Huffman Algorithm for Antivirus Quarantine

Sonny Lazuardi Hermawan 13511029
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511029@std.stei.itb.ac.id
sonnylazuardi@gmail.com

***Abstract*— This paper is about using Huffman Algorithm for Antivirus to secure malware from computer users. Since system of software is becoming more and more sophisticated, the malware will try to innovate to keep their existence in the system. We need to secure the infected file so that user would not be in danger. Quarantine is one solution to secure the infected file. The best way of antivirus to do quarantine is by encrypting the infected file. By using Huffman algorithm in encrypting the infected file, we will get some benefits such as data compression, safe encryption for the infected file, and a small running time.**

***Index Terms*— Huffman, algorithm, malware, quarantine, compression, encryption.**

I. Introduction

A. *Computer and Malware*

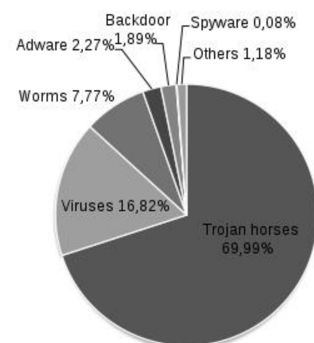
Computer has becoming a part of human life. Actually, computer has taken role to help the people to get their needs. Computerized inventory system manages the supply chains for food we bought. Computer-controlled water system dispenses the water. Computer manages financial transactions to pay for it all. Students need a computer for learning and doing their task. Those things prove that computers is holding main role in society's infrastructure. Have you ever imagined when computer stop working and losing data? There will be a big loss if it happens. As the computer system grows, malware will never stop infecting computer system. They always come with new way to spread themselves and avoiding the antivirus detection. There are three characteristics of malware [2].

1. *Self-replicating* malware actively attempts to propagate by creating new copies, or instances, of itself. Malware may also be propagated passively, by a user copying it accidentally, for example, but this isn't self-replication.

2. The *population growth* of malware describes the overall change in the number of malware instances due to self-replication. Malware that doesn't self-replicate will always have a zero population growth, but malware with a zero population growth may self-replicate.

3. *Parasitic* malware requires some other executable code in order to exist. "Executable" in this context should be taken very broadly to include anything that can be executed, such as boot block code on a disk, binary code in applications, and interpreted code. It also includes source code, like application scripting languages, and code that may require compilation before being executed.

Based on the characteristics, there are some types of viruses that may harm your computer and your data. This is the statistics for malware spreading based on the types.



http://en.wikipedia.org/wiki/File:Malware_statics_2011-03-16-en.svg
Fig. 1 Malware spreading by types

Since system of software is becoming more and more sophisticated, the malware will try to innovate to keep their existence in the system. The traditional method of antivirus that uses virus definition database for scanning viruses would have taken a lot of time to be updated. Antivirus now has already had some heuristics method to anticipate the threat of malware. Even they have succeeded to suspect a virus; they have to ensure that the file is really a malware and analyze the file so that the virus file can be cleaned later. The problem is how to secure the file that may not be a malware or may be a system file so that it can be restored later? The answer is quarantine.

II. QUARANTINE

After finding a suspected virus file, antivirus software needs to quarantine the infected file. Quarantine is only a temporary measure, and may only be done until the user decides how to handle the file (e.g., giving approval to disinfect it) [2]. In the other words, Antivirus software may have generically detected a virus, but have no idea how to clean it. Quarantine will be used until the antivirus software has been updated and can deal with the virus that was discovered.

Actually, Quarantine is a simple technique to secure the suspected file by copying the file to the "quarantine" directory, removing original infected file, and disallowing user to access the infected file. But then come the problem, user may have changed the permission to the infected file and they can be executed even in a quarantine. That means it is not solving the dangerous file, it just move the danger from one place to another. The user may have the virus back if they are not aware of what the quarantine is. So, there come some solutions to secure the infected file.

