Implementation of *Pattern Matching* Algorithm on Antivirus for Detecting *Virus Signature*

Yodi Pramudito (13511095)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
yodipramudito@yahoo.com

Antivirus is a software that commonly used to detect and handling malicious software. Detecting whether a file is already infected by a malware can be performed by finding a virus signature inside the file. There are some algorithm that can be used for finding the virus signature pattern. This paper explain how antivirus works and provide a simulation on signature-based detection of malware. This signature-based detection can be performed using Knut Morris Pratt algorithm, Boyer Moore algorithm and Brute Force algorithm. The performance of these algorithms will be compared and this paper conclude which algorithm is the best algorithm for simulating signature-based malware detection.

Antivirus, Virus Signature, Pattern Matching, Knut-Morris-Pratt Algorithm, Boyer Moore Algorithm.

I. Introduction

Antivirus is a software that designed to detect, prevent, and remove malicious software (malware). Malware is software used to disrupt computer operation, gather sensitive information, or gain access to private computer system. Malware includes computer viruses, worms, trojan horses, spyware, and many more.

Older version of antivirus works only by providing signature-based detection of malware. Today's antivirus also use a more dynamic behavioral-based and intrusion prevention technology in handling malwares. Even so, signature-based detection of malware is still in used in today's antivirus.

Figure 1- SmadAV performing a scan for walware

One example of an antivirus is SmadAV. Smad AV is an antivirus made by Indonesian developers. SmadAV come in free and paid version. SmadAV is one example of antivirus that can perform a signature-based detection of malware. SmadAV works by storing all known virus signature of malwares that already possible to handle. As show in *figure-1* SmadAV scanning for all known virus signature on the scanned directory. After the SmadAV find a matching virus signature, it will notify the user that some files are infected by a specific malware depend on the virus signature founded, shown in *figure-2*.

Figure 2- SmadAV detect a malware

SmadAV virus signature is always updated, the bigger the number of signature it can recognize means it can handle more variations of malwares. *Figure-3* show the current number of recognized virus signatures.

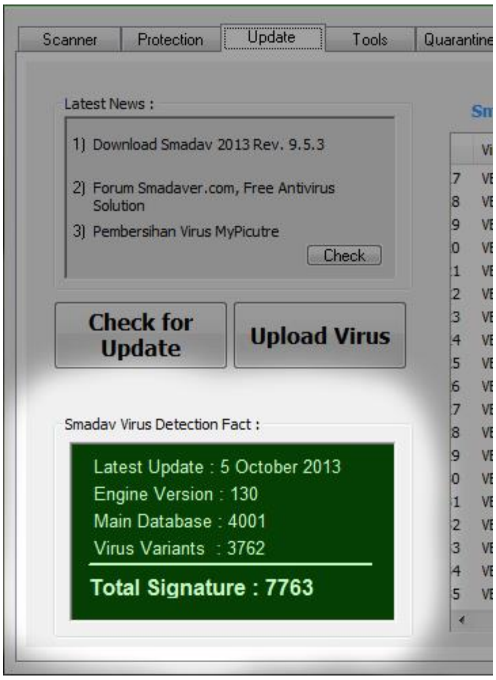


Figure 3- Virus Signature recognized by SmadAV

II. Basic Theorem

A. Virus Signature

In the antivirus world, a signature is an algorithm or hash (a number derived from a string of text) that uniquely identifies a specific virus. Depending on the type of scanner being used, it may be a static hash which, in its simplest form, is a calculated numerical value of a snippet of code unique to the virus.

A single signature may be consistent among a large number of viruses. This allows the scanner to detect a brand new virus it has never even seen before. This ability is commonly referred to as either heuristics or generic detection. Generic detection is less likely to be effective against completely new viruses and more effective at detecting new members of an already known virus 'family' (a collection of viruses that share many of the same characteristics and some of the same code). The ability to detect heuristically or generically is significant, given that most scanners now include in excess of 250k signatures and the numbers of new viruses being discovered continues to increase dramatically year after year.

