# General Method of Program Code Obfuscation
# (draft)

Gregory Wroblewski

Wroclaw 2002

# Contents

# List of Figures

# List of Tables

# Acknowledgments

WRITING of PhD dissertation is often a long and gravel road, but you can go through it with help of people you meet. Thanks to my guide: professor Janusz Biernat, my road became straight and easier to go. Thanks to my wife Joanna I get enough support to avoid thoughts about giving up. I place special thanks to my Parents, thanks to their sacrifice and huge help I found quiet environment to finish my work.

# Abstract

OBFUSCATION of machine code programs is a form of protection of programs' code against unauthorized reading. The problem of obfuscation is quite fresh, because first papers connected directly to obfuscation appeared only few years ago, yet some advanced publications can be already found. We reviewed them, describing in details most important papers of Christian Collberg and Chenxi Wang.

We proposed a formal model of program based on the analysis of changes in the usage of computer's resources utilized by the program. The model appeared to be useful for development of obfuscation methods working on the low level of programming. We showed that obfuscating transformation has some interesting properties and prooved, that for machine programs it is possible to create a single-pass algorithm of obfuscation. Describing own classification of obfuscating transformations we described different methods of obfuscation from the low level point of view. Obtaining results of research on typical properties of structure of todays computers' programs we created an efficient method of redundant code generation, required during the process of obfuscation. On the base of theoretical analysis and experience of another scientists we proposed a basic algorithm of machine programs obfuscation, which was implemented for the RISC and CISC type processors.

To estimate efficiency of the obfuscation process we proposed three analytical methods of quality measurements and results of empirical research. We created three algorithms of machine programs' complexity measurement. For the implementation we showed results of quality measurements, performed using analytical and empirical methods. The empirical measurements were done on three different groups of programmers. From the final results it can be concluded, what should be the form of an algorithm of obfuscation, giving almost one hundred percent safe protection against unauthorized analysis. In the final conclusions we estimated values of parameters of an obfuscation process, giving such good efficiency.

# Section 1

# Introduction

FAST development of multimedia and internet technology in recent years created huge need for research in methods of protection of intelectual property of software producers. According to common practice we have legal and technical methods. Legal methods are all possible acts creating appropriate law, which makes legal actions against illegal users and retailers ([15]). Technical methods can be divided as follows (figure 1.1):

- server-side execution – allows to avoid sending of final code to the user; can be used only in presence of high availability of broadband networks
- code authentication ([11]) – most efficient when authentication data are sent via network; user has complete code, which in theory can be mangled – authetication procedures can be removed
- program encoding – protects against tampering of program; in present programs used very often ([3]); main drawback is that decoder can be written and used as an universal tool
- code obfuscation – transformation of executable code, making analysis hard (this implies changes are hard) ([25], [72]); it should be used with other methods of software protection, because it does not protect against everything (ex. unauthorized users)

This dissertation covers last method only – youngest and not researched well yet – code obfuscation.

## 1.1 Goals and Assumptions

According to needs evolving from research and first experiences I advance following hypothesis:

- obfuscation of machine code is less complex than obfuscation on high-level of programming
- quality of such algorithm is not worse than quality of algorithms proposed so far

To proof the hypothesis I developed an efficient method of obfuscation of machine code programs. I defined efficiency using analytical and empirical research, which allowed to compare the method with other proposed methods. Comparison of obtained results with results of other authors was choosen as the criterion of verification for the second part of the hypothesis. First

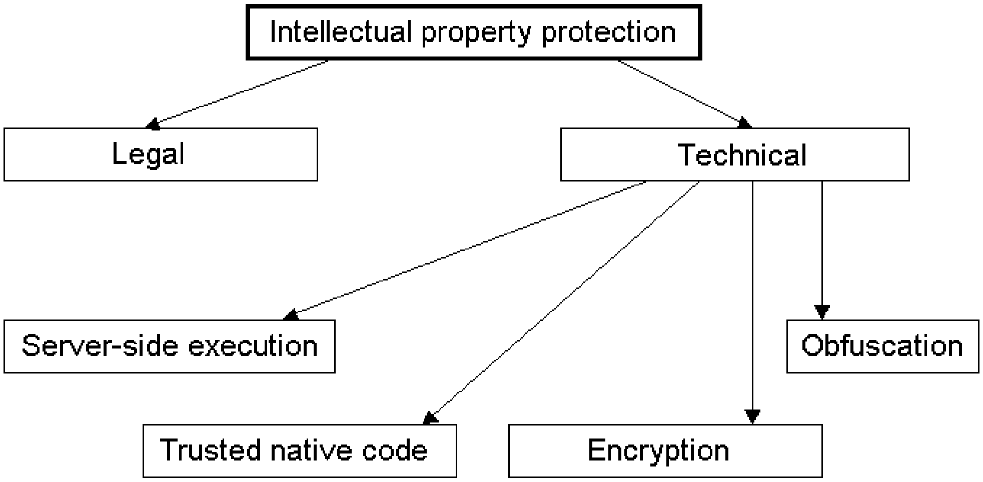


Figure 1.1: Methods of software protection according to [25].

part was verified by estimation of computer's resources required to complete compared processes of obfuscation.

I have choosen low-level of programming – machine code – because:

- it is much harder to analyze machine code that code of program written in high-level language
- obfuscation of compiled code can remove some properties inherited from high-level language[1], which can make decompilation impossible and analysis much harder
- parsing of machine code is much easier than parsing of most popular high-level languages, it makes obfuscation algorithm simple
- there are no traces in scientific publications that obfuscation of machine code was investigated

Because of syntactic simplicity of assembler languages, grammar based formal approach (known from [61] or [2]) was not used, but instead a simple own formal background was developed.

Realization of the main goal was decomposed on a set of sub-tasks:

1. Adaptation of selected analytical methods of measurement of programs complexity to machine code level.
2. Work out of empirical method measuring efficiency of obfuscating methods.
3. Creation of a formal background for the algorithm of obfuscation working on the machine code level (for class COSH, according to Treleaven's classification).
4. Carry out research and measurements of typical properties of programs for present processors.

[1] For an example repeating fragments of code or similar control flow patterns for particular type of a loop.

5. Development of an efficient method of machine code obfuscation.
6. Implementation of the developed method for most popular architectures (required for effectiveness tests).
7. Performing of some experiments and mesurements and providing appropriate conclusions.

Execution of presented tasks was described in consecutive chapters of dissertation.

As the subject of research a single function (procedure, method, etc.) of a high level language was selected. It should be executed in the sequential way and will be analyzed from the machine code level. The selection was made on the practical basis, because most present programs are written in the high level languages, while strong protection is required only for small fragments.

## 1.2 Current State of the Art

Scientific approach to the problem of program code obfuscation is so young, that it is hard to say about its history. The first trials of systematic research connected with obfuscation are dated in the late 1990. The main impulse causing more interesting in this subject was fast development of Java language technologies. The fact, that programs written in this language and compiled to Java Virtual Machine code can be in a trivial way transformed form the executable form to source code, caused a great need for creation of theories and tools making obfuscation of executable code possible. The most advanced work in this area was done by Collberg, Thomborson, Low and also by Chenxi Wang (as the PhD dissertation) [72]. Publication [76] is the first one connected directly with obfuscation of machine code programs.

In [25] a first detailed classification of obfuscating transformations was shown and some analytical methods of quality measures were proposed. A general algorithm of obfuscation was described, which can be applied to most popular high level programming languages. Yet final results obtained after implementation and details of implementation were not included, in opposition to similar, but no so sophisticated papers [49] and [50].

Different approach was taken in [72]. A particular obfuscating transformation was researched there in pure theoretical basis and next implemented on the source code level of C programming language. Empirical research of efficiency of the obtained algorithm were not performed.

Specialized methods of programs' code obfuscation are also present in publications about software protection or software watermarking. Some of the description of methods of software protection ([6], [34], [53], [64]) contain techniques of obfuscation, even the authors do not mention it directly. The same we found in descriptions of software watermarking methods as well ([29], [57]).

In many papers obfuscation is mentioned as one of the methods of encoding or steganography ([5], [35], [36], [42], [51], [71]). Separate group make publications approaching directly or indirectly the problem of unobfuscation, which is strongly connected with decompilation ([14], [17], [19], [22], [23], [43], [44]). Similar approach can be found in work dedicated to reverse engineering ([20], [63], [77]).

In recent years the first attempts to create a theoretical basics of programs' obfuscation has appeared, which are based on cryptographic theories. Major accomplishment in this area are results from [37] and [9], where a proof is shown, that one hundred percent efficient (in an idealized approach) methods of programs' obfuscation do not exist. Some alternative models of further theoretical research were proposed either and suggestion has been made, that known heuristic approaches ([25], [49], [72]) may indicate, that a class of efficient obfuscating transformations must exist. Our dissertation is the next example of a heuristic approach.

## 1.3 Overview of Chapters

Next chapter shows a review of scientific papers about code obfuscation. Works of Collberg and Chenxi Wang were described in details and general definitions of obfuscating transformation were given.

Chapter 3 contains description of the format background developed especially for dissertation needs. An alternative definition of obfuscating transformation was given and some theoretical conclusions, useful in constructing of obfuscating algorithm, were presented.

In chapter 4 a typical methods of measurement of quality of obfuscating transformations were described, together with adaptation to the low level of programming. A background of empirical research methodology is also included.

Chapter 5 contains classification and description of obfuscating transformations – basic elements of obfuscation process.

The method of creation of obfuscating algorithms with sample algorithm was presented in chapter 6. Remarks about implementation and discussion about efficiency was also included.

Summary, proposals of future research and final conclusions are included in chapter 7.

Data concernig research and experiments are presented in consecutive appendixes:

- appendix A – listings of test programs
- appendix B – methodology of research of machine code programs properties
- appendix C – examples of obfuscation for selected test programs
- appendix D – web sources of information about code obfuscation

In the appendixes detailed data are included, which allow to reproduce all described experiments.Some intermediate results of research are also described.

# Section 2

# What is Code Obfuscation?

THERE is not a common formal definition of the obfuscation process in current publications. Obfuscation, being a transformation of program into program, can be understand as the special case of data coding. The further analysis shows, that there are a lot of similarities between obfuscation and cryptography, but still we cannot treat these two techniques as equivalent.

## 2.1 Definition of Obfuscation Process

Depending on context different definitions of the obfuscation process can be found. Analyzing obfuscation from the security point of view and describing obfuscating transformation as a ”one-way translation”, the following definition was given in [72]:

**Definition 2.1** *Let **TR** be translating process, such that $P \overset{\mathbf{TR}}{\to} B$ translates source program $P$ into a binary program $B$. TR is a one-way translation, when time required for reconstruction of program $P$ from program $B$ is greater from a specific constant $T$.*

There exists full analogy between one-way translation and cryptography: cryptographic schemas work with assumption, that coding is easy and reverse process is computationally complex (without having the key).

