute force algorithm is $2n$. The time complexity of the searching phase of brute force algorithm is O$(m \times n)$.

Algorithm-The following describes the algorithm for the brute force algorithm for string matching. The various notations that have been used are as they were described above. (3.1)

```
$for\ i\ from\ 0\ to\ n-m\ do$
    $flag \coloneqq true$
    $for\ j\ from\ 0\ to\ m-1\ do$
        $if\ x[j] \neq y[i+j]\ then$
            $flag \coloneqq false$
            $break$
        $endif$
    $endfor$
$endfor$
```

Main points-Here we outline the main features of the above algorithm which can serve as a quick overview of the above algorithm. [9]

I. No preprocessing phase.

II. Constant space required. No extra memory required other than the memory storage for pattern and text.

III. Always shifts the window by one position to the right.

IV. Character comparisons can be done in any order.

V. Searching phase is O$(m \times n)$ time complexity.

VI. $2n$ expected character comparisons.

This algorithm is a very basic algorithm. It does not give good results in real time network intrusion detection systems and hence is not used in network intrusion detection systems.

3.2.2 Karp-Rabin Algorithm

More sophisticated string matching algorithm employ more sophisticated skipping, that is, they shift the window by as many positions as possible. This allows lesser number of character comparisons than the brute force algorithm. Instead of using this technique, Karp-Rabin algorithm uses hashing of text and pattern to speed up the testing of equality. So instead of checking at each position of the window, the equality of the pattern is tested with the text only when they “look alike”- that is when their hashed values are equal to close to each other.

For this method to work we define a $hash$ function which will be used for hashing both the pattern as well as the text. It is desirable that the $hash$ function should have the following properties- [9]

I. It must be efficiently computable.

II. It must be highly discriminating for strings.

III. $hash(y[j+1 \,..\, j+m])$ must be easily computable from $hash(y[j \,..\, j+m-1])$ and $hash(y[j+m])$:

$hash(y[j+1 \,..\, j+m]) = rehash(y[j], y[j+m], hash\ (y[j \,..\, j+m-1])$

For a word w of length m let $hash(w)$ be defined as follows:

$$hash(w[0 \,..\, m-1]) = (w[0] \times 2^{m-1} + w[1] \times 2^{m-2} + \ ... + w[m-1] \times 2^{0})\ mod\ q$$

where q is a large number. Then,

$$rehash(a,b,h) = ((h - a \times 2^{m-1}) \times 2 + b)\ mod\ q$$

Algorithm- For the Karp-Rabin algorithm we present a function below which implements the algorithm. This function can be coded and called from the main program. This function returns the index of the text where the pattern is found, else returns null. [10]

```
$function\ KarpRabin(string\ text[0 \,..\, n-1]\ string\ pattern[0 \,..\, m-1])$
    $hpattern := hash(pattern[0 \,..\, m-1])$
    $htext := hash(text[0 \,..\, m-1])$
    $for\ i\ from\ 0\ to\ n-m\ do$
        $if\ htext = hpattern\ then$
            $if\ text[i \,..\, i+m-1] = pattern\ then$
                $return\ i$
```

```
                    $endif$
            $endif$
            $htext := hash(text[i + 1\,..\,i + m])$
    $endfor$
    $return\ null$
```

In this algorithm we see that character comparisons are not made for all the positions of the window. First the $hash$ values are calculated for both the pattern and well the text of length equal to that of the pattern. Then these two $hash$ values are compared. If they are equal only then all the characters of the text and pattern are compared. This comparison is necessary because we cannot say for sure whether two strings are the same based only on their $hash$ values.

Main points- The main points of this algorithm can be described as follows [9]:

I. This algorithm uses a hash function.
II. Preprocessing phase in O(m) time complexity and constant space.
III. Searching phase in O$(m \times n)$ time complexity.
IV. O$(m + n)$ overall expected running time.

This algorithm is an improvement over the Brute force algorithm because it doesn't compare all the characters every time. Only when the $hash$ values of the pattern and text in the window are equal, the character comparisons are made. This gives considerable improvement of performance over the Brute force approach.

3.2.3 Boyer-Moore Algorithm

This is an algorithm that searches for the location i of the first occurrence of pattern in text. During the search operation, the characters of pattern are matched starting from the last character of pattern. The information gained by starting the match at the end of the pattern often helps in proceeding in large jumps of the window. Thus this algorithm has the property that, in most cases, not all of the first i characters of the text are examined. Further, the number of characters actually examined decreases as a function of the number of characters in pattern. [11]

Algorithm- This algorithm comes in various forms. The main part of this algorithm is that it uses two main techniques for shifting the window from one position to another. They are known as the good-suffix shift and the bad-character shift. These techniques and the general algorithm have been described below in a generic description. Boyer-Moore algorithm can be implemented in a variety of ways but the main essence of the algorithm is the two shifting techniques.

The first notable difference of this algorithm is that it starts matching the characters from right to the left instead from left to right as in most string matching algorithms. When there is a mismatch, the window needs to be shifted and this algorithm uses two pre-computed functions corresponding to the good-suffix shift and the bad-character shift.

