An innovative algorithm for code Obfuscation

Aslam Muhammad[1], Zia-ul Qayyum[2], Ahmad Ashfaq[3], Waqar M. M.[4] Martinez-Enriquez A. M[5]., Afraz Z. Syed[6]

[1,3,4,6]Department of Computer Science & Engineering, UET, Lahore Pakistan
[2]Department of Computing and Technology, IQRA University, Islamabad, Pakistan
[5]Department of CS, CINVESTAV, D.F. Mexico
(Corresponding author email: maslam@uet.edu.pk)

Abstract: The cod obfuscation is a process of transforming any program into an incomprehensible form for protecting it from malicious attempts. To achieve this objective, many algorithms are found in the literature. Some of them based on program instructions reordering and block reordering which are difficult to implement as well as resource requirements are very high. In addition, some associated constructs are needed to run such applications and hence demand more user expertise. In this research paper, we propose a new user friendly obfuscation algorithm based on insertion of zero impact instructions and additional code insertion. Obfuscation can be carried out of any code however we choose assembly language programs as reverse engineers always translate high level language codes to it for stealing the intellectual properties. The algorithm is implemented in Microsoft visual basic for Intel machines.

[Aslam Muhammad, Zia-ul Qayyum, Ahmad Ashfaq, Waqar M. M., Martinez-Enriquez A. M., Afraz Z. Syed: **An innovative algorithm for code Obfuscation**. Life Science Journal. 2012;9(1):527-533] (ISSN:1097-8135). http://www.lifesciencesite.com. 79

Keywords: Reverse engineering; Source code; Structural complexity; Insertion; Compilation; Obfuscation.

1. Introduction

The need of the safety of intellectual property of software developers has become clear in current years of rapid development of multimedia technologies. Now a days it has become difficult from attacker's perspective to understand the source code due to its availability in binary formats, however the reverse engineering process has made the attackers to understand the correct behavior of the software and to take out the actual logic out of the program [14], so code obfuscation came into being and this is a technique which employed to reduce the risk of the theft of this intellectual property. This research paper focuses on the methodology proposed for obfuscation. Reverse engineering is a mechanism which prevents the implementation of piracy prevention methodology. This technique actually allows the user to by pass the code detecting key and starts from the process of disassembling the code written in any language. After extracting the logic, de-compilation procedure is applied and high level abstraction from this assembly code is found. Most of the research on code obfuscation has been fixed on perplexing this de-compilation phase. In contrast of focusing on this disassembly stage our goal is to perturb the disassembly procedure to make the program harder to disassembly. Results obtained by majorly reverse engineering tools being used portray the effectiveness of our method.

Compilation is the process of translating a source code written in any language to machine code. This process consists of series of steps; each step produces some low level representation than upper level step. Reverse engineering is a dual process of recovering high level structure and semantics from a machine code program. The compilation and reverse engineering processes are shown in Fig 1.

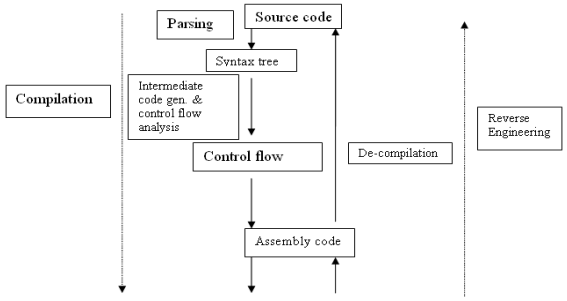

Figure1. The Processes of Compilation and Reverse Engineering

The whole reverse engineering process can be divided into two parts [2].

(i) Disassembly: which produces assembly code from machine code
(ii) De-compilation: which reconstruct the high level semantic structure of the program from the assembly code?[5]

The most of prior work regarding code obfuscation was focused on different aspects of de-compilation; our goal in this paper is to increase the difficulty level of disassembling the program [2]

2. Techniques being used for software code protection

There are numerous publications on software obfuscation and their accomplishment. A complete nomenclature of obfuscating transformations was introduced in 1997 by Collberg et al. [8]. To gauge the consequence of an obfuscating transformation, Collberg defined three metrics: cost, potency, and resilience. Software complexity metrics ([1, 6, 11, 8, 21, 13, 1]), which were designed to decrease the complexity of the software, can be used to assess this in spite of subjective metric. In compare to potency that gauges the power of the obfuscating transformation in opposition to humans, flexibility defines how beautifully it withstands an assault of an automatic de-obfuscator. This metric technique evaluates both the programmer effort (that is much effort was required to develop a de-obfuscator) and the de-obfuscator attempt (the try of space and time necessary for the de-obfuscator to run). An ideal obfuscating transformation has high strength and resilience values, but small costs in terms of extra memory usage as well as greater than before execution time. In practice, a trade-off among potency, resilience and costs has to be compromised.

