SERP'02
147SP

General Method of Program Code Obfuscation

Gregory Wroblewski

Institute of Engineering Cybernetics

Wroclaw University of Technology

Wroclaw, Poland

Mail: 14608 NE 42nd Pl #407
Bellevue, WA
98007 USA

E-mail: gwroblew@hotmail.com

Phone: +1 (425) 707 6974
Fax: +1 (425) 936 7329

Keywords: Obfuscation, Security, Reverse Engineering.

General Method of Program Code Obfuscation

Gregory Wroblewski
Institute of Engineering Cybernetics
Wroclaw University of Technology
Wroclaw, Poland
E-mail: gwroblew@ict.pwr.wroc.pl

Abstract *Obfuscation can be a simple tool for software protection. In this paper we present a method of machine code obfuscation, which can be applied to most present processors. The obfuscation method is based on a theory, which led to two useful theorems. The proposed algorithm of obfuscation was implemented and tested using analytical and empirical approaches. The obtained results give the first estimation of the maximum possible efficiency of the obfuscation process.*

Keywords: obfuscation, security, reverse engineering, machine code

1 Introduction

Although the security through obscurity method seems to be not very popular in current software science ([3]), obfuscation is present almost everywhere. Each time someone wants to store important data he encodes it, which in fact is just pure obfuscation (doing this he assumes that an intruder can access his data). It looks somehow strange that no one obfuscates important pieces of programs or does it in a simple way. We foresee that protection of programs through obfuscation will be the next step in software security development. Obfuscated program code can be the last defense, when all other protections fail.

Until now few people have researched the problem of program obfuscation. Collberg, Thomborson and Low have laid down some basic definitions and implemented a complex obfuscating algorithm of JAVA programs ([4], [5]). Chenxi Wang proposed a robust algorithm suitable for most high level languages ([10]). In this paper we present a more general approach, doing obfuscation on machine code level. This paper is a brief overview of [11].

2 Basic definitions

The general theory of machine code obfuscation is based on typical mathematic background. We assume that a computer architecture is given in the form of an instruction set and a context – vector of some variables, describing possible states of the computer. An instruction I can be written as a simple vector function:

$$I(\mathbf{c_1}) = \mathbf{c_2} \quad \text{for} \quad c_1 \in S$$

where S is the set of all possible states.

Obviously in present computers instructions change only very small part of the global context. Sequences of instructions are called programs (ex. $P = (I_1, I_2, I_3, ..., I_n)$). Only static programs, which are executable in the finite time will be analyzed. By default instructions are executed in a forward direction. Particular instructions may change the sequence of execution.

A program may use even the whole context of a computer, producing huge amount of information as the output. For an user only part of this information is important. To provide the important input and output information we define a vector of context usage.

Definition 1 *Vector* $\mathbf{v}$ *of usage of context* $\mathbf{c}$ *is a vector of values from set* $\{0, 1\}$*. Vectors* $\mathbf{c}$ *and* $\mathbf{v}$ *have the same dimension. The values of* $\mathbf{v}$ *have the following meaning:*

- 1 – *the corresponding context variable holds value important for the user*
- 0 – *the variable holds a value, which is not important*

We will perform typical logical operations on the usage vectors: AND (writing $\mathbf{v_1 v_2}$) and OR (writing $\mathbf{v_1} + \mathbf{v_2}$). We will use also the "zero" vector – $\mathbf{0}$.

To analyze a program $P(\mathbf{c_1}) = \mathbf{c_2}$ in the proposed model, one must define the usage vectors for $\mathbf{c_1}$ and $\mathbf{c_2}$. Instead of writing static vector of context variables we will write these defined vectors of context usage: $P(\mathbf{v_1}) = \mathbf{v_2}$, remembering that every program always transforms a context into a context. Usage information shows us which part of the context is transformed (elements of $\mathbf{v_1}$ or $\mathbf{v_2}$ equal to 1) and which part remains identical or unimportant (elements of $\mathbf{v_1}$ or $\mathbf{v_2}$ equal to 0).

