Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 70 (2015) 757 – 763

Procedia Computer Science

4th International Conference on Eco-friendly Computing and Communication Systems

Different Obfuscation Techniques for Code Protection

Chandan Kumar Behera[a*], D. Lalitha Bhaskari [b]

[a,b]Department of Computer Science & Systems Engineering, Andhra University, Visakhapatnam, India

Abstract

With the advancements in digital technology, the threat of unimaginable level of duplicating and illegal reproducing of software also increases. Therefore the piracy rate is increasing proportionally. This scenario has clearly placed the threat for the software manufacturers and leads to the development of numerous software protection techniques. The numerous software protection techniques have been developed and one of such software protection techniques is code obfuscation. The code obfuscation is a mechanism for hiding the original algorithm, data structures or the logic of the code, or to harden or protect the code (which is considered as intellectual property of the software writer) from the unauthorized reverse engineering process. In general, code obfuscation involves hiding a program's implementation details from an adversary, i.e. transforming the program into a semantically equivalent (same computational effect) program, which is much harder to understand for an attacker. None of the current code obfuscation techniques satisfy all the obfuscation effectiveness criteria to resistance the reverse engineering attacks. Therefore the researchers as well as the software industries are trying their best to apply newer and better obfuscation techniques over their intellectual property in a regular process. But unfortunately, software code is not safe, i.e. still it can be cracked. This paper presents some of the obfuscation methods, which can help to protect the sensitive code fragments of any software, without alteration of inherent functionalities of the software. The proposed obfuscation techniques are implemented in assembly level code, with taking care of the theory of optimizing transformations. The assembly code represents the data dependencies and comfort to analyse the data after disassembling the executable as compared to the decompiled code.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Organizing Committee of ICECCS 2015

Keywords: Program obfuscation; software protection; code transformation; Assembly code; byte level manipulation.

* Corresponding author.
*E-mail address:*ckb.iitkgp@gmail.com

1877-0509 © 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Organizing Committee of ICECCS 2015
doi:10.1016/j.procs.2015.10.114

1. Introduction

IT industries spend billions of dollars annually for preventing from security attacks such as tampering and malicious reverse engineering. Because of huge application and development on internet technologies and multimedia, the vast necessity for research on security and protection has been formed. Every organization is having its own intellectual property and it's a big challenge for them to protect their data, i.e. software piracy or injection of malicious code etc. And also, their data processing through the application is confidential, so its uncovering may damage the software purchaser's business directly. There are two general ways to protect the intellectual property, legally or technically. Legally means getting copyrights or signing legal contracts against creating duplicates etc. And technically means the owners of the software will give the solution for protection with that particular software. Previously, securing data means use of firewalls and gateways in the operating system itself or on the network. But, for defending from outsiders, the better idea is to use these mechanisms or methods within the application software. One of those types of methods is obfuscation, which is a novel area of research in the field of software protection and gaining more importance in this present digital era.

Obfuscation consists of code transformations that make a program more difficult to understand by changing its structure, while preserving the original functionalities, not suitable also to reverse-engineering. Encryption and firewalls are some of the common solution to diminish the threat of the attackers who try to crack the application. But, these approaches do not help to protect the software, when the attacker is him/herself the end-user. Among the various techniques available for protecting code from different attacks, code obfuscation is one of the most popular alternative, for preventing from code comprehension, code tampering etc. So, code obfuscation is a largely adopted solution, and many different obfuscation approaches has been proposed. This is also a type of software protection against unauthorized reverse-engineering.

However, a determined attacker, after spending enough time to inspect obfuscated code, might locate the functionality to alter and succeed in her/his malicious purpose. For this reason, obfuscation techniques are implemented with other approaches, such as code replacement/update, code tampering detection, protections updating (by that the attackers get a limited amount of time to complete their objective) etc. Practically, encryption, protection by server-side, hardware-based security solutions, different signed native codes, tamper proofing, watermarking, software aging, packing are some of the most commonly used methods to avoid or challenge the detection engines. However, the provider should estimate how long the obfuscation would resist, i.e., the time taken by an attacker to understand the code. According to that, some other obfuscation methods can be implemented on the original code, so that; the adversary will not be able to get the algorithm or logic of the code.

