Detecting Metamorphic viruses by using Arbitrary Length of Control Flow Graphs and Nodes Alignment

Essam Al daoud
Zarka Private University, Jordan
essamdz@zpu.edu.jo

Ahid Al-Shbail
Al al-bayt University, Jordan
ahid_shbail@yahoo.com

Adnan M. Al-Smadi
Al al-bayt University, Jordan
smadi98@aabu.edu.jo

Abstract

Detection tools such as virus scanners have performed poorly, particularly when facing previously unknown virus or novel variants of existing ones. This study proposes an efficient and novel method based on arbitrary length of control flow graphs (ALCFG) and similarity of the aligned ALCFG matrix. The metamorphic viruses are generated by two tools; namely: next generation virus creation kit (NGVCK0.30) and virus creation lab for Windows 32 (VCL32). The results show that all the generated metamorphic viruses can be detected by using the suggested approach while less than 62% are detected by well known antivirus software.

Key words: metamorphic virus, antivirus, control flow graph, similarity measurement.

1. Introduction

Virus writers use better evasion techniques to transform their virus to avoid detection. For example, polymorphic and metamorphic are specifically designed to bypass detection tools. There is strong evidence that commercial antivirus are susceptible to common evasion techniques used by virus writers[1]. Metamorphic Virus can reprogram itself. it use code obfuscation techniques to challenge deeper static analysis and can also beat dynamic analyzers by altering its behavior, it does this by translating its own code into a temporary representation, edit the temporary representation of itself, and then write itself back to normal code again. This procedure is done with the virus itself, and thus also the metamorphic engine itself undergoes changes. Metamorphic viruses use several metamorphic transformations, including Instruction reordering, data reordering, inlining and outlining, register renaming, code permutation, code expansion, code shrinking, Subroutine interleaving, and garbage code insertion. The altered code is then recompiled to create a virus executable that looks fundamentally different from the original. For example, The source code of the metamorphic virus Win32/Simile is approximately 14,000 lines of assembly code. The metaphoric engine itself takes up approximately 90% of the virus code, which is extremely powerful[2]. W32/Ghost contains many procedures and generates huge number of metamorphic viruses, it can generate at least 10! = 3,628,800 variations[3].

In this paper, we develop a methodology for detecting metamorphic virus in executables. we have initially focused our attention on viruses and simple entry point infection. However, our method is general and can be applied to any malware and any obfuscated entry point.

2. Related works

Lakhotia, Kapoor, and Kumar believe that antivirus technologies could counter attack using the same techniques that metamorphic virus writers use; identify similar weak spots in metamorphic viruses [4]. Geometric detection is based on modifications that a virus has made to the file structure. Peter Szor calls this method shape heuristics because is far from exact and prone to false positives [5]. In 2005 Ando, Quynh, and Takefuji introduced a resolution based technique for detecting metamorphic viruses. In their method, scattered and obfuscated code is resolved and simplified to several parts of malicious code. Their experiment showed that compared with emulation, this technique is effective for metamorphic viruses which apply anti-heuristic techniques, such as register substitution or permutation methods[6]. In 2006 Rodelio and others use code transformation method for undoing the previous transformations done by the virus. Code transformation is used to convert mutated instructions into their simplest form, where the combinations of instructions are transformed to an equivalent but simple form [7]. Mohamed and others use engine-specific scoring procedure that scans a piece of code to determine the likelihood [8]. Bruschi, Martignoni, and Monga proposed a detection method control flow graph matching. Mutations are eliminated through code normalization and the problem of detecting viral code inside an executable is reduced to a simpler problem[9]. Wong and Stamp experimented with Hidden Markov models to try to detect metamorphic malware. They concluded that in order to avoid detection, metamorphic viruses also need a degree of similarity with normal programs and this is something very challenging for the virus writer[10].

3. The proposed method

This section introduces new procedures to extract partial control flow graph of any binary file. Two main points are considered during the development of the suggested algorithms, first point is to reorder the flow of the code by handling "*jmp"* and *"call*" instructions, and second point is to use one symbol for all alternatives and equivalent instructions. The output of Algorithm 1 is stored in the matrix *ALCFG* and contains arbitrary number of the nodes. Moreover the sequence of the nodes is represented by using symbols to be used in the similarity measurement.

Algorithm 1: Construction of Arbitrary length of Control Flow Graph (*ALCFG*)

Input: Disassembled portable executable file (x), the number of the file lines (*n*), the start location (j), the required number of the nodes (*m*).

