Research Journal of Applied Sciences, Engineering and Technology 5(6): 2053-2058, 2013
DOI:10.19026/rjaset.5.4749
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: July 27, 2012 Accepted: September 17, 2012 Published: February 21, 2013

Research Article

A Novel Disassemble Algorithm Designed for Malicious File

Jianmin Pang, Yichi Zhang, Chao Dai, Di Sun and Qiang Wang
National Digital Switching System Engineering and Technology Research Center,
Zhengzhou, 450002, China

Abstract: In order to avoid being static analyzed, hacker rely on various obfuscation techniques to hide its malicious characters. These techniques are very effective against common disassembles, preventing binary file from being disassembled correctly. The study presents novel disassemble algorithm which based on analyzed Control Flow Graph (CFG) and Data Flow Graph (DFG) information improve the ability of the disassembly. The proposed algorithm was verified on varied binary files. The experiment shows that the method not only improves the accuracy of disassemble but also greatly deal with malicious files.

Keywords: Control flow graph, disassemble, obfuscation, reverse slice

INTRODUCTION

Modern reverse engineering techniques automatically recognize library functions, local variables, stack arguments, data types, branches, loops, etc. This technology has been widely used in malicious detecting and vulnerability scanning. For virus analysis, we disassemble binary executable file to obtain its machine instructions. Unfortunately, hackers use many techniques to disguise malware against reverse analysis (Mila, 2004; Chao *et al.*, 2008).

The term obfuscation refers to techniques that preserve the program's semantics functionality while, at the same time, making it more difficult for the analyst to extract the program's function (Abhishek, 2009). These difficulties caused by follow reasons:

- Modern computer based on Von Neumann architecture. In this architecture don't distinguish between data and instructions. With the development of compile technology, data can be defined at any time (Arun *et al.*, 2005). So code sections commonly contain data such as jump tables or string constants.
- The Intel x86 instructions set architecture contains variable length instructions that can start at arbitrary memory address (Intel-64 and IA-32, 2009). In this case, disassemble indentify the address of the next instruction depend on disassembled instructions. When the disassemble is initially off by a byte, the following instructions are erroneous.
- To determine the address of branches often according to experience and assumptions (such as: the next instruction of Call was return location). These assumptions for normal software are right, but don't suit malware (Konstantin, 2005).

In this study, we present a novel disassemble algorithm which based on analyzed Control Flow Graph (CFG) and Data Flow Graph (DFG) information improve the ability of the disassembly. The proposed algorithm was verified on varied binary files. The experiment shows that the method not only improves the accuracy of disassemble but also greatly deal with malicious files.

LITERATURE REVIEW

Common disassemblers translate binary code into correspond machine instruction using lookup tables. Generally, we can divide the disassembly algorithms into two categories: the linear sweep algorithm and the recursive traversal algorithm.

The linear sweep algorithm (Cullen and Saumya, 2003) disassembles from the start of a program's entry point and then sweeps on the whole code section and disassemble each instruction encountered in the process.

The main weakness of the linear sweep algorithm is that it is very difficult to distinguish between code and data under the Von Neumann architecture, because data can be embedded in code section which may lead to the misinterpretation of data into code. Due to overlapping instructions, misalignment can lead to an alternate sequence of instructions that does not reflect the instructions that are actually executed at runtime. Alternate instruction streams that are a consequence of misalignment have a tendency to realign with the correct stream after few instructions (Nathan *et al.*, 2008);

Corresponding Author: Jianmin Pang, National Digital Switching System Engineering and Technology Research Center, Zhengzhou 450002, China

together with the fact that the x86 instruction set is so densely coded that most byte sequences constitute valid code, this can make disassembly errors introduced by misalignment hard to spot. At present, disassembles which use the linear sweep algorithm comprise OBJ Dump and Windbg, etc.

Compared to the linear sweep algorithm, recursive traversal algorithm (Cullen and Saumya, 2003) decides the next instruction to disassemble via the control flow of the program. When encountered the control transfer instruction (such as jmp, call, ret, etc.), it will not continue to disassemble sequentially, but starts to disassemble from the target address of the control transfer instruction. As for the non control transfer instruction, this algorithm would deal with them just like linear sweep. This allows the disassemble to skip over data bytes mixed into code sections. On the downside, this strategy is not guaranteed to process all bytes in the executable, since not all code locations are accessed through direct branches from the entry point.