Further, obfuscation methods include code re-ordering, transformation to replace meaningful identifier names in the original code with meaningless random names (identifier renaming), junk code insertions, unconditional jumps, conditional jumps, transparent branch insertion, variable reassigning, random dead code, merge local integers, string encoding, generation of bogus middle level code, suppression of constants, meshing of control flows and many more.

Basically, obfuscation is different from encryption in many ways. Primarily, it does not require any inverse transformation. Next, it's not necessary for an attacker to look for the original code all the time, because the attack can be succeed without having the original code of the software. And at last, cipher text will be worthless without the key, as an obfuscated program can perform without any additional information.

If we discuss another aspect, mostly, software code is portable and distributed across the networks, which can be un-trusted also. So the protection mechanism must be included within the software, but it should be hardware independent also. The primary function of any software protection techniques is to detect the pirate, corrupt or misuse of the code or application. Based on this, it is supposed that 'code obfuscation' is the simple and foremost source code protection tool in the area of software protection and security. The main idea behind these obfuscation techniques are to hide the original code from the adversary, as the code will be transformed, but its functionality will be similar to the original code; but much more difficult to analyse or understand.

For analyzing a code, any disassemblers or de-compiler is needed to use on the executable code. But, it's obvious that the disassembled code will be not similar to the original code, as it is not possible to get back with all same functionalities. As most of the code analysis techniques have been researched and experimented on Assembly

code or with low level code; so, Assembly level programming is considered in this paper. In the next section, few obfuscation techniques are discussed.

According to Collberg et al [1], Code obfuscation refers to a class of techniques that transform a source program into a target program, such that both programs have the same behaviour, but the targeted program is difficult to reverse engineer by any attacker. The above observations motivated us to design the obfuscation schemes, which protect sensitive code fragments with satisfying all efficiency criteria mentioned in [1]. It is also possible to combine more than one obfuscation techniques such that we may achieve better strength. The idea moved towards obfuscation, because of these following remarks.

- Obfuscated application software never goes through any interruption due to network limitations. Also it does not need any hardware to encrypt or decrypt the code. So there is no requirement of digital signature for software application to authenticate, whether the application is having secure code from the reliable source or not.
- Encryption techniques need some specific hardware to act effectively, with the signed code to put some restriction, which should not be dependent on hardware platform. This use of dedicated hardware, also expend the cost to the software purchaser.
- De-compilation will be harder for the obfuscated code, even after spending enough effort and time. But it is possible to get back the algorithms and data structures. So, the main intention is to increase the time and effort, by that it is efficiently infeasible for an adversary to reverse engineer the obfuscated code or software.
- By obfuscation, it is possible to entangle the code and eliminate the majority of logical links, so that, the transformed code becomes complex enough for analysis and unauthorized modifications.
- There are so many complex encryption algorithms, which are not possible to implement, because of the limitations of memory and the bandwidth over network.
- In the recent progresses, the theory of obfuscation still in need of evaluation of the quality of practical obfuscating transformations in a quicker and easier way.

By putting together, some of the above mentioned aspects in software protection techniques, it can be considered that the ‘code obfuscation’ concept may be a stronger as well as a better tool for securing the software.

To analyze the code, it is very much necessary to find the order in which all the instructions are executed. That order of execution can reveal by control flow graph. Generally, if the control flow graph is complicated, that means the code of the program is also complicated. But, the important point is the control flow graph is useful, if loops and conditional statements are available in the program. In another case, if there are no conditional statements or loops in the code, then the whole code will be treated as one block. In this paper, the mentioned code obfuscation techniques will be useful for fraction of code (if there is no conditional statement) or the code within a basic block. So, it is believed that, our obfuscation techniques can be implemented to those parts of code, for which making control flow graph is not possible or there is no conditions available.