Output: *ALCFG* m×m matrix and node sequence array *NodeSeq* contains m nodes

Steps:

1- Call prepare *op* matrix (the size of *op* matrix is *n×4*)

2- Call prepare the matrices Labels and JumpTo (the size is *c×2* and *e×3*)

3- Call Construct the matrix *ALCFG*

Algorithm 2: Prepare *op* matrix (the size of *op* matrix is *n×4*)

Input: Disassembled portable executable file (*x*), the number of the file lines (*n*), the start location (*j*), the required number of the nodes (*m*).

Output: *op* matrix of size *n×4* (this matrix contains the jump instructions and the labels)

1- Load the matrix *op[n][4]* from the file *x*, where the opcode *i*, is stored at the row *i*, the column *op[i][1]* will be used to store the labels (for simplicity we will consider each label as an opcode), the column *op[i][2]* will be used to store the instructions (*mov ,jmp, add,…*), the column *op[i][3]* will be used to store the first operand, the column *op[i][4]* will be used to mark the rows that are processed, assume that default value is 0.

2- Delete the rows that do not contain label or jump instructions (jump instructions such as *call, ret, jmp, ja, jz, je*…). In this step a special action must be consider if the *"ret"* instruction is preceded directly by push instruction, in this case *"ret"* is replaced by *"jmp"* and its operand is replaced by the value which has pushed.

3- Rename all the conditional jump instructions to the names in the Table 1.

4- Add to the end of the matrix a row contains *op[n+1][2]="end*"

5- Delete the rows that contain inaccessible label (this means that *op[i][3]* does not equal to this label for all *i*)

6- Delete the rows that contain unreachable operand (this means that *op[i][1]* does not equal to this operand for all *i*)

Algorithm 3: Prepare the matrices *Labels* and *JumpTo*

Input: *op* matrix of size *n×4*

Output: The matrix *Labels* of size *c×2* and the matrix *JumpTo* of size *e×3*

```
Do the following while count <= m
   If op[j][4]=1 then
      stack2.pop j
      if  j = -1 then stack1.pop j
      if  j= -1 then break
   else if op[j][2]="call" then
      stack1.push j+1; j=z+1 where
            op[z][1]= op[j][3]
   else if op[j][2]="ret" then
      stack1.pop  j
   else if op[j][2]= "jmp" then
      j=z+1 where op[z][1]= op[j][3]
   else if op[j][2]="A" ,"N", .. or "L"  then
     stack2.push z ,where op[z][1]= op[j][3]
     JumpTo [e][1]= op[j][3];
     JumpTo [e][2]= m;
     JumpTo [e][3]= op[j][2]
     m=m+1;e=e+1; j=j+1
   else if op[j][1] <> "null" then     //label
     Labels[c][1]= op[j][1];
     Labels[c][2]= m
     c=c+1;  m=m+1;  j=j+1
   else if op[j][2]="end"and m<=count then
     stack2.pop j
     if j = -1 then break
```

Algorithm 4:Construct the matrix ALCFG

Input: The matrix *Labels* of size c×2 and the matrix *JumpTo* of size e×3

Output:ALCFG represented as m×m matrix and nodes sequence NodeSeq contains m nodes

```
1- Fill the upper minor diagonal of matrix
   ALCFG by 1
2- Fill the array NodeSeq by "K"  // labels
3- for each row  i  in the matrix JumpTo
     x=JumpTo[i][2];
     NodeSeq[x]= JumpTo[i][3]
      for each row  j  in the matrix Labels
        if JumpTo[i][1]= Labels[j][1]  then
          y=  Labels[j][2] ;  ALCFG[x][y]=1
```

Table 1. the instructions and corresponding symbols

Instructions	Symbol
JE, JZ,	A
JP, JPE	R
JNE,JNZ	N
JNP, JPO	D
JA, JNBE, JG, JNLE	E
JAE,JNB,JNC, JGE, JNL	Q
JB, JNAE, JC, JL, JNGE	G
JBE, JNA, JLE,JNG	H
JO, JS	I
JNO, JNS, JCXZ, JECXZ	L
LOOP	P
LABEL	K
GAP	M

All above algorithms can be implemented very fast and can be optimized. The worst case of algorithm 2 is *5n* where *n* is the number of the lines in the disassembled file, the worst case of algorithm 3 is *n* and the worst case of algorithm 4 is $(m/2)^2$ where $m \leq n$. Therefore; the total complexity of algorithm 1 is $O(n)+O(m^2)$.

