Research Journal of Applied Sciences, Engineering and Technology 5(4): 1183-1186, 2013
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: June 20, 2012 Accepted: August 08, 2012 Published: February 01, 2013

Adaptive File Comparison Technique for Secured Data Transmission Environment

[1]S. Srinivasan and [2]C. Chandrasekar
[1]Department of Computing, SASTRA University, Kumbakonam, Tamilnadu, India
[2]Periyar University, Selam, Tamilnadu, India

Abstract: Huge amount of data are stored in the network security environment and quantities of data in these fields tend to increase year on year. For this reason efficient File matching algorithms should be used which minimizes computer storage which results in minimum response time on searching process. There are many problems involved in file content matching technique. This technique works for finding the file content similarity. In this study, we propose the new algorithm called Adaptive File Comparison Technique, which converts the source and destination file contents into tokens and finally it makes a matrix format data called sparse matrix. The comparison technique compares the sparse matrices of source and destination files to find the content similarity and dissimilarity which leads to an efficient content matching technique rather than comparing the entire file content.

Keywords: Adaptive file comparison technique, databases, file matching, sparse matrix

INTRODUCTION

Computer network has been widely used and played important roles in people's life. Along with its popularity is the serious network security problem. With each passing day there is more critical data accessible in some form over the network.

Keyword matching is one of the most efficient network security techniques. It compares the network packets or flow with the defined keywords utilizing string matching algorithm. The keyword is a string indicating network intrusion, attack, virus, Spam or dirty network information. Keyword matching is wildly used in Network Intrusion Management System (NIMS), anti-virus, anti-spam, network content filtering and so forth.

String matching is one of the most important areas that have been studied in computer science. The general string matching algorithm is well studied. Since 1977, with the publication of the Boyer-Moore (BM) algorithm and Knuth-Morris-Pratt (KMP) algorithm, there have been several hundred studies published that deal with exact string matching.

While be applied to network security, string matching algorithm's speed is an enormous pressure as in other application areas. And the keywords are known, so they could be pre-processed before searching. There are also some specific characteristics in network security application. First, the keywords pre-processing only be done while the network security equipment or software start up or the keywords update. So the pre-processing time could be neglect when designed or selected the string matching algorithm. Second, we concentrate on the practical performance of the algorithms not the theoretical bounds. Third, the anti-attack capacity must be concerned to protect network security equipment itself. In the content matching technology, the main drawback is that the entire file content is considered for the evaluation and it may takes too much of time to find a tiny change in the file, this is a serious issue which could be solved through the proposed Adaptive File Comparison Technique. It compares the file content by creating sparse matrices from the content and this approach provides similarity and dissimilarity details can be obtained effectively.

LITERATURE REVIEW

String matching deals with a string $T = t_1 t_2 ... t_n$ (the text) of size n and searches it for all occurrences of another, shorter string (Boyer and Moore, 1977; Knuth *et al*., 1977) $P = p_1 p_2 ... p_m$ (the pattern) of size m (usually n>m). Both strings build over a finite set of character called an alphabet denoted by Σ.

String-matching algorithms scan the T with the help of the sliding window mechanism. The size of the window is gene rally equal to m. To a j (1< = j< = n), the window is positioned on $t_j t_{j+1} \ldots t_{j+m-1}$. The position j is called a check point. At each check point, the characters of the window are compared with the characters of the pattern (Chakkrit, 2007). After a whole match of the pattern or after a mismatch, the window is shifted along the T according to the heuristics of each algorithm.

Corresponding Author: S. Srinivasan, Department of Computing, SASTRA University, Kumbakonam, Tamilnadu, India

```
Existing algorithm:
FileComparison ((File1Content, File2Content)

Input   : FileContents
Output: FileComparisonResult

begin:
        File1data = ""; File2data = "";

        File1data = FormatContent (File1Content);
        File2data = FormatContent (File2Content);
        if (Compare (File1data, File2data))
        {
          //data content are exactly same
          FileComparisonResult = "True";
        }
        else
        {
          //data content are different
          FileComparisonResult = "False";
        }
        return FileComparisonResult;
       end if;
end begin;
```

Fig. 1: Existing Algorithm for FileComparison

- Boyer-Moore (BM) Algorithm
- Boyer-Moore-Horspool (BMH) Algorithm
- Reverse Factor (RF) Algorithm
- Karp-Rabin (KR) Algorithm
- Knuth-Morris-Pratt (KMP) Algorithm

These are the string matching algorithms (Horspool, 1980) and the same concept to be extended with the little modification to file matching algorithm.

