Implementation of String Matching in Network Intrusion Detection System

Yusuf Rahmat Pratama - 13516062
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516062@std.stei.itb.ac.id

***Abstract*—String Matching algorithm can be used in implementing a Network Intrusion Detection System (NIDS). With the String Matching algorithm and some optimizations, additional security measures, and complementing algorithms, NIDS could contribute in protecting systems connected by networks from suspicious activities by detecting such activities from the network traffic. In particular, String Matching is used in matching incoming or outgoing traffic with some predefined known malicious codes, which NIDS will alert if a match is positive.**

***Keywords*—algorithm, NIDS, security, string matching**

I. Introduction

With the rising dependencies of connected computer networks systems in human's lives, their usage has tremendously increased in the past years. Such events participated in the ever-increasing exposure of those systems, resulting in many attempts of breaching them, with intentions of personal gains, etc. These security threats therefore require development of prevention methods to identify potential threats that flow through the system and protect the systems.

One device type or software that is used to contribute in accomplishing this is with an Intrusion Detection System (IDS), which functions as a monitoring software that analyzes the stream of traffic in a network or system for potential security threats by recognizing pre-defined malicious patterns, checking for policy violations, detecting deviations from regular traffic, and many more.

Many existing algorithms can be used for implementing IDS, with further tweaks and optimizations to fulfill IDS' purpose. One of the algorithms that are usable is the String Matching algorithm, which is the algorithm to find a defined set of strings in a larger text or string. This algorithm can be used by IDS as a checker for malicious patterns that are pre-defined, in which when a traffic or program contains one or more of the patterns defined, the system will notify the management system of the detection of potential security threat.

The String Matching algorithm itself is one of the most general algorithms in computer science. The algorithm has a general purpose of finding patterns, whether exact-matching or defined otherwise, in a larger set of texts or strings, i.e. in a document, paper, etc. String Matching algorithm has many various algorithms with different approaches in string matching, which can be classified according to its input type of text and pattern, the number of patterns that are used, etc.

Therefore, the rest of the paper is organized as follows: Section II presents the definition of the string matching algorithm. Then, Section III describes the Intrusion Detection System and specifically the Network Intrusion Detection System as the example of implementation of string matching. Finally, Section IV discusses the implementation of string matching algorithms in the Network Intrusion Detection System.

II. String Matching

According to [1], string matching can be defined as a basic pattern matching problems, in which given text $T = t_1t_2...t_n$ and pattern $P = p_1p_2...p_n$, string matching is defined as to verify whether string P is a substring of T.

The approaches to string matching can be classified based on many criteria. String matching can be classified based on the preprocessing of the text and/or pattern, in which preprocessing can be used to achieve faster string matching. The classification can be divided into four categories [1]:

1. Neither the pattern nor the text are preprocessed. Example of algorithm in this category is the Elementary algorithms, which is the most basic algorithms used, such as Brute Force.
2. The pattern is preprocessed. Example of this category is the Pattern Matching automata.
3. The text is preprocessed. Factor automata, and index methods belongs in this category.
4. Both the pattern and the text is preprocessed. The pattern matching automata and factor automata also belongs in this category, and also signature methods.

String matching algorithm can also be classified based on the number of patterns used in each iteration of the algorithm. This classification is divided into three categories:

1. Single pattern algorithm.
   Example of this algorithm is the Brute force search, the

Knuth-Morris-Pratt algorithm [2], the Boyer-Moore string search algorithm [3], and the Rabin-Karp algorithm [4].

2. Finite set of patterns algorithm.
   The Aho-Corasick string matching algorithm [5], and the Commentz-Walter algorithm [6] are examples of this category.

3. Infinite set of patterns algorithm.
   This category can be accomplished using regular expression (Regex), in which the patterns cannot be enumerated finitely.

Another classification of the string matching algorithm is by their matching strategy, as categorized in [7] into four categories:

1. Prefix matching, which matches patterns starting from the prefix. Knuth-Morris-Pratt algorithm and the Aho-Corasick string matching algorithm are prefix matching.
2. Suffix matching, which matches patterns starting from the suffix. Boyer-Moore algorithm and Commentz-Walter algorithm are suffix matching.
3. Best factor matching, which matches patterns with the best factor first.
4. Other strategy, such as Brute Force which checks the pattern one by one.

As usage in the implementation of the string matching in this paper, the algorithms that will be discussed are the Brute Force algorithm, the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm.

2.1 Brute Force Algorithm

The brute force algorithm is the simplest and relatively inefficient approach to the string matching algorithm. The algorithm simply iterates the text per character, comparing it to the character in the pattern. If there is a character difference, the algorithm moves one character in the text ahead and re-compare the text with the pattern.