Given a program $P(\mathbf{v_1}) = \mathbf{v_2}$, transforming usage of a context $\mathbf{c}$ according to description provided by vectors $\mathbf{v_1}$ and $\mathbf{v_2}$, one can calculate usage of the context after every instruction of P. It can be easily done with standard data flow analysis algorithms ([8]).

In [11] we classified all instructions, according to obfuscation needs, as: operations, branches and special. In most present assembler languages this classification can be projected onto the following groups of instructions:

- operations – arithmetic, logic, shifts and rotations, copying data, operating on bits and bitfields
- branches – unconditional, conditional, with/without return
- special – changing execution state (ex. RESET, HALT), input/output

There are also two important groups of operations:

- reversible operations – when there is an operation or program P_I such that:

$$I(\mathbf{v_1}) = \mathbf{v_2} \iff \bigvee_{P_I} \bigwedge_{\mathbf{c_1},\mathbf{c_2} \in S} P_I(\mathbf{v_2}) = \mathbf{v_1}$$

- irreversible operations – otherwise

In real machines there are instructions which cannot be classified as reversible or not (it depends on the current use of context).

We will use symbol $||$ to show concatenation of two programs. Given an *a priori* condition $P = P_1||P_2$ it is understood that program P was divided in any place on two programs P_1 and P_2.

Definition 2 *Program $P_1(\mathbf{v_1}) = \mathbf{v_{11}}$ is equivalent to program $P_2(\mathbf{v_2}) = \mathbf{v_{22}}$ in the context usage: input $\mathbf{v}$, output $\mathbf{v}'$, if:*

$$\mathbf{v}(\mathbf{v_1} + \mathbf{v_2}) = \mathbf{v} \quad \wedge \quad \mathbf{v}'(\mathbf{v_{11}} + \mathbf{v_{22}}) = \mathbf{v}' \quad \wedge$$
$$(\bigwedge_{\mathbf{c},\mathbf{c}' \in S} P_1(\mathbf{v}) = \mathbf{v}' \quad \wedge \quad P_2(\mathbf{v}) = \mathbf{v}')$$

which will be written as follows:

$$P_1 \equiv P_2(\mathbf{v}) = \mathbf{v}'$$

Using presented background we proposed ([11]) an alternative definition of obfuscating transformation (in comparison to [4]).

Definition 3 *Obfuscating transformation $\mathcal{T}$ is such a change of program $P(\mathbf{v}) = \mathbf{v}'$ into program $\mathcal{T}(P)$, that there is program P', such that program $\mathcal{T}(P)||P'$ is equivalent to program P in context usage: input $\mathbf{v}$, output $\mathbf{v}'$.*

$$\mathcal{T}(P)(\mathbf{v}) = \mathbf{v}_{\mathcal{T}} \ \textit{ is obfusc. tr. of } P(\mathbf{v}) = \mathbf{v}' \iff$$
$$(\bigvee_{P'} P'(\mathbf{v}_{\mathcal{T}}) = \mathbf{v}' \quad \wedge \quad \mathcal{T}(P)||P' \equiv P(\mathbf{v}) = \mathbf{v}')$$

Separation of the obfuscated program on two parts: $\mathcal{T}(P)$ and P', allows to bind reversible operations to obfuscating transformation.

In construction of obfuscating algorithms we used two properties of obfuscating transformations (proved in [11]).

Theorem 1 *If $\mathcal{T}$ is an obfuscating transformation, then $\mathcal{T}(P)$ can contain only:*

- *programs equivalent to instructions from P*
- *reversible operations*
- *instructions which change only part of the context not used in the program P*

Theorem 2 *For any program P divided in any place into two programs $P = P_1||P_2$, which are obfuscated by two different obfuscating transformations $\mathcal{T}_1(P_1)$ and $\mathcal{T}_2(P_2)$, there is a program P_X such that $\mathcal{T}(P) = \mathcal{T}_1(P_1)||P_X||\mathcal{T}_2(P_2)$ is obfuscating transformation.*

Using theorems 1 and 2 it is possible to create a pure sequential algorithm of obfuscation.

3 Evaluation of obfuscating transformations

There are two methods of evaluation of obfuscating transformations. Analytical method extracts information from program structures before and after obfuscation. It can be used to compare different algorithms of obfuscation, but it cannot answer the basic question: how well is a program protected from tampering? Only empirical research can measure this property.