Preventing disassembling is almost impossible in situations where attackers have complete control over the host containing this software; the ordinary solution is to make the effect of disassembling valueless for additional static analysis by preventing the retransformation of control flow graph. To the end of this [6] and [5] use so-called branch functions to conceal the target of CALL instructions: The explained methods substitute CALL instructions with jumps (JMP) to a general function (branch function), which function is to call is decided on run time. Under the supposition of a static analyzer being the branching function is a black box, the call objective is not exposed until the real execution of the code. This successfully prevents the control flow graph being rebuilding using static analysis. Nevertheless, the idea of a branching function does not defend against dynamic examination. An attacker can run the software on a variety of inputs and watch its performance. Medou et al. [13] argues that newly anticipated software protection models would not survive attacks that unite static and dynamic examination techniques. Even now, Dynamic analysis can be made harder using code obfuscation.

One more approach to watch cryptographic keys embedded within software is the utilization of White-Box Cryptography (WBC), which attempts to build a decryption methods that becomes challenging against white-box" attacker, who is smart enough to monitor each step of the decryption method. In WBC, the code is implemented as a arbitrary system which is dependent on any key for lookup tables. The implementation of white-box DES was firstly brought into the scene by Chow et al. [7]. On the Basis of this approach, further AES and DES white-box implementations have been suggested, but all have been broken. Billet et al. [4], Wyseur et al. [4], Jabob et al. [14], and Michiels Gorissen [4] introduced a method of white-box used cryptographic method which is able to make any software resistant against any attempt of tempering it. In this approach, the software code which is purely executable is used for the white-box lookup table for the purpose of cryptographic key. If the code has been changed it would message as an unacceptable key. On the other hand, owing to the lack of safe WBC implementations, the safety and security of the construction is ambiguous. Hardware-based methods will permit to shield the real execution of Program from the attacker completely. However, this merely moves hits to the hardware tamper resistance, while challenges like support difficulty for inheritance systems and elevated costs are rising. For that reason, hardware-based code protection techniques are obsolete

2.1 Research Challenges

Collberg's and Chenxi Wang's Algorithms are extensively used in the techniques of obfuscation. Due to the inclusion of some high level constructs, building of secondary structures, Inclusion of classic procedures as input and requirement of additional sources to make it user friendly, these algorithms are hard to implement and problems are solved theoretically. Implementation of the proposed algorithm using IZII and ACI techniques in the form of user friendly software which has not been implemented up till now is the objective of the paper as available literature presents theoretical solutions using IR and BR techniques which need some associated constructs to be implemented which makes the program very hard to compile. Designed software's simplicity is unique in a way it takes an assembly language and on a single click converts it into an obfuscated program completely different from original one and un-understandable as far as its logics are concerned. Furthermore obfuscated programs get compiled with the production of same out put as was before with out any problem. At the end the comparison of complexity measures of theoretical solutions discussed above and our implemented solution is given in tabular form.

3. Code Obfuscation Techniques

We have two types of methods to protect the software property (I) legal and (II) technical [12]. Legal methods include all possible laws which are being acting against illegal users and retailers making them face some legal actions in the form of fine and punishment, where as technical methods are Server-side execution, Code authentication, Encryption and Obfuscation. Here in this research paper we focuses only on the last method i.e. code obfuscation. Before going into the detail of the obfuscation method, we preceded our research on the basis of following hypothesis

1. In the complexity point of view machine code is more suitable than any high level language code.
2. In quality perspective, the algorithm is not bad than any so far proposed algorithms.

To prove these hypotheses we developed very effective obfuscation method for machine code. In our research work low level of programming was chosen because of the following reasons.