2. Different Code Obfuscation Techniques

In this section, the code fraction, which mentioned italic, are tried to obfuscate.

i) By data transformation:

Program-1 (Original Code)	**Program- 1 (Obfuscated Code)**
.model small .code Mov DH, 65 Mov AH, 2 *Mov DL, 75* Int 21h Mov AH, 4CH Int 21h end	.model small .code Mov DH, 65 Mov AH, 2 *db 10110010 b* *Dec BX* Int 21h Mov AH, 4CH Int 21h end

In the obfuscated code, the machine code is written for 'Mov DL'. The binary form of the number 75 is 01001011. For decrementing the content a register, the instruction is '01001reg' and the register BX represented as '011'. So the binary number can be replaced as the assembly code 'Dec BX'.

ii) By reflecting of carry:

Program-2 (Original Code)	Program-2 (Obfuscated Code)
.model small .code Mov AH, 2 Mov DL, 65 Mov BL, 70 L1: Int 21h Add DL, 1 *Cmp DL, BL* *JLE L1* Mov AH, 76 Int 21h end	.model small .code Mov AH, 2 Mov DL, 65 L1: Int 21h Add DL, 1 Mov BL, DL *Add BL, 185* *JNC L1* Mov AH, 76 Int 21h end

In the above example, in place of JLE, jump with no carry (JNC) is used. If the carry flag is clear, then JNC transfers the control to the level. It should be only taken care of the boundary, which is in between -128 to +127. According to this the both mentioned programs give the same output.

iii) By using indirect addressing:

Program-3 (Original Code)	Program-3 (Obfuscated Code)
.model small .code Lea SI, L1 Mov AH, 2 Mov BL, 70 Mov BH, 71 Mov CL, 72 Mov CH, 73 Mov DH, 74 L1: *Mov DL, CH* Int 33 Mov AH, 76 Int 33 end	.model small .code Lea SI, L1 Mov AH,2 Mov BL,70 *Add CS:[SI+1], AH* Mov BH, 71 Mov CL, 72 Mov CH, 73 Mov DH, 74 L1: *Mov DL, BL* Int 33 Mov AH, 76 Int 33 end

In the original program, the content of the register CH will be moved to DL. So the output is 'I'. In the obfuscated program, by the instruction 'Mov DL, BL', the value of BL, which is 70, should be moved to DL. But, because of 'Add CS: [SI+1], AH', the content of AH will be added to BL. So, 72 should be stored in BL. But, the instruction jumps to the next two instructions. So the content of CH moves to DL and the output will be 'I' in place of H.

iv) Use of register addressing:

Program-4 (Original Code)	Program-4 (Obfuscated Code)
.model small .code Mov AH, 2 *Mov AL, 180* Mov DL, AL Int 33 *Mov BH, 186* Mov DL, BH Mov AH, 2 Int 33 Mov AH,76 Int 33 end	.model small .code *Mov AX, 47796* Mov CH, AH Mov AH, 2 Mov DL, AL Int 33 Mov DL, CH Mov AH, 2 Int 33 Mov AH, 76 Int 33 end

The output of the above original code is the contents of the registers AL and BH. According to the above example, the values are 180 and 186 respectively. In the obfuscated program, register BH is not used and in place of that, the register AH is used and for storing the content of AH, the register CH is used. The content of AL and BH in the original program are stored in the register AX of the obfuscated program, i.e. the single value 47796 can be calculated as (186 X 256 + 180). This is by calculated as content (AH) x 2^8 + content (AL). In the original code, the register BH is used in place of the register AH.

v) Combining binary instructions with Assembly code:

Program-5 (Original Code)	Program-5 (Obfuscated Code)
.model small .code Mov DH, 85 Mov AH, 2 Mov CL, 80 *Mov DX, 35537* Int 21h Mov DL, DH Int 21h Mov AH, 76 Int 21h end	.model small .code Mov DH, 85 Mov AH, 2 Mov CL, 80 db 10111010b Mov DL, CL Int 21h Mov DL, DH Int 21h Mov AH, 76 Int 21h end

In the above programs, it is shown that, machine code and the decimal numbers are converted into assembly code. The machine code '10111010' mentioned in the obfuscated code can be read as' 1011-1-010'. i.e. 'Mov DX', according to the format '1011wreg number'. Again, the 'number' of the format can be converted into some assembly code. According to this example 1000101011010001 is the binary form of 35537and can be read as 100010-10-11-010-001 of the format 'Mov-dw-11-DL-CL'. i.e. 'Mov DL,CL'. There are some other examples mentioned below:

Mov CX, 520 ⇨ db 10111001 b db 00001000 b db 00000010 b	Mov CL, 70 ⇨ db 10110001 b db 01000110 b	Mov CX, 70 ⇨ db 10110001 b db 01000110 b db 00000000 b

vi) By combining binary and decimal numbers with Assembly code instructions:

Program-6 (Original Code)	Program-6 (Obfuscated Code)
.model small .code Mov BL, 80 Mov BH, 85 Mov AH, 2 *Mov BL, 178* *Mov BH, 98* Mov DL, BH Int 21h Mov DL, BL Int 21h Mov AH, 76 Int 21h end	.model small .code Mov BL, 80 Mov BH, 85 Mov AH, 2 *db 10110011b* *Mov DL, 183* *db 98d* Mov DL, BH Int 21h Mov DL, BL Int 21h Mov AH, 76 Int 21h end

In the original program, according to the instruction 'Mov BL,178', the binary code of 'Mov BL' is 10110011, which is mentioned in the obfuscated code. The binary form of 178 is 10110010, which can be elaborated as '1011-0-010' and it is 'Mov DL' as the assembly code. So, there is a binary instruction, then an assembly code instruction and after that a decimal number is there in the obfuscated program, which is similar to the italic assembly code instructions of the original program.

vii) Use of decimal numbers in between Assembly code instruction:

Program-7 (Original Code)	Program-7 (Obfuscated Code)
.model small .code Mov AH, 2 *Mov DL,* 70 Int 33 Mov AH, 76 Int 33 end	.model small .code Mov AH, 2 *db 178d* *db 70d* Int 33 Mov AH, 76 Int 33 end

In the original program, the binary form of the italic assembly code 'Mov DL' is 10110010 and its decimal value is 178, which is used in the obfuscated code (mentioned italic).

viii) Using binary instructions in place of Assembly code:

Program-8 (Original Code)	Program-8 (Obfuscated Code)
.model small .code Mov BL, 70 Mov CL, 80 Mov DH, 90 Mov AH, 2 *Mov DL, CL* Int 21h Mov AH, 76 Int 21h end	.model small .code Mov BL, 70 Mov CL, 80 Mov DH, 90 Mov AH, 2 *db 10001010b* *db 11010001b* Int 21h Mov AH, 76 Int 21h end

In the original code, by the instruction 'Mov DL, CL', the content of one register moves to another register. The machine code of this is '100010dw 11reg_1,reg_2'. i.e. Mov represents 100010, d=1(for correct) and w=0 (for byte). So, the machine code 10001010 is generated, which is mentioned in the obfuscated program. The next part of the binary instruction is '11reg_1,reg_2'. In this example, the registers DL and CL are used. The machine codes for DL and CL are 010 and 001 respectively. i.e. the machine code for '11DL, CL' is '11010001', which is mentioned in the second italic code of obfuscated program.

ix) Use of binary instructions in between Assembly code:

Program-9 (Original Code)	Program-9 (Obfuscated Code)
.model small .code Mov BL, 70 Mov CL, 80 Mov DH, 90 Mov AH, 2 *Mov DL, 178* Int 21h Mov AH, 76 Int 21h end	.model small .code Mov BL, 70 Mov CL, 80 Mov DH, 90 Mov AH, 2 *db 10110010b* *db 10110010b* Int 21h Mov AH, 76 Int 21h end

In the first binary instruction, first 4 bits represent 'Mov' and the next 4 bits represent to the register DL. The second binary instruction is a numerical value 178(calculated as 11 X 16 + 2).