3.1 Analytical methods

In article [4] Collberg, Thomborson and Low proposed three measures:

- potency – measure of complexity added to obfuscated program, in most cases it describes how hard it is to understand a program
- resilience – measures how well a given transformation protects a program from an automatic deobfuscator
- cost – describes amount of resources a program must use after obfuscation to execute

Definition 4 *For a given complexity measure $E(P)$, potency of obfuscating transformation $\mathcal{T}(P)$, $\Pi(\mathcal{T},P)$ is defined as:*

$$\Pi(\mathcal{T},P) = \frac{E(\mathcal{T}(P))}{E(P)} - 1$$

To calculate potency of obfuscating transformation some typical measures of program complexity can be used. In case of machine code we selected three well known measures:

1. Measure of length E_L – describes how long is a program and how complicated are its instructions, according to the formula (valid for two-address machines) for $P = (I_1, I_2, I_3, ..., I_n)$:

   $$E_L(\mathcal{T},P) = \sum_{i=1}^{n} \frac{\text{no. of arguments in } I_i}{2}$$

   The formula was found after some experiments and choosen to diversify measure of length for selected programs.
2. Measure of depth E_D – is an integer number of maximum nesting level of jumps in the measured program.
3. Measure of flow E_F – is a rational number of average number of references to local variables per fragment of P between two branches. Flow and depth can be calculated by a simple recursive algorithms ([11]).

To represent potency in the form of single number the average potency of obfuscation was defined:

$$\Pi_A(\mathcal{T},P) = \frac{\frac{E_L(\mathcal{T}(P))}{E_L(P)} + \frac{E_D(\mathcal{T}(P))}{E_D(P)} + \frac{E_F(\mathcal{T}(P))}{E_F(P)} - 3}{3}$$

We used the original form of resilience ([4]):

$$R(\mathcal{T},P) \in \{trivial, weak, strong, full\}$$

and the original definition of cost:

$$C(\mathcal{T},P) \in \{free, cheap, costly, dear\}$$

Eventually we presented results of the quality tests of obfuscating transformations in the form of value of average potency and description of resilience and cost.

3.2 Empirical method

Empirical tests of obfuscating transformations were done on selected groups of persons trying to understand a given obfuscated program. I selected three groups for tests: students, software engineers and crackers. The result of empirical test is given in form of the shortest time required in a selected group for understanding of obfuscated program.

4 Obfuscation of machine code

To create low level obfuscating transformations we classified all simple possible transformations (figure 1).

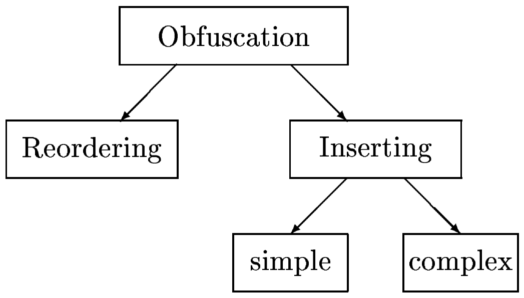

Figure 1: Classification of obfuscating transformations as seen on the low level of programming.

This classification is a consequence of the presented theoretical background. Treating program P as a sequence of instructions, when P

is divided on two parts $P = P_1||P_2$, it is possible to apply only the following obfuscating transformations: insert additional instructions, exchange one part of P, for an example P_1 with P_C, reorder instructions of P, reorder blocks of P. To diversify large possible group of inserting transformations we defined simple insertion and complex insertion. Simple one uses only information from neighbourhood instructions, while complex uses a larger part of program.

To construct an efficient method of insertion we used an interesting property of machine code, which comes from data dependencies occurring between instructions located close to each other in a program.