- The analysis of the machine code is harder than the code written in HLL.
- Decomposition of the machine code becomes impossible due to some inherited properties in compiled code from high level language [12].
- Obfuscation algorithm becomes very simple due to easier parsing methods in machine code.
- Investigation of machine code obfuscation was not found in reported literature.

Actualization of the main goal was decomposed into the following sub tasks.

I. Specific analytical methods for measuring the complexity of the programs were adopted.
II. Some empirical methods for measuring the obfuscating method's efficiency were worked out.
III. A background for obfuscation algorithm for machine code was created.
IV. An efficient method for obfuscation of machine level code was developed.
V. This developed method was implemented for most renowned architecture.
VI. After performing some measurements and experiment some appropriate and valuable conclusions were drawn.

Our approach is the combination of the obfuscation techniques working for both dynamical and static reverse engineering. Where static reverse engineering is the process of reverse engineering any software automatically with out executing it actually. Machine code can be translated into assembly language by an attacker using a disassembler and the control flow graph could be redesigned with out execution of the code. We can make reconstruction of static flow more difficult by the insertion of indirect jumps that hides the details of jump target and the utilization of branching functions. A universal method for creating an algorithm of obfuscation working in machine level code can be designed by knowing two fundamental elements: Obfuscation transformation and the results of research in the structure of a specific program. Using this methodology an algorithm of obfuscation was designed and then implemented. Following four activities were under our discussion while proposing our new methodology.

- Program's instructions reordering (IR).
- Blocks reordering of the program (BR).
- Insertion of zero impact instructions (IZII).
- Additional code insertion (ACI).

All above activities would be independent of each other, which means a different result would be produced by changing the order of execution. On the basis of methods given above a sample algorithm is designed which works on Intel* 8086 machines. From above given methods only two IIZI and ACI methods were selected for implementation because these methods have great influence on the quality of the obfuscation [11].

3.1 Structure of the Algorithm

To initialize the code obfuscation algorithm following steps are followed:

- All global objects are assigned some starting values.
- Some additional local variables are added into the source code and base addresses are found using assembly language instructions.
- Memory is allocated to obfuscated program
- The process of data flow analysis is launched
- Main loop of the program as shown in Fig 3 is started.

As stated above that this research paper focuses only on the implementation of two techniques used for obfuscation i.e IZII and ACI. Detail of these is given below.

3.1.1 Insertion of zero impact instructions (IZII)

Some instructions whose over all result is zero are embedded in the program at particular place. Zero impact instructions are of following three types.

3.1.1.1 Jumps of over all zero effects

At particular location within the program is searched out and a set of jump instructions which ultimately brings the instruction pointer at the point where it was before the insertion are inserted. This makes the program lengthy and hard to understand. For example.

```
mov  ecx, 5
     cmp  eax,ebx
     jg  greater
     jl less
     mov  edx,2
greater:
     mov  edx,2
less:
     mov edx, 2
     sub  edx, 10
```

3.1.1.2 Insertion of free elements

Free elements means any register whose value does not effect the over all result of the program, For example **add. W d3,d2** as d3,d2 do not have any significance in the program. This free element is made independent of all dependencies between last inserted instructions and the current instruction and then instruction is added to the program.

3.1.1.3 Insertion of opaque constructs

Opaque constructs are the set of instructions which are not clear to understand. For example

Functio	Ranges of parameters
Funcl(a	a=1,2,…,255 b = 0,1,…,5
Func2(	a =1,2 b = 32, 33,…,127
Func3(	a = 0,2, 3, 6

Adding few global variables aa, bb, cc, we can make new opaque constructs. Into the function Func2 (a,b,c) we can insert for an example the expression bb = (a + b + c) AND bb, which value will be always less than 100000. A not used element and a place of jump from opaque construct are chosen and after drawing the opaque construct type instructions are inserted.

3.1.2 Additional code insertion (ACI)

3.1.2.1 Insertion of reversible operations

Reversible operation is one that gives the value as was before for example following two instructions are reversible:

AX = BX-5

AX= BX+5

A used element and an operation to be inserted are picked up. All dependencies on used elements get cleared and the set of instructions performing some reversible operation is inserted.