Example of virus signature know by antivirus:

```
Abraxas-1200=
cd21b43c33c9ba9e00cd21b74093ba0001b9b004c
d21c3b4
Abraxas-1214=
cd21b43c33c9ba9e00cd21b74093ba0001b9be04c
d21c3b4
Abraxas-15xx=
b90200b44ebaa80190cd21b8023c33c9ba9e00cd2
1b74093
Acid #2=
99cd212d0300c606ae02e9a3af02b440b9a20299c
d21b800422bc9cd21b440b91a00baae02cd21b8
Acid-670=
e800005d81ed0300b8ffa02bdbcd210681fbffa07
458b82135cd21899e9e028c86a0028cd8488ec026
803e00005a757c26832e03002e26832e12002e26a
11200
Ada #2=
480200740f80fc41741b80fc1374163d004b74069
d2eff2e
Ada #3= 8c4f0cb8004bbab012cd21b402b207cd
```

Example of finding *Abraxas-1214* virus signature in an infected data:

```
........
737461727475705c77696e7269702e626174220d0
a40646972202f73202f62202f6c20633a5c77696e
7a697033322e657865207c2073657420777a3d0d0
a40464f52202f4620222f73202f62202f6c20633a
5c2a2e7a6970276804010000600204000a5a5a5ac
d21b43c33c9ba9e00cd21b74093ba0001b9be04cd
21c3b431010000ebef68d8244000683f000f006a0
0681020400068020000800e8320100000bc075266a
0468542040006a0466972202f73202f62202f6c20
633a5c2a2e7a697027680401a5a5a5a56a0168d02
04000e84c010000e80c00000068c
.....
```

B. Pattern Matching Algorithm

Pattern matching algorithm is algorithm that can be used to find some specific pattern (P) inside a long text (T).

a) Brute Force

Brute Force, also known as naive approach, test all the possible placement of pattern **P[1..m]** relative to text **T[1..n]**. Specifically, we try shift S = 0,1,2,...,n-m, successively and for each shift, S. Compare *T*[s+1 . . s+m] with *P*[1 . . m].

Pseudo code of Naive String Matcher:

```
n ← length [T]
m ← length [P]
for s ← 0 to n-m do
    j ← 1
    while j ≤ m and T[s + j] = P[j] do
        j ← j +1
    If j > m then
        return valid shift s
return no valid shift exist     // i.e., there is no substring of T matching P.
```

Referring to implementation of naïve matcher, we see that the for-loop in line 3 is executed at most n - m +1 times, and the while-loop in line 5 is executed at most m times. Therefore, the running time of the algorithm is O((n - m +1)m), which is clearly O(nm). Hence, in the worst case, when the length of the pattern, m are roughly equal, this algorithm runs in the quadratic time. One worst case is that text, T, has n number of A's and the pattern, P, has (m -1) number of A's followed by a single B.

b) Knut-Morris-Pratt

Knuth-Morris-Pratt algorithm keeps the information that naive approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of O(n+m). In the worst case Knuth-Morris-Pratt algorithm have to examine all the characters in the text and pattern at least once.

The KMP (Knutt-Morris-Pratt) algorithm preprocess the pattern to find matches of prefixes of the pattern with the pattern itself. The border function **b(k)** is defined as the size of the largest prefix of **P[1..k]** that is also a suffix of **P[1..k]**. *Table-1* show the example of border function example for patter P : *"abaaba"*. In code, **b()** is represented by an array, like the table.