Definition 5 *Let program $P = (I_1, I_2, ..., I_n)$ be n instructions long. Let $\mathbf{v_i}$ be usage vector before instruction i and $\mathbf{v_i^*}$ after instruction i. Probability of dependency between instructions distant of d instructions in the program P is defined as:*

$$p(P,d) = \frac{\sum_{i=1}^{n-d} c_i}{n-d} \qquad c_i = \begin{cases} 1 \textit{ when } \mathbf{v_i^* v_{i+d}} \neq \mathbf{0} \\ \quad \textit{and} \quad \bigwedge_{j=1,2,\dots,d-1} \\ \quad \mathbf{v_{i+j}^* v_{i+d}} = \mathbf{0} \\ 0 \textit{ in other case} \end{cases}$$

Dependencies were calculated in context *(registers, variables in local memory, global memory)* using algorithm described in [11].

The property is shown on figure 2. It turns out that the probability of data dependency is smaller for larger distance between instructions.

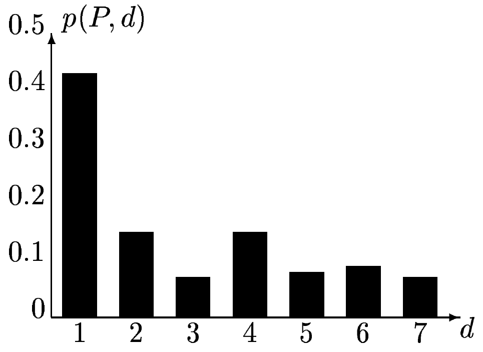

Figure 2: Probability of data dependency between two instructions as function of distance between them.

Presented property can be reproduced in an easy random way, by adding artificial dependencies to randomly generated instructions. Results of such model are shown on figure 3.

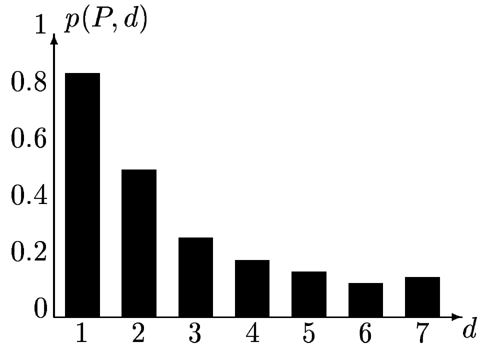

Figure 3: Probability of data dependecy between two instructions of random program with added dependencies.

It can be found from experiments that programs do not use the whole available context all the time. According to theorem 1 we can insert the two types of instructions: any instructions using free part of context or reversible operations changing used part of context.

Insertion of new instruction causes obfuscated program to grow longer. To make control of this growth we introduced rescaling factor S:

$$S = \frac{|\mathcal{T}(P)|}{|P|}$$

where |...| returns the length of a program.

Sample methods of complex insertion are presented in [4]. Among them inserting of opaque constructs seems to be most important, because it is the only way to increase significantly resilience of obfuscating transformations. The original definition of opaque construct can be found in [5]. In [5] an efficient method of creating opaque constructs is described. It is based on a high level approach (uses pointers, objects and system calls). In [11] we described a more general method.

Any automatic method of creating opaque constructs is weak, because it generates fragments having a similar pattern. This fragments still cannot be deleted automatically, but can be found easier and then removed by a human. It looks like opaque constructs are analogous to cryptographic keys: they should be unique and kept secret.

There are different possible ways to merge opaque constructs and additional inserted code (based on context usage). Both techniques are required to make the algorithm of obfuscation potent and resilient.

5 Sample algorithm

In the sample algorithm three important methods were selected in the general group of inserting instructions:

- insertion of any operations changing free part of context
- insertion of reversible operations changing used part of context
- insertion of opaque constructs

Two techniques of obfuscation were choosen for implementation: insertion and reordering of blocks. Like in [10] we treated program as a set of functions, while the algorithm obfuscates a single function only.

The implemented algorithm uses the following assumptions:

- input is a single function
- the program is obfuscated in context consisting of processor's registers and local stack
- for the whole obfuscated context full data flow analysis was done
- external function calls are not present
- the input program does not contain instructions using elements outside the obfuscated context
- there are only static jumps inside the input program
- following input parameters are given:
    1. Rescaling factor S, $S > 1$
    2. Frequency of reordering R_B – how often reordering of blocks must be applied, $0 \leq R_B < 1$
    3. Logical flags for different methods of code insertion: using free part of context I_N, reversible operations I_D and opaque constructs I_O (zero means insertion disabled), $I_N, I_D, I_O \in \{0, 1\}$

Table 1 shows global objects used in the algorithm. The algorithm uses the following conditions:

- condition of insertion: $C_I = (M < N_I \cdot S)$
- condition of reordering: $C_R = (\frac{N_B}{N_I} < R_B)$

Algorithm 1 *The main loop of program code obfuscation:*

1. *If conditions C_R and C_I are true, do block reordering.*

Table 1: Global objects used in the algorithm of obfuscation.