Most general available definition of the obfuscation process can be found in papers [25] [26] and others written by the same authors. Accorting to their idea obfuscation process is a transformation of a computer program into a program.

**Definition 2.2** *Let $\mathcal{T}(P)$ be program, transformation of program $P$. $\mathcal{T}$ is an obfuscating transformation, if $\mathcal{T}(P)$ has the same observable behavior as $P$. In addition $\mathcal{T}$ must follow conditions:*

- *if program $P$ fails to terminate or terminates with an error condition, then $\mathcal{T}(P)$ may or may not terminate*
- *otherwise $P$ termiates and $\mathcal{T}(P)$ must terminate and produce the same output as $P$*

Above definition does not imply how a transformation of program should work in order to be *obfuscating* transformation. Authors of paper [25] make proposition of a classification of all obfuscating transformations, according to their experience and current state-of-the-art in obfuscation techniques.

## 2.2 A Taxonomy of Obfuscating Transformations

Basic criterion of classification is target of application. Four groups bound to obfuscation of some information were created here ([25]):

- layout obfuscation – its source and/or binary structure (ex. change of exported functions names on random character strings)
- data obfuscation – some local and global structures
- control obfuscation – of main skeleton creating a program
- preventive obfuscation – protecting from decompilators and debuggers

In every group there are lot of specific methods, which were classified and described with details in [25]. Because the original description is rather verbose, I presented short overview only.

Without doubts classification made by Collberg does not cover all possibilities of layout obfuscation. The reason comes from specialization of the developed algorithm [2]. Techniques of identifiers coding and removing of comments are described in a clear-cut way, but into the category of "changing format" we can put more transformation, not only applying to source code, but also to binary code. For Java programs it can be change of layout of blocks inside a method, while for machine code binary program typical would be modification of executable file (ex. increasing of code sections).

Classification of data obfuscation methods is much more complicated. They were divided on three main groups:

1. Storage and encoding obfuscation – change of representation and methods of usage of variables, for an example:
   - split variables – representation of a variable in the form of more than one variables and identity mapping
   - promote scalars to objects – ex. integer variables to objects with complex methods, instead of trivian addition or multiplication
   - convert static data to procedure – ex. value 1 to formula $\frac{b+1-a}{\cos(a+\pi-b)}$, with assumption that $a = b$ and both values are precise
   - change encoding – ex. logical values TRUE and FALSE for boolean variables
   - change a variable lifetime – ex. from local to global or from local to the element of object
2. Aggregation obfuscation – merge independent data and split dependent data, for an example:
   - merge scalar variables – storing in single consistent area, requres changes in references to these vairables
   - modify inheritance relations – ex. adding redundant objects to already present structures

[2] High level object oriented languages, ex. Java.

  - split, fold, merge arrays – complication of access to arrays

3. Ordering obfuscation – reordering of internal objects layout, for an example:
   - reorder variables – local, global or in structures (originally this order is random very seldom)
   - reorder methods – as they are part of objects
   - reorder arrays – ex. non-standard representation of multi-dimensional arrays

Techniques presented in the groups are just examples good for obfuscation of structures in an object oriented programming language. Indeed, it is hard to find methods, inheritance and even arrays in pure machine level program.

In classification of control obfuscation methods also three groups of transformations were created:

1. Computations obfuscation – change in main structure of control flow, for an example:
   - reducible to non-reducible flow graphs – ex. insertion of contructions does not occurring in high level languages, [2] p. 606
   - extend loop condition – addition of conditions not changing behavior of program
   - table interpretation – ex. creation of a simple virtual processes and addition of pseudo-code interpreters
2. Aggregation obfuscation – spliting and merging fragments of code, for an example:
   - inline method – instead of method call, independent obfuscation of every inserted copy
   - outline statements – artificial creation of global procedures
   - clone methods – similar to inline method
   - unroll loop – possible only for short loops with constant counter
3. Ordering obfuscation – reordering of blocks, loops and expressions, with preservation of dependecies, for an example:
   - reorder block – ex. in branching instructions
   - reorder loop – possible often in nested loops
   - reorder expression – very cheap when there are no dependecies between two exepressions (no extra code required)

Even the authors admitted in [25], that most of presented obfuscating transformations are eventually inserting of a redundant code into the obfuscated program. Only selection of programming language makes them so sophisticated and created so many methods.

Apart from such a general approach, in study of other authors, ex. Chexi Wang [72], much simpler classifications can be found, prepared for a specific application.

For an example in the same work obfuscating transformations are classified as:

- intra-procedural transformations

  - degeneration of control flow by change of static branches on dynamic
  - loose injection of data aliases

- inter-procedural transformations
  - change of function calls into indirect calls, using pointers to functions
  - creation of aliases to function pointers
  - injection of data dependent aliases between different procedures

In this classification methods of creation of pointers to data aliases were distinguished, because it has been proved ([32]), that elimination of such constructions is a task computationally very complex.

## 2.3 Algorithms of Obfuscation

Until now most general and most advanced algorithm was presented in [25]. It was adapted to high level programming languages, especially object oriented languages. Below, the main part of obfuscation procedure is presented.

### 2.3.1 Collberg's Algorithm

The original algorithm is quite large. Even the authors have given just its basic data structures and general methodology. Main part of the algorithm gets as input the following objects:

(a) application $\mathcal{A}$, consisted of source files or executable objects $C_1, C_2, ...$

(b) standard libraries $L_1, L_2, ...$ defined in given programming language

(c) set of obfuscating transformations $\{\mathcal{T}_1, \mathcal{T}_2, ...\}$

(d) mapping $P_t$, from every transformation $\mathcal{T}$ into a set of language constructions inserted by $\mathcal{T}$ into application

(e) set of functions measuring quality of $\mathcal{T}$ related to a code $S$

(f) set of input data $I = \{I_1, I_2, ...\}$ of application $\mathcal{A}$

(g) two numerical values: AcceptCost $> 0$ and ReqObf $> 0$, where first given information about acceptable increase of resources required by program after obfuscation and second – about required level of obfuscation

Before execution of the main loop, some auxiliary structures has to be built:

1. Load elements of application $C_1, C_2, ...$, which means:
   - for source code: full lexical, syntactical and semantic analysis of code[3] (exactly like in case of compilation)
   - for executable code: making of analogous structures/analysis (used only in case of objects including full or almost full source code information)

[3] Less efficient algorithm may use syntactical analysis only.

2. Load libraries $L_1, L_2, ...$ called directly or indirectly by obfuscated application.

3. Build internal representation of the whole application. Selection of internal representation depends on source code language type and complexity of used obfuscating transformations. Typical set of data structures contains:

    - control flow graph for every procedure in $\mathcal{A}$
    - call graph for every procedure in $\mathcal{A}$
    - inheritance graph for all classes in $\mathcal{A}$
    - global data flow graph in $\mathcal{A}$

4. Create auxiliary mappings using additional algorithms:

    - $R(M)$, for every procedure $M$ in $\mathcal{A}$ return duration of execution of $M$
    - $P_S(S)$, for every fragment of code $S$ in $\mathcal{A}$ return set of language constructions used in $S$
    - $I(S)$, for every fragment of code $S$ in $\mathcal{A}$ return a priority of obfuscation of $S$
    - $A(S, \mathcal{T})$, for every fragment of code $S$ in $\mathcal{A}$ return level of accuracy of applying $\mathcal{T}$ to code $S$

5. Execute main loop of obfuscation (algorithm 2.1).

6. Process final internal structures to generate obfuscated application $\mathcal{A}'$.

Algorithms created with presented method prepare the environment for execution of the main loop of obfuscation. The main loop is just a simple template based on two additional functions, because most things were done during preparation.

**Algorithm 2.1** *Apply obfuscating transformations to application. In every step select a fragment of code $S$ and appropriate transformation $\mathcal{T}$ and apply it to $S$. Processing ends, when required level of obfuscation of acceptable cost of execution of the destination code is reached.*

```
REPEAT
    $S :=$ SelectCode($I$)
    $\mathcal{T} :=$ SelectTransformation($S, \mathcal{A}$)
    Apply $\mathcal{T}$ to $S$ and update appropriate structures reprezenting the application (ex. data and control flow graphs)
UNTIL NotFinished(ReqObf, AcceptCost, $S, \mathcal{T}, I$)
```

### 2.3.2 Chenxi Wang's Algorithm

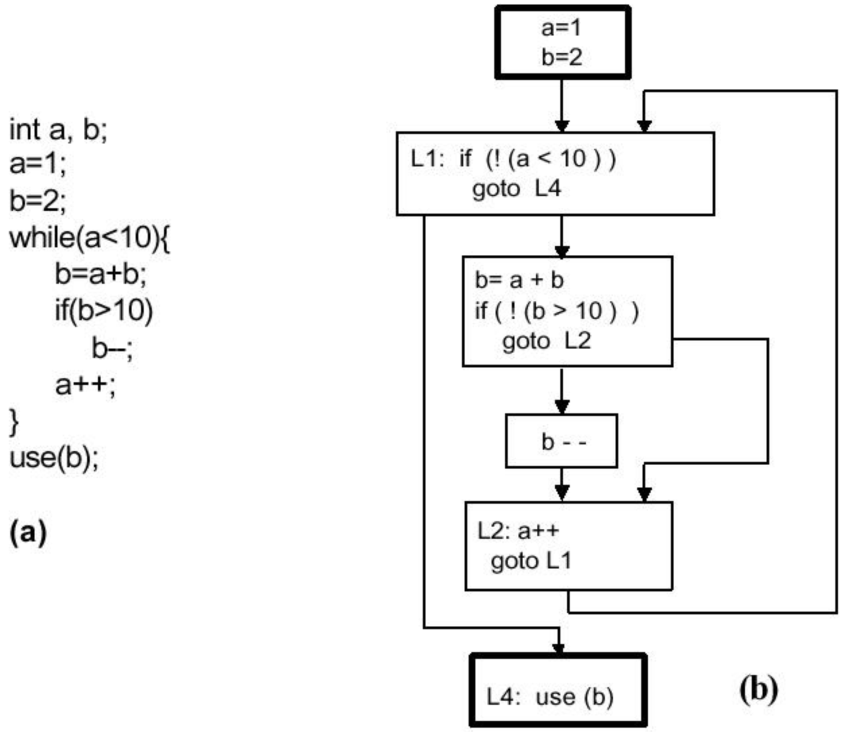
In the paper [72] a more specialized algorithm of obfuscation is shown (in comparison to general algorithm presented by Collberg). The algorithm has been divided on the two parts: interprocedural and intraprocedural. We are interested in obfuscation of single functions only, so only interprocedural part will be described.

