A program behavior recognition algorithm based on assembly instruction sequence similarity

Baojiang Cui[1], Chong Wang[2] , GuoWei Dong[3], JinXin Ma[4]

[1] School of Computer Science, Beijing University of Posts and Telecommunications, National Engineering Laboratory for Mobile Network Security, China
cui_bjiang@163.com

[2] School of Computer Science, Beijing University of Posts and Telecommunications, National Engineering Laboratory for Mobile Network Security, China
wangchong756@126.com

[3] China Information Technology Security Evaluation Center, Beijing 100085,China
dgw2008@163.com

[4] China Information Technology Security Evaluation Center, Beijing 100085,China
majinxin2003@126.com

Abstract. The analysis on assembly instruction sequence plays a vital role in the field of measuring software similarity, malware recognition and software analysis, etc. This paper summarizes the features of assembly instructions, builds a six-group model and puts forward an algorithm of calculating similarity of assembly instructions. On that base a set of methods of calculating similarity of assembly instruction sequence are summarized. The preliminary experimental results show that it has high efficiency and good effect.

1 Introduction

In recent years, malicious software has a increasing spread, a dizzying variety and fast pace of change, which produce a great deal of trouble and loss for users. A report[1] in Business Software Alliance recently shows that 430 million new pieces of malware were discovered in 2015,up 36 percent from 2014 and organizations experiences some form of malware attack every seven minutes. Thus it can be seen, how to identify the behavior of the program has become an important research field of information security and plays a significant role in the program similarity and malware analysis field.

The traditional program identification technology can be divided into static recognition[2,3] and dynamic recognition[4,5]. Static recognition means using a disassembler to turn the executable program code into assembly language and discover certain acts by matching the sequence of bytes and extracting signatures and constant feature of the program algorithm. Static recognition has an

© Springer International Publishing AG 2017

L. Barolli et al. (eds.), *Advances on Broad-Band Wireless Computing, Communication and Applications*, Lecture Notes on Data Engineering and Communications Technologies 2, DOI 10.1007/978-3-319-49106-6_2

advantage of low overheads, but it can t recognize the use of multi-state, deformation, encryption, confusion and other means of malicious programs.

Dynamic recognition is in an isolated emulation environment to run suspicious files, scanning system calls and analyzing instruction steam and data available. Thereby it can substantially eliminate the effects of Packers and code obfuscation. The drawback is the high overhead, low efficiency and low accuracy.

This paper presents a program behavior recognition algorithm based on assembly instruction sequence similarity. In this work we record the assembly instruction steam, extract and analysis the information of instructions and generalise the model of assembly instruction sextuple. Based on this model, we design a matching algorithm for the assembly instruction steam sequence. Ultimately, we complete the induction which is from the assembly instructions to the abstract logic behavior of program from bottom to top, using dynamic program behavior feature extraction technology, to achieve the purpose of identifying the unknown program.

2 Structure of assembly instruction

Typically, the general format of assembly instructions are:

[label field:] opcode field [source operand, destination operand] [; comment]

Among them, the square brackets [] means that contents are optional, depending on the circumstances.

(1) Label field: It is located at the beginning of the statement and represents the address of this statement. Numeral itself consists of one to eight letters and numbers, representing the command position in memory.
(2) Opcode field: It represents the function of the instruction and indicates that the operation to be executed by a computer.
(3) Operand: Operand is the assembly instructions operating data or address that data is located. This part can be divided into operand itself, the address of operand or the information related to the operands.
(4) Comments: Comments begin with semicolon (;). It illustrates the program features to enhance readability.

3 Abstract coding of assembly instructions

Compared to static assembly instructions, the system function call sequences and the like, dynamic assembly instructions flow covers the entire information of the program behavior during the execution. It is an ideal object of analysis which we research to identify the program behavior. Through the above analysis shows that the structure of the assembly instructions, simple assembly instructions structures do not have abstraction. The semantic information that assembly instructions implied can not be identified and calculated by programs. Therefore, in order to extract the assembly instructions operational semantics, we design a abstract code to describe assembly instructions and name the model of assembly instruction sextuple In.

$$\mathrm{In} = < \mathrm{op}_{\mathrm{code}}, \mathrm{op}_{\mathrm{num}}, \mathrm{psw}, \mathrm{op}_{\mathrm{wr}}, \mathrm{deep}, \mathrm{time} > \quad (3\text{-}1)$$

This model fully considers the abstract feature information between different instructions. After extracting the semantic component feature of each assembly instruction, we quantify each element in the model of assembly instruction sextuple, using quantitative value to express the information in each part.