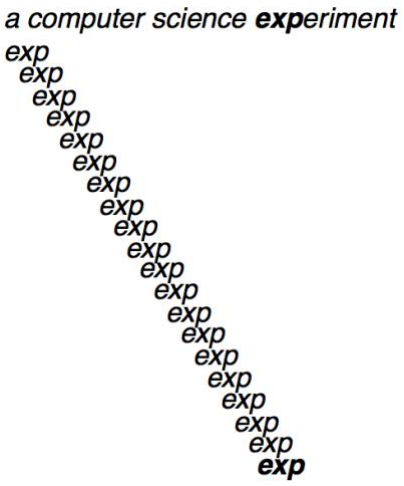

Fig. 1. A brute force algorithm example

In the average case, the brute force algorithm normally only has to iterate one or two characters to determine if it is in the right or wrong position, therefore the average complexity of the algorithm is $O(n + m)$, with n as the length of the text and m as the length of the pattern.

However, when the number of the character variation is small, such as bit matching, the brute force algorithm doesn't work efficiently, as there are many redundancies that the algorithm does. The worst-case scenario results in a complexity of $O(mn)$ which is very inefficient compared to the average case.

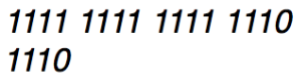

Fig. 2. A bit matching which results in worst-case scenario for brute force algorithm

2.2 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt Algorithm is an improvement of the brute force algorithm, which decreases the redundancy matching by storing information of the pattern to determine where the next match could begin, without storing the previously scanned text itself. Therefore, the algorithm could bypass the redundancy checking, while only taking $O(m)$ memory without remembering the previously scanned characters [2].

The main differentiator of this algorithm is that the Knuth-Morris-Pratt Algorithm uses the *next* table or *failure function* which is pre-computed before the searching algorithm takes place. Each element with index i in the *next* table stores the size of the largest prefix of the pattern until the i^{th} character (*pattern[0..i]*) which is also a suffix of the pattern until the i^{th} character excluding the first index (*pattern[1..i]*). The function ensures that the algorithm won't match any character in the text more than once.

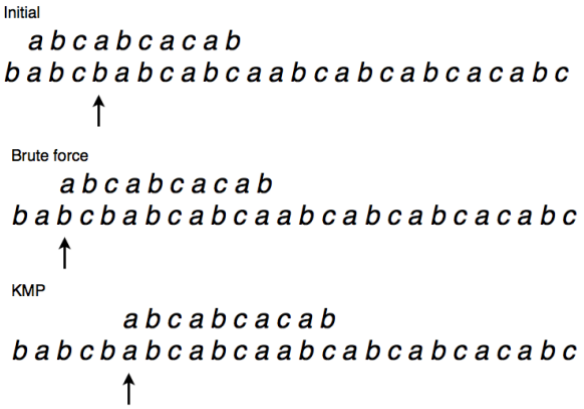

Fig. 3. Difference of efficiency between Brute Force and Knuth-Morris-Pratt algorithm

From Fig. 3, when the scanning of the pattern and the text resulted in a difference, the brute force algorithm

directly shifts the pattern once and restart the comparison from the beginning. In the Knuth-Morris-Pratt algorithm, it ensures that every character in the text is only compared once, so in Fig. 3, the algorithm know that the last character of the text that was checked was *a b c x* with $x \neq a$. From this, the algorithm deduces that no matter what the value of *x* is (as long as it is not *a*), the pattern can be shifted immediately four places to the right. The efficiency of the Knuth-Morris-Pratt compared to brute force algorithm can be seen more clearly when the failure function is bigger.

The complexity of the Knuth-Morris-Pratt algorithm is *O(n + m)*, with n as the length of the text and m as the length of the pattern. This complexity is relatively more efficient than brute force algorithm's, which could rise up to *O(nm)*. As the previous definition suggests, the Knuth-Morris-Pratt algorithm is especially efficient for processing very large text and with relatively small variation of the characters because the algorithm ensures that every character in the text is only compared once, not redundantly compared as with brute force algorithm.

However, the Knuth-Morris-Pratt could be considered inefficient when the text contains very large variation of characters, because such events would lead to a lot of mismatch (with relatively small amount of failure function), so the algorithm will work similarly as the brute force algorithm, with additional memory complexity of *O(m)* to store the table function.