1. Renaming the infected file

The simplest solution to this problem is by renaming the infected file, so it can't be directly executed. By changing the extension name of a file, it will not be executed as an executable (.exe). The weakness of this solution is that the virus body is still there. Once the infected file renamed back or executed by other program, the virus would run again in the system. They just have a little effort to be alive again.

2. Render files in Invisible directory

Another solution is to render the files in the quarantine directory invisible - what can't be seen can't be copied. Antivirus software can accomplish this feat using file-hiding techniques like stealth viruses and rootkits use. However, this may not be the best idea, as viruses may then try to hide in the quarantine directory, letting the anti-virus software cloak their presence. There could also be issues with false positives produced by virus-like behavior from anti-virus software [2].

3. Encrypt infected file

One solution is to encrypt quarantined files by some trivial means, like an XOR with a constant. The virus is thereby rendered inert, because an executable file encrypted this way will no longer be runnable, and copying the file does no harm. Also, an encrypted, quarantined file is readily accessible for disinfection [2].

Based on three solutions above, we can conclude that we must secure the infected file so that it can't be accessed by user and can be easily cleaned after updating antivirus. The best way to do this is in solution number 3. We can secure the infected file by encrypting the file itself. But is that XOR encryption is really the best solution? Can we use other encoding method?

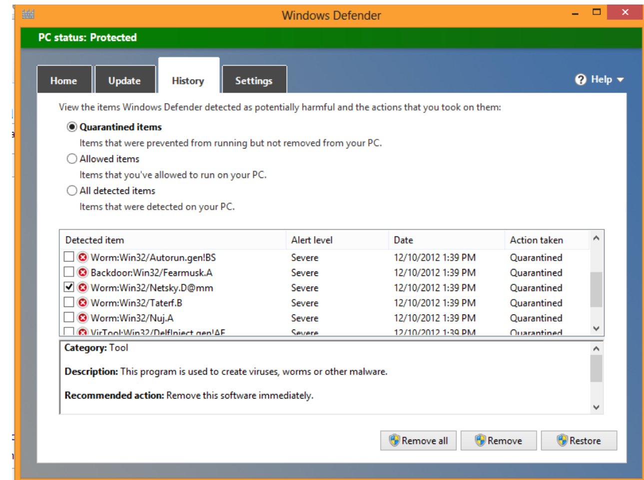


Fig. 2 Windows Defender Quarantine

III. HUFFMAN ALGORITHM

In data storage, file with big size will take a lot of space of storage that we have. This problem can be solved with the encoding of file content in concise way, so that the space needed become smaller. This encoding method is also called data compression. Data compression is done by encoding each character in file content with shorter code [3].

Huffman algorithm is a very popular way to represent data with minimum memory needed to store the data. Huffman algorithm was created to reduce data redundancy in a file. Huffman algorithm was developed by David A. Huffman while he was a Ph.D., student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" [5].

A. *Huffman Encoding*

Coding system which is widely used is ASCII encoding. With ASCII, each character will be coded in 8-bit binary. These are some example of ASCII character.

DEC	OCT	HEX	BIN	Symbol
65	101	41	01000001	A
75	113	4B	01001011	K
84	124	54	01010100	T
85	125	55	01010101	U

Table I ASCII table

Based on the encoding rule above, string 'KAKAKTUA' can be represented by this sequence of bit:

010010110100000101001011010000010100101101010100010101010101000001

From the ASCII encoding, 8 characters need 8 x 8 = 64 bit (8 byte). To minimize the sum of bit, length of code for each character must be shorten as short as possible, especially for the character which has the biggest frequency. This thought is the basic of the Huffman encoding. These are the steps to get the Huffman code

.

1. Count the frequency of occurrence of each character in the string;
2. Choose two symbols that has the fewest probability
3. Make a Huffman tree from the symbol from bottom up using greedy algorithm;
4. Determine the Huffman code for each symbol based on the convention bit;
5. Convert the string into new bit representation from the Huffman code for each symbol.