1) operation code

$\mathrm{op}_{\mathrm{code}}$ includes the operation code itself and their type. First, the operation code can be divided into six kinds according to their semantics, including data transfer instructions(MOV, LEA), arithmetic instructions(ADD, SUB), bit operation instructions(AND, OR), string operation instructions (STOS, CMPS), jump control instructions(JA, CALL), advanced control instructions(CLC, BOUND). Each category can be divided into subcategories in accordance with the address of the operand and the specific operating behaviors in the table below.

Category	Subcategory
Data transfer	General data transfer operation、Stack data transfer operation、Data Exchange operation
Arithmetic	Add and sub、Multiplication and division、Extended operations
Bit operation	General bit operation、Bit test operation、Bit scanning operation、Shift operation
String operation	String transfer operation、String store operation、String compare operation、
Jump control	Function call、Unconditional jump、Conditional jump、Loop control

Table 1 Classification of operation code

In quantifying the operation code, if two operation codes are totally same, it will return 1 in operation coding function. If two operation codes belong in the same category, which means there are similar places in meaning , it will return 0.5 in operation coding function. If they are not in the same category, it will return 0.1.

2) operand coding

$\mathrm{op}_{\mathrm{num}}$ contains the number of operands of the instruction and three characteristics of the operand:

- Type：Operand is a field of assembly instructions. There are three types of operand can be put in this field, which are immediate, memory and register. In addition, memory space is divided into stack and non-stack space.
- Importance: If memory or register as an independent operand, will be marked as "important"; on the contrary, if it is as an integral part of the operand address, were marked as "not important." This similarity will be quantized in subsequent alignments.
- Tags: Tags here represent some information related to operands, recording associated memory or register. For example, EAX is a 32-bit general-purpose registers, EAX will be marked as 32, and joined the AX, AH, AL in the label. In quantifying operands, first we check each read-write mode of operands are same or not. Important will be marked if two operands are totally same and the returned value will be 0.2. If they are not same, the returned value will be 0.1.Next we check their storage type, If they are same, the result will increase by 0.1, otherwise it does not affect the result.

3) Flag register coding

Flag registers are called the program status word. We can know your current state of the CPU by them. For example, OF represents the overflow flag, it is used to check the result from addition or

substraction operation is overflowed or not. psw is a nine-tuple structure which describes the flag register set. It is written $< cf, pf, af, zf, sf, tf, if, df, of >$. It corresponds to nine flag registers, describing the impact operation code makes. If the operation code affected one register, the returned value increases by 1/9.

4) Read and write operations coding

$\mathrm{op_{wr}}$ is a triad $< w, r, t >$ which describes read-write mode in the assembly instruction, representing the operand of read , write and take . For example, the assembly instruction MOV EBX，DWORD PTR [EAX+0X4]. There are four operands in it. The read-write mode of the first one EAX is t. The read-write mode of the second one 0x4 is r. Then two operands comprise the third operand DWORD PTR [EAX+0X4], and its read-write mode is r. The read-write mode of the last one EBX is w.

5) Depth of nesting function coding

deep represents the function which the assembly instruction belongs to is invoked in deep layer. Even for the same instructions, the meanings of different nested depths are very different. Such as PUSH SP, the value in SP in functions of different nested depths differs greatly.

6) Timestamp coding

time indicates the exact point of execution. When $\mathrm{time} = 10$, it means that it is the 10th assembly instruction when the program executes.