The input of the algorithm is a typical procedure of a high level language (figure 2.1(a)). Obfuscation process of every procedure has been divided into three stages:

- dismantling of control flow graph

- flattening of control flow graph
- addition of structures with data aliasing

In the first stage the conrol flow graph is built and every loop construction is substituted with `if...goto` construction (figure 2.1(b)). The main goal of this stage is to simplify program analysis in the next stages.



```
int a, b;
a=1;
b=2;
while(a<10){
    b=a+b;
    if(b>10)
        b–;
    a++;
}
use(b);
```

Figure 2.1: Example of dismantling of a control flow graph in the Chenxi Wang's algorithm [72].

There are branches and nodes in the dismantled contol flow graph. If we enumerate all branches and add a variable holding number of currently executed branch, then it is possible to flatten the control flow graph into the form like on the figure 2.2. Change of value of variable `swVar` causes jump from one branch into another.

The process of flattening can be reversed in an easy way, mainly because data causing jumps between the branches are constant. The solution proposed in [72] adds data aliases in the form of a redundant global array `g[]` (figure 2.3). The array contains data required to switch branches during execution and not important random data. When switching occurrs values in the array are changed in the way, that important data remain unchanged. It is proved in [72] that static analysis of such a program in computationally complex.

### 2.3.3 Other Algorithms of Obfuscation

Additional examples of code obfuscation algorithms are programs obfuscating source and executable form of Java classes. Unfortunately these programs are described only from commercial

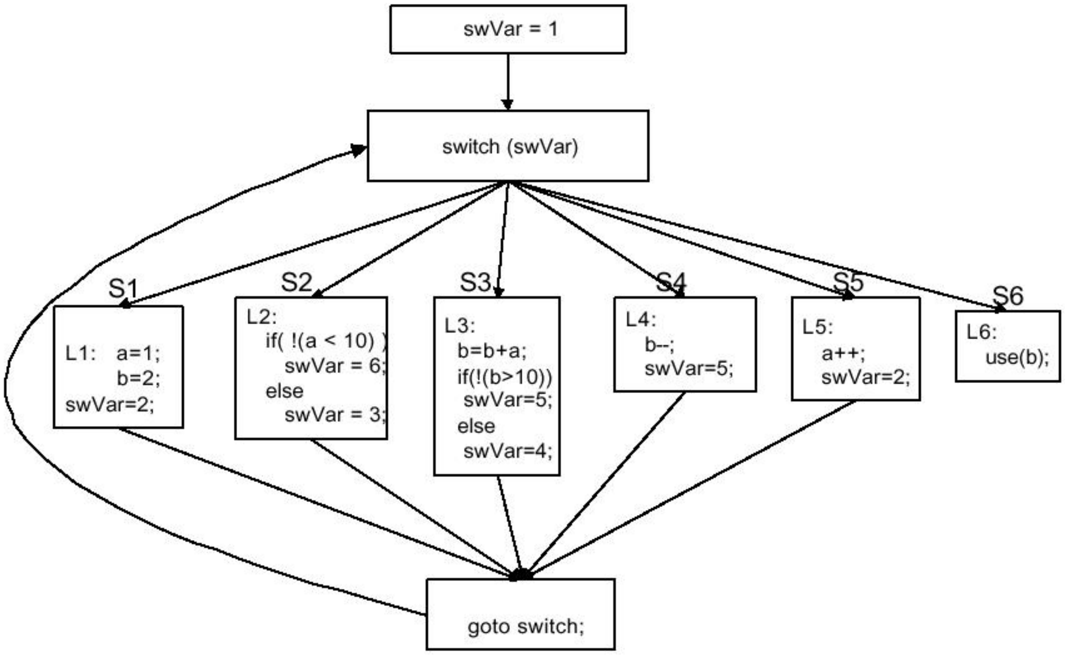


Figure 2.2: Example of a control flow graph flattening in Chenxi Wang's algorithm of obfuscation [72].

point of view, while technical details are not available. Although there are many obfuscators to choose from and competition is strong, there are no solutions going beyond general schema. Typical techniques in these schema are: obfuscation of identifiers, injection of constructions not present in Java language (ex. loops with `goto`), simple reordering of program structure and addition of some redundant code. Most application take not obfuscated executable files for Java Virtual Machine as input. The files are first decompiled, which makes impossible obfuscation of already obfuscated code. Typical examples of programs obfuscating Java language code are products: Dash-O Pro from *preEmptive solutions*[4] and RetroGuard from *Retrologic*[5].

Another examples can be taken from environment of enthusiasts of source code obfuscation of C programming language (figure 2.4)[6]. Used techniques could be named in Collberg's classification as program layout obfuscation. To make programs obfuscated in such a way they use automatic tools and "manual" transformations. Yet in practice this technique given very weak protection, because there are lot of programs formatting source codes in the automatic way. They ofter remove completely all efects of obfuscation. Even additional and redundat code can be removed automatically or "manualy" after the formating, because it becomes highly visible.

[4] `http://www.preemptive.com`

[5] `http://www.retrologic.com`

[6] Lot of examples can be found on pages `http://www.ioccc.org`.

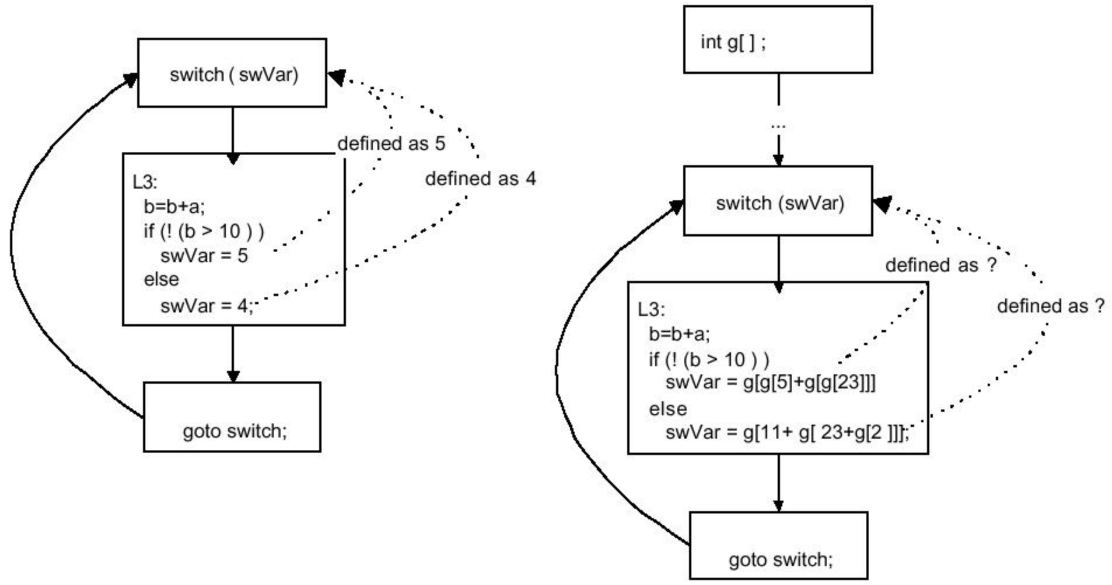
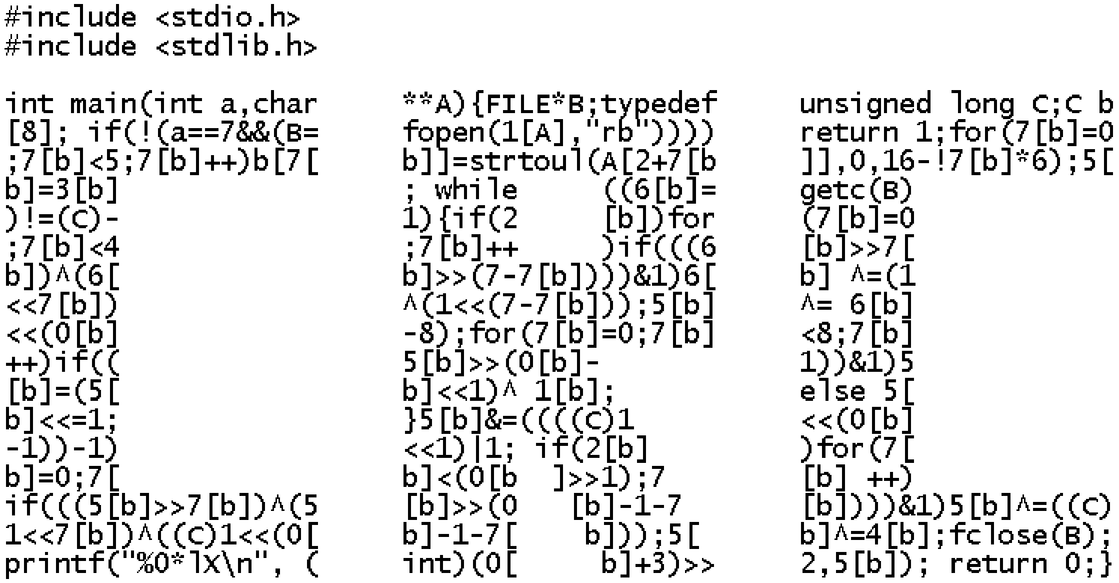


Figure 2.3: Example of adding of data aliases into flattened control flow graph in Chenxi Wang's algorithm of obfuscation [72].



```
#include <stdio.h>
#include <stdlib.h>

int main(int a,char     **A){FILE*B;typedef     unsigned long C;C b
[8]; if(!(a==7&&(B=     fopen(1[A],"rb")))      return 1;for(7[b]=0
;7[b]<5;7[b]++)b[7[     b]]=strtoul(A[2+7[b     ]],0,16-!7[b]*6);5[
b]=3[b]                 ; while     ((6[b]=     getc(B)
)!=(C)-                 1){if(2     [b])for     (7[b]=0
;7[b]<4                 ;7[b]++     )if(((6     [b]>>7[
b])^(6[                 b]>>(7-7[b])))&1)6[     b] ^=(1
<<7[b])                 ^(1<<(7-7[b]));5[b]     ^= 6[b]
<<(0[b]                 -8);for(7[b]=0;7[b]     <8;7[b]
++)if((                 5[b]>>(0[b]-            1))&1)5
[b]=(5[                 b]<<1)^ 1[b];           else 5[
b]<<=1;                 }5[b]&=((((C)1          <<(0[b]
-1))-1)                 <<1)|1; if(2[b]         )for(7[
b]=0;7[                 b]<(0[b  ]>>1);7        [b] ++)
if(((5[b]>>7[b])^(5      [b]>>(0   [b]-1-7      [b]))&1)5[b]^=((C)
1<<7[b])^((C)1<<(0[     b]-1-7[    b]));5[      b]^=4[b];fclose(B);
printf("%0*lX\n", (     int)(0[     b]+3)>>     2,5[b]); return 0;}
```

Figure 2.4: Example of source code obfuscation in C programming langugage.