The Traditional algorithm which is to be followed to perform file comparison process is stated in Fig. 1.

Normally File matching algorithm each byte or character or block of data is converted in to an matrix representation (Lecroq, 1992; Nimisha and Deepak, 2012). The data is analysised by individual string character (Sunday, 1990) is compared with one another.

METHODOLOGY

In order to overcome the identified limitations in the existing work and to improve the performance of the proposed application, this study introduces the following techniques. They are:

0	11	0	0	0
1	0	0	1	12
0	0	0	17	0
53	0	0	0	0
0	7	19	0	2

Fig. 2: Source matrix

0	7	0	0	0
1	0	0	1	12
0	0	0	17	0
53	0	0	0	0
0	7	19	0	2

Fig. 3: Destination matrix

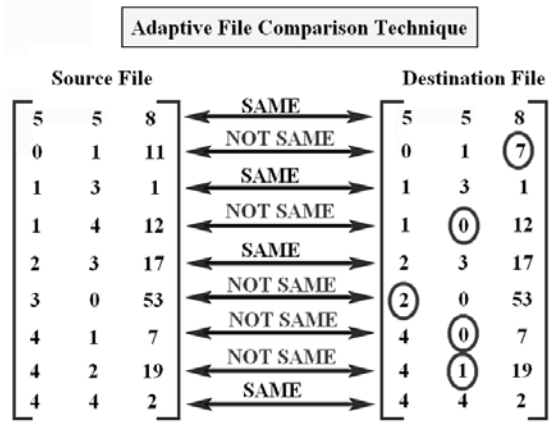

Fig. 4: Adaptive file comparison technique

- Token Computation
- Adaptive file comparison technique

Token computation: In order to achieve assurance of data storage verification, our scheme entirely relies on the pre-computed token verification. Before comparing the content, each file is computed with the user pre-computation technique to generate certain number of short verification tokens. Each token covering a certain row in the file is converted into matrix by using the allocated tokens.

Need token computation: Input file and output file is converted into square matrix of size nXn. In this square matrix, token is allocated for each and every row; allocated token must be a unique identifier which helps to perform the comparison technique effectively.

Adaptive File Comparison Technique (AFCT): We proposed a new algorithm named "Adaptive File Comparison Technique". Here original file is available in the database in the network environment. The new file is compared with old file with this algorithm.

Each file is converted in matrix; the matrix is once again converted in to sparse matrix format. Here source file which is stated in the following Fig. 2 is the original file and compared with destination file which is stated in the following Fig. 3. In this below example the source file row 0, column 1 and the element 11 has changed in destination file element has 7.

For example the Source and Destination file data is represented in the matrix format and is stated in the following Fig. 2 and 3.

In this example the original matrix is converted in to Sparse Matrix. The Sparse matrix format is represented in the Fig. 4. In that first row element is 5 58, the first 5 indicates row size of the matrix, next 5

Proposed algorithm:

```
AdaptiveFileComparisonTechnique (File1Pathinfo, File2Pathinfo)

Input    : FilePathinfo, NoiseValueInfo
Output : FileComparisonResult

Begin   :
DefaultInfo = ”InvalidDataFormat”; File1Data = ””, File2Data = ””;
DataSimillarityCount = 0; DataNonSimillarityCount = 0;
Simillaritypercent = 0.0; NonSimillaritypercent = 0.0;

  File1 = FileNameExtraction (File1Pathinfo);
  File2 = FileNameExtraction (File2Pathinfo);
  if (File1.ISContentEmpty || File1.ISNull && File2.ISContentEmpty || File2.ISNull)
 {
    FileComparisonResult = DatafaultInfo;
    Print FileComparisonResult;
  }
 else
 {
   File1Data = LoadNONNOISEVALUES (File1); File1Data =
   ConvertTOSPARSEMATRIXFORMAT (File1Data);
   File2Data = LoadNONNOISEVALUES (File2);
   File2Data = ConvertTOSPARSEMATRIXFORMAT (File2Data);
   for (i = 0, j = 0; i<File1Data.Size(), j<File2Data.Size(); i++, j++)
            {
            If (File1Data (i).EQUALS (File2Data (j)))
            {
            DataSimillarityCount++;
            }
            else
            {
            DataNonSimillarityCount++;
            }
                        }
Simillaritypercent = CALCULATEPERCENTSIM (DataSimillarityCount,
DataNonSimillarityCount);
NonSimillaritypercent = CALCULATEPERCENTNONSIM (DataSimillarityCount,
DataNonSimillarityCount);

Print (“Simillarity percentage = “Simillaritypercent);
Print (“NonSimillarity percentage = “ NonSimillaritypercent);
}
end begin;
```

Fig. 5: Algorithm for adaptive file comparison technique

indicates the column size of the matrix and the 8 indicates the total number of Non-zero elements in the matrix. Other values in the row indicate the concern row value, column value and value of Non-zero elements.