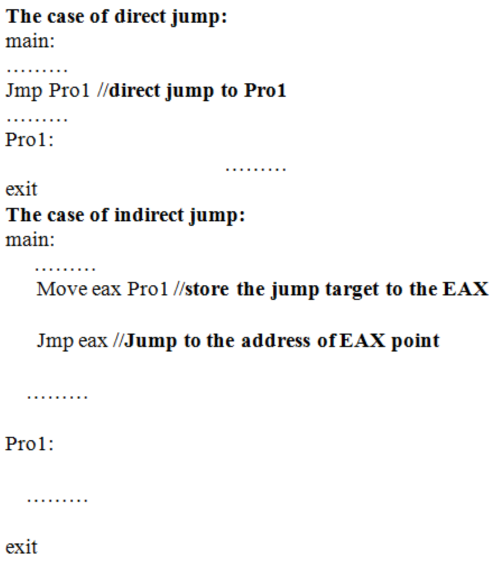
The obfuscation with indirect jumps just makes use of the disposal of control transfer instruction to achieve the goal of confusing the disassemble. It is used by IDA Pro. It can handle obfuscations in certain cases by getting across junk bytes partly (such as junk bytes after jump instruction) while linear disassemblers can not deal with this. Unfortunately, neither of them can identify the indirect jump target address. That’s why a novel algorithm should be introduced.

Indirect jump obfuscation: One of the main problems when analyzing assembly language is indirect branch instructions. These instructions correspond to jump statements, but jump target is calculated at runtime. In binary, any address in the code section is a potential target of an indirect branch. There are no explicit symbols in code section. If failure to statically resolve the target of an indirect branch instruction, it will leads to an incomplete control flow graph. So we couldn’t obtain completely machine code.

Through above analysis can be see that hacker adopt the method which changing the direct jump to the indirect jump. Figure 1 shows the detail of the method.

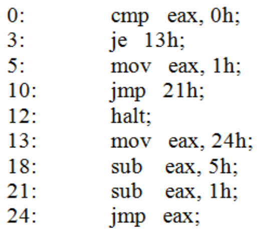
In Fig. 1, we can find that the specific address (Pro1) of the direct jump (Jmp Pro1) can be calculated by disassemble. At the same time, we can according to the target address for append edge which is from Jump instruction (Jmp Pro1) to target addressing (Pro1).

For the indirect jump case, disassemble analyze the indirect jump instruction found the jump target stored on the register (eax). Due to the value of register is dynamic generated; we couldn’t get the specific address. If we adopted recursive traversal algorithm, we couldn’t disassemble the target of the jump. On the other side, we adopted linear sweep algorithm without considering the CFG. Once the data stored in the code section, the data will trade as code disassemble, which will lead to disassemble derail.



```
The case of direct jump:
main:
………
Jmp Pro1 //direct jump to Pro1
………
Pro1:
                         ………
exit
The case of indirect jump:
main:
   ………
   Move eax Pro1 //store the jump target to the EAX

   Jmp eax //Jump to the address of EAX point

  ………

Pro1:

  ………

exit
```

Fig. 1: Changing direct jump to indirect jump



```
0:      cmp   eax, 0h;
3:      je  13h;
5:      mov   eax, 1h;
10:     jmp  21h;
12:     halt;
13:     mov   eax, 24h;
18:     sub   eax, 5h;
21:     sub   eax, 1h;
24:     jmp   eax;
```

Fig. 2: The segment of indirect jump codes

The obfuscation is so simple, but the efficacy is terrific to disassemble. Although, some disassemble adding constant propagation function against this obfuscation, which is no use for data hidden assignment.

DISASSEMBLE FRAMEWORK

To deal with indirect jump problem, we mainly rely on the partial CFG and DFG. By analyze those graph, we can extract the target of the jump instruction. Specific say, indentify the target of indirect jump is divided three steps: firstly, ascertain the variables that related the jump target. In one process, reverse slice based on the built CFG to extract related variables. Secondly, depending on the reverse slice result calculate the address of jump target in one branch. Lastly, append the edge of the indirect jump target. Then calculate the impact on the original DFG. If the DFG changes, then recalculate DFG till appear fixed point.

Before propose the method that extracts the indirect jump target, we introduce the typical example of the indirect jump, as shown in the Fig. 2.

From the Fig. 2 we can see an indirect jump instruction (jmp eax) in this code fragment. Disassembling this code fragment, we can get the indirect jump instruction. But we couldn’t get the address of the indirect jump. After that, control flow interruption, so that disassemble couldn't continue to work.