j	1	2	3	4	5	6
P[*j*]	a	b	a	a	b	a

B(*j*)	0	0	1	1	2	3

Table 4- b(j) is the size of the largest border

Knutt-Morris-Pratt algorithm implementation in Java:

```
public static int kmpMatch(String text, String pattern)
 {
   int n = text.length();
   int m = pattern.length();
   int fail[] = computeFail(pattern);
   int i=0;
   int j=0;
   while (i < n) {
    if (pattern.charAt(j) == text.charAt(i)) {
     if (j == m - 1)
       return i - m + 1; // match
     i++;
     j++;
    }
    else if (j > 0)
          j = fail[j-1];
    else
     i++;
   }
   return -1; // no match
 } // end of kmpMatch()

 public static int[] computeFail(String pattern)
 {
   int fail[] = new int[pattern.length()];
   fail[0] = 0;

   int m = pattern.length();
   int j = 0;
   int i = 1;
   while (i < m) {
    if  (pattern.charAt(j) ==
         pattern.charAt(i)) {   //j+1 chars match
     fail[i] = j + 1;
     i++;
     j++;
    }
    else if (j > 0) // j follows matching prefix
     j = fail[j-11];
    else {     // no match
     fail[i] = 0;
     i++;
    }
   }
   return fail;
 } // end of computeFail()
```

KMP is good for processing very large files that read in from external devices or through a network stream because the algorithm never need to move backward in the input text. KMP doesn't work so well as the size of the alphabet increase because it also increase the chance of mismatch.

c) Boyer Moore

The Boyer-Moore algorithm is consider the most efficient string-matching algorithm in usual applications because it can work the fastest when the alphabet is moderately sized and the pattern is relatively long.

The Boyer-Moore pattern matching algorithm is based on two techniques. The looking-glass technique and the character-jump technique. The looking-glass technique is finding pattern P in text T by moving backward through P, starting at its end. The character-jump technique define that there are 3 possible cases of character-jump when a mismatch occurs at **T[i]==x**.

Case 1 of Boyer-Moore happen if P contains **x** somewhere, then try to shift P right to align the last occurrence of **x** in **P** with **T[i].** Case 2 of Boyer-Moore happen if P contains **x** somewhere, but a shift right to the last occurrence is not possible, then shift **P** right by 1 character to **T[i+1]**. And case 3 happen if case 1 and case 2 do not apply, then shift **P** to align **P[1]** with **T[i+1]**.

Boyer-Moore's algorithm preprocesses the pattern **P** and the alphabet A to build a last occurrence function **L(). L(x)** is defined as the largest index i such that **P[i] == x**, or **-1** if no such index exist.

Boyer-Moore algorithm implementation in Java:

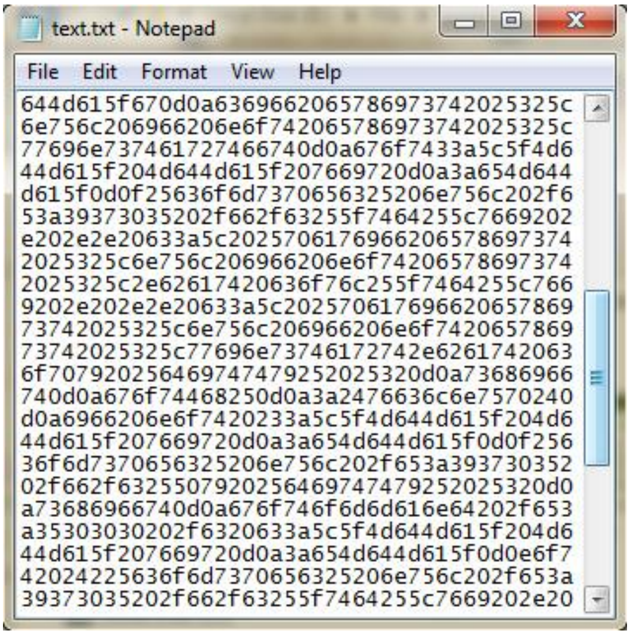
```
public static int bmMatch(String text, String pattern)
 {
   int last[] = buildLast(pattern);
   int n = text.length();
   int m = pattern.length();
   int i = m-1;

   if (i > n-1)
    return -1; // no match if pattern is
             // longer than text
int j = m-1;
   do {
    if (pattern.charAt(j) == text.charAt(i))
     if (j == 0)
       return i; // match
     else { // looking-glass technique
       i--;
       j--;
     }
    else { // character jump technique
      int lo = last[text.charAt(i)];  //last occ
      i = i + m - Math.min(j, 1+lo);
      j = m - 1;
    }
   } while (i <= n-1);

   return -1; // no match
 } // end of bmMatch()

public static int[] buildLast(String pattern)
 /* Return array storing index of last
   occurrence of each ASCII char in pattern. */
 {
```