Through the proposed algorithm we can easily identify that the data is updated or not by the content of the first line. If the first line in both the source and destination file sparse matrix are same then there may not be any updates in the file but we need to check the entire content in both the source and destination sparse matrices but if we identified any change in the first line then it denotes that there is a update in the file content.

The proposed algorithm is stated in the following Fig. 5 which produces and efficient output on the comparison technique. If needed for further verification we have to compare the entire element in the matrix. The final result will be generated after the completion of the entire process.

PERFORMANCE ANALYSIS

The proposed Adaptive File Comparison Technique is implemented and studied thoroughly and the performance of this proposed system is analyzed with the existing traditional file content comparison technique in terms of execution, processing time and level of accuracy. From the results, our proposed work performs well as compared with the existing approach. Result is stated that the proposed technique performs better than the existing algorithm in terms of its execution time.

EXPERIMENTAL RESULTS

We analyze the performance of our proposed approach by the limitation of Execution time.

From the Fig. 6 and 7, it is observed that the proposed work finds best results with minimum response time as compared with that of existing work.

Through the results, we also revealed that our proposed system achieves best performance for both the low and high volume of file content.

CONCLUSION

In this study, we presented a new algorithm Adaptive File Comparison Technique as a fast file comparison algorithm. Using our new proposed algorithm leaded to two main advantages compared to other compared algorithms. Experimental result show that our proposed algorithm the search with less number of comparisons and faster elapsed searching time between the file. Therefore our proposed algorithm is suitable for searching any kind of database as well as in any other files with the extension of text and java file searching applications.

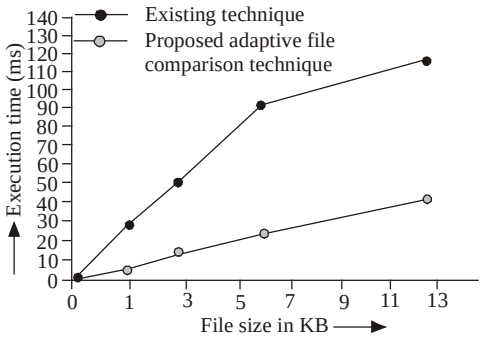

Fig. 6: Evolution of adaptive file comparison technique

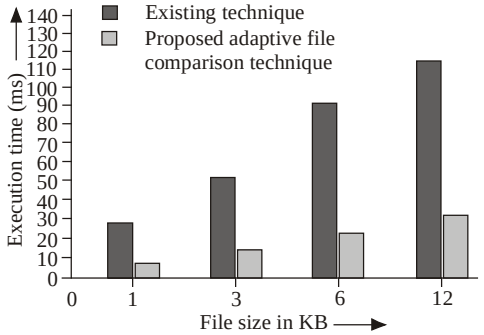

Fig. 7: Execution time based report with various file sizes

RECOMMENDATIONS

This study is applicable only for text and java files, it can be expanded by using word document, XL sheets, PDF, image, etc.

REFERENCES

Boyer, R. and J. Moore, 1977. A fast string searching algorithm. Commun. ACM, 20(10): 762-772.

Chakkrit, S., 2007. A comparison and analysis of name matching algorithms. Int. J. Appl. Sci. Eng. Technol., 45: 252-257.

Horspool, R.N., 1980. Practical fast searching in strings. Software Prac. Exp., 10(6): 501-506.

Knuth, D., J. Morris and V. Pratt, 1977. Fast pattern matching in strings. SIAM J. Comput., 6(2): 323-350

Lecroq, T., 1992, A variation on the Boyer-Moore algorithm. Theoret. Comp. Sci., 92(1): 119-144.

Nimisha, S. and G. Deepak, 2012. String matching algorithms and their applicability in various applications. Int. J. Soft Comp. Eng., 1(6).

Sunday, D.M., 1990. A very fast substring search algorithm. Commun. ACM, 33(8): 132-142.