The first thing to get Huffman code is making the probability table of each symbol. The process of making Huffman code is by making binary tree which is called Huffman tree.

Symbol	Frequency	Probability
K	3	3/8
A	3	3/8
T	1	1/8
U	1	1/8

Table II Probability Table and the Huffman Code for string 'KAKAKTUA'

Choose two symbols with the lowest probability. In the example above we choose T and U). Those two symbols is then combined to a node of symbol TU for the parent of symbol T and U with probability of 1/8 + 1/8 = 2/8. This new symbol is treated as new node and included for the searching of the next symbol which has lowest probability.

Choose the next two symbols (the new symbol included) with the lowest probability. In this example the next two symbols are TU (2/8) and K (3/8). Do the same things as the step before. The result is the new symbol TUK with the probability of 2/8 + 3/8 = 5/8.

Do the same procedure for the next two symbols which have the lowest probability. The next two symbols are A (3/8) and TUK (5/8). The combination of them is ATUK with the probability of 3/8 + 5/8 = 8/8.

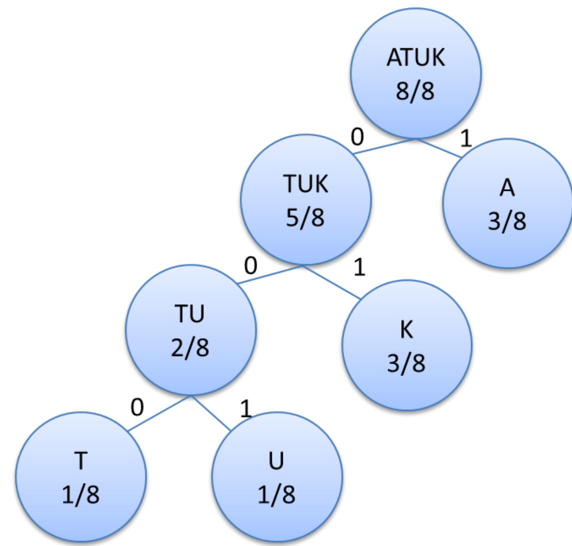


Fig. 3 Huffman tree for strings 'KAKAKTUA'

Leaf in the Huffman tree represented the symbol which is used in the strings. Each code for symbol give label 0 for the left branch and label 1 for the right branch. After making the path from root to leaf, we can get the code for each symbol. From the Huffman tree above, we get these codes for each symbol.

Symbol	Huffman Code
K	01
A	1
T	000
U	001

Table III Huffman code for each symbol

With the Huffman coding, the string 'KAKAKTUA' can be represented with these sets of bit:

011011010000011

The symbols that often exist are represented with shorter code than the other symbols. Code for every symbol can't be the prefix of the other code because it will cause the ambiguity in the decoding process. The symbols that often exist have the code with the fewest sum of bit. There is no symbol that has the prefix of other symbols. Huffman code is not unique, this means code for each characters is different on every strings depends on the frequency of the character itself. Besides, the decision whether the node in the Huffman tree is placed left or right will also determine the final code but it's not influenced the length of code.

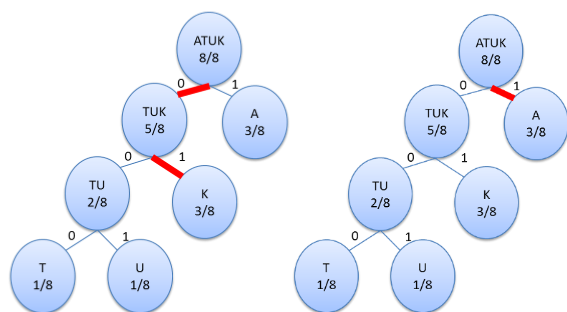
With this Huffman coding, the sum of bits to make the string 'KAKAKTUA' is only 15 bit. If we compare with the ASCII code, we get the difference of 64 – 15 = 49 bit.