Suppose, while the character comparison, we get to a point when the characters in the pattern and the text do not match, that is, $x[i] \neq y[i+j]$. Also, in this condition, we have already matched some of the trailing characters of the pattern with that of the text, that is, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1]$. In this case, we employ the good-suffix shifting. We align the segment $y[i+j+1\,..\,j+m-1] = x[i+1\,..\,m-1]$ of the text with the right most occurrence in the pattern under the condition that it is preceded by a character different from the character at which the mismatch had occurred, that is, different from $x[i]$. This type of shifting is known as the good-suffix shift because there exists a suffix of the pattern which has already matched with the text. After the shifting, instead of starting the character comparisons from the extreme right of the pattern, we can start the comparison from $x[i]$. If there is no such suffix, then we need to align the longest suffix of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x.

Under the bad-character shift, we align the character $y[i+j]$ with the right most occurrence in $x[0\,..\,m-2]$. In some cases, $y[i+j]$ does not occur anywhere in x. In these cases, the window is just shifted by one position, that is, the window is shifted from $y[i+j]$ to $y[i+j+1]$.

The bad character shift can sometimes be negative, that is, the window can get shifted to the left instead to the right. So the Boyer-Moore algorithm uses the maximum of the bad-character and the good-suffix shift.

Main points- The main points of the Boyer-Moore algorithm are [9]-

I. Performs the comparisons from right to left.
II. Preprocessing phase in O$(m +$σ$)$ time and space complexity.
III. Searching phase in O(mn) time complexity.

IV. $3n$ character comparisons in the worst case when searching for a non periodic pattern.

V. O($\frac{n}{m}$) best performance.

3.2.4 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm also works on the principle that when a mismatch occurs the pattern itself contains sufficient information so that the window can be shifted sufficiently so that unnecessary character comparison does not take place. The advantage of using this algorithm over the Boyer-Moore algorithm is that in this algorithm only one table needs to be constructed from the pattern instead of two tables as is the case with the Boyer-Moore algorithm. So the time spent on preprocessing of the pattern is lesser in this case.

Algorithm- As described above, we need to construct a table while preprocessing. This table is known as the **partial table**. In the algorithm that we describe next, we indicate this partial table by **T**. The partial table indicates where we need to start the next search if this attempt of searching ends in a mismatch. The entries in T can be constructed quite easily. If while comparing, we get a mismatch at $y[i+j]$ with $x[i]$, then the next possible match will start at $y[i+j-T[i]]$. Here $T[i]$ can be interpreted as the amount of backtracking that needs to be done.

For this to work well, some values of T needs to be defined before-hand to give consistent results. First is $T[0] = -1$. This will indicate that if we get a mismatch for $x[0]$, then we do not backtrack but that we inspect the next character. Second is that though the next match can start at $i + j + T[i]$ but need not check any of the $T[i]$ characters after that and we can continue searching from $x[T[i]]$.

Following is the algorithm for constructing the partial table [12]-

```
$function\ kmp\_table(string\ x, string\ y)$
    $pos := 2$
    $cnd := 0$
    $T[0] := -1, T\ [1] := 0$
    $while\ pos < m\ do$
        $if\ x[pos-1] = x[cnd] then$
            $T[pos] := cnd + 1$
            $pos := pos + 1$
            $cnd := cnd + 1$
        $endif$
        $else\ if\ cnd > 0\ then$
            $cnd := T[cnd]$
        $endif$
        $else$
            $T[pos] := 0$
            $pos := pos + 1$
        $endelse$
    $endwhile$
```

The searching algorithm uses the partial table as constructed by the above algorithm by using the following the algorithm.[12]

```
    $function\ kmpSearch(string\ x, string\ y, T)$
        $i := 0$
        $j := 0$
        $while\ (i + j) < n\ do$
            $if\ x[i] = y[i + j]\ then$
                $j := j + 1$
                $if\ j = m\ then$
                    $return\ i$
                $endif$
            $endif$
            $else$
                $i := i + j - T[j]$
                $if\ T[i] > \ -1\ then$
                    $j := T[j]$
                $endif$
                $else$
                    $j ≔ 0$
                $endelse$
            $endelse$
        $endwhile$
        $return\ m$
```

Main points- The main points of the Knuth-Morris-Pratt algorithm are outlined below[9]

I. Performs the comparisons from left to right.

II. Preprocessing phase in O(m) space and time complexity.

III. Searching phase in O$(n + m)$ time complexity.

IV. Delay bounded by $\log(m)$.

3.3 Conclusion

In this chapter, we described the various algorithms which we have studied and implemented. The results which we have obtained about the running time of these algorithms with different input sizes have been discussed in a later chapter. Suffice is to say that all these algorithms are comparable to each other and their performance varies with the type of input given.

Chapter 4

Suffix Trees and Aho-Corasick Algorithm

4.1 Introduction

In this chapter, we discuss the Aho-Corasick algorithm. In this project, the main emphasis is on this algorithm. This algorithm uses a very important and popular concept in computer science which is known as **suffix trees**. Aho-Corasick algorithms uses suffix trees and this determines the performance of the implementation. Suffix trees can have many kinds of implementations and depending on the kind of implementation of the suffix tree, the running times of the Aho-Corasick algorithm varies.