Reverse slice: When the indirect jump instruction was found, we denote the place of the instruction and put the instruction in the stack (TI). After the program CFG built, we take out an indirect Jump Instruction (JI) from the TI and recode the address of JI. Pick up all instruction in the basic Block (BJI) which is included JI. Calculate the variables related to the jump target of JI and then put those variables into the Set (SV). Reverse filtrate the SIB according to the SV, which store in RI appending to SR. Put the variables related by RI to the SV as reverse slice standard.

After deal with a basic block, looking up its father node (SFB) stored in the father stack (SSB). IF SSB is not empty, pop the basic block of SSB as current B reverse slice. Otherwise, declare all basic block in the process have analyzed completely. At the end of the algorithm, we can get the instruction which related by indirect jump target.

On the basis of the relevant references, the algorithm which reverse slice get the indirect jump target relay on the CFG in a proc. The input of the algorithm is the address of indirect jump target, we can obtain the basic block and proc related JI. The output is the reverse Slice Result (SR), which store the data flow of indirect jump target.

Target address calculate: Through reverse slice process, we can get the instructions related to the target address. The target address calculates stage is based on the reverse slice results. In this calculate stage, we need recode every changed variables for calculate jump target. In addition, those reverse slice instructions belong to multi-branch, which will eventually influence the target result. So calculate the target have two cases: in the first case, every branch keeps other branches calculate result alone, process simply and calculate the target by the branch self; in second case, when fall across the value of a variable depend on another branch, we should suspend until related branch analysis result come back.

According above idea, Fig. 3 shown the flow of the algorithm of calculates the indirect jump address. The

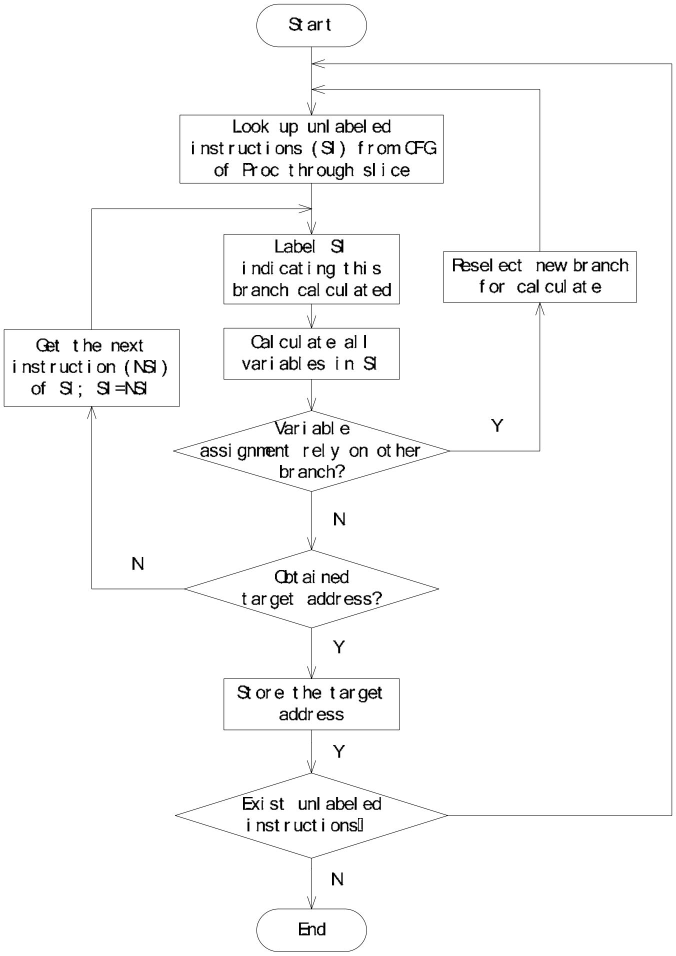

Start
Look up unlabeled instructions (SI) from CFG of Proc through slice
Label SI indicating this branch calculated
Reselect new branch for calculate
Get the next instruction (NSI) of SI; SI=NSI
Calculate all variables in SI
Variable assignment rely on other branch?
Y
N
N
Obtained target address?
Y
Store the target address
Y
Exist unlabeled instructions?
N
End