In this case, Huffman code save 49/64 * 100% = 76.56% of space. Huffman code typically saves space from 20% up to 90%.

B. *Huffman Decoding*

The preceding method is used to make Huffman code. How to read the Huffman code and getting the strings back? To recognize the information from Huffman code, we need to reverse the process of encoding which is often called decoding. The reverse process, i.e. to reconstruct the sequence of symbols in the source, is called decoding [4]. Huffman code is also a compression algorithm, so the decoding is just like decompression. The decompression algorithm involves the operations where the codeword for a symbol is obtained by 'walking; down from the root of the Huffman tree to the leaf for each symbol.

Take the previous example for the decoding process. We want to decode 011011010000011 to the original string using the Huffman tree.



(a) 011011010000011 (b) 011011010000011

Fig. 4 Decoding string from Huffman tree

Figure 4 shows the first two steps of decoding the string. It shows symbol K and A. The decoder reads 0s or 1s bit by bit. Current bit is highlighted (yellow). In step (a), starting from the root of the Huffman tree, we move along the left branch one edge down to the left child since a bit 0 is read. Then, we move along the right branch to the right child since a bit 1 is read. When we reach a leaf, the symbol K at the leaf is output. This process starts from the root again until it reaches the leaf. In step (b) it will move to the right child since a bit 1 is read and also it reaches the leaf. Then, the process begins from the root again. So, we can conclude the decoding process in this three points (Algorithm) [4].

1. Read the coded string bit by bit. Starting from the root, then traverse one edge down the tree to a child according to the bit value. If the current bit read 0 we move to the left child, otherwise, to the right child.
2. Repeat this process until we reach a leaf. If we reach a leaf, we will decode one character and restart the traversal from the root.
3. Repeat this read-and-move procedure until the end of the message.

IV. Huffman for Executable Files

Many people say that Huffman algorithm is the best for the text file compression. Huffman is not good for binary file like executable files because they have their own compression method. This is not a wrong statement, but we will try how Huffman code works for binary file like executable file.

A. *Lossless compression*

If we are talking about executable file, we are dealing with sensitive data which means every single bit in the file can't be removed. If it loses some data, it may not be executed anymore. Lossless data compression is a class of data compression algorithms that allows the exact original data to be reconstructed from the compressed data. The lossless compression only allows constructing an approximation of the original data, in exchange for better compression rates. The process of Huffman lossless compression on executable is like text file Huffman encoding. The different is we must split each binary code from the executable file as a character and encode it with Huffman method.

For the comparison of Huffman compression with other compressor, we see Zhuff compression tool that uses Huffman algorithm. Zhuff is compression software designed for speed (especially decompression speed). It is based around LZSS & Huffman mechanisms [7].

Pos.	Compressor Name	Compressed Size (bytes)	Compress Ratio (%)	Bits per Byte (b/B)
027	WinRAR 4.1b3	1283777	66.83	2.6533
167	WINZIP 8.0	1732476	55.24	3.5806
203	Zhuff 0.2	1961239	49.33	4.0534

http://www.maximumcompression.com/data/exe.php

Table IV Comparison of Huffman compression for executable file with other compressor

The data above comes from the compression of an executable file Acrobat Reader 5.0 executable (acrord32.exe) with total file size of 3870784 bytes. If we compare Huffman compression (Zhuff), it is below the other popular compressor such as WinRAR and WinZip. But, if we see the compression ratio, it's not very bad, Huffman can save 49.33% spaces for executable file. It proves that Huffman can be used for executable file that we will use it for quarantining an executable virus file.

V. Huffman for Quarantine

Now, we are trying to decode the virus file into the Huffman code. In this case, we will not trying to detect a

virus file, but we assume that the file is a positive virus file. Then, we will do quarantine for the virus file. The Huffman code will be used for quarantine of antivirus.