We first discuss suffix trees and some idea about some of the standard implementations of suffix trees. Next, we discuss the Aho-Corasick algorithm.

4.2 Suffix Trees

A suffix tree, in computer science, is an important data structure used for many string applications like string matching algorithms, longest common substring problem, regular expressions etc. Suffix trees store strings in such a way that the common substring among the strings are stored only once. That is, the common substrings in the various strings are stored and at the point where the strings differ, they branch out. In these trees, the edges are labeled with characters from the strings.

4.2.1 Definition

The suffix trees for a string S is a tree whose edges are labeled with strings, such that each suffix of S corresponds to exactly one path from the tree's root to leaf. [13] The suffix tree for the string S of length m is defined as a tree such that [14] –

I. All the paths from the root of the tree to the leaves have one-to-one relationship with the suffixes of S.
II. All edges are labeled and denote non-empty strings.
III. All the internal nodes (with the possible exception of the root) have two children.

4.2.2 Functionality

A suffix tree has the following functionalities [13]-

I. A suffix tree for a string S can be built in O(n) time.

II. The string searching operation, after the suffix tree has been built, takes $O(m)$ time where m is the length of the substring.

III. The suffix tress can be implemented in a variety of ways. Some of which are the following-

a. Unsorted Arrays
b. Hash Maps
c. Balanced Search trees
d. Sorted Arrays

In our study, we just make of the hash maps implementation and a standard set implementation of the suffix trees.

4.2.3 Example

Now, we show an example of a suffix tree. The following figure shows a suffix tree for the string "Bananas". The tree (or trie) contains all the suffixes of "Bananas" including the word itself- BANANAS, ANANAS, NANAS so on and ending with an S. [15]

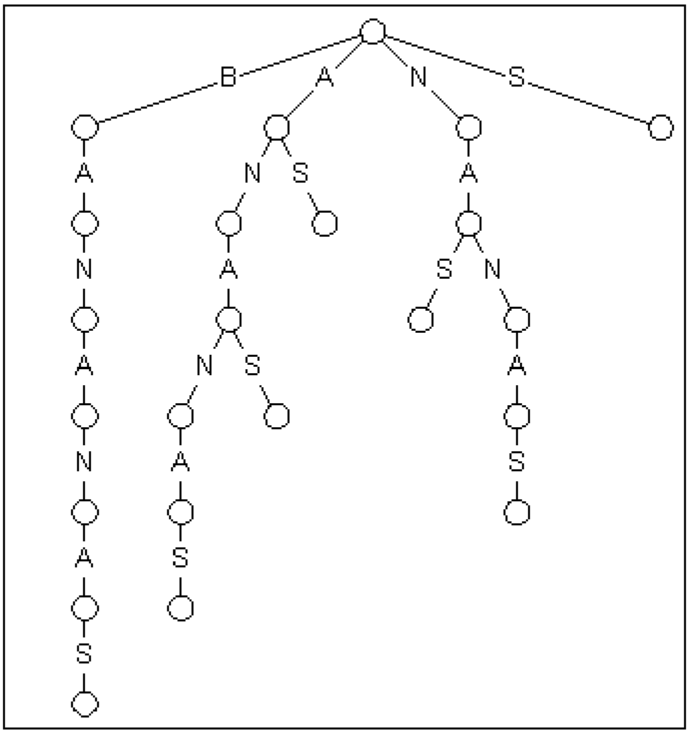

Figure 4.1 : Suffix Tree for "Banana"

The use of a suffix tree, in this way, can be that all the occurrences of any substring can be search simultaneously in a given text.

4.3 Aho-Corasick Algorithm

This algorithm is the most important algorithm in our project. The experimental results presented in the next chapter are for the various kinds of implementations of this algorithm.

This is a kind of “dictionary-matching” algorithm [16] which finds the various strings from a finite set of strings in a text that needs to be inspected. This algorithm can work simultaneously for all the strings in the set and locate these in the text, that is, this algorithm can match all the strings “at once” thus reducing the time taken for the searching. All the other algorithms that we have discussed above match only one pattern at a time. But this algorithm matches all the patterns at once. So this algorithm is better suited for a network intrusion detection which has to match all the patterns from the rules in the incoming traffic.

This algorithm constructs a trie with suffix tree like structure for all the patterns that are to be matched. The suffix tree contains a set of links from each node representing a pattern to the node containing the longest proper suffix. This suffix tree is used during the run time for matching purposes. The suitability of this algorithm to a network intrusion detection system can be further justified by the fact that the suffix tree for all the patterns of the rules can be constructed offline, that is, before its deployment, and then the suffix tree can be used for the real time searching with the network data. Only when new rules or new patterns are added to the set of rules, then the suffix trees need to be re-constructed.

So in a network intrusion detection system, the automaton (the suffix tree) is known before hand, and so the searching phase’s run time is linear in the length of the input plus the number of matched entries.