4 Similarity measurement algorithm of assembly instructions

After quantifying each element in the sextuple model, we can deduce the calculating formula that describes the smilarity between two assembly instructions.

$$\mathrm{SimIn} = \mathrm{SimC} * (\mu_b * \mathrm{SimB} + \mu_s * \mathrm{SimS}) \quad (4\text{-}1)$$

Among them, SimC represents the semantic similarity of instruction, referring to the similarity of $< \mathrm{op_{code}}, \mathrm{op_{num}} >$. SimB represents the behavior similarity of instruction, referring to the similarity of $< psw, \mathrm{op_{wr}} >$. SimS represents the structural similarity of instruction, referring to the similarity of $< deep, time >$. μ_b、μ_s represents the weights to SimB、SimS. Since $< \mathrm{op_{code}}, \mathrm{op_{num}} >$ plays an important role in the sextuple, SimC itself is a multiplication factor. These variables need to fulfill the below conditions:

$$\mu_b + \mu_s = 1，\ 0 \le \mathrm{SimC} \le 1，\ 0 \le \mathrm{SimB} \le 1，\ 0 \le \mathrm{SimS} \le 1 \quad (4\text{-}2)$$

The computational method above, we take into full account the semantic information, the behavior information and the structural information. It can be more accurate and comprehensive to display the similar situation between two instructions.

5 Similarity measurement algorithm of assembly instruction sequence

After having calculated the similarity between two assembly instructions , we start to calculate the similarity betweeen two assembly instruction sequences. Here we adopt the idea of the longest common sequence(LCS) problem which is a dynamic programming.

$\mathrm{InS}(1, \mathrm{n})$ represents a assembly instruction sequence that consists of n assembly instructions, and it is numbered from 1. So the similarity between two assembly instruction sequences $\mathrm{SimInS}(\mathrm{n_1}, \mathrm{n_2})$ is defined below:

$$\mathrm{SimInS}(\mathrm{i}, 0) = 0 \ \ (1 \le \mathrm{i} \le \mathrm{n_1}) \quad (5\text{-}1)$$

$$\mathrm{SimInS}(0,\mathrm{j}) = 0 \ \ (1 \le \mathrm{j} \le \mathrm{n}_2) \tag{5-2}$$

$$\mathrm{SimInS}(\mathrm{i},\mathrm{j}) = \max\begin{cases} \mathrm{SimInS}(\mathrm{i}-1,\mathrm{j}) \\ \mathrm{SimInS}(\mathrm{i},\mathrm{j}-1) \\ \mathrm{SimInS}(\mathrm{i}-1,\mathrm{j}-1)+\mathrm{SimIn}(\mathrm{i},\mathrm{j}) \end{cases} \quad (1 \le \mathrm{i} \le \mathrm{n}_1, 1 \le \mathrm{j} \le \mathrm{n}_2) \tag{5-3}$$

Pseudo-code as follows:

```
Input:  Assembly instruction sequence  InS1;
        Assembly instruction sequence  InS2;
Output: The similarity of this two sequences;
function Sim(InS1,InS2)
  for i = 1 to end do
    for j = 1 to end do
      t=max(SimInS[i-1][j],SimInS[i][j-1])
      SimInS[i][j]=max(t,SimIns[i-1][j-1]+SimIn[i][j])
    end for
  end for
 return SimIns[n1][n2]
 end function
```

$\mathrm{SimInS}(\mathrm{i},\mathrm{j})$ represents the global similarity of two assembly instruction sequence and it has not been normalized. To normalize this result, we define $\mathrm{SimInS}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2})$ as the partial similarity of two assembly instruction sequence $\mathrm{InS}_{1(\mathrm{i}_1,\mathrm{i}_2)} =< I\mathrm{n}_{\mathrm{i1}}, \mathrm{In}_{\mathrm{i1+1}}, \mathrm{In}_{\mathrm{i1+2}} \ldots \mathrm{In}_{\mathrm{i2}} >$, $\mathrm{InS}_{2(\mathrm{j}_1,\mathrm{j}_2)} =< I\mathrm{n}_{\mathrm{j1}}, \mathrm{In}_{\mathrm{j1+1}}, \mathrm{In}_{\mathrm{j1+2}} \ldots \mathrm{In}_{\mathrm{j2}} >$.We get the following equation:

$$\mathrm{SimInS}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2}) = \mathrm{SimInS}(\mathrm{i2},\mathrm{j2}) - \mathrm{SimInS}(\mathrm{i1}-1,\mathrm{j1}-1) \tag{5-4}$$

So the normalized similarity of two assembly instruction sequence are as follows:

$$\mathrm{len}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2}) = \max(\mathrm{i2}-\mathrm{i1}+1, \mathrm{j2}-\mathrm{j1}+1) \tag{5-5}$$

$$\mathrm{NorSimIns}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2}) = \frac{\mathrm{SimInS}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2})}{\mathrm{len}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2})} \tag{5-6}$$

$\mathrm{len}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2})$ indicates the length of the instruction sequence interval. The assembler instruction sequence similarity normalized result $\mathrm{NorSimIns}(\mathrm{i1},\mathrm{i2},\mathrm{j1},\mathrm{j2})$ can be get by the partial similarity divide the interval length.