2.3 Boyer-Moore Algorithm

According to [3], the idea behind the Boyer-Moore algorithm is that more information is gained by matching the pattern from the right than from the left. This results in the possibility of the Boyer-Moore Algorithm to make a bigger jump while minimizing the amount of checking required, especially if the text and the pattern resemble natural language.

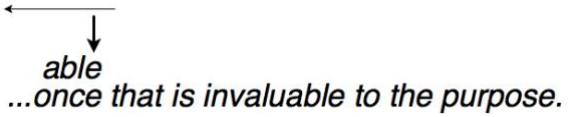

Fig. 4. Boyer-Moore algorithm matches pattern from the right

The Boyer-Moore algorithm uses a *last occurrence function* which is pre-processed before the matching algorithm takes place. The *last occurrence function* maps all the characters variation from the text, then it saves the last index of each character in the pattern, with a -1 value if the character doesn't exist.

The algorithm starts by comparing normally from right to left. However, when it detects a mismatch, the algorithm checks between the three possible cases in order:

1. If the character *x* from the text that is mismatched exists in the pattern and has a last occurrence index that is smaller than the current pointer of the pattern, then the algorithms shifts the pattern such that the last occurrence of *x* in the pattern is aligned with *x*. The algorithm then starts comparing normally from the rightmost of the pattern.

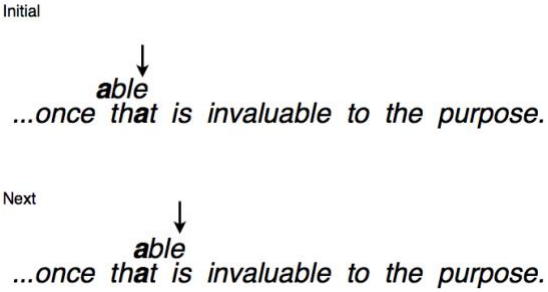

Fig. 5. First case of the Boyer-Moore algorithm mismatch

2. If the character *x* from the text that is mismatched exists in the pattern but has a last occurrence index that is larger than the current pointer of the pattern, then the algorithm shifts the pattern by one character of the text. Then the algorithm starts comparing normally from the rightmost of the pattern.

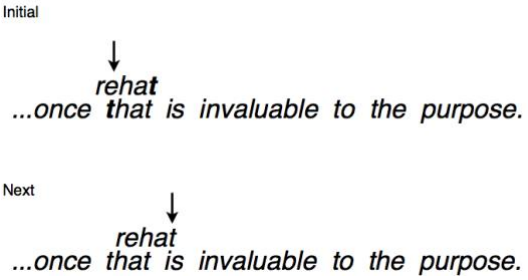

Fig. 6. Second case of the Boyer-Moore algorithm mismatch

3. If the first and second case doesn't apply, which is if *x* doesn't exist in the pattern, then the algorithm shifts the pattern such that the first character of the pattern aligns with the character of the text after *x*.

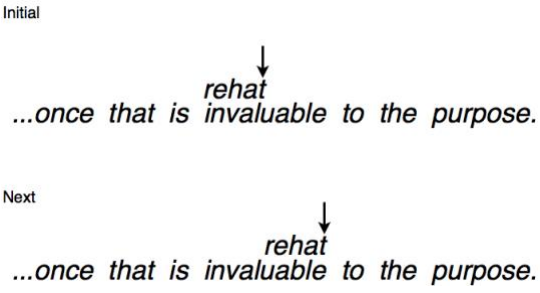

Fig. 7. Third case of the Boyer-Moore algorithm mismatch

The Boyer-Moore algorithm is very efficient in text

with large characters variation compared to brute force algorithm. However, the algorithm suffers in text with small variation of characters, as it implies less "jump" capability of the algorithm, and because of the possibilities of the algorithm to check on a character multiple times, the redundancy can increase dramatically in this situation. The worst-case scenario of Boyer-Moore algorithm is *O(nm)*, with n as the length of the text, and m as the length of the pattern.

III. Intrusion Detection System

According to [8], "Intrusion Detection Systems (IDS) are security tools that, like other measures such as antivirus software, firewalls and access control schemes, are intended to strengthen the security of information and communication systems."

An Intrusion Detection System achieve this by monitoring a network or system for malicious activity or potential security threat. The system then raises an alarm if it detects such activity. Intrusion Detection System serves mainly as monitoring software, which alert other security components that could handle such problems. However, there are some variations of Intrusion Detection System which also has the protection aspects without external software.

Reference [9] defined the Common Intrusion Detection Framework (CIDF) as a generalized structure of an intrusion detection system. According to CIDF, an Intrusion Detection System can be decomposed into four types of components:

1. Event generators
2. Analyzers
3. Databases
4. Response units

Consequently, a hypothetical Intrusion Detection System could be designed with CIDF which takes form of the schematic on Fig. 8.