Algorithm- This algorithm, as well, works on the same concepts of aligning the text with the pattern (in this case the suffix tree) and window shifting. This algorithm also uses the concept of shifting the window as much as possible to remove unnecessary comparisons.

We can now discuss the algorithm with reference to a network intrusion detection system. The suffix tree containing the different patterns from the network filtering rules is aligned to the right most end of the packet payload text. The suffix tree moves from the right end to the left while the character comparisons are done from the left end to the right. Each time the suffix tree needs to be repositioned.

As described earlier, this algorithm also uses the same shifting techniques as that of the Boyer-Moore algorithm, namely, the good suffix shift (in this case, it is known as good prefix shift) and the bad character shift. The main difference between the Boyer-Moore algorithm and Aho-Corasick algorithm is that instead of sliding a single pattern along the text, this algorithm slides a tree of patterns (the suffix tree) while performing the shifting techniques.

We now discuss the sliding techniques as they relate to the Aho-Corasick algorithm.

I. **Bad character shift**- In this shifting type, if a mismatch occurs then the suffix tree is shifted to align to the next occurrence of the character in some other pattern in the suffix tree. If the particular character in question is not in any pattern, then the window is shifted by the length of the smallest pattern in the suffix tree.

II. **Good prefix shift**- This shifts the window to the next occurrence of a complete prefix that has already been encountered as a substring of some other pattern, or the window can be shifted to the next occurrence of some prefix of the correctly matched text as the suffix of some other pattern in the tree.

This is the general description of the Aho-Corasick algorithm. As we see, it is almost same to the Boyer-Moore algorithm. But with the inherent nature of comparing all the patterns at once, the practical run time performance of this algorithm is much better than the Boyer-Moore algorithm.

Aho-Corasick algorithm’s performance depends a lot on the suffix tree’s performance. While the construction phase of the suffix tree can be overlooked as it can be done offline, but still a more optimized implementation of a suffix tree can gain in terms of performance. This is exactly what we show in the next chapter where we discuss our results with different implementations of the suffix tree. We show that we get enhanced performance if we use a better hash set implementation of the suffix tree as compared to the standard sorted set implementation of the suffix tree.

4.4 Conclusion

In this chapter, we discuss the most important algorithm, as it pertains to our project, the Aho-Corasick algorithm. The most important part of the Aho-Corasick algorithm is the suffix tree, as has been discussed that the performance of the algorithm depends on a large extent to the implementation of the suffix tree. Thus, a brief idea about the suffix tree has also been discussed.

Lastly, the importance and suitability of the Aho-Corasick algorithm to a network intrusion detection system has been justified because of the virtue of the algorithm to match all the patterns simultaneously.

Chapter 5

Various Implementations of suffix trees and Experimental results

5.1 Introduction

In this chapter, we discuss the various implementations of the suffix trees, as we had said in the previous chapter. We make a comparable study of the Aho-Corasick algorithm using the various implementations of the suffix trees. Moreover, we also make a comparable study of the various algorithms presented in Chapter-2.

But the main focus of this chapter and of this project is the Aho-Corasick algorithm with the various implementations of the suffix trees. The various implementations of the suffix trees has bee done using a standard sorted set, a sparse hash set and a dense sparse set. We compare the performance of the Aho-Corasick algorithm with these implementations of the suffix tree and with inputs of various sizes.

5.2 Experimental Results of the various algorithms

In this section, we show the experimental results that we have obtained with the various algorithms discussed in Chapter-2. The various algorithms that we have implemented and whose results are presented in this section are the Brute-Force algorithm, the Karp-Rabin algorithm, the Boyer-Moore algorithm and the Knuth-Morris-Pratt algorithm.

To present the results of the running time of these algorithms, we vary the input size, where the input is the words in the English dictionary. The number of patterns to be matched remains the same. The running time (in milliseconds) for the various algorithms are recorded in the following table.

Input- The words of the English dictionary. The number of words are varied, thus varying the input size. The number of patterns to be matched and the size of the pattern is kept constant for a particular input size.

Output- The running time for the various algorithms for varying input size. The time is measured in milliseconds.

Input Size	Running Time (in milliseconds)			
	Brute-Force	Karp-Rabin	Knuth-Morris-Pratt	Boyer-Moore
20000	15	15	17	15
60000	46	45	46	40
100000	74	73	79	68
140000	102	102	108	102
180000	129	132	139	119
200000	144	144	159	133

We also draw a graph for the above tabulated data.