Definition 1: A skeleton signature of a binary file is the nodes sequence *NodeSeq* and the matrix *ALCFG*.

To illustrate the previous procedures; consider the input is the virus *Z0mbie III*, where Figure 1 is part from the source code of *Z0mbie III*, Figure 2 is the *op* matrix, figure 3 is the *Labels* matrix and figure 4 is *JumpTo* matrix of the first 20 nodes of the virus *Z0mbie III*.

```
start:
                ….
               pop    esi
               sub    esi, $-1-start
               push   esi
                        ….
               jne    tsr_complete
               shl    edi, 9
                        ….
               je     tsr_complete
tsr:
               int 3
               call   c000_rw
               pusha
               mov    ecx, virsize
              call    c000_ro
tsr_complete:
               out    80h, al
             ….
```

Figure 1: part from *Z0mbie III*

	N	tsr_complete	0
	A	tsr_complete	0
tsr			0
	call	c000_rw	0
	call	c000_ro	0
	H	__cycle_1	0
	E	__mz	0
tsr_complete			0
restore_program			0
	A	__exit	0
	A	restore_program	0
	N	__exit	0
cf8_io			0
-	-	-	-
-	-	-	-
-	-	-	-

Figure 2: the *op* matrix

1	tsr_complete	N
2	tsr_complete	A
3	__cycle_1	H
4	__mz	E
9	__exit	A
10	restore_program	A
11	__exit	N
12	__exit	Q
14	__exit	G
15	__mz	A
17	__exit	N
19	__exit	I
20	__cycle_2_next	G

Figure 3: The *Labels* Matrix

5	tsr
6	cf8_io
7	tsr_complete
8	restore_program
13	__cycle_1
16	__mz
18	__cycle_2

Figure 4: The *JumpTo* Matrix

The following is the skeleton signature of *Z0mbie III* which is consist from the sequence of the first 10 nodes *NodeSeq* and the matrix *ALCFG*:

N A H E K K K K A A

$$ALCFG_{10\times10} = \begin{bmatrix} 1 & & & 1 & & & & & & \\ & 1 & & 1 & & & & & & \\ & & 1 & & & & & & & \\ & & & 1 & & & & & & \\ & & & & 1 & & & & & \\ & & & & & 1 & & & & \\ & & & & & & 1 & & & \\ & & & & & & & 1 & & \\ & & & & & & & & 1 & \\ & & & & & & & 1 & & \end{bmatrix}$$

4. Similarity Measure Function

To detect the metamorphic viruses that preserve its control flow graph during the propagation, we can simply compare *ALCFG* matrices, but if the control flow graph is changed during the propagation then a similarity measure function must be used. Unfortunately the current similarity measurement functions such as Euclidean distance, Canberra distance or even measurements based on neural network can not be used; the reason is the random insertion and deletion in the nodes sequence of the generated control flow graph. In this section we propose a new similarity measure function to detect the metamorphic viruses. Consider the following definitions:

Definition 2: The diagonal sub-block of size $m\times m$ of the matrix *ALCFG* which has the size $n\times n$ is the matrix *A* and denoted by $A \in ALCFG$, where the first row and column start at $i+1<n$, the last row and column end at $i+m<=n$ and *i* is any integer number less than n.

Definition 3: Let $ALCFG_p$ denotes to *ALCFG* matrix of size $n\times n$ of the program *P* and $ALCFG_V$ denotes to *ALCFG* matrix of size $m\times m$ of the virus *V*.

Definition 4: The matrices $ALCFG_S$ and $ALCFG_V$ are similar if the following conditions are satisfied:
1- Alignment$(NodeSeq_S, NodeSeq_V) = c \geq T$
2- DelMis&Comp$(ALCFG_S, ALCFG_V)=1$

We will denote to the similarity measure function by *j* such that:

$$j(ALCFG_s, ALCFG_v) = \begin{cases} c & if \quad \text{satisfied} \\ 0 & else \end{cases}$$

Definition 5: The program *P* is infected by the virus *V* if and only if $j(ALCFG_s, ALCFG_v) = c$,where $ALCFG_s$ **p** $ALCFG_p$.

For simplicity we will focus on viruses that use simple entry point infection, therefore *i=0*. However our approach can be applied to any obfuscated entry point

Algorithm 5: Check whether the program *P* is infected by the virus *V* or not.