Fig. 3: The algorithm of calculate the indirect jump address

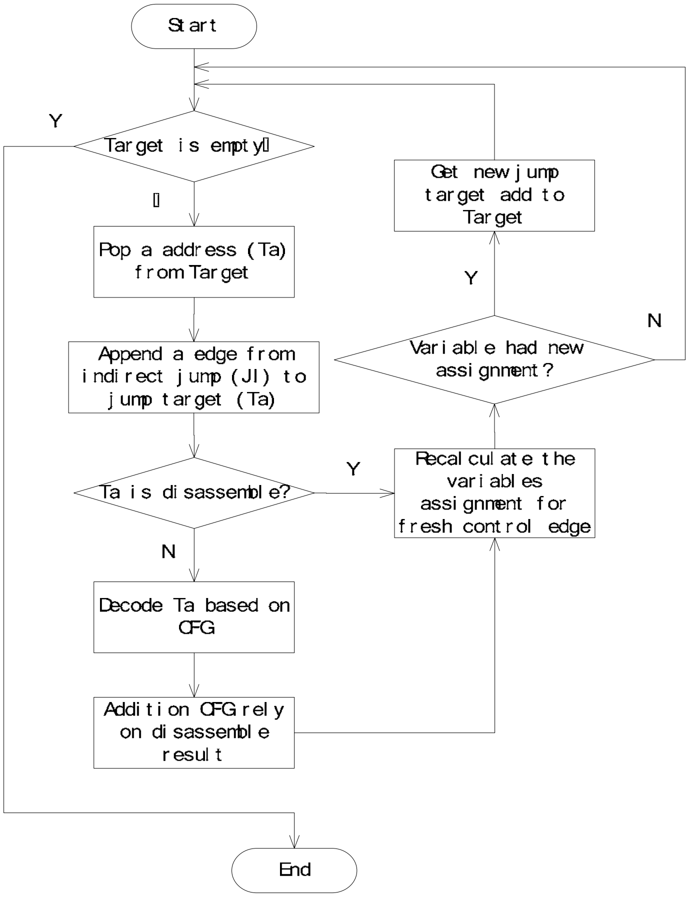

Start
Y
Target is empty□
□
Pop a address (Ta) from Target
Append a edge from indirect jump (JI) to jump target (Ta)
Ta is disassemble?
N
Decode Ta based on CFG
Addition CFG rely on disassemble result
End
Y
Recalculate the variables assignment for fresh control edge
Variable had new assignment?
Y
Get new jump target add to Target
N

Fig. 4: The algorithm of append control flow of jump target

input of the algorithm is the instructions of reverse Slice Result (SR); analyze the CFG and DFG to get the possible indirect jump address (Target).

For the above case, calculate the indirect jump target. In the process have two branch and assignment to EAX in each branch. So we need calculate two branches which assignment to EAX.

Append control flow: Through indirect jump calculating, we can identify the possible jump target. After that, append the fresh ascertain target to the CFG. The newly edge of DFG will indicate disassemble to decode uncover code section. After that, reanalyze the DFG of process, judge whether the newly appending edge affect the DFG. So we will repeat step two recalculate the assignment of variable. If have new assignment, we need append new target address and related CFG until doesn't have new DFG (fix point appeared). Figure 4 shown the algorithm of appends control flow.

At the same time, analyze original DFG look up fresh target address. The input of this algorithm is the set (Target) of indirect jump address, which re-decode the jump target rely on the new appended CFG and re-analyze the DFG. The output of the algorithm is the completely CFG of program and its machine codes.

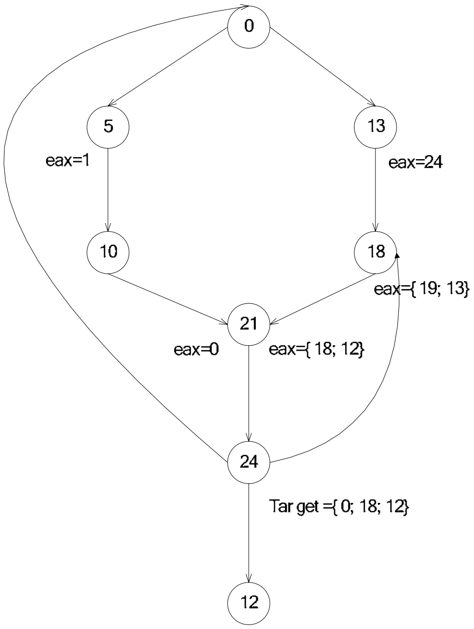


Fig. 5: The outcome of appended control flow

We can get the indirect jump target and add two edges to CFG. The two edges both start from the