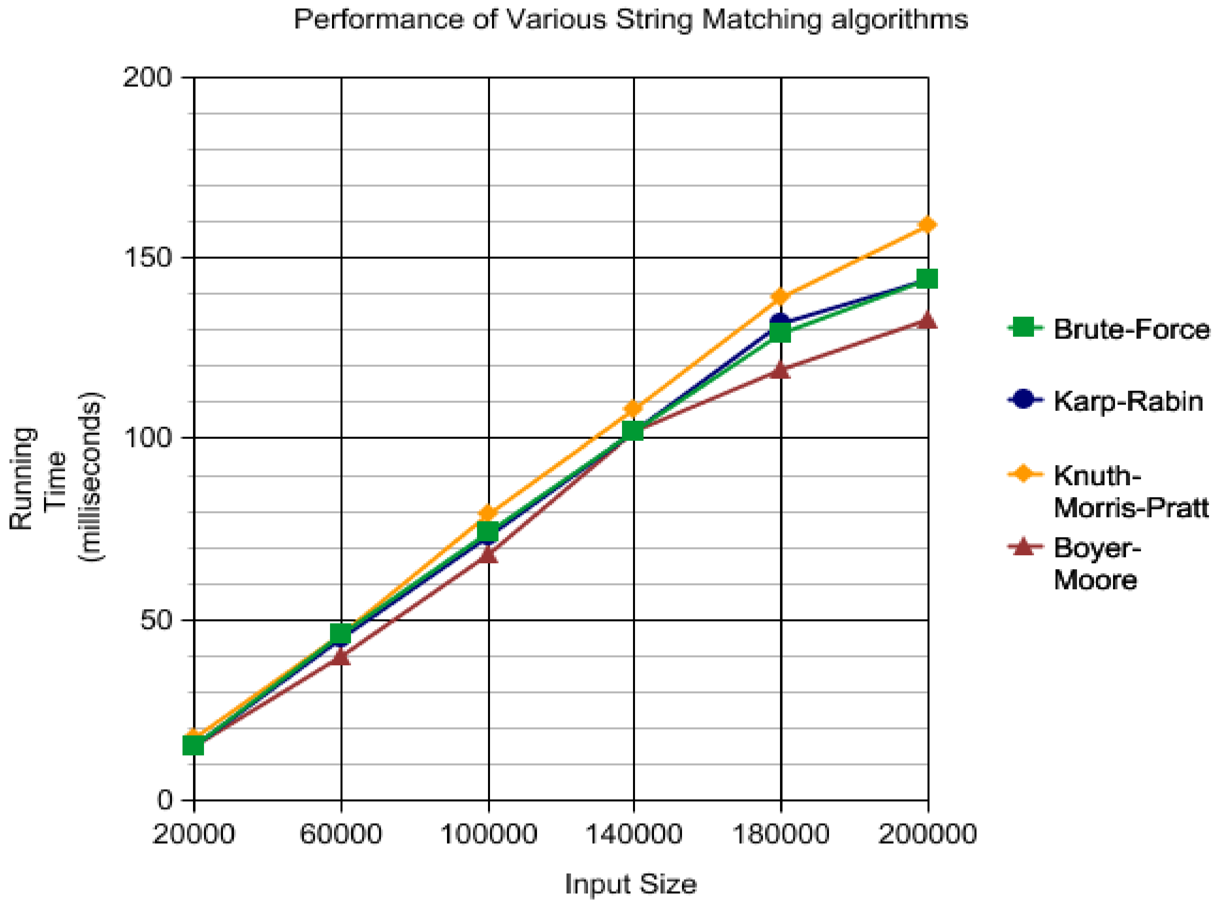

Figure 5.1: Graph showing input size vs Running time

Inference- From the table and the graph we see that there is no trend in the performance between the algorithms. Only that KMP performs the worst and Boyer-Moore performs the best.

5.3 Implementations of the suffix trees

In this section, we discuss the various implementations of the suffix trees. We mainly conduct our study with three implementations of the suffix tree. One of them is an implementation already present in the C++ Standard Template Library, while the other two are hash table implementations of the suffix trees. These hash table implementations are libraries taken from the Google code repository, which can be found here, http://code.google.com/p/google-sparsehash/. This is a freely downloadable library which incorporates various memory-efficient and performance efficient hash map implementation.

We now discuss the various implementations that have been used in the following subsections. We do not discuss the standard set of the C++ STL as it is a part of the language itself. Suffice is to say that the standard set uses a red-black tree internally and gives many functions which can be, and have been, used in the programs to use the functionality provided by the standard set. We now discuss the other implementations that have been used.[17]

5.3.1 Sparse Table

A sparse table is a container which holds values of any type, more specifically, it is a random container. Sparse table is so called because it implements a sparse array, an array which uses less memory for unassigned elements. To explain this, we can say that, in many practical cases, arrays are sparsely populated, that is, most of the indices of the array are unassigned and the indices that have been assigned values take large amount of memory as they might have been assigned values of the struct type. For instance, if we allocate an array a of size 5 and assign $a[2] = [bigstruct]$, the $a[2]$ will take a lot of memory while $a[0], a[1], a[3]$ and $a[4]$ will not because they are unassigned and do not contain any value.

A sparse table t instead consisting of a contiguous array consists of **groups**. Each group consists of a fixed number of indices of the sparse set, say, M indices. The first group keeps track of the elements $t[0]..t[M-1]$, the second group will know about the elements $t[M]..t[2M-1]$. Actually, all the operations that are to be done on a particular element is passed onto the particular group which contains that element and the group does the operation.

When we come down to the level of a group for a particular operation, we are dealing with that group only. A group consists, in addition to the vector which stores the results, a bitmap of size M which stores whether the particular index in question is assigned any value or is unassigned. For example, for a particular group if we ask for a lookup() operation, then first the group consults the bitmap. If $bitmap[i]$ is 0, then the index is unassigned and the operation fails. If $bitmap[i]$ is 1, then the index is assigned and the operation is carried on.