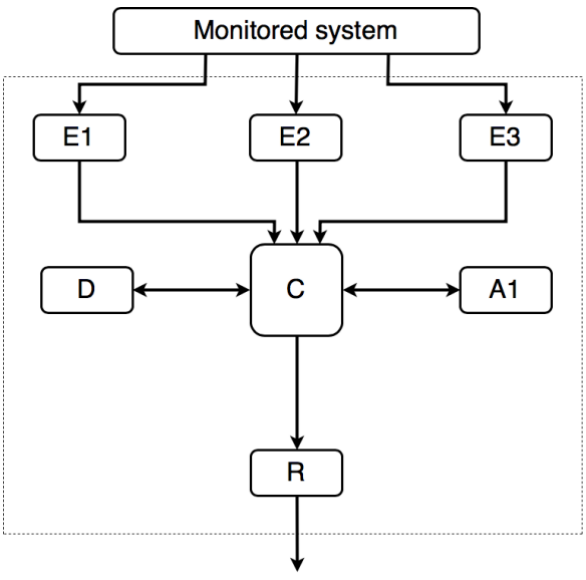

Fig. 8. An example of a hypothetical IDS according to CIDF architecture

Each of the component in the Common Intrusion Detection Framework can be defined as follows [9]:

1. Configuration and Directory Service
   Labeled C in Fig. 8, the configuration and directory service ties the other components together in the CIDF interface. However, this component is optional, as if a component can address its target directly than this service component is not necessary.

2. Event Generators
   Event generators functions as sensors that acquire events such as data flow or traffic from the target system outside of the intrusion detection system environment, convert them to IDS-supported format, and provide the data to the rest of the system. The event generators are labeled as Ei in Fig. 8.

3. Event Analyzers
   Labeled as Ai in Fig. 8, their function is to receive the data from other components, analyze them according to the specified requirements, and return data to other components containing the summary of the input, whether it is considered safe or potentially hazardous.

4. Event Databases
   Event databases simply store data and push queried data. The component is sometimes unnecessary, but it could give persistence to the system's data. Databases can also store pre-defined malicious codes or functions for the system to intrude. Event databases are labeled Di in Fig. 8.

5. Response units
   Response units function as a reaction unit to the events monitored by the system. They carry out various tasks based on the events identified. Some functions that could be added to response units are killing process, blocking access, resetting connection, etc. They could also function to notify other security systems of the intrusion that is detected by the system. The response units are labeled R in Fig. 8.

Generally, Intrusion Detection Systems can be classified according to the place of implementation. As such, it is categorized into two types:

1. Network Intrusion Detection Systems
   Network intrusion detection systems are usually placed inside the network (typically in the strategic points of traffic within the network) and monitors the traffic. The systems analyze the traffic by matching the traffic to pre-defined data of malicious activity, which usually contains known and previous implemented attacks on the system. The systems could also analyze for deviations of traffic

from normal conditions for detecting malicious activity. When a potentially dangerous activity is detected, the systems will raise an alarm to the administrator, or directly resolve the activity, according to the implementation the systems.

2. Host Intrusion Detection Systems
   Host intrusion detection systems are placed inside individual hosts or devices inside a network. The systems only monitor traffic from the particular device only. The functionalities are similar to the network intrusion detection systems.

IV. String Matching in Network Intrusion Detection Systems

String matching algorithms are particularly useful for some aspects of the implementation of Network Intrusion Detection Systems. Recall that Network Intrusion Detection Systems function by matching the traffic with pre-defined known malicious activities, which could be applied using string matching algorithms using pre-defined code data as patterns.

Using the CIDF architecture, a simple Network Intrusion Detection System implementing string matching module can be modeled as such in Fig. 9.