Table 1: Sample for experiment

ID		Name	OBJ dump	ID Apro	Radux
Normal file	1	At.exe	1.000	1.000	1.000
	2	Comrereg.exe	1.000	1.000	1.000
	3	Csrss.exe	0.990	1.000	1.000
	4	Calc.exe	1.000	1.000	1.000
	5	Dcomcnfg.exe	1.000	1.000	1.000
	6	Write.exe	1.000	1.000	1.000
	7	Notepad.exe	0.960	1.000	1.000
	8	Msdaenum.dll	0.950	0.940	0.980
	9	Msdaer.dll	0.910	0.950	0.960
	10	Msdasc.dll	0.880	0.920	0.950
		AVG	0.969	0.981	0.989
Malicious file	1	Spybot	0.860	0.930	0.980
	2	Arianne.1052	0.820	0.890	0.960
	3	Drox	0.620	0.900	0.960
	4	Format A. F	0.780	0.760	0.990
	5	Trinoo	0.960	0.950	0.930
	6	Virtual root	0.960	0.960	0.970
	7	Weird	0.530	0.920	0.950
	8	Eriz.401	0.680	0.860	0.980
	9	Gildo	0.750	0.900	0.930
	10	Silly.c	0.860	0.820	0.940
		AVG	0.782	0.889	0.959

indirect jump instruction (address: 24), with one side pointing to the address: 0 and the other side to the address: 18. After appending control flows, recalculate the DFG, judge the original data flow whether changes. After added control flow edge which from address: 24 to address: 18, the register eax was bring new assignment in address: 18. So recalculate the indirect jump address, getting the new target address: 12. Then append new control flow from address: 24 to address: 12. At this time, data flow unchanged and fix point appeared algorithm end. Figure 5 is shown the final outcome.

EXPERIMENTAL RESULTS

To verify proposed method, we build the prototype system (Radux). We do our experiment on PE format files to measure the disassemble accuracy. The experiment results will compare with OBJ dump and IDA Pro which were popular disassemblers.

The total number of sample in space are 20, which are divided into benign program and virus program Virus samples used in the experiments downloaded from the well-known website VX Heavens (Nathan *et al*., 2008). Benign program are selected from operating system and legal software. The programs we choose are the PE files under Windows directory after the installation of Windows XP for the first time. The experiment results showed in Table 1.

As shown by the result in Table 1, Radux have decode capable at PE files. Using proposed method not only deal with normal files with higher accuracy, but also handle malware. The disassemble accuracy is higher than other popular disassemblers. In addition the common disassemblers can deal with properly, but doesn't cope with malwares. This is because normal software didn't using confusion methods to interfere with disassembler. But hacker usually adopted obfuscation methods to escape detect.

CONCLUSION AND RECOMMENDATIONS

Disassemble is usual method to obtain software function. Hacker often deceives disassemblers by disguising its actual behavior using obfuscation. This study proposes a disassemble algorithm based on CFG and DFG for malware. This technique has been implemented as part of our system Radux. Experimental results prove that the approach is effective on disassemble malicious binary code.

Self-modifying code is usually encountered in static analysis on executables. Static analysis of self-modifying code will be the next focal point.

ACKNOWLEDGMENT

This study is supported by the National High-Tech Research and Development Plan of China under Grant No. 2006AA01Z408, national project: 2009ZX01036-001-001 and the scientific and technological project of Henan Provincial of China under Grant No. 092101210501.

REFERENCES

Abhishek, S., 2009. Identifying Malicious Code Through Everse Engineering. Springer, New York, pp: 147-150, ISBN: 978-0-387-09824-1.

Arun, L., U.K. Eric and V. Michael, 2005. A method for detecting obfuscated calls in malicious binaries [J]. IEEE Trans. Softw. Eng., 31(11): 955-967.

Chao, D., P. Jianmin and Z. Rongcai, 2008. Research on deobfuscation against malicious code obfuscated with conditional jumps [A]. International Conference on Information Technology and Environmental Systems Sciences.

Cullen, L. and D. Saumya, 2003. Obfuscation of execurable code to improve resistance to static disassembly. Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), Washington DC, pp: 290-299.

Intel-64 and IA-32, 2009. Architectures Software Developer’s Manual. Intel Corporation.

Konstantin, R., 2005. Efficient Static Analysis of Executables for Detecting Malicious Behaviors [D]. Polytechnic University, Brooklyn.

Mila, D.P., 2004. Code obfuscation and malware detection by abstract interpretation [D]. Ph.D. Thesis, Department of Computer Science, University of Verona (Itali), Verona, Italy.

Nathan, E.R., Z. Xiaojin, P.M. Barton and H. Karen, 2008. Learning to analyze binary computer code. 23rd Conference on Artificial Intelligence (AAAI-08), Chicago, Illinois.