A. Virus Experiment

The experiment needed to prove that Huffman can be done in quarantining a virus. For example, we will try to quarantine a Worm.Win32.VB.kz (Winamps.exe). It is an Indonesian worm which has popular name of 'Amburadul'. This worm can spread and duplicate itself to the system and make self-defense by auto running at the startup.

Firstly, we will use Zhuff compression tool to get information of how much Huffman works for this executable virus file.

```
*** Zhuff v0.8, by Yann Collet (Nov 28 2011) ***
Compressed Filename is : D:\Dangerous\Koleksi Virus\Winamps.ex
Detected : 4 cores
Compressing D:\Dangerous\Koleksi Virus\Winamps.exe.bak using 1
ion level = 0)
Compression completed : 63KB --> 12KB  (19.30%) (12351 Bytes)
Total Time : 0.02s ==> 3.8MB/s
(CPU : 0.03s = 184%)
```

Fig. 5 Compression of Worm.Win32.VB.kz file

From the figure above, we can see the Huffman compression for this executable virus file. The actual size of virus file is 63KB can be compressed into 12KB by using Huffman compression. It can save 80.7% spaces. The time needed for the encoding is 0.02 seconds in speed of 3.8MB/s. But, Zhuff is not a pure Huffman algorithm. Let's check about the compression of pure Huffman encoding.

B. Huffman for Virus File

We can get the source of Huffman encoding and decoding from planet-source-code.com by Fredrik Qvarfort written in VB (Visual Basic) [8]. This is the pure Huffman encoding for the file. We can use this source for our antivirus with the credits to Fredrik. This Huffman encoding process is the same with the process of Huffman encoding explained above. The encoded file will have prefix of "HE3", and the Huffman tree will be saved in the first part of the file. If the destination file is larger than the source file, it will leave uncompressed and will give "HE0" prefix. Meanwhile, the decoding process includes the identification of the file, extracting Huffman tree, and decode the actual data. The identification process checks the prefix of file whether it is "HE3" or not. If it is "HE0", this will return the uncompressed data. Then, it will create a Huffman tree and decode the data from the Huffman tree. The usage of the Huffman encoding and decoding is by filling the parameter for original file path and compressed/decompressed file path.

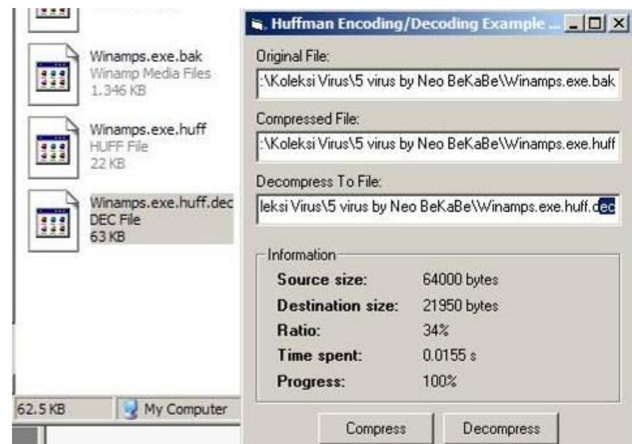


Fig. 6 Huffman Encoding process of the virus file

The figure above shows the result of the Huffman encoding of the virus file (Winamps.exe.bak). The compression ratio is 34% from 64KB to 21KB. This pure Huffman encoding can save 66% spaces. It means Huffman encoding could be done for quarantining viruses.

The next procedure is decoding the file. In the decoding process, it will identify whether the file is compressed using Huffman encoding or not. Then it will create Huffman tree and begin the decoding process using the Huffman tree.