3.1.2.2 Insertion of meaningless code

Meaningless means the insertion of those registers and flags which are not used in the program. A block of code that has no meaning is added at particular location these meaningless codes may be a string, array or declaration of some not used variables. This meaningless code reduces the readability of the original code. Locations where all above insertions are made are found by our software. Original assembly program is read from top to bottom and places are found where IZII and ACI are implemented. These places are fixed on the following parameters.

- Where the loop is being started
- Where loop is being terminated
- Where mathematical operation appears
- Look for not used registers
- Where move operation is being performed
- Where an array is declared

The architecture of main loop of the algorithm is given below in Fig 3.

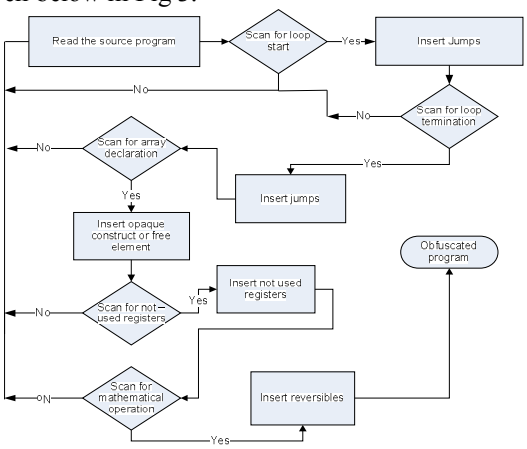

Figure 3: The main architecture of the algorithm for code obfuscation

4. Implementation of the Algorithm

The implementation phase of our research is the most important phase which we have contributed in present research. To get good performance and efficiency we designed proper data structure to store all information needed to run the program and by use of .NET technology we implemented above mentioned algorithm. Screen short of our obfuscated machine is given in Fig 4. Left hand side is the given source file and on the right hand side the program has

been obfuscated which is larger in size, difficult to understand but same in functionality.

The complete structure of program of code obfuscation consists on following six modules:

Step1. Loading of the source program- Program is loaded into the obfuscator

Step2. Basic analysis of the program - Investigation of the number of local variables and all parameters along with the jump addresses.

Step3. Data flow analysis - calculation of physical address of every instruction using pointers in the program and assembly language is very rich to provide physical address of each instruction..

Step4. Optimization of data flow - Calculation of loop holes in the context.

Step5. Obfuscation of the program – By scanning (reading each instruction and deciding the type of insertion) whole program from top to bottom an obfuscated program is obtained.

Step6. Saving of the obfuscated program – All transformations of the program is saved.

The basic thing which has been processed in this algorithm is an instruction. As obfuscated and source program are stored in the form of array structures so it is also possible to obfuscate any already obfuscated program and this is called iterative method of obfuscation[11].

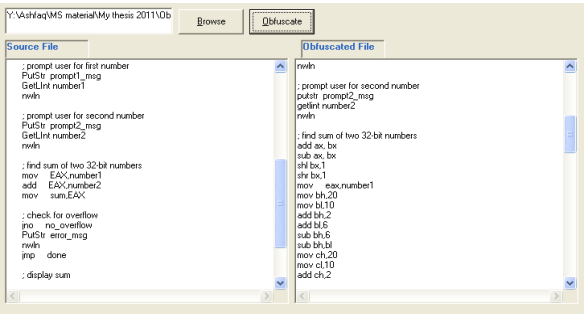

Figure 4 Screen shot of obfuscating machine

5. Results discussion

The most important parameter of obfuscating algorithm is its efficiency which reflects the power of the obfuscation process on the capacity to read the exact meaning of obfuscated program. Efficiency of obfuscation is directly proportional to the average complexity of the program Complexity as defined by Basili, is a measure of the resources expended by a system while interacting with a piece of software to perform a given task [15]. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task. For practical tests and research we have taken 5 sample programs of various tasks and obfuscated process was applied on them then the efficiency along with different parameters is measured. Detailed result discussion of all these programs is given below.

All given test programs were obfuscated with the methodology proposed by us and an analysis of obfuscated and un-obfuscated programs are made on the basis of following three parameters.

(i) Resources used by the computer for an un-obfuscated and an obfuscated program.

(ii) Efforts put by human to extract the logic from an un-obfuscated and an obfuscated program.

(iii) Structure of an obfuscated and an un-obfuscated program.

Comparative study of the resources used by the computer to run and store an obfuscated and an un-obfuscated program is carried out and the results are accumulated in table 5.1.Execution time-Calculated by C++ function **ibmts_calcTimeStamp ().** Storage is the space taken by the program on hard disk.