We now discuss the various operations available in sparse table which are important for our use. The various operations are-

I. **find( )-** This method finds the appropriate element in the sparse table. Finding the appropriate vector element is the most expensive part of the lookup operation. This operation takes $\mathrm{O}(M)$ time where M is the group size.

II. **insert( )-** This operation inserts a particular element into the sparse table. Insert operation starts with the lookup operation. If the lookup operation succeeds, then it is a matter of just inserting the new element in the place of the previous element which has been returned by the lookup operation. When the lookup operation fails, then the code needs to find the appropriate group and the appropriate place where the insertion will take place. After finding the position i where to do the insertion, the insertion operation is done and $bitmap[i]$ for the particular group is set to 1.

III. **delete( )-** This operation deletes an element from the sparse table. This operation is similar to the insert operation, in the way that it also begins with a lookup operation. If the lookup operation fails, the this delete is of no significance and is ignored. However, if the lookup operation succeeds then the returned element is deleted, $bitmap[i]$ is set to 0 and all the elements above i are moved down by one position.

Resource usage- We now discuss a little bit about the resource usage of the sparse table. For a sparse table of size N, the space overhead is $N + \frac{48N}{M}$ bits. For the default value of M, which is 48, this is exactly 2 bits per entry. If we increase the value of M, then the overhead of space decreases but the operations discussed above takes longer time. On the other hand, if we decrease the value of M then the space overhead increases but the operations become somewhat faster.

5.3.2 Sparse Hash Set

Sparse hash set is a concrete implementation for use as a suffix tree in place of a standard set. It is basically a hash table which uses a sparse table, which was discussed above, to implement the underlying array. So the basic way that the sparse hash set behaves is the same as that of the sparse table. Here as well the underlying array (or table, in this case) is implemented in the form of groups. All the operations, as discussed above, are done at the group level.

In particular, a sparse hash set uses quadratic internal probing as the collision resolution technique which is employed during the hashing operation. This data structure always stores only one element at one position in the sparse hash set. If the position at which it tries to store the element is already occupied as returned by the lookup operation, then it is not allowed to store the element at that position and it must look for some other position where it must store the element. This is resolved by the quadratic internal probing.

We now discuss the operations of the sparse hash set that we use in our programs.

I. **insert( )-** This function inserts an element into the sparse hash set. To understand better, how the insertion takes place, we take an example. Suppose t is a sparse hash set, then the call to this function like $t.insert(foo)$ proceeds in the manner as described now. foo is first hashed and converted to an integer i, which indicates the index where to insert. Assuming that the default size of the table is 32, we look at $t.sparsetable[i\%32]$. If this place is unoccupied, we insert the element in this place else we look at position $t.sparsetable[(i+1)\%32]$, $t.sparsetable[(i+3)\%32]$, $t.sparsetable[(i+6)\%32]$ and so on. In general, we go on probing the next triangular number.

   If the table is now "too full", then the table is grown in size (generally to twice the value of the present size) and all the values in the tables are re-hashed and inserted in the new table using the method described above. Also if the table is "too empty", then a new table is created whose size is half of that of the size of the current table, and the values are re-hashed into the new table.

II. **find( )-** The function to find a particular value in the table is the same as described for a sparse table above.

III. **delete( )-** This is the function used to erase a value from the sparse hash table. It is almost the same as that for the sparse table. The only difference is that whenever a value is deleted from the table, instead of just removing the value from the table, we replace that position with a default value. This is done so as to remove exceptions that may crop up during the lookup operation.

Resource Usage- Mostly, the resource usage for space will be the same for sparse hash set as that in the sparse table because the sparse hash set uses a sparse table as the underlying array. Time required for the operations is also largely determined by the sparse table implementation but the point to be kept in mind is that some time is also spent on the probing of various indices to find a value due to the quadratic probing that has been used. But 4-5 probes should be enough for a lookup and thus the time spent in such probing can be negligible when compared to the time due to the sparse table implementation.

One point that needs to kept in mind is the space required while shrinking and expanding the table. It will need double the space then, because we need to keep in memory both the new table and the old table containing the values. To decrease the memory requirements, it should be kept in mind to delete the values from the old table as soon as they are hashed and entered in the new table. This is done as to reduce space overhead.

5.3.3 Dense Hash Set

The dense hash set also employs a hash table like the sparse hash set. The hash table aspects of the dense hash set is the same as that of the sparse hash set. The only difference is that while a sparse hash set employs a sparse table for the underlying array, a dense hash set employs, as its underlying array, a simple C array. This implies that the space overhead is more in case of dense hash set especially if large structures are to stored but the time requirements for the various operations are less in case of dense hash set as the sparse table needs to spend more time on memory management which is not the case with simple C arrays.