Input: The program *P*, the matrix $ALCFG_V$ and a threshold *T*, where V is a virus in the database

Output: yes if infected or no if the program is not infected

1- Disassemble the program P (In this study the software IDA Pro 4.8 is used, but this process can be implemented and embedded in one software)
2- Call Algorithm 1 to find $ALCFG_p$ and $NodeSeq_p$ (in this study the first sub block is processed which is equivalent to the simple entry point. However to check all the possible entry points we have to process all $m \times m$ sub block in the matrix $ALCFG_P$)
3- Call Algorithm 6 to find The Percentage *c* and the sequence *A*
4- If $c \geq T$ then
   Call algorithm 7 to Delete the mismatch nodes and compare the matrices
   If algorithm 7 retrun 1 then
      Return "Yes"
   Else
      Return "No"
   Else
      Return "No"

Algorithm 6: The Alignment of two sequences. *Alignment (,)*

Input: The sequences $NodeSeq_S$ and $NodeSeq_V$

Output: The Percentage *c* and the sequence *A*, where *c* represents the percentage of the match node to the total number of the nodes and *A* contains the index of the mismatched nodes

1- Apply Needleman-Wunsch-Sellers algorithm **on** the sequences $NodeSeq_S$ and $NodeSeq_V$
2- Store the index of mismatch nodes in the array A
3- Find c= number of matched nodes*100/ total number of nodes

Algorithm 7: Delete the mismatch nodes and compare. *DelMis&Comp(,)*

Input: $ALCFG_S$, $ALCFG_V$ and the mismatched sequence *A*.

Output: *0* or *1*

1- If mismatch with gab then delete the row *i* and the column *i* from the matrix $ALCFG_S$ for all *i* in the mismatched nodes, and delete the last rows and columns from $ALCFG_V$ where the number of the deleted rows and columns equal to the number of the gabs
2- If mismatch with symbol then delete the row *i* and the column *i* from the matrices $ALCFG_S$ and $ALCFG_V$ for all *i* in the mismatched nodes.
3- Rename the matrices to $ALCFG_s^{\,d}$ and $ALCFG_v^{\,d}$.
4- If $ALCFG_s^{\,d} = ALCFG_v^{\,d}$ then
   Return *1*
   Else
   Return *0*

The most expensive step in the previous algorithms is Needleman-Wunsch-Sellers algorithm which can be implemented in m^2 operation, and the total complexity of all procedures is $O(n)+O(m^2)$. Therefore the suggested method is much faster than the previous methods; for example the cost of finding the isomorphic sub graph in [9] is well known *NP*-complete problem.

To illustrate the suggested similarity measure function, assume that we like to the check weather the program *P* is infected by the virus *Z0mbie III* or not, assume that the threshold *T=70* and m=10

(note that: to reduce the false positive we must increase the threshold and the number of the processed nodes), the first 10 nodes that are extracted from *P* and the *ALCFG* matrix are (the skeleton signature of *P*):

N A H A E K K K K A

$$ALCFG_s = \begin{bmatrix} 1 & & & 1 & & & & & & \\ & 1 & & 1 & & & & & & \\ & & 1 & & & & & & & \\ & & & 1 & 1 & & & & & \\ & & & & 1 & & & & & \\ & & & & & 1 & & & & \\ & & & & & & 1 & & & \\ & & & & & & & 1 & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \end{bmatrix}$$

By using algorithm 6 the nodes of *P* aligned with the nodes of *Z0mbie III* as following:

N A H A E K K K K A -
N A H - E K K K K A A

c= number of matched nodes*100/ total number of nodes**=9*100/10=90 >*T*.**

The mismatch occur with gabs; therefore column 4 and row 4 must be deleted from $ALCFG_S$, column 10 and row 10 must be deleted from $ALCFG_V$. Since matrices after deletion are identical, we conclude that the program P is infected by a modified version of *Z0mbie III* and $j(ALCFG_s, ALCFG_v) = 90\%$.