Program	Un-obfuscated program		Obfuscated program	
	Execution time (microseconds)	Storage Kb	Execution time (microseconds)	Storage Kb
Bubble sort	8889	4	9015	5.1
Sum Array	538	6.2	635	6.9
Sum of col and rows	745	5.4	810	6.1
Transpose of matrix	3548	3.9	3612	4.9
Array search	7345	4.7	7502	6.2

Table 5.1 Comparison of resources used by computer

Space taken by obfuscated program is enhanced due to the fact of addition of some extra code. Execution time remains almost same with slight difference which proves that our method of obfuscation effect no more on execution time and the performance of the obfuscated program remain unaffected. A tentative comparison of different people who try to extract the logic of obfuscated and un-obfuscated programs is given in table 5.2.

Table 5.2: An average time taken by different people to extract the logic of the program

Program	Un-obfuscated program			Obfuscated program		
	Student (h)	Engineer (h)	Cracker (h)	Student (h)	Engineer (h)	Cracker (h)
Bubble sort	0.21	0.18	0.11	3.13	2.30	2.01
Sum Array	0.29	0.19	0.12	3.43	3.12	2.10
Sum of col and rows	0.39	0.25	0.17	4.11	2.37	1.46
Transpose of matrix	0.40	0.23	0.10	4.27	3.31	2.51
Array search	0.32	0.17	0.06	2.46	1.19	0.47

It is obvious from this table that obfuscated program becomes harder to the people of different walks of life for extracting the logic hidden in the program and obfuscated program becomes less readable. Third parameter on the basis of which our obfuscator was tested is structural complexity. It is the measure of length E_L and flow E_F.

Measure of length E_L(See figure 1) describes specific length of program P containing N instructions, considers also number of arguments in instructions, according to the formula given below.

$$E_L(\tau,P)=\sum_{i=1}^{N} C_i \quad dlac_i = \begin{cases} 0 & When instruction has no arguments \\ 0.5 & When instruction has one argument \\ 1 & When instruction has n arguments \end{cases} \qquad (1)$$

Values of C_i were selected empirically, starting from the rule that value 1 corresponds to instructions which have most often occurring number of arguments. Remaining values were selected in a way creating diversified values of measure E_L for selected test programs.

Measure of flow E_F (see Equation 2) is a rational number, describing the average number of references to local memory in basic block of program by the formula given below. Basic block is defined as continuous sequence of instructions lying between two nodes of control flow graph.

$$E_F(\tau,P)=\frac{1}{M}\sum_{i=1}^{M} a_i \qquad (2)$$

Where M is number of basic block in program, a_i is number of references to local memory in block i. A comparison of structural complexity of obfuscated and un-obfuscated programs is given in table 5.3.

Table 5.3: Values of Structural complexity measures

Program	Un-obfuscated program		Obfuscated program	
	E_L	E_F	E_L	E_F
Bubble sort	0.67	7	0.88	3
Sum Array	0.70	3	0.79	1
Sum of col and rows	0.58	10	0.87	6
Transpose of matrix	0.66	17	0.78	11
Array search	0.75	6	0.91	4

After obfuscation, value of E_L is increased which shows that the program's length is increased after obfuscation making it difficult to understand and decreased value of logic flow E_F indicates that logic of the program becomes harder for reader to understand

6. Conclusion and future work

Implementation of our proposed algorithm for Intel architecture. The proposed approach looks very promising in the following areas of comparison.

- scalability - Describe the controllability of an obfuscation process by user
- flexibility - How easy is to use an implemented algorithm in different development environment or programming language
- portability - It describes easiness to transfer an implemented algorithm from one machine to another
- The low complexity of our algorithm is due to easiness of semantic analysis of machine languages and simplicity of implementation of data flow analysis on the low level of programming: Other available algorithms are not very portable, use very specific opaque constructs

Our proposed algorithm allows obfuscating already obfuscated programs. Programs obfuscated in such a way will have significantly different control flow graph (in comparison to programs obfuscated one time only), in the way dependent on the kind of inserted opaque constructs. The main drawback of all developed algorithms of obfuscation so far is the fact, that they remove all effects of code optimization, done by compilers. In modern processor it causes very often breaking of data processing stream, which slows down execution. That's way critical loops and highly optimized fragments should not be obfuscated. Our method of program code obfuscation is very general - it does not depend on specific properties of any computer architecture, but to general idea of context and instruction only. To convert an implemented algorithm for a new machine, it is only required to handle the specifications of its architecture (like special instructions set).