## 2.4 How to Unobfuscate?

When we talk about obfuscation, it is natural to ask: is it possible to unobfuscate an obfuscated program? The answer is positive: in most cases reversible process can be performed, taking the obfuscated program and returning original program. Yet there is no easy answer to question: how to unobfuscate? It can be observed, that obfuscation is a contrary process to code optimization. Another area close to unobfuscation is decompilation of programs. Either optimization of compiled code ([2], [61]) or decompilation of machine code to high-level languages representation ([14], [16], [17], [19], [18], [21], [22], [69]), are covered very wide in scientific literature.

Sample classification of methods used in unobfuscation of code was presented in [25]. Using own experience and results of current state-of-the-art research authors selected following methods:

- identifying and evaluating opaque constructs[7] – allows to detect and remove inserted opaque constructs ([32])
- identification by pattern matching – comparison of fragments with database of patterns to detect inserted parts
- identification by program slicing – is one of the classic methods of decompilation, can detect unimportant fragments of code ([18], [69])
- statistical analysis – analysis of partial results in code extracted during program execution ([73])
- evaluation by data flow analysis – classic method of optimization, allows to bind separated fragments of original program and remove inserted code ([8], [17], [19], [40])
- evaluation by theorem proving – allows to obtain result of program without its execution, useful only for simple constructions ([47])

It can be seen now, that there are lot of well described techniques, efficiently unobfuscating program code. The only problem which arises very often is very high computational complexity unobfuscating algorithms (with no chances for simplification). This weak place is used during obfuscation and only using such property can guarantee an efficient protection against unauthorized analysis of obfuscated program code.

## 2.5 Other Problems Connected with Obfuscation

There are three important problems of software theory and practice, strongly connected with obfuscation through common goals, methods and procedures:

1. Software protection – code obfuscation techniques has its place in general theory of software protection increasing securing of a intelectual property ([15], [36]).
2. Software watermarking – is a technique which allows to indentify a program (thanks to introduced code – markers), even after some modifications of its contents ([28], [29], [30]).
3. Complexity measurements of software – are part of theoretic and practic quality research of programs; are used as basic tool in tests of quality of obfuscation algortihms ([1], [7], [13], [31], [39], [54], [56]).

[7] **opaque construct** – a program construction, which is very difficult to static analysis (exponential or higher computational complexity)

In case of software protection usually (ex. [72]) following scenarios of attacks are distinguished:

- denial-of-service of program – most often by giving some tampered data as input, causing program to misbehave
- program tampering – introduction of changes in program contents, causing change of behavior, required by the attacker
- impersonation during authentification – giving impersonated data during authorization, impossible to verify by program

Program obfuscation is one of methods to protect against unauthorized modification of its contents. Main advantage of obfuscation lays in its resistance to impersonation and denial-of-service methods.

Obfuscation can be also a method of insertion of identifiers – watermarkers. Obfuscated program has properties of encrypted message with elements of error correction. Lot of changes are needed to make obfuscated program unidentifiable.

Comparison of programs is the very old and well documented problem. It arose as a need of evaluation of software engineers work ([31]). Similarly obfuscating algorithms can be evaluated. In the same way like engineer, an algorithm gets defined task and produces executable program. Used methods of complexity measurement for generated programs will be analogous to methods used in case of software engineers.

# Section 3

# Theoretical Background

TO construct and describe the general algorithm of code obfuscation we created a simple formal system defining in a clear way such elements of present computers like instruction or program. The system uses classic mathematical background: sets, variables, functions, Cartesian product, vectors, spaces, etc.

In the table 3.1 a short description of used notation is shown. In general the notation follows common standards. We assume, that indexes of vectors' elements (small letters $i$ or $k$) denote dimension in an $N$-dimensional space (take values from the set $\{1, 2, ..., N\}$). Special attention should be paid to writing $\mathbf{v}_1 \overset{!i}{=} \mathbf{v}_2$, which is the short form of expression $\bigwedge_{k \neq i} v_1^k = v_2^k \ \wedge \ v_1^i \neq v_2^i$.

Table 3.1: Description of used notation.

| Notation | Example | Description |
|---|---|---|
| small letter | $v$ | dimension $v$, element of vector $\mathbf{v}$ |
| capital letter | $W$ | set of word's values $W$ |
| small bold letter | $\mathbf{v}$ | vector $\mathbf{v} \in \mathbf{S}$ of elements from sets $W_i$ |
| capital bold letter | $\mathbf{S}$ | set of vectors $\mathbf{S} = W_1 \times W_2 \times ... \times W_N$ |
| small letter + indexes | $v_j^i$ | element $i$ of vector $\mathbf{v}_j \in \mathbf{S}$ (dimension) |
| $\times$ | $W_1 \times W_2$ | operator of the Cartesian product |
| $\times_{i=1}^{N}$ | $\times_{i=1}^{N} V_i$ | Cartesian product of sets: $V_1 \times V_2 \times ... \times V_N$ |
| $\alpha$ | $\alpha$ | value describing unimportant element of vector $\mathbf{v}$ |
| $\bar{\alpha}$ | $\bar{\alpha}$ | vector of values $\alpha$ |
| capital letter + (...) | $I(\mathbf{v})$ | vector function executing instruction $I$ |
| capital letters + indexes | $I_1 \vert I_2$ | sequence of instructions $I_1$, $I_2$; composition $I_2(I_1(\mathbf{v}))$ |
| capital letter $\mathcal{N}$ | $\mathcal{N}$ | set of natural numbers |
| funkcion + upper index | $I(\mathbf{v})^i$ | element $i$ of vector $I(\mathbf{v})$ |
| equality + index | $\mathbf{v}_1 \overset{!i}{=} \mathbf{v}_2$ | equality of $\mathbf{v}_1$ and $\mathbf{v}_2$, except for element $i$ |

## 3.1 Basic definitions

In our analysis we will consider the class of machines described in Treleaven's classification by the abbreviation COSH (*control driven, shared data*), which is a generalization of classical von Neumann model. According to this model computer consists of the following elements:

- memory – ordered set of words holding information, in general these are locations (cells) of the memory and registers of processor
- processor – a device transforming information by execution of instructions
- bus (unimportant in our analysis)

State of memory can be changed only by processor executing an instruction. The possible ways of execution are described by set of instructions **I** where every instruction has assigned its object of execution. Such description of computer is called instruction set architecture (ISA).

**Definition 3.1** *Architecture of computer $\mathcal{A}(\mathbf{I}, \mathbf{S})$ we call a set of instructions $\mathbf{I} = \{I_1, I_2, ..., I_M\}$, which is set of functions $I_i : \mathbf{S} \rightarrow S'_i$, where:*

$$\mathbf{S} = \{(v^1, v^2, ..., v^N) : v^1 \in W_1, v^2 \in W_2, ..., v^N \in W_N\} = W_1 \times W_2 \times ... \times W_N$$

*is space of vectors, created from all possible values of dimensions (words) $v^1$, $v^2$, ..., $v^N$, and $S'_i \subset \mathbf{S}$ is anti-domain of function $I_i$.*[8]

Set **S** is called *space of states* of given computer. State of an architecture can be for an example a configuration of internal devices storing information. Each instruction has in addition a property of control forwarding: instruction decides, which instruction must be executed next. In present computers instructions change usually only small number of dimensions (elements of vectors) in the space of states, mapping most of them one-to-one. Given an instruction we can determine changed dimensions, called *output context* and changing dimensions, called *input context.*

**Definition 3.2** *Input context $S_I$ is a set of vectors, consisted of values of all dimensions used by given function $I$ (having influence on the function):*

$$S_I = \{\times_{i=1}^{N} V_i \ \ \textit{such that} \ \ V_i = W_i \ \textit{if} \bigvee_{\substack{\mathbf{v}_1 \overset{!i}{=} \mathbf{v}_2 \in \mathbf{S} \\ k \neq i}} (I(\mathbf{v}_1)^k \neq I(\mathbf{v}_2)^k \quad \vee \quad I(\mathbf{v}_1)^i \neq v_1^i)$$
$$\textit{otherwise } V_i = \{\alpha\}\}$$

*where symbol $\alpha$ is an unique value (not present in sets $W_1, W_2, ..., W_N$) meaning a word not important for given function.*

**Definition 3.3** *Output context $S_O$ is a set of vectors, consisted of values of all dimensions changed by given funcion $I$:*

$$S_O = \{\times_{i=1}^{N} V_i \ \ \textit{such that} \ \ V_i = W_i \ \textit{if} \bigvee_{\mathbf{v}_1, \mathbf{v}_2 \in \mathbf{S}} (I(\mathbf{v}_1) = \mathbf{v}_2 \quad \wedge \quad v_1^i \neq v_2^i)$$
$$\textit{otherwise } V_i = \{\alpha\}\}$$

It can be seen, that space $S_O$ contains only dimensions, which are not mapped by identity mapping (change of particular element of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$).

[8] In real computers in it often true, that $S_i^{'} = \mathbf{S}$.

**Example 3.1** *Let us consider a simplest computer with space of states* $\mathbf{S} = W_1 \times W_2 \times W_3$*, where* $W_1 = W_2 = W_3 = \mathcal{N}$ *and one hypothetical instruction, adding two natural numbers from sets* $W_1$ *and* $W_2$*, according to mapping:*

$$I : (v^1, v^2, v^3) \rightarrow (v^1 + v^2, v^2, v^3)$$

*for* $v^1 \in W_1, v^2 \in W_2, v^3 \in W_3$*. It can be easily seen, that for given instruction*

$$S_I = W_1 \times W_2 \times \{\alpha\}$$

*because only two dimensions have influence on the final results, and*

$$S_O = W_1 \times \{\alpha\} \times \{\alpha\}$$

*because only the first dimension holds the final result.*

Domain of every instruction contains anti-domain of any instruction, that's why we can consider sequences of instructions, thus define a *program.*

**Definition 3.4** *Program* $P$ *is an instruction or a sequence: (instruction, program) or a sequence: (program, instruction):*

$$P = I \quad \vee \quad I|P \quad \vee \quad P|I$$

Above definition describes either static or dynamic program (a process). Further only static programs will be analyzed, which define not the sequence of execution, but a way of context processing. This way does not have to be equivalent to the execution process, because some instructions can be executed many times and some may not be executed at all. Executed sequence is defined as dynamic program. We assume that all analyzed programs satisfy the stop condition, i.e. they produce assigned result in the finite time.