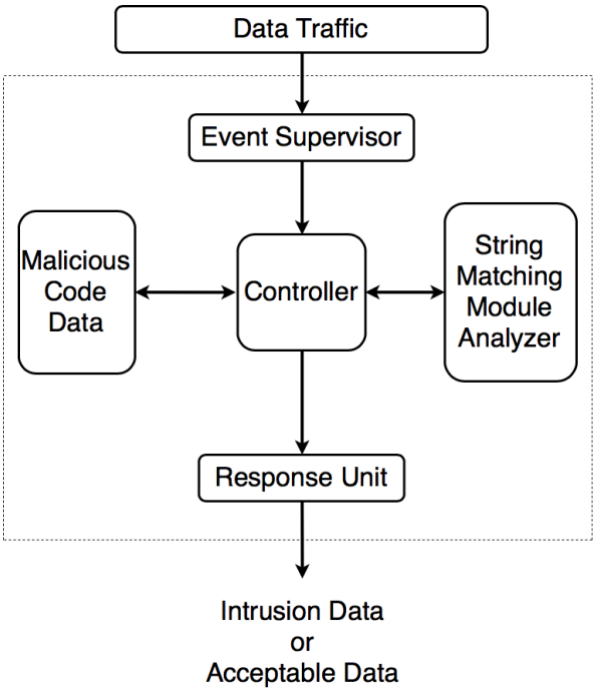

Fig. 9. A Network Intrusion Detection System model with string matching implementation

Fig. 9. describes the string matching algorithms as an Event Analyzer, which receives input of the data traffic that has been pre-processed by the Event Supervisor, matching the data with the malicious code data that are pre-defined and stored in the Databases, and returns whether the data contain malicious activities or if they are acceptable.

The Response Unit then labels the data according to the results from the analyzer. If the data is an intrusion data, the response unit will also send alerts to the management software for further action on the labelled data.

Realization of the simple Network Intrusion Detection System is done using the Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm for the string matching analyzer. Because both algorithms are exact pattern matching, every data traffic has to be detected and intruded by the system. Furthermore, the traffic has to be processed by the system before analyzed by the algorithm.

Experiments suggested that for the simple string matching analyzer to work effectively, the data have to be processed such that:

1. The redundancies both in the data traffic that are monitored and the malicious data pattern stored are minimalized or removed completely. This is because data redundancy could lead to incorrect matching result when using exact pattern matching algorithm.

2. The stored malicious data are written as simple as possible without sacrificing the meaning and the matching of the data, because complex-written stored data could increase the complexity of the algorithm, slowing the systems down.

However, ultimately the usage of exact pattern matching as analyzer for the network intrusion detection system is limited, as further modifications of the malicious code implemented within the traffic could bypass the analyzer completely.

Consequently, other methods of pattern matching could be implemented, such as using regular expression, where patterns are matched according to the expression defined, eliminating the need of exact match, and with the right expression, could identify malicious code whatever the modifications might be.

Furthermore, to increase the effectiveness of the systems, other forms of analyzers could also be used concurrently with the string matching module, such as deviation recognition, or other varieties of secure analyzers.

V. Acknowledgment

The writer would like to thank God for without Him nothing could ever be achieved. The writer would also like to thank Mr. Rinaldi Munir as the coordinator of the Algorithm Strategies lectures and initiator of the task, as well as Mrs. Ulfa as Algorithm Strategies' lecturer. The writer thanks every author in which his or her works are referenced in the writer's paper. Last but not least, the writer thanks his parents for continually supporting the writer's journey in life.

References

[1] Melichar, Borivoj, Jan Holub, and J. Polcar. Text Searching Algorithms. Vol. 1. 2 vols., 2005.

[2] Knuth, D.E., Morris, J.H., and Pratt, V.R. *Fast pattern matching in strings*. SIAM Journal on Computing Vol. 6, No. 2. Philadelphia: SIAM, 1977

[3] Boyer, R.S., and Moore, J.S., *A fast string searching algorithm*. ACM Vol. 20, No. 10. New York: Association for Computing Machinery, 1977.
[4] Karp, R.M.; Rabin, M.O. "Efficient randomized pattern-matching algorithms". *IBM Journal of Research and Development* Vol. 31, No. 2. New York: IBM, 1987
[5] Aho, A.V., and Corasick, M.J. *Fast pattern matching: An aid to bibliographic search*. ACM Vol. 18, No. 6. New York: Association for Computing Machinery, 1975.
[6] Commentz-Walter, Beate (1979). *A String Matching Algorithm Fast on the Average*. International Colloquium on Automata, Languages and Programming. LNCS. **71**. Graz, Austria: Springer. pp. 118–132.
[7] Navarro, G., and Raffinot, M. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. New York: Cambridge University Press New York, 2002
[8] Garcia-Teodoro, P., Diaz-Verdejo, J., Macia-Fernandez, G., Vazquez, E. "Anomaly-based network intrusion detection: Techniques, systems, and challenges.". Computers & Security, Vol 28, Issues 1-2, 18-28. New York: Elsevier, 2009
[9] Staniford-Chen S., Tung B., Porrar P., Kahn C., Schnackenberg D., Feiertag R., et al. The common intrusion detection framework. Internet draft, 1998.

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, 13 May 2018

Yusuf Rahmat Pratama - 13516062