It can be seen that efficient obfuscation is also possible with low-level approach. Using the results from empirical research, we estimated parameters of obfuscation required to obtain well protected software. The main aim of the work, production of efficient algorithm of code obfuscation on the assembler level, simpler than algorithms making full analysis of structures of programs written in high-level languages, was made with satisfying conditions. Our obfuscator was tested on three parameters resources used by computer, human efforts, and structural complexity and found our algorithm efficient and very complex for reverse engineering.

Our main contribution is the implementation of the proposed algorithm using IZII and ACI in the form of user friendly software which has not been implemented up till now. Whole literature available provides only theoretical solutions. Software designed by us is so simple in use that it takes an assembly language file and on a single click converts it into an obfuscated program completely different from original one and un-understandable as far as its logics are concerned. Screen shot given in Figur4 is

the demonstration of whole software. Left hand side is the source file and on the right hand side is the program that has been obfuscated.

Obfuscation of high level language may be carried out by finding more cut off points where insertions could be made. Furthermore obfuscation process could be made intelligent by merging obfuscation of all languages in single software and allowing it to select obfuscating technique automatically.

Acknowledgements:

Authors are grateful to the Department of Computer Science and Engineering, University of Engineering and Technology Lahore, Pakistan

Corresponding Author:
Dr. Muhammad Aslam
Department of Computer Science and Engineering
University of Engineering and Technology
Lahore, Pakistan
E-mail: maslam@uet.edu.pk

References

1. Basili, V.R. Qualitative software complexity models: A summary. In Tutorial on Models and Methods for Software Management and Engineering. IEEE Computer Society Press, Los Alamitos, Calif., 1980.
2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 2001.
3. F... Allen, Control flow analysis, SIGPLAN Notices 5(7):1-19, July 1970.
4. B.S. Baker, n algorithm for structuring flowgraphs, Journal of the ACM, 24(1):98-120, January 1997.
5. Ooaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vad-han, Ke Yang, On the (Impossibility of Obfuscating Programs, Advances in Cryptology — CRYPTO'01, Springer Lecture Notes in Computer Science vol 2139, pp 1-18, Santa Barbara, CA, November 2001.
6. V. Basili, D. Hutchens, An Empirical Study of a Complexity Family, IEEE Transactions on Software Engineering, Volume 9, No 6, November 1983, pp 664-672
7. Rex Jaeschke, Encrypting C source for distribution, Journal of C Language Translation, 2(1), 2005.
8. Christian Collberg, Clark Thomborson, Watermarking, Tamper-Proofing, and Obfuscation -Tools for Software Protection, Technical Report #170, Department of Computer Science, The University of Auckland; also: Technical Report 2000-03, Department of Computer Science, University of Arizona (2000)
9. Christian Collberg, Clark Thomborson, Software Watermarking: Models and Dynamic Em-beddings, Technical Report, Department of Computer Science, The University of Auckland (1998)
10. Christian Collberg, Clark Thomborson, On the Limits of Software Watermarking, Technical Report #164, Department of Computer Science, The University of Auckland (2000)
11. Christian Collberg, Clark Thomborson, Douglas Low, Breaking Abstractions and Unstructur-ing Data Structures, IEEE International Conference on Computer Languages, ICCL'98, Chicago, IL, May 1998
12. Christian Collberg, Clark Thomborson, Douglas Low, Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, SIGPLAN-SIGACT POPL'98, ACM Press, San Diego, CA, January 2003
13. Christian Collberg, Clark Thomborson, Douglas Low, Taxonomy of Obfuscating Trans-formations, Technical Report #148, Department of Computer Science, The University of Auckland, 2006
14. Frederick B. Cohen, Operating System Protection through Program Evolution, 1992
15. Cristina Cifuentes, Doug Simon, Antoine Fraboulet, ssembly to High Level Language Translation, Technical Report 439, Department of Computer Science and Electrical Engineering, The University of Queensland, August 1998.

2/12/12