But with the employment of C arrays in dense hash set, one important question needs to be addressed- how to distinguish between unassigned positions and deleted positions. In a sparse table, this is accomplished by a $bitmap$ and default values for the deleted elements. In this case, thus, we will require two default values, one for unassigned positions and one for deleted positions. When a new dense hash set is made, all the positions in the array are initialized with the default value to be used for unassigned positions.

Resource Usage- The space overhead might a little more than that required by a sparse hash set. But this memory overhead is kind of offset by the performance of the operations. The operations in a dense hash set is much more faster than the sparse hash set because of the usage of simple C arrays in the former case while the latter case uses sparse table.

5.4 Experimental Results for variants of Suffix Tree

In this section, we show the various results that we have obtained with various implementations of the suffix trees in the Aho-Corasick algorithm. The Aho-Corasick algorithm is run with inputs of various sizes and with various implementations of the suffix trees and the running time of these variants are compared with each other and also with the implementation of the Boyer-Moore algorithm.

We now present the results of the study done. The suffix tree of the Aho-Corasick algorithm has been implemented using the following three containers-

I. Standard Set
II. Sparse Hash Set
III. Dense Hash Set

We take the implementation with the standard set as the reference and the same implementation has been tweaked to use the other two containers as well. We also study the performance of the Boyer-Moore algorithm as the Aho-Corasick algorithm is derived from the Boyer-Moore algorithm.

Input- Words from the English dictionary. The number of words is varied and the number of words is referred to as the input size. The number of patterns to be searched is kept at a constant of 50.

Output- The time, in seconds, spent for running the program.

We now give the table for the various running time recorded with varying input sizes for the various programs.

Input Size	Running time (in seconds)			
	Boyer-Moore	Standard Set	Sparse Hash Set	Dense Hash Set
5000	4.23	3.85	3.90	3.81
10000	12.79	9.72	10.04	9.25
20000	25.01	21.64	22.58	20.04
25000	31.15	25.51	26.95	24.91
50000	56.42	47.87	49.32	45.73
60000	70.63	60.01	62.93	58.20
75000	75.22	63.87	66.80	62.17
90000	98.40	85.53	89.21	81.92
100000	112.63	97.48	102.17	93.60

We now plot the graph for the above table.