According to the above transfer equation,we can draw a sketch map describing the algorithm.

Each grid represents the similarity between two assembly instructions. The arrows represent the value of the source in the position. Choosing dark boxes indicate the path traversed when calculating ParSimInS (i1, i2, j1, j3).

	0	$\mathrm{Sim}_{\mathrm{i1}}$	$\mathrm{Sim}_{\mathrm{i2}}$	Sim_{13}
0	0	0	0	0
$\mathrm{Sim}_{\mathrm{j1}}$	0	↘ 0.5	↘ 0.6	→ 0.6
$\mathrm{Sim}_{\mathrm{j2}}$	0	↓ 0.5	↘ 1.2	→ 1.2
$\mathrm{Sim}_{\mathrm{j3}}$	0	↘ 0.8	↓ 1. 2	↘ 1.9

Figure 5-1 The algorithm sketch map

So we can get the result from the figure above:

$$\mathrm{ParSimInS(i1,i2,j1,j3)} = 1.2,\ \ \mathrm{NorSimIns(i1,i2,j1,j3)} = \frac{\mathrm{ParSimInS(i1,i2,j1,j3)}}{\mathrm{len(i1,i2,j1,j3)}} = \frac{1.2}{3} = 0.4$$

6 Optimum of algorithm

6.1 Reduce of time complexity

The current average time complexity of the algorithm is $\mathrm{O(n^2)}$. n is the number of instructions in the program. In practical applications, a program contains a large number of assembly instructions, so now the time complexity of the algorithm is high, efficiency is low. So we find a Instruction - Basic Block mode to reduce the time complexity of the algorithm.

The basic block is a sequence of instructions executed in the program. During the running time, it maybe execute the same basic block. First we calculate the similarity between assembly instructions for the basic element in the static state. Then we calculate the similarity between basic blocks for the basic element, while establishing the similarity between the two-dimensional table of basic blocks. We treat $< index(\mathrm{BBL_1}), \mathrm{index(BBL_2)} >$ as key value, thus the number between two basic blocks is only once. When we find these two basic blocks has been calculated, the query time is only O(1), eliminating many repetitive calculations. Time complexity has been reduced. In summary, the similarity of two basic blocks $\mathrm{BBLS_1(1,n)}$ and $\mathrm{BBLS_2(1,n)}$ may be expressed as:

$$\mathrm{SimBBL(BBL_1, BBL_2)} = \mathrm{SimInS(i1,i2,j1,j2)} \tag{6-1}$$

$\mathrm{BBLS(1,n)}$ represents that there are n basic block sequence $< BB\mathrm{L_1}, \mathrm{BBL_2}, \mathrm{BBL_3} \ldots \mathrm{BBL_n} >$, we can get the equation from above.

$$\mathrm{SimBBLS(i,0)} = 0 \ \ (1 \le \mathrm{i} \le \mathrm{n_1}) \tag{6-2}$$

$$\mathrm{SimBBLS(0,j)} = 0 \ \ (1 \le \mathrm{j} \le \mathrm{n_2}) \tag{6-3}$$

$$\mathrm{SimBBLS(i,j)} = \max\begin{cases} \mathrm{SimBBLS(i-1,j)} \\ \mathrm{SimBBLS(i,j-1)} \\ \mathrm{SimBBLS(i-1,j-1)} + \mathrm{SimBBL(i,j)} \end{cases} \quad (1 \le \mathrm{i} \le \mathrm{n_1}, 1 \le \mathrm{j} \le \mathrm{n_2}) \tag{6-4}$$

The partial similarity $\mathrm{SimBBLS(i1,i2,j1,j2)}$ between two basic block sequence can be expressed as:

$$\mathrm{SimBBLS(i1,i2,j1,j2)} = \mathrm{SimBBLS(i2,j2)} - \mathrm{SimBBLS(i1-1,j1-1)} \tag{6-5}$$