Object	Type	Description
L_S	list	addresses of jumps forward
L_O	list	operations on current context
N	integer	length of input program
N_B	integer	number of reordered blocks
M	integer	length of output program
N_I	integer	current obfuscated instruction

2. *If there are jumps to the current position on L_S, balance context for each jump, using operations from L_O.*
3. *Balance context for the current instruction (if required). It is done by reversing appropriate operations from L_O.*
4. *If current instruction is a jump backwards, balance context to the one from destination of the jump.*
5. *If current instruction is a jump forward, add it to L_S.*
6. *Store current content of L_O for the future context balancing.*
7. *Copy obfuscated instruction and apply changes to its context (if required).*
8. *As long as C_I is true, insert new instructions.*
9. *If copied instruction was an unconditional jump, empty L_O.*

It has been proved ([11]) that the proposed algorithm satisfies conditions of the obfuscating transformation.

6 Results and comparison

The presented algorithm was implemented on two different processors: Intel x86 and MIPS. For both architectures the same analytical and empirical measurement was done, but in this paper only results for Intel are shown (for MIPS results see [11]).

We have choosen six different programs for tests:

- HASH – simple hashing
- MATRIX – multiplication of matrices
- INSORT – sorting by simple insertion
- BUBSORT – bubble sorting
- CODETEST – calculating checksum of a program

- DECODE – decoding part of a program

Results of complexity measurement are shown in table 2. Values E_L are normalized by division by number of instructions in the measured program. In the case of empirical research we have choosen as reference data the average time needed for analysis of a program by a human. A student and a programmer needed about 15 minutes, a cracker – about 10 minutes.

Table 2: Complexity measurement of sample programs for Intel x86.

Program	E_L	E_D	E_F
HASH	0.88	1	3.0
MATRIX	0.87	3	4.0
INSORT	0.88	3	3.6
BUBSORT	0.90	3	4.7
CODETEST	0.90	4	3.7
DECODE	0.93	1	7.3
Average	**0.90**	**4**	**4.3**

As the analytical measure of quality of an obfuscating transformation the average value of potency Π_A is calculated. In the basic test we used all implemented techniques of code insertion: instructions using free elements of context, reversible operations on used elements of context and opaque constructs. In addition we set the following parameters of obfuscation: rescaling of program – 10, frequency of block reordering – 0.2. After obfuscation (table 3), programs became more homogeneous. This can be seen from smaller distortions of values E_L and E_F. Average potency of obfuscated programs grew only half of rescaling value. The resilience of obfuscated programs hardly depends on the use of opaque constructs. The cost of proposed algorithm can be classified as cheap.

Table 3: Complexity measures of obfuscated test programs.

Program	E_L	E_D	E_F	Π_S
HASH	8.4	11	3.1	6.19
MATRIX	8.3	21	3.9	4.84
INSORT	8.1	14	4.2	4.01
BUBSORT	8.2	18	3.5	4.29
CODETEST	8.4	28	3.4	4.75
DECODE	8.5	19	3.2	8.53
Average	**8.3**	**32**	**3.7**	**4.89**

Empirical tests were done only on program DECODE. In the first attempt the program was obfuscated with the use of block reordering and inserting of instructions using free elements of context only. The rescaling value was set to 5. After obfuscation time needed for analysis became significantly longer (table 4), but for most crackers guessing the correct meaning of the program was quite an easy task.

Table 4: Results of empirical quality tests of simple obfuscating transformation.