5. Implementation

The metamorphic viruses are taken from VX Heavens search engine and generated by two tools; namely: Next Generation Virus Creation Kit (NGVCK0.30) and Virus Creation Lab for Windows 32 (VCL32) [11]. Since the output of the kits was already in the asm format, we used Turbo Assembler (TASM 5.0) for compiling and linking the files to generate exe's, which are later disassembled using IDA pro 4.9 Freeware Version. Algorithm 4 is implemented by using MATLAB 7.0. The NGVCK0.30 has advanced assembly source-morphing engine, and all variants of the viruses generated by NGVCK will have the same functionality, but they have different signatures. In this study; 100 metamorphic viruses are generated by using (NGVCK). 40 viruses are used for analyzing and 60 viruses are used for testing, let us call the first group A1 and the second group T1. After applying the suggested procedures on A1 we note that all the viruses in A1 have just seven different skeleton signatures when T=100 and m=20 and have four different skeletons when T=80 and m=20 and have three different skeletons when T=70 and m=20. T1 group is tested by using 7 antivirus software; the results are obtained by using the on-line service [12]. 100% of the generated viruses are recognized by the proposed method and by McAfee, but none of the viruses are detected by using the rest software. Another 100 viruses are generated by using VCL32, where all of them are obfuscated manually by inserting dead code, transposition the code, reassigning the registers and substituting the instructions. The generated viruses are divided into two groups, A2 and T2, A2 contains 40 viruses for analyzing and T2 contains 60 viruses for testing. Again 100% of the generated viruses are detected by the proposed method, 84% are detected by Norman, 23% are detected by McAfee and 0% are detected by the rest software. Figure 5 describes the average detection percentage of the metamorphic viruses in T1 and T2.

6. Conclusion

The antivirus software trying to detect the viruses by using variant static and dynamic methods. However; all the existing methods are not adequate. To develop new reliable antivirus software some problems must be fixed. This paper suggested new procedures to detect the metamorphic viruses by using arbitrary length of control flow graphs and nodes alignment. The suspected files are disassembled, the opcode encoded, the control flow analyzed, and the similarity of the matrices is measured by using a new similarity measurement. The implementation of the

suggested approach show that all the generated metamorphic viruses can be detected while less than 62% are detected by other well known antivirus software.

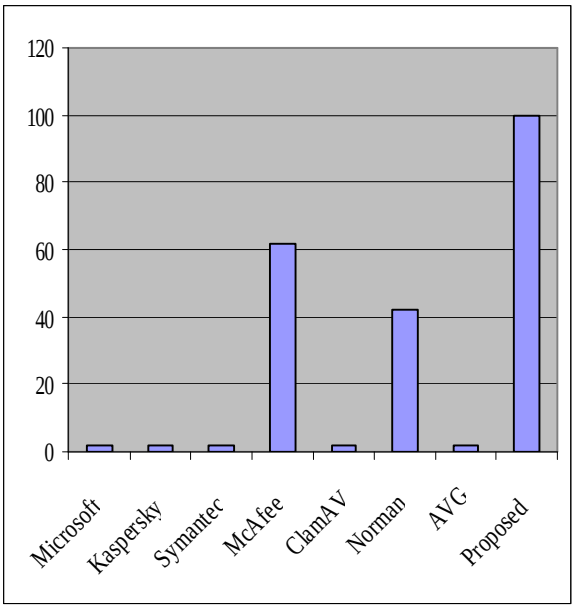

Figure 5. The average percentage of the detected viruses from group T1 and T2

References

[1] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware Normalization," Technical Report # 1539 at the Department of Computer Sciences, University of Wisconsin, Madison, 2005.

[2] F. Perriot, "Striking Similarities: Win32/Simile and Metamorphic Virus Code", Symantec Corporation 2003.

[3] E. Konstantinou, "Metamorphic Virus: Analysis and Detection Technical Report," RHUL-MA-2008-02 Department of Mathematics Royal Holloway, University of London, 2008.

[4] A. Lakhotia, A. Kapoor, and E. U. Kumar, "Are metamorphic computer viruses really invisible?," part 1. Virus Bulletin, 2004, pp 5-7.

[5] P. Szor, *The Art of Computer Virus Research and Defense.* Addison Wesley Professional, 1 edition, February 2005.

[6] R. Ando, N. A. Quynh, and Y. Takefuji, "Resolution based metamorphic computer virus detection using redundancy control strategy," In WSEAS Conference, Tenerife, Canary Islands, Spain, Dec. 2005, pp16-18.

[7] R. G. Finones and R. T. Fernande, "Solving the metamorphic puzzle". Virus Bulletin, March 2006, pp 14-19.

[8] M. R. Chouchane and A. Lakhotia, "Using engine signature to detect metamorphic malware," In WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode, New York, NY, USA, 2006, pp 73-78.

[9] D. Bruschi, L. Martignoni, and M.Monga, "Detecting self-mutating malware using control flow graph matching," In DIMVA, 2006, pp 129-143.

[10] W. Wong and M. Stamp, "Hunting for metamorphic engines," Journal in Computer Virology, 2(3), 2006, pp 211-229.

[11] http://vx.netlux.org/ last access March 2009.

[12] http://www.virustotal.com/ last access March 2009.