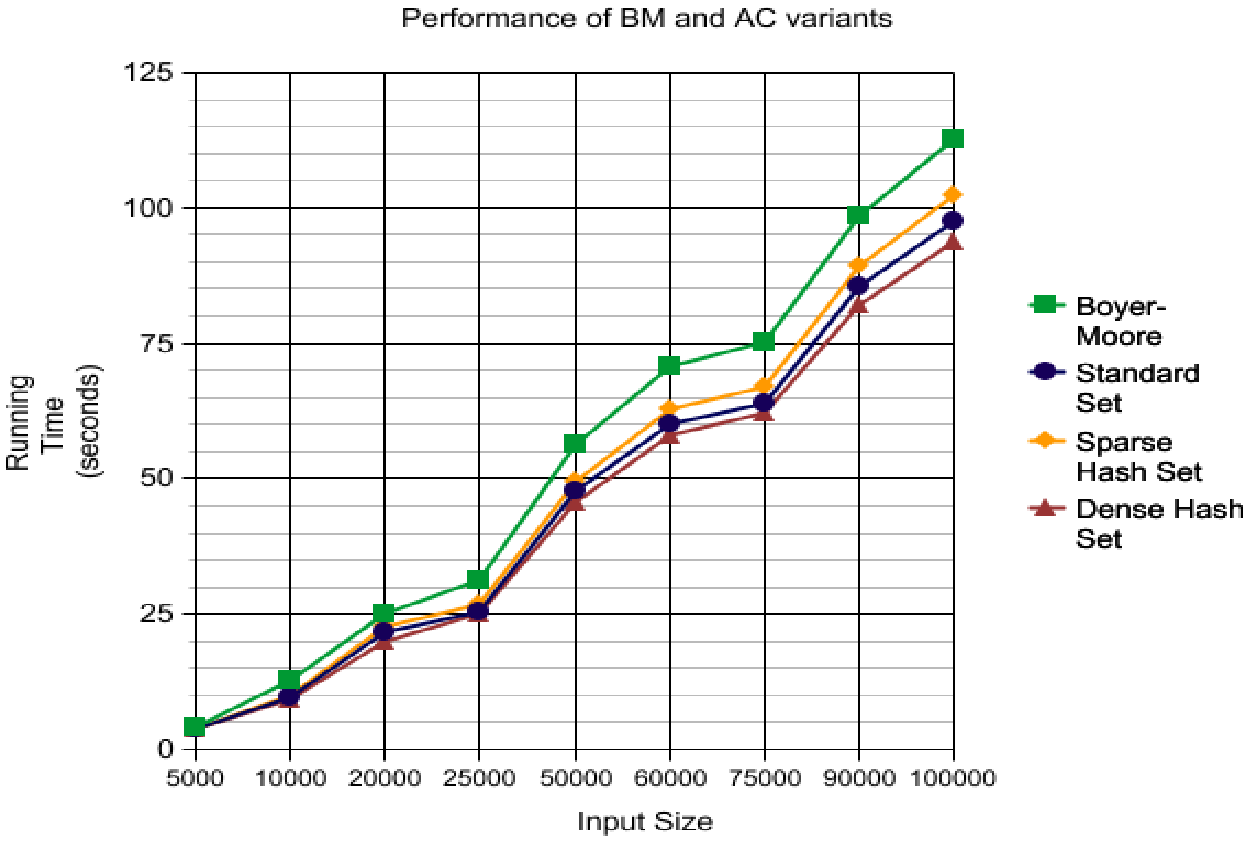

Figure 5.2: Graph showing Input Size vs Running Time

Inference- From the table and the graph, we find that we find that a dense hash set gives the best performance. This better performance gets more pronounced as the input size increases. Clearly, all the variants of the Aho-Corasick algorithm performs better than the Boyer-Moore algorithm and rightly so. One more inference which can be made is that a sparse set gives lower performance when compared to the standard set. This has been explained before as the sparse set needs more time for its operation as it employs sparse table where memory management takes time. But still a sparse set can be used in practical situations as it uses less memory and the performance does not decrease by any great magnitude.

5.5 Conclusion

In this chapter, we gave the various experimental results of the algorithms discussed in chapter 3. The important part in this chapter was the various implementations of the suffix tree and the results of the Aho-Corasick algorithm using the various implementations of the suffix tree and a comparable study of them. We have reached a conclusion that a dense hash set implementation gives the best performance.

Chapter 6

Conclusion

6.1 Conclusion

In this thesis, we have presented and discussed our learning and the experimental results which we have obtained as a result of our undergoing the project.

We have first discussed the theory behind the shortcomings of the traditional security systems, the needs for a network intrusion detection system. Then we moved on to discuss the various string matching algorithm that can be employed in a network intrusion detection system. We, then, concentrated on a better string matching algorithm which can be most practically suited for a network intrusion detection system, the Aho-Corasick algorithm. Then our project consisted of using various implementations of the suffix tree. We show the experimental results for both the normal string matching algorithms as well as for the Aho-Corasick algorithm along with the various implementations of the suffix tree. From the experimental results, we find that Boyer-Moore performs the best among the normal string matching algorithms. The Aho-Corasick algorithm performs better than the Boyer-Moore algorithm and among the variants we find that the one which uses the dense hash set implementation of the suffix tree performs the best.

6.2 Future Work

As our future work, we wish to find more optimized implementations of the suffix tree. As the main performance of the Aho-Corasick algorithm is determined by the operations of the suffix tree, both the construction as well as the implementation of the suffix tree can lead to better performance of the Aho-Corasick algorithm and thus lead to better network intrusion detection system.

REFERENCES

[1] http://csrc.ncsl.nist.gov/publications/nistpubs/800-94/SP800-94.pdf Guide to Intrusion Detection and Prevention Systems (IDPS), NIST CSRC special publication SP 800-94, released 02/2007.

[2] Stefan Axelsson, Research in Intrusion-Detection Systems: A Survey. TR: 98 17. December 15, 1998.

[3] Alex Lukatsky,Protect Your Information with Intrusion Detection. ISBN 1931769117

[4] Wikipedia. Network Intrusion Detection System. http://en.wikipedia.org/wiki/Network_intrusion_detection_system

[5] wps.prenhall.com/esm_clarke_gsgis_4/7/1848/473320.cw/index.html Pearson Education Inc., publishing as Pearson Prentice Hall.

[6] Wikipedia. Snort (Software) http://en.wikipedia.org/wiki/Snort_%28software%29

[7] Martin Roesch, Snort- Lightweight Intrusion Detection for Networks, Stanford Telecommunications, Inc, 13th LISA conference, 1999, 229-223.

[8] Antonatos S, Anagnostakis K G, Markatos E P. Generating realistic workloads for network intrusion detection systems. *Software Engineering Notes*, 2004, 29(1): 207-215.

[9] Christian Charras, Thierry Lecroq, Handbook of Exact String Matching Algorithms, King's College Publications, 2004, ISBN :0954300645.

[10] Wikipedia. Rabin-Karp string search algorithm. http://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_string_search_algorithm

[11] Robert S. Boyer, J Strother Moore, A Fast String Searching Algorithm, Association for Computing Machinery Inc., 1977.

[12] Wikipedia. Knuth-Morris-Pratt Algorithm. http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

[13] Wikipedia. Suffix Tree. http://en.wikipedia.org/wiki/Suffix_tree

[14] Gusfield, Dan, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. USA: Cambridge University Press. ISBN: 0521585198. 1997.

[15] Mark Nelson, Fast String Searching with Suffix Trees, Dr. Dobb's Journal, August 1996.

[16] Wikipedia. Aho-Corasick string matching Algorithm. http://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_string_matching_algorithm

[17] Craig Silverstein: Implementation of sparse_hash_map, dense_hash_map and sparsetable. http://google-sparsehash.googlecode.com/svn/trunk/doc/implementation.html, 2005.

[18] YU Jianming, XUE Yibo, LI Jun, Memory Efficient String Matching Algorithm for Network Intrusion Management System, Tsinghua Science and Technology, ISSN 1007-0214, October 2007.