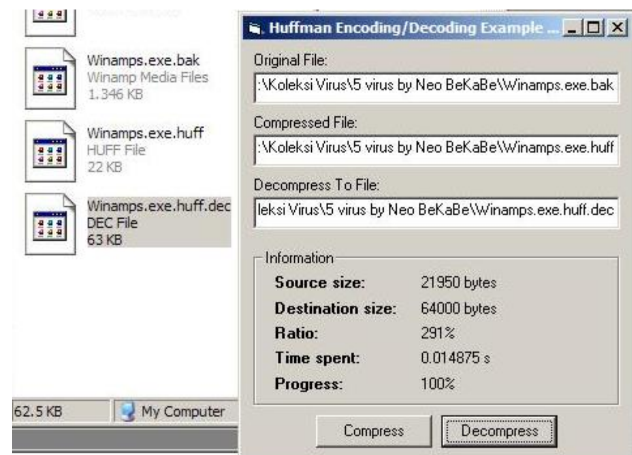


Fig. 7 Huffman Decoding process of the virus file

The decoding result is the same with the original file. We can prove by the size and the content of the decoded file and the original file. The time needed for decoding is faster than the encoding process that is 0.014s. It means that Huffman compression succeeds to convert the quarantined file back to the original file without losing any data (lossless compression).

C. Huffman Coding Benefits

Based on the experiment above, we can get three important factors which is the benefits from Huffman encoding for quarantine of antivirus.

1. Encryption

The basic function of quarantine is to secure the executable virus file to be executed by the user. One of the ways to do this is by using encryption for the executable file. The encryption will prevent the user by directly access the original virus file. From the experiment of virus quarantine above, we get the encrypted virus file after encoding it using Huffman coding. Two figures below show the original (after decoding) file and the encrypted file (after encoding).

(a) Huffman Encoded File (b) Huffman Decoded File

Fig. 8 Encryption by Huffman Coding

2. Size

Viruses have various file size. There are some viruses that have a big file size. When we want to do quarantine for that files, we will lose a lot of space just for holding the dangerous file in our disk. It means that we need compression for quarantined virus file so that it doesn't eat a lot of space in our disk. From the experiment above, we get 66% saved space (compression ratio 34%). It means we can save more spaces after doing quarantine and use it for other purposes.

3. Speed

Doing encryption and compression need some times. This Huffman coding, based on the experiment above, shows a relatively small time to encode and decode (±0.01s). The bigger the file being compressed, the slower the compression will be. But, Huffman algorithm have a fairly efficient running time, because it has complexity of O(n log n).

VI. Conclusion

Huffman algorithm can be used for quarantine in antivirus because of these three things.

1. Huffman algorithm can be used for encrypting the virus file so that the file cannot be directly executed by user.
2. Huffman algorithm can reduce the virus file size so that it can save more spaces in the user's disk.
3. Huffman algorithm has a fairly efficient running time so that it can quarantine some viruses in a little time.

VII. Acknowledgment

I want to say thanks for lecturer of Discrete Structure, Mrs. Harlili and Dr. Ir. Rinaldi for the patience to teach. I also thanked my parents for the support to me. Thanks to Mr. Hirin for sharing a lot about virus and antivirus with me. Thanks to Fredrik Qvarfort for the Huffman encoding and decoding in Visual Basic source code.

References

[1] C. W. Ko. *Method and apparatus for detecting a macro computer virus using static analysis*. United States Patent. February 2004.

[2] J. Aycock, *Computer Viruses and Malware*, Springer 2006.

[3] R. Munir. *Diktat Kuliah IF 2091 Struktur Diskrit 4th ed*. Program Studi Teknik Informatika STEI ITB. 2008.

[4] M. I. Pu. *Fundamental Data Compression*. Elsevier 2006.

[5] http://www.huffmancoding.com/my-uncle retrieved at 16 December 2012 14:48

[6] http://www.maximumcompression.com/data/exe.php retrieved at 17 December 2012 07:00

[7] http://fastcompression.blogspot.com/p/zhuff.html retrieved at 17 December 2012 07:10

[8] http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=11000&lngWId=1 retrieved at 17 December 2012 08:00

Declaration

I hereby declare that the paper is my own writing, not an adaptation, nor translation from another person's paper, and not a form of plagiarism.

Bandung, 18 Desember 2012

Sonny Lazuardi Hermawan
13511029