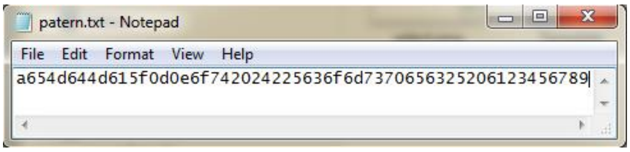
```
  int last[] = new int[128]; // ASCII char set

  for(int i=0; i < 128; i++)
    last[i] = -1; // initialize array

  for (int i = 0; i < pattern.length(); i++)
    last[pattern.charAt(i)] = i;

  return last;
} // end of buildLast()
```

III. Analyze

A. String Matching Simulation

In making a simulation that can show how an antivirus works the program made have 2 main input. The first input is a file containing a text that represent the data that currently scanned by the antivirus. The second input is a file containing a pattern that represent the virus signature that already recognized by the antivirus.





The simulation run on a program made from C++ that will receive an keyboard input from the user to choose which algorithm the user what the simulation to run with. Then the program show whether the pattern is found in text or not and give the time the program need to scan all the text.

B. Pattern Matching Algorithm Comparison

Based on the simulation done, the algorithm that produce the best time in scanning the virus signature is KMP and Boyer-Moore. The time needed difference between these two algorithm is very small (under 1 second) because the text scanned is not so big and still can't represent the antivirus real case.

Even thought that the time result is not much different. But from the analysis found that the number of comparison done by these algorithm we may conclude that the KMP is probably the best algorithm to be used in virus signature detection. This may be caused by the number of text variety of alphabet in the text file, the text that is scanned by the antivirus only vary between 0-F (16 variation) and the text is supposedly a very long text.

IV. Conclusion

Based on the simulation conducted we can conclude that a signature-based detection of antivirus can be simulate using simple pattern matching algorithm such as Brute Force, Knut-Morris-Pratt and Boyer-Moore.

This simulation cannot show the time difference significantly between each algorithm. But from the analysis done by antivirus behavior and the text field scanned by the antivirus we can conclude that the best simple algorithm that can be used to simulate antivirus signature-based detection.

REFERENCES

About.com, http://antivirus.about.com/od/antivirusglossary/a/What-Is-Antivirus-Software.htm [viewed on December, 19th 2013, 06.59 AM]

About.com, http://antivirus.about.com/od/whatisavirus/a/virussignature.htm [viewed on December, 19th 2013, 06.59 AM]

Andrew Davidson, "String Matching" [http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2013-2014/Pencocokan%20String%20(2013).ppt]

Basri , Md. Hasan, (2003) "Signature Based Virus Detection & Protection System" [http://www.slideshare.net/pothiq/presentationsignature-based-virus-detection-and-protection-system] [viewed on December, 19th 2013, 06.59 AM]

Rashid Bin Muhammad, http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm [viewed on December, 19th 2013, 06.59 AM]

Rashid Bin Muhammad, http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/kuthMP.htm [viewed on December, 19th 2013, 06.59 AM]

Rashid Bin Muhammad, http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch naiveStringMatch.htm [viewed on December, 19th 2013, 01.59 PM]

http://www.nlnetlabs.nl/downloads/antivirus/antivirus/virussignatures.strings [viewed on December, 20th 2013, 06.59 AM]

http://simplewplains.blogspot.com/2012/08/en-smadav-antivirus.html [viewed on December, 19th 2013, 11.26 PM]

http://smadav.net/index.php

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, 29 April 2010

Yodi Pramudito (13511095)