It is derived from definition 3.4, that program is a sequential composition of instructions. Domain and anti-domain of program satisfy conditions of domain and anti-domain of instruction (given in definition 3.1), because anti-domain of every instruction is included in domain – space of states of given machine. Thus definitions 3.2 and 3.3 can be applied to all possible programs.

## 3.2 Operations on Contexts

Composition of instructions is a function either, so for a program we can determine input and output context as well, using given definitions. We can also combine contexts of instructions included in program, obtaining in the final result input and output context of the whole program. Combining we use operation being a specific sum of contexts, defined as follows:

**Definition 3.5** *Sum of contexts* $S_1$ *and* $S_2$ *is a set of vectors consisted of elements of vectors from space* $S_1$ *or* $S_2$ *not being the value* $\alpha$*. For given contexts* $S_1$ *and* $S_2$ *the sum of contexts (noted by the operator* $\Cup$*) is calculated from the formula:*

$$S_1 \Cup S_2 = \{\times_{i=1}^{N} V_i \quad \textit{such that} \quad V_i = W_i \quad \textit{if} \quad \bigwedge_{\substack{\mathbf{v}_1 \in S_1 \\ \mathbf{v}_2 \in S_2}} (v_1^i \in W_i \quad \vee \quad v_2^i \in W_i)$$
$$\textit{otherwise } V_i = \{\alpha\}\}$$

Similarly we define a specific product of contexts:

**Definition 3.6** *Product of contexts $S_1$ and $S_2$ is a set of vectors consisted of elements of vectors from space $S_1$ or $S_2$ not being the value $\alpha$ neither in $S_1$ nor in $S_2$. For given contexts $S_1$ and $S_2$ the product of contexts (noted by operator $\Cap$) is calculated from the formula:*

$$S_1 \Cap S_2 = \{\times_{i=1}^{N} V_i \quad \textit{such that} \quad V_i = W_i \quad \textit{if} \quad \bigwedge_{\substack{\mathbf{v}_1 \in S_1 \\ \mathbf{v}_2 \in S_2}} (v_1^i \in W_i \quad \wedge \quad v_2^i \in W_i)$$
$$\textit{otherwise } V_i = \{\alpha\}\}$$

Using the product of contexts we define relation of inclusion for contexts:

**Definition 3.7** *Context $S_1$ is included in context $S_2$, if all elements of its vectors, which are not the value $\alpha$ in $S_1$, are not also the value $\alpha$ in $S_2$. Relation of inclusion is noted with use of operator $\Subset$:*

$$S_1 \Subset S_2 \Leftrightarrow S_1 \Cap S_2 = S_1$$

Next used operation is a specfic difference of contexts:

**Definition 3.8** *Difference of contexts $S_1$ and $S_2$ is a set of vectors consisted of only these elements of vectors from space $S_1$, which are in space $S_2$ the value $\alpha$. For given contexts $S_1$ and $S_2$ difference of contexts (noted by operator $\ominus$) is calculated from the formula:*

$$S_1 \ominus S_2 = \{\times_{i=1}^{N} V_i \quad \textit{such that} \quad V_i = W_i \quad \textit{if} \quad \bigwedge_{\substack{\mathbf{v}_1 \in S_1 \\ \mathbf{v}_2 \in S_2}} (v_1^i \neq \alpha \quad \wedge \quad v_2^i = \alpha)$$
$$\textit{otherwise } V_i = \{\alpha\}\}$$

**Theorem 3.1** *Given a program $P = I_1|I_2$ and instructions' input contexts: $S_{I1}$, $S_{I2}$ and output contexts: $S_{O1}$, $S_{O2}$, the input context $S_{IP}$ of the program $P$ can be calculated from the formula:*

$$S_{IP} = (S_{I2} \ominus S_{O1}) \Cup S_{I1}$$

**Proof.** Using definition of input context (definition 3.2) given for the program $P$:

$$S_{IP} = \{\times_{i=1}^{N} V_i \quad \text{such that} \quad V_i = W_i \quad \text{if} \quad \bigvee_{\substack{\mathbf{v}_1 \overset{!i}{=} \mathbf{v}_2 \in \mathbf{S} \\ k \neq i}} (P(\mathbf{v}_1)^k \neq P(\mathbf{v}_2)^k \quad \vee \quad P(\mathbf{v}_1)^i \neq v_1^i)$$
$$\text{otherwise } V_i = \{\alpha\}\}$$

let us consider all possible cases of combination of input contexts, knowing that $P = I_1|I_2$ (table 3.2).

According to definitions 3.2 and 3.3 given element of input or output context may belong either to set $W_i$ or set $\{\alpha\}$. Analyzing dependencies of $S_{IP}$ from contexts $S_{I1}$, $S_{I2}$, $S_{O1}$, $S_{O2}$ we find, that context $S_{IP}$ depends only on contexts $S_{I1}$, $S_{O1}$ and $S_{I2}$. For given elements of these three contexts values of element of context $S_{IP}$ can be obtained from definition: element is used by the program $P$ (is different from the value $\alpha$), if transforming a vector from $\mathbf{S}$ changes at least one element. Taking into account, that the program $P$ is composition of instructions $I_1$ and $I_2$ we may expand, that an element is used by the program $P$, if is used by the instruction

Table 3.2: The truth-table for combination of input contexts.

| $S_{I1}$ | $S_{O1}$ | $S_{I2}$ | $S_{IP}$ |
|---|---|---|---|
| $\alpha$ | $\alpha$ | $\alpha$ | $\alpha$ |
| $\alpha$ | $\alpha$ | $W_i$ | $W_i$ |
| $\alpha$ | $W_i$ | $\alpha$ | $\alpha$ |
| $\alpha$ | $W_i$ | $W_i$ | $\alpha$ |
| $W_i$ | $\alpha$ | $\alpha$ | $W_i$ |
| $W_i$ | $\alpha$ | $W_i$ | $W_i$ |
| $W_i$ | $W_i$ | $\alpha$ | $W_i$ |
| $W_i$ | $W_i$ | $W_i$ | $W_i$ |

$I_1$ or is used by the instruction $I_2$, but was not changed by instruction $I_1$. The change of an element in instruction $I_1$ can be detected as appropriate value in context $S_{O1}$ being different from $\alpha$.

According to definitions 3.5 and 3.8 we may state, that formula:

$$S_{IP} = (S_{I2} \ominus S_{O1}) \Cup S_{I1}$$

represents mapping shown in the table 3.2, thus it calculates input context of the program $P$. ■

**Theorem 3.2** *Given a program $P = I_1|I_2$ and instructions' output contexts: $S_{O1}$, $S_{O2}$, the output context $S_{OP}$ of the program $P$:*

***a)*** *if after composition of instructions $I_1$ and $I_2$ none of elements from the domain of the program $P$ is mapped by indentity mapping and it was not mapped by identity mapping by both instructions, i.e.:*[9]

$$\bigwedge_{i=1,2,\ldots,N} \Big( \bigvee_{\mathbf{v}_1,\mathbf{v}_2 \in \mathbf{S}} ((I_1(\mathbf{v}_1) = \mathbf{v}_2 \quad \vee \quad I_2(\mathbf{v}_1) = \mathbf{v}_2) \quad \wedge \quad v_1^i \neq v_2^i) \Rightarrow \\ \Rightarrow \bigvee_{\mathbf{v}_3,\mathbf{v}_4 \in \mathbf{S}} (P(\mathbf{v}_3) = \mathbf{v}_4 \quad \wedge \quad v_3^i \neq v_4^i)\Big) \tag{3.1}$$

*is the sum of contexts $S_{O1}$ and $S_{O2}$:*

$$S_{OP} = S_{O1} \Cup S_{O2}$$

***b)*** *otherwise must be calculated directly from the definition 3.3, but the following property is true:*

$$S_{OP} \Subset (S_{O1} \Cup S_{O2})$$

[9] The condition means, that after combination the given program or its part "do nothing".

**Proof.**

**a)** Let us write definition 3.3 for the program $P = I_1|I_2$:

$$S_{OP} = \{\times_{i=1}^{N} V_i \text{ such that } V_i = W_i \text{ if } \bigvee_{\mathbf{v}_1,\mathbf{v}_2 \in \mathbf{S}} (I_1|I_2(\mathbf{v}_1) = \mathbf{v}_2 \ \wedge \ v_1^i \neq v_2^i) \\ \text{otherwise } V_i = \{\alpha\}\}$$

expanding composition of instructions on two operations:

$$S_{OP} = \{\times_{i=1}^{N} V_i \text{ such that } V_i = W_i \text{ if } \bigvee_{\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3 \in \mathbf{S}} (I_1(\mathbf{v}_1) = \mathbf{v}_3 \ \wedge \ I_2(\mathbf{v}_3) = \mathbf{v}_2 \\ \wedge \ v_1^i \neq v_2^i) \\ \text{otherwise } V_i = \{\alpha\}\} \tag{3.2}$$

If the assumption (3.1) is true, it is also true, that:

$$(v_1^i \neq v_3^i \ \wedge \ v_3^i \neq v_2^i) \Rightarrow v_1^i \neq v_2^i \tag{3.3}$$

thus the equivalence is true:

$$v_1^i \neq v_2^i \Leftrightarrow (v_1^i \neq v_3^i \ \vee \ v_3^i \neq v_2^i) \tag{3.4}$$

Inserting (3.4) into (3.2) we obtain:

$$S_{OP} = \{\times_{i=1}^{N} V_i \text{ such that } V_i = W_i \text{ if } \bigvee_{\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3 \in \mathbf{S}} (I_1(\mathbf{v}_1) = \mathbf{v}_3 \ \wedge \ I_2(\mathbf{v}_3) = \mathbf{v}_2 \ \wedge \\ (v_1^i \neq v_3^i \ \vee \ v_3^i \neq v_2^i)) \\ \text{otherwise } V_i = \{\alpha\}\}$$

and after some simple logic transformations:

$$S_{OP} = \{\times_{i=1}^{N} V_i \text{ such that } V_i = W_i \text{ if } \bigvee_{\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3 \in \mathbf{S}} ((I_1(\mathbf{v}_1) = \mathbf{v}_3 \ \wedge \ v_1^i \neq v_3^i) \\ \vee \ (I_2(\mathbf{v}_3) = \mathbf{v}_2 \ \wedge \ v_3^i \neq v_2^i)) \\ \text{otherwise } V_i = \{\alpha\}\} \tag{3.5}$$