Although entire software-based methods cannot afford perfect protection, we anticipate that the above mentioned techniques significantly raise the difficulty for reverse-engineering as well as for signature-based detection and code detection through pattern matching, especially when these techniques will be combined with the existing code obfuscation techniques.

Conclusion

Due to the increasing piracy of the software, a novel attempt is made to discuss and implement some of the obfuscation methods in this paper. Normally after obfuscation, the complexity of the code increases according to logically as well as structurally because of the insertion, removal or rearrangement of the code. The techniques presented have been found to be effective. Here the initial step is taken to obfuscate the code without much increasing the complexity. These mentioned obfuscation techniques have been implemented and analysed.

The future work is aimed at the development of a framework for automation of the presented techniques and to provide as a plug-in to support other obfuscation techniques. Also, the aim has been set to implement the proposed idea for large scale software protection and improvement.

References

1. Christian Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Dept. of Computer Science, Univ. of Auckland, 1997.
2. Jean-Maries Borello and Ludovic Me, Code Obfuscation Techniques for Metamorphic Viruses, Springer, 2008.
3. C. Collberg, J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection. Addison-Wesley Professional, 2009.
4. Hideaki Goto, M. Mambo, H.ShiZuya, Y. Watanabe, Evaluation of Tamper-Resistant software deviating from structured programming rules, ACISP, LNCS 2119, pp-145-158, Springer- verilag Berlin Heidelberg, 2001.
5. P. Mishra, A taxonomy of software uniqueness transformations, master's thesis, San Jose State University, 2003.
6. LeventErtaul , Suma Venkatesh, JHide - A Tool Kit for Code Obfuscation, Software Engineering and Application, Cambridge, USA,2004.
7. Christopher W. Fraser, Eugene W. Myers, Alan L. Wendt , Analyzing and Compressing Assembly Code, ACM SIGPLAN '84 Symposium on Compiler Construction SIGPLAN Notice8 VoL 19, No. 8, 1984.
8. D. Libes, Obfuscated C and Other Mysteries. Wiley, 1993.
9. Robbie Harwood, Maxime Serrano, Lecture 26: Obfuscation, 15411: Compiler Design, 2013
10. G. Wroblewski., General method of program code obfuscation. In Proc. International Conference on Software Engineering Research and Practice (SERP 02), pages 153–159, 2002.
11. N.George, G.Charalambous, Applied Binary Code Obfuscation, 2009.
12. I. V. Popov, S. K. Debray, and G. R. Andrews, Binary obfuscation using signals, In Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, pages 19:1–19:16, Berkeley, CA, USA, 2007.
13. Marius Popa ,Binary Code Disassembly for Reverse Engineering, Journal of Mobile, Embedded and Distributed Systems, ISSN 2067 – 4074 , vol. IV, no. 4, 2012.
14. M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, B. Preneel, On the effectiveness of source code transformations for binary obfuscation. In H. R. Arabnia and H. Reza, editors, Software Engineering Research and Practice, pages 527–533. CSREA Press, 2006.
15. Giovanni Vigna, Static Disassembly and Code Analysis, Malware Detection. Advances in Information Security, Springer, Heidelberg, vol. 35, pages. 19 – 42, 2007.
16. C. Linn, S. K. Debray., Obfuscation of executable code to improve resistance to static disassembly, In S. Jajodia, V. Atluri, and T. Jaeger, editors, ACM Conference on Computer and Communications Security, pages 290–299. ACM, 2003.
17. M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, and B. Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation", Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP, 2006.