Group	**People [number]**	**Best time of answer [h]**
students	46	—
engineers	9	3
crackers	7	1

In the second attempt the full coverage of the proposed algorithm was used. Simple opaque constructs were used (based on arithmetic and logic operations). Rescaling value was set to 5. The obfuscated program turned out to be more resilient to human analysis (table 5). Crackers who managed to find the correct answer used the *brute force* method, reading disassembled listing and writing comments.

Table 5: Results of empirical quality tests of full obfuscating transformation.

Group	**People [number]**	**Best time of answer [h]**
students	14	—
engineers	7	—
crackers	5	2

For programs rescaled with higher ratio brute force method could lead to very long time of analysis. Experienced crackers proposed a different solution of how to unobfuscate a long program. The general form of such an algorithm looks like this:

- build full data flow graph (local usage of context)
- optimize data flow graph – reduces simple reversible operations and instructions using free elements of context
- scan control flow graph in the search for opaque constructs – half-automatic step, because in some cases the cracker must make the decision if a part is an opaque construct or not

Presented results obtained in the empirical research are a first attempt and cannot be referenced to any work. Until now only analytical measurement was taken into account ([4]) or the only measured value was cost ([10]).

In comparison with other known algorithms of obfuscation the proposed approach looks very promising (table 6). Its low complexity comes from easiness of semantic analysis of machine languages and quite simple implementation of data flow analysis. Other areas of comparison have the following meanings:

- portability – how easy is to transfer an implemented algorithm from one machine to another
- flexibility – how easy is to use an implemented algorithm in different development environment or programming language
- scalability – how much an obfuscation process can be controlled by user

Algorithm described in [4] is not portable, because it was designed especially for use with the Java Virtual Machine. Algorithm from [10] uses very specific opaque constructs making it not very scalable.

Table 6: Comparison of three algorithms of code obfuscation.

Property	**Collberg** [4]	**Chenxi Wang** [10]	**Wróble wski** [11]
Complexity	high	medium	low
Portability	no	yes	yes
Flexibility	medium	medium	high
Scalability	good	poor	good

7 Summary

The proposed method of program code obfuscation is very general – it does not depend on specific properties of any computer architecture. To convert an implemented algorithm for a new machine, it is only required to handle specific properties of its architecture (like special instructions). Low complexity of the proposed algorithm makes it an efficient tool for software watermarking ([6], [7]) and computer viruses.

It can be seen that efficient obfuscation is also possible with a low-level approach. Using the results from empirical research, we estimated the length of a well protected obfuscated short program (including opaque constructs and entangled code "covering" them). This information cannot be found using only analytical approach, like in [5] or [10]. The result is: $1,400,000$ to $70,000,000$ instructions.

References

[1] David Aucsmith, *Tamper Resistant Software: An Implementation*, Information Hiding, Springer Lecture Notes in Computer Science vol. 1174, 1986, pp. 317-333

[2] Cristina Cifuentes, *A Structuring Algorithm for Decompilation*, XIX Conferencia Latinoamericana de Informatica, Buenos Aires, Argentina, August 1993, pp. 267-276

[3] Michael Howard, David LeBlanc, *Writing Secure Code*, Microsoft Press, 2002

[4] Christian Collberg, Clark Thomborson, Douglas Low, *A Taxonomy of Obfuscating Transformations*, Technical Report #148, Department of Computer Science, The University of Auckland, 1997

[5] Christian Collberg, Clark Thomborson, Douglas Low, *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs*, SIGPLAN-SIGACT POPL'98, ACM Press, San Diego, CA, January 1998

[6] Christian Collberg, Clark Thomborson, *Software Watermarking: Models and Dynamic Embeddings*, Technical Report, Department of Computer Science, The University of Auckland, 1998

[7] 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

[8] Thomas Pittman, James Peters, *The Art of Compiler Design: Theory and Practice*, Prentice-Hall Inc., New Jersey 1992

[9] Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson, *Software Tamper Resistance: Obstructing Static Analysis of Programs*, Technical Report, University of Virginia, Department of Computer Science, 2000

[10] Chenxi Wang, *A Security Architecture for Survivability Mechanisms*, PhD Dissertation, University of Virginia, Department of Computer Science, October 2000

[11] Gregory Wroblewski, *General Method of Program Code Obfuscation*, PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002, (under final revision)