The normalized representation is:

$$\mathrm{len(i1,i2,j1,j2)} = \max(\mathrm{i2-i1+1, j2-j1+1}) \tag{6-6}$$

$$\mathrm{NorSimBBLs(i1,i2,j1,j2)} = \frac{\mathrm{SimBBLS(i1,i2,j1,j2)}}{\mathrm{len(i1,i2,j1,j2)}} \tag{6-7}$$

6.2 Reduce of space complexity

The current space complexity of the algorithm is $\mathrm{O(n^2)}$. n is the number of instructions in the program. The memory which the program needs is relatively large. As can be seen in Figure 5-1, During the calculation, when we use the dynamic programming algorithm to calculate $\mathrm{SimInS(i,j)}$, the information we need is only the last layer and this layer. That means the information before has been

useless, so a two-dimensional array is sufficient to achieve the purpose of the calculation of each state. In this case,we reduce the space complexity and improve efficiency in the use of memory.

To sum up, after optimizing the time complexity and space complexity, the program can calculate the degree of similarity between the basic blocks, and thus from the bottom to up, we can also calculate the similarity between functions. Ultimately the similarity between any two applications can be calculated, so we achieve the purpose of identifying the behavior of the program.

7 Experimental results and analysis

To verify the accuracy, time complex and space complex of this algorithm, we adopt the programs form the program online judge system of Beijing University of Posts and Telecommunications. Users can submit a variety of source code on the same topic on it. We select 2000 samples of 20 kinds.

First we gather and classify all the samples. For each type of the programs, we select the most representative program. We don t describe the clustering algorithm detailedly here. Then we discriminate and classify many unknown programs. The result is correct when the original source code belongs to the same category, otherwise the result is wrong.

Final results are as follows: (Ins represents the average number of assembly instructions in all programs in that category, BBL represents the average number of basic blocks in all programs of this category, ETBO represents the elapsed time before optimization, ETAO represents the elapsed time after optimization, MBO represents the memory before optimization, MAO represents the memory after optimization).

Program behavior	Accuracy	Ins	BBL	ETBO	ETAO	MBO	MAO
Program behavior	100%	1567	305	10.000s	0.078s	1896kb	1888kb
Find the max	96%	108	20	0.093s	0.016s	1892kb	1888kb
Encryption	99%	555	152	1.077s	0.062s	1896kb	1888kb
Simple selection	98%	26	11	0.015s	0.001s	1888kb	1888kb
Sort the structure	94%	1953	288	14.118s	0.639s	1896kb	1888kb
Reverse a string	93%	647	89	2.792s	0.296s	1896kb	1888kb
Length of string	99%	244	64	0.390s	0.031s	1892kb	1888kb
Matrix operations	96%	858	177	4.275s	0.078s	1896kb	1888kb
Average	98%	744	138	4.095s	0.150s	1894kb	1888kb

Table 2 Experimental results

Experimental results show that the average accuracy rate is 98%.When we use the Instruction - Basic Block mode and scroll array, the time complexity and space complexity is reduced by several tens of times.

8 Conclusion

This paper puts forward a program behavior recognition algorithm based on the similarity of assembly instructions. We treat the assembly instructions as our basic size and generalise the model of assembly instruction sextuple. Based on this model, we design a matching algorithm for the assembly

instruction sequence to detect the program behavior. Meanwhile, the algorithms are optimized for time complexity and space complexity, the experimental results show that this method can detect the basic behavior of the program, with the advantage of high accuracy. Next we hope to detect instruction semantics for further, constantly optimize the time complexity and maximize efficiency.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (No. 61272493，61502536，U1536122).

References

1.http://www.bsa.org/~/media/Files/StudiesDownload/BSA_GSS_A4.pdf

2. Gröbert F, Willems C, Holz T. Automated Identification of Cryptographic Primitives in Binary Programs[J]. Lecture Notes in Computer Science, 2011:41-60.

3.Jingwei Zhang. Research on Public Key Cryptographic Algorithm Recognition Technology [D]. The PLA Information Engineering University, 2011.

4. LI Xiang, KANG Fei, SHU Hui. Cryptographic Algorithm Recognition Based on Dynamic Binary Analysis. Computer Engineering, 2012, 38(17): 106-109,115.

5.Caballero J, Yin H, Liang Z, et al. Polyglot: automatic extraction of protocol message format using dynamic binary analysis[J]. Ccs 07 Proceedings of Acm Conference on Computer & Communications Security Acm, 2007:317--329.