Expressions $\bigvee_{\mathbf{v}_1,\mathbf{v}_3 \in \mathbf{S}} (I_1(\mathbf{v}_1) = \mathbf{v}_3 \ \wedge \ v_1^i \neq v_3^i)$ and $\bigvee_{\mathbf{v}_2,\mathbf{v}_3 \in \mathbf{S}} ((I_2(\mathbf{v}_3) = \mathbf{v}_2 \ \wedge \ v_3^i \neq v_2^i)$ describe full[10] dimensions of output contexts $S_{O1}$ and $S_{O2}$ (according to definition 3.3). We may write then:

$$\bigvee_{\mathbf{v}_1,\mathbf{v}_3 \in \mathbf{S}} (I_1(\mathbf{v}_1) = \mathbf{v}_3 \ \wedge \ v_1^i \neq v_3^i) \Rightarrow \bigwedge_{\mathbf{v}_1 \in \mathbf{S}_{O1}} v_1^i \in W_i$$

and

$$\bigvee_{\mathbf{v}_2,\mathbf{v}_3 \in \mathbf{S}} ((I_2(\mathbf{v}_3) = \mathbf{v}_2 \ \wedge \ v_3^i \neq v_2^i) \Rightarrow \bigwedge_{\mathbf{v}_2 \in \mathbf{S}_{O2}} v_2^i \in W_i$$

[10] Not containing the value $\alpha$ only.

and next use above statements, inserting them into (3.5):

$$S_{OP} = \{\times_{i=1}^{N} V_i \;\text{ such that }\; V_i = W_i \;\text{ if }\; \bigwedge_{\substack{\mathbf{v}_1 \in S_{O1} \\ \mathbf{v}_2 \in S_{O2}}} (v_1^i \in W_i \quad \vee \quad v_2^i \in W_i)$$
$$\text{otherwise } V_i = \{\alpha\}\}$$

obtaining as the final result definition 3.5 given for the sum of contexts $S_{O1}$, $S_{O2}$:

$$S_{OP} = S_{O1} \Cup S_{O2}$$

**b)** If assumption (3.1) is not true, then expression (3.3) is false, thus equivalence (3.4) is also false and we cannot finish the proof like in the case (**a**), that's why statement (3.2) cannot be simplified to the sum of output contexts and context $S_{OP}$ must be calculated directly from definition.
Yet we observe, that equivalence (3.4) is responsible for "admission" of an $i$ element into context $S_{OP}$, according to formula (3.2). Thus ignoring the equivalence we can only add some additional elements into the sum $S_{O1} \Cup S_{O2}$, comparing with the correct context $S_{OP}$. All elements not being the value $\alpha$ in $S_{OP}$, will not be $\alpha$ in $S_{O1} \Cup S_{O2}$ as well. So it must be true, that:

$$S_{OP} \Subset (S_{O1} \Cup S_{O2}) \tag{3.6}$$

■

The case describe by relation (3.6) occurs, when second part of combined program "cancels" all operations executed in the first part on an element of context. Such behaviour would be strange in the obfuscated program and even if present, then for small number of "canceled" elements, according to proof of the case **(b)**, we can apply formula using sum of contexts without bigger consequences for the obfuscation process.

**Example 3.2** *An architecture of computer is given with the space of states* $\mathbf{S} = W_1 \times W_2 \times W_3 \times W_4$*, where* $W_1 = W_2 = W_3 = W_4 = \mathcal{N}$*, and two instructions adding two natural numbers, according to mapping:*

$$I_1 : (v^1, v^2, v^3, v^4) \rightarrow (v^1, v^2, v^1 + v^2, v^4)$$

*and*

$$I_2 : (v^1, v^2, v^3, v^4) \rightarrow (v^1, v^3 + v^4, v^3, v^4)$$

*where* $v^1 \in W_1, v^2 \in W_2, v^3 \in W_3, v^4 \in W_4$*. Input contexts of instructions are equal to:*

$$S_{I1} = W_1 \times W_2 \times \{\alpha\} \times \{\alpha\}$$
$$S_{I2} = \{\alpha\} \times \{\alpha\} \times W_3 \times W_4$$

*and output contexts:*

$$S_{O1} = \{\alpha\} \times \{\alpha\} \times W_3 \times \{\alpha\}$$
$$S_{O2} = \{\alpha\} \times W_2 \times \{\alpha\} \times \{\alpha\}$$

*A program $P = I_1|I_2$ was created. According to theorem 3.1 input context of the program $P$ has form:*

$$S_{IP} = W_1 \times W_2 \times \{\alpha\} \times W_4$$

*and input context of the program $P$ (according to theorem 3.2):*

$$S_{OP} = \{\alpha\} \times W_2 \times W_3 \times \{\alpha\}$$

*so the program $P$ uses elements $v^1$, $v^2$, $v^4$ of every vector $\mathbf{v} \in \mathbf{S}$ and changes elements $v^2$ and $v^3$.*

As it can be seen from the example, summing of output contexts leads to growth of programs' output context, which for longer programs makes their output context equal to the analyzed subspace of given computer's space of states. Yet in practice input and output contexts are given together with a program. A program does not contain information, which dimensions of the output context are real results of calculations (used by an user) and which are just some temporary calculations. In the example 3.2 the program $P$ adds three natural numbers. On the end we get also a temporary result: sum of two natural numbers. Thus giving the output context *a priori* is justified by logic analysis and practice.[11]

It should be also noticed, that output context given with program does not have to be equal to the output context calculated from the definition 3.2 (example 3.3). Typically present programs generate a lot of temporary calculations, which are unimportant ont the end of program's run.

## 3.3 Instructions and Operations

On the begining of the obfuscation process a program is given $P(\mathbf{v}_1) = \mathbf{v}_2$ and its input $S_I$ and output $S_O$ context. In a typical present computer the space of states $\mathbf{S}$ has milions or even bilions of dimensions, that's why for obfuscation we choose a subspace $S'$ of the space $\mathbf{S}$ (it can be for an example set of processor's registers only). To define input and output contexts in an easy way in the limited subspace $S'$, we introduce the following notation:

$$S_{IO} = \{(a, b, d)\}$$

where $S_{IO}$ is the defined space, a context of program for the machine defined by subspace $S' = \{(a, b, c, d, e) : a \in A, b \in B, c \in C, d \in D, e \in E\}$. Above notation is a short version of the form $S_{IO} = \{(a, b, \alpha, d, \alpha) : a \in A, b \in B, d \in D\}$. Usually even subspace $S'$ has a lot of dimensions, while analyzed subspace is a small fragment only. Not writing the unimportant dimensions in input and output contexts simplifies the formal analysis and it is some kind of analogy to the idea of vector subspace.[12]

**Example 3.3** *Given a machine program written in the assembler of Motorola 68000 processor and input context $S_I = \{(d0, d1, d2, d3, d4)\}$ and output context $S_O = \{(d2, d4, d5)\}$:*

[11] An interested analogy is here the wave function in quantum mechanics. Without an observer (user) examined process described by a given wave function is not determined, similarly like result of program's calculations.

[12] Subspace $\{(x, y, 0)\}$ of the vector space $\{(x, y, z)\}$, $x, y, z \in \mathbf{R}$ is a vector subspace, while the subspace $\{(x, y, 5)\}$ is not.

```
move.w  d0,d5      ; d5 = d0
add.w   d1,d5      ; d5 = d5 + d1
add.w   d0,d1      ; d1 = d1 + d0
sub.w   d3,d5      ; d5 = d5 - d3
sub.w   d1,d2      ; d2 = d2 - d1
```

Using only definition the output context of the program in the example would have had form $S'_O = \{(d1, d2, d5)\}$. Absence of the element $d1$ in $S_O$ means, that result of calculations included in this element is not important. On the other hand presence of the element $d4$ in the input and output context means, that it is important for some further analysis.

Given a program $P(\mathbf{v}_1) = \mathbf{v}_2$ without given input and output context, we assume, that there exists an input contest $S_1$ and output context $S_2$. If a program is given and a context (without additional "input" or "output"), then we assume that input and output contexts are equal (the same space).

For every program $P(\mathbf{v}_1) = \mathbf{v}_2$, given its input and output context, it is possible, using theorem 3.2, to determine output context after every instruction of the program ([61]).

To satisfy needs which appeared during construction of the general algorithm of code obfuscation, we classified all instructions according to their processing specificity:

- operations – instructions which do not change sequence of program execution and do not use elements of machine context, which have:
    - conncetion to an environment outside the architecture
    - ability of changing execution of other operations (crossing natural input context for given operation, ex. for addition variables holding numbers to add are such part of context)
- branches – instructions changing sequence of execution
- special – instructions, which do not qualify to above groups

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 point of return

- special
  - changing execution state (ex. RESET, HALT)
  - input/output services

There are also two important groups of operations:

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

  $$I(\mathbf{v}_1) = \mathbf{v}_2 \text{ is reversible} \Longleftrightarrow \bigvee_{P_I} \bigwedge_{\mathbf{v}_O \in S_O} \bigvee_{\mathbf{v}_I \in S_I} P_I(\mathbf{v}_O) = \mathbf{v}_I$$

  where $S_I$ is input context for $I$, output context for $P_I$ and $S_O$ is output context for $I$, input context for $P_I$; above condition means, that the function realizing instruction $I$ is a one-to-one mapping
- irreversible operations – otherwise

which could be also mapped onto assembler instructions. In real processors some intructions cannot be classified as reversible or irreversible, because reversibility depends on the current context.

Assuming that currently most popular machines are two-address kind and that in three-address machines two-address instructions are used most often (see appendix C), for practice reasons we introduced following classification (reversibility should be considered in context consisted of arguments of instructions, without a register containing onverflow or carry flags):

- reversible operations
  - arithmetic (multiplication and division with additional conditions)
  - logical of type EX-OR, EX-NOR and NOT
  - rotations
  - exchange of value
- irreversible operations
  - addition with carrying; subtraction with borrowing
  - copying (with exception of exchanging values)
  - logical AND, OR
  - shifts

Above classification will have an additional justification and application when we present the general algorithm of obfuscation.

A sequence of instruction can be treated as reversible or irreversible too. In some exceptional cases, using appropriate context, a sequence of irreversible operations can be reversible. Although the concept of reversible sequence of instructions is more general than concept of reversible operation, reversible sequences containing irreversible operations occurr very rare in practical applications and reversibility of such sequences is created mainly by an extra copy of one of input arguments. Because of this we will consider only reversible operations and their interactions with programs.

**Example 3.4** *Given machine program written in assembler of microprocessor Intel 80386:*

```
stc             ; C = 1
adc  eax,ebx    ; eax = eax + ebx + C
```

is reversible in context $\{(eax, ebx)\}$, although both operations are irreversible in their contexts: first – $\{(C)\}$, second – $\{(eax, ebx, C)\}$. Reversibility of such program comes from non-used part of context, in the example it is carry flag $C$.

**Example 3.5** *Given machine program written in assembler of microprocessor Intel 80386. Every instruction was classified as follows:*

```
        mov   ecx,10     ; ecx = 10, irreversible
  _loop:
        lodsd            ; eax = [esi], irreversible
        add   edx,eax    ; edx = edx + eax, reversible in context {(eax, edx)}
        loop  _loop      ; branch
        in    al,dx      ; special, input/output
```

In further analysis of programs often used transformation will be separation or concatenation of programs:

**Definition 3.9** *Program $P$ is the concatenation of programs $P_1 = I_1|I_2|I_3|...|I_n$ and $P_2 = I_1^*|I_2^*|I_3^*|...|I_m^*$, when:*

$$P = P_1|P_2 = I_1|I_2|I_3|...|I_n|I_1^*|I_2^*|I_3^*|...|I_m^*$$

Given an *a priori* condition $P = P_1|P_2$ it is understood, that program $P$ was split in any place on two programs $P_1$ and $P_2$.

## 3.4 Equivalence of Programs

A natural definition of programs' equivalence, based on the idea of machine's state, would not include contexts of programs, so its applications could be very limited. Introducing input and output contexts we obtained a little bit more complicated definition of programs' equivalence.

**Definition 3.10** *Program $P_1$ is equivalent to program $P_2$ in the input context $S_I$, output context $S_O$, if both programs map the same dimensions not being the value $\alpha$ in $S_I$, on dimensions not being value $\alpha$ in $S_O$. To write the definition using symbols we define two auxiliary sets of indexes' values:*

$$J = \{x : \bigvee_{\mathbf{v}^x \in S_I} \mathbf{v}^x \neq \alpha\}$$

$$K = \{x : \bigvee_{\mathbf{v}^x \in S_O} \mathbf{v}^x \neq \alpha\}$$

*Programs $P_1$, $P_2$ are equivalent, if:*

$$\bigwedge_{\mathbf{v}_1, \mathbf{v}_2 \in \mathbf{S}} (\bigwedge_{j \in J} v_1^j = v_2^j \Leftrightarrow \bigwedge_{k \in K} P_1(\mathbf{v}_1)^k = P_2(\mathbf{v}_2)^k)$$

Given definition satisfies condition of symmetry, but we must remember, that two programs does not have to be equivalent in their contexts (obatined from theorems 3.1, 3.2 or given *a* *priori*). Such approach seems to be justified in the case of computer programs, especially in the area of obfuscation. The definition guarantees, that program $P_1$ equivalent to program $P_2$ in contexts: input $S_I$, output $S_O$, realizes exatly the same mapping on the used dimensons of the input context. Additional operations executed by program $P_1$ are redundant from the program's $P_2$ point of view.

In the symbolic form we note equivalence of program $P_1$ to program $P_2$ in contexts: input $S_I$, output $S_O$, by:

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

The placement of arguments in the given relation remains important, because still its symmetry depends on given contexts.

**Lemma 3.1** *If* $P_3 \equiv P_1(\mathbf{v}) = \mathbf{v}_X$ *and* $P_4 \equiv P_2(\mathbf{v}_X) = \mathbf{v}'$*, then it is true, that:*

$$P_3|P_4 \equiv P_1|P_2(\mathbf{v}) = \mathbf{v}'$$

**Proof.** From the basic assumptions we get according to definition 3.10:

$$J = \{x : \bigvee_{\mathbf{v}^x \in S} \mathbf{v}^x \neq \alpha\}$$
$$K = \{x : \bigvee_{\mathbf{v}^x \in S_X} \mathbf{v}^x \neq \alpha\}$$
$$L = \{x : \bigvee_{\mathbf{v}^x \in S'} \mathbf{v}^x \neq \alpha\}$$

and

$$\bigwedge_{\mathbf{v}_1,\mathbf{v}_2 \in \mathbf{S}} ((\bigwedge_{j \in J} v_1^j = v_2^j \Leftrightarrow \bigwedge_{k \in K} P_3(\mathbf{v}_1)^k = P_1(\mathbf{v}_2)^k) \quad \wedge$$
$$(\bigwedge_{k \in K} v_1^k = v_2^k \Leftrightarrow \bigwedge_{l \in L} P_4(\mathbf{v}_1)^l = P_2(\mathbf{v}_2)^l))$$

thus:

$$\bigwedge_{\mathbf{v}_1,\mathbf{v}_2 \in \mathbf{S}} (\bigwedge_{j \in J} v_1^j = v_2^j \Leftrightarrow \bigwedge_{l \in L} P_3|P_4(\mathbf{v}_1)^l = P_1|P_2(\mathbf{v}_2)^l)$$

We obtained definition 3.10. After simplification we get:

$$P_3|P_4 \equiv P_1|P_2(\mathbf{v}) = \mathbf{v}'$$

■

## 3.5 Definition of the Obfuscation Process

Using described above basic definitions we created an alternative (in opposite to for an example [25]) definition of obfuscating transformation.

**Definition 3.11** *Obfuscating transformation $\mathcal{T}$ is such a change of program $P(\mathbf{v}) = \mathbf{v}'$ into program $\mathcal{T}(P)$, that there is program $P'$, restoring original output context of program $P$ and program $\mathcal{T}(P)|P'$ is equivalent to program $P$ in contexts: input $S$,output $S'$.*

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

Sequence of instructions $P'$, returning the condition of equivalence is called *completion of obfuscating transformation.* It is not possible to obfuscate all instructions of given architecture, because for some instructions context cannot be changed (like for input/output instructions). In general case it is hard to say if such a change of usage will be an reversible operation, providing correctness of transformation according to given definition.

Separation of the obfuscated program on two parts: $\mathcal{T}(P)$ and $P'$, allows to bind reversible operations to obfuscating transformations. Definition based on the assumption of equivalence of programs $P$ and $\mathcal{T}(P)$ would be less practical, because it would describe only enhancement of contexts in $\mathcal{T}(P)$ related to contexts from $P$.

The condition of equivalence $\mathcal{T}(P)|P' \equiv P(\mathbf{v}) = \mathbf{v}'$ implies necessity of keeping in programs $\mathcal{T}(P)(\mathbf{v}) = \mathbf{v}_{\mathcal{T}}$ and $P'(\mathbf{v}_{\mathcal{T}}) = \mathbf{v}'$ identity of mapping. It guarantees that if $\mathcal{T}(P)|P'(\mathbf{v}_1) = \mathbf{v}_2$, then $S \subset S_1$ and $S' \subset S_2$ and $\mathcal{T}(P)|P'(\mathbf{v}) = \mathbf{v}'$.

**Example 3.6** *Given sequence of instructions of processor Intel 80386 considered in the context $\{(eax, ebx, ecx, edx)\}$:*

```
mov  eax,[esi]    ; eax = [esi]
mov  edx,[edi]    ; edx = [edi]
add  eax,edx      ; eax = eax + edx
sub  ebx,edx      ; ebx = ebx - edx
```

was obfuscated with a transformation, which generated sequence:

```
xchg eax,ebx      ; eax <-> ebx
mov  ebx,[esi]    ; ebx = [esi]
mov  edx,[edi]    ; edx = [edi]
add  ebx,123      ; ebx = ebx + 123
xchg ecx,edx      ; ecx <-> edx
add  ebx,ecx      ; ebx = ebx + ecx
sub  edx,321      ; edx = edx - 321
sub  eax,ecx      ; eax = eax - ecx
```

The completion of this transformation is sequence:

```
add  edx,321     ; edx = edx + 321
xchg ecx,edx     ; ecx <-> edx
sub  ebx,123     ; ebx = ebx - 123
xchg eax,ebx     ; eax <-> ebx
```

which brings back original usage of context $\{(eax, ebx, ecx, edx)\}$.

## 3.6 Properties of Obfuscating Transformations

Obfuscating transformations meeting conditions of definition 3.11 have two interesting properties, which are backbone of contruction of general algorithm of code obfuscation.

**Theorem 3.3** *If $\mathcal{T}$ is an obfuscating transformation, then $\mathcal{T}(P)$ can contain only:*

- *programs equivalent to instructions from $P$, i.e. such instructions $P_I$, that if $I$ is a part of program $P$ and $I(\mathbf{v}_1) = \mathbf{v}_2$ then $P_I \equiv I(\mathbf{v}_1) = \mathbf{v}_2$*
- *reversible operations*
- *instructions, which do not change the output context in program $P$, i.e. such instructions $I(\mathbf{v}_{11}) = \mathbf{v}_{22}$, for which it is true, that for given beginning fragment of program $P$, $P'(\mathbf{v}_1) = \mathbf{v}'_2$, always $S_{22} \cap S'_2 = \{\mathbf{0}\}$*

**Proof.** (indirect) Let $I_P$ be sum of sets: type of instructions in program $P$, reversible operations and instructions, which do not change the output context in $P$, $I_N$ be the set of irreversible operations changing this context, and $I_M$ – set of all instructions of given machine. From the classification presented on the page 23 we obtain:

$$I_P \cup I_N = I_M$$

and

$$I_P \cap I_N = \varnothing$$

Contradiction of the thesis of theorem is:

$$\mathcal{T}(P) \text{ is obfuscating tr. of } P \Longrightarrow \bigvee_{I \in \mathcal{T}(P)} I \notin I_P$$

Also $I \in I_M$, because it must be an instruction of the given machine, which implies that:

$$I \in I_N$$

Using definition of reversible operation we can writen, that an instruction $I(\mathbf{v}_1) = \mathbf{v}_2$ is irreversible operation, when:

$$\bigvee_{\mathbf{v}_1, \mathbf{v}_{11} \in S_1} \bigvee_{\mathbf{v}_2 \in S_2} (I(\mathbf{v}_1) = \mathbf{v}_2 \quad \vee \quad I(\mathbf{v}_{11}) = \mathbf{v}_2)$$

This condition is contradictory with part of the definitions of equivalence of programs, so according to definitions 3.11 $\mathcal{T}$ is not an obfuscation transformation (construction of completion of $\mathcal{T}(P)$ becomes impossible). ■

Some of instructions of real computers, like „`adc eax,ebx`" in the example 3.4, can be recersible operation (`eax = eax + ebx + C`) and instruction extending a context of obfuscated program at the same time (using carry flag $C$ when the program context is $\{(eax, ebx)\}$).

In the proof of second property of obfuscating transformation the following lemma will be useful:

**Lemma 3.2** *For any pair of contexts $\mathbf{v}_1$, $\mathbf{v}_2$ of given machine, there is a program $P_X(\mathbf{v}_1) = \mathbf{v}_2$.*

**Proof.** It is known that $\mathbf{v}_1$ i $\mathbf{v}_2$ belong to the set $\mathbf{S}$, set of all possible states of the machine:

$$\mathbf{v}_1, \mathbf{v}_2 \in \mathbf{S}$$

Only execution of an instruction causes change of state of the machine, so the set $\mathbf{S}$ is created as the result of execution of all possible programs. Thus $\mathbf{v}_1$ and $\mathbf{v}_2$ could only be created as the result of execution of a programs. To make possible repetition of any execution of any program without external initialization of the machine[13], we assume:

$$\bigwedge_{S_A, S_B \subset \mathbf{S}} (\bigvee_{P} P(\mathbf{v}_A) = \mathbf{v}_B \Longleftrightarrow \bigvee_{P'} P'(\mathbf{v}_B) \to \mathbf{v}_A)$$

We know now, that there exist $P_1(\mathbf{v}) = \mathbf{v}_1$ and $P_2(\mathbf{v}) = \mathbf{v}_2$ (where $\mathbf{v}$ is some choosen state of the machine). In addition there exists $P_1'(\mathbf{v}_1) = \mathbf{v}$, so program $P_X = P_1'|P_2$ must transform $\mathbf{v}_1$ into $\mathbf{v}_2$. ■

Second property of obfuscating transformation will be shown first on the example.

**Example 3.7** *Given program in assembler of processor Intel 80386 considered in context* $\{(eax, ebx, esi, edi$

```
mov  eax,[esi]    ; eax = [esi]
add  eax,ebx      ; eax = eax + ebx
mov  [edi],eax    ; [edi] = eax
inc  ebx          ; ebx = ebx + 1
jz   jump         ; if ebx = 0,
                  ;    do jump
```

was split into two independent programs (with the same context):

```
mov  eax,[esi]    ; eax = [esi]
add  eax,ebx      ; eax = eax + ebx
```

and

```
mov  [edi],eax    ; [edi] = eax
inc  ebx          ; ebx = ebx + 1
jz   jump         ; if ebx = 0,
                  ;    do jump
```

and next they were separately obfuscated using different obfuscating transformations. As the result following programs were obtained:

```
sub  ebx,10       ; ebx = ebx - 10
                  ; change of usage!
mov  eax,[esi]    ; eax = [esi]
add  eax,20       ; eax = eax + 10
                  ; change of usage!
add  eax,ebx      ; eax = eax + ebx
```

and

[13] External in relation to instructions of programs, for an example using a RESET button.

```
; values in eax and ebx were not restored!
mov  [edi],eax    ; [edi] = eax
inc  ebx          ; ebx = ebx + 1
mov  eax,10       ; eax = 10
jz   jump         ; if ebx = 0,
                  ;    do jump
```

After concatenation the obfuscated programs would not give correct result, equivalent to result returned by original program. Value in regiester `eax` is increased by 10 on the end of the first part of obfuscated program and the second part ignores this change. It follows from definition of the obfuscating transformation, which does not impose to return the original context after obfuscation, but only requires that performed changes must be reversible. If we insert now between two programs the sequence of instructions:

```
sub  eax,10      ; eax = eax - 10
add  ebx,10      ; ebx = ebx + 10
```

then we obtain program working correctly. Analysis of this example leads to the following theorem:

**Theorem 3.4** *For any program $P$ split 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 of $P$.*

**Proof.** From the lemma 3.2 we know, that for any pair of states the program $P_X$ exists. During the proof we use definition of obfuscating transformation and the fact, that output context of program $P_1$ is input context of program $P_2$. From the assumptions we get:

$$P(\mathbf{v})= \mathbf{v}_P \Longrightarrow (P_1(\mathbf{v}) = \mathbf{v}_X \quad \wedge \quad P_2(\mathbf{v}_X) = \mathbf{v}_P)$$
$$\bigvee_{\mathcal{T}_1,\mathcal{T}_2} (\mathcal{T}_1(P_1)(\mathbf{v}) = \mathbf{v}_1 \quad \wedge \quad \mathcal{T}_2(P_2)(\mathbf{v}_X) = \mathbf{v}_2)$$

Definition of $\mathcal{T}_1(P_1)$ (3.11) implies that:

$$\mathcal{T}_1(P_1) \text{ is obfuscating tr.} \Longrightarrow \bigvee_{P'} P'(\mathbf{v}_1) = \mathbf{v}_X \Longrightarrow P_X = P'$$

so the sequnce of instructions $P_X$ is the complection of transformation $\mathcal{T}_1(P_1)$. In addition $\mathcal{T}_1(P_1)|P' \equiv P_1(\mathbf{v}) = \mathbf{v}_X$, so:

$$\mathcal{T}_1(P_1)|P'(\mathbf{v}) = \mathbf{v}_X$$

which implies:

$$\mathcal{T}(P) = \mathcal{T}_1(P_1)|P_X|\mathcal{T}_2(P_2)(\mathbf{v}) = \mathbf{v}_2$$

which means, that $\mathcal{T}(P)$ preserves output context of program $P_2$. From definition of $\mathcal{T}_2(P_2)$ we get:

$$\mathcal{T}_2(P_2) \text{ is obfuscating tr.} \Longrightarrow \bigvee_{P''} P''(\mathbf{v}_2) = \mathbf{v}_P$$

and $\mathcal{T}_2(P_2)|P'' \equiv P_2(\mathbf{v}_X) = \mathbf{v}_P$. According to lemma **??** and definition 3.11 $\mathcal{T}(P)$ is obfuscating transformation (preserved resersibility of changes introduced into program $P$ and equivalence of programs). ■

The main conclusion coming from the thorem is fact, that it is possible to create pure sequential algorithm of obfuscation, which performs obfuscation with single-pass – instruction after instruction, without any need of updating a structures describing control flow in the obfuscated program.

# Section 4

# Evaluation of Obfuscating Transformations

THERE are two complementary methods of verification of performed obfuscation process quality. Analytical methods extract information taking obfuscation algorithm parameters, source program and obfusated program. They are best in comparison of different algorithms of obfuscation, but they cannot answer the basic question: how efficient is given algorithm (an absolute value)? They fail because human factor has the great impact and only empirical research can give reasonable answer. Empirical research has of course only statistical meaning, but it can be tuned with appropriate research methodology.

## 4.1 Analytical methods

In [25] Collberg, Thomborson and Low have proposed three measures, describing how efficient and useful is given obfuscating transformation:

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

Different combinations of proposed measures are used to obtain one general measure. It is assumed that all three measures are „orthogonal" (figure 4.1), which means that it is possible to construct algorithms calculating them in such a way, that any change in value of one will not influence others. It turns out, that in general this cannot be done easily (chapter 4.1.4).

### 4.1.1 Potency of transformation

To define potency of obfuscating transformation we introduce first measure, describing how much sequence of instructions $P_1$ is more complicated (unreadable) than sequence $P_2$. Problem of measureing programs complexity is quite old (ex. see [38]). In the frame of software theory lot of different measures were created, which can be applied according to current needs. Unfortunately lot of them cannot be use in case of general algorithm of obfuscation based on low-level

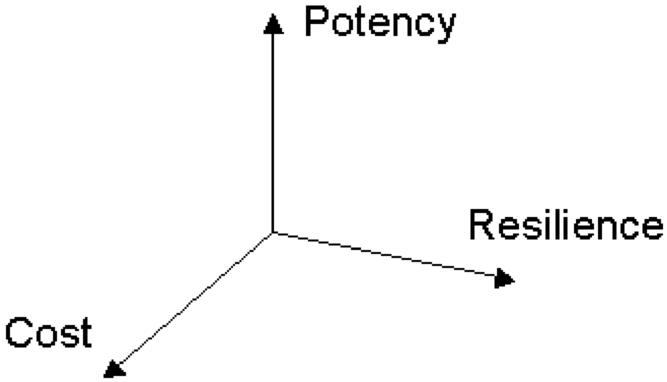


Figure 4.1: Assumed „orthogonality" of obfuscating transformations measures, according to [25].

programming, because they use high-level programming languages constructions. Examples of measures which can be used are shown in table 4.1.

Table 4.1: Overview if typical measures of program's complexity

| **Measure** | **Description** | **Author** |
|---|---|---|
| Length of program | number of instructions + number of arguments | Halstead [38] |
| Nesting level | number of nested conditions | Harrison [39] |
| Data flow | number of references to local variables | Oviedo [59] |

**Definition 4.1** *For given complexity measure $E(P)$ potency of obfuscating transformation $\mathcal{T}$ in relation to obfuscated program $P$, $\Pi(\mathcal{T}, P)$ is defined as:*

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

It is given, that $\mathcal{T}$ is *strong obfuscating transformation*, when $\Pi(\mathcal{T}, P) \gg 0$, for selected group of complexity measures.

In application with machine code measures from table 4.1 are defined as follows[14]:

1. Measure of length $E_L$ – describes specific length of program $P$ containing $N$ instructions, considers also number of arguments in instructions, according to formula (for two-address machines):

$$E_L(\mathcal{T}, P) = \sum_{i=1}^{N} c_i \quad \text{dla} \quad c_i = \begin{cases} 0 & \text{when instruction } i \text{ has no arguments} \\ 0.5 & \text{when instruction } i \text{ has one argument} \\ 1 & \text{when instruction } i \text{ has two arguments} \end{cases}$$

[14] Specific adaptation to machine code is required, because original description of these measures contains constructions typical for high-level programming languages: blocks, loops, etc. In addition we ignore calls to subroutines (only single function is analyzed).

Values $c_i$ 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 diverified values of measure $E_L$ for selected test programs (appendix A). For three-address machines $c_i = 1$ corresponds to instructions having three arguments. Exceptions in an architecture, ex. three-address instructions in two-address machines were ignored, by choosing for them also $c_i = 1$. Alleged arguments were ignored too, because they appear with very small frequency in programs.

2. Measure of depth $E_D$ – is an integer number, describing nesting level of conditional branches in static program. We propose to calculate $E_D$ with following recursive algorithm:

**Algorithm 4.1** *Input of main procedure is number of currently processed instruction* **i** *and current nesting level* **nest**. *Algorithm uses global array* ***FLAG***, *initiazed on* 0. *This array hold information if given instruction was already processed. On the beginning given procedure is called with parameters equal* 0, *which means first instruction of program and nesting level zero.*

```
(a) copy value of nest into variable njmp
(b) check in FLAG[i], if current instruction was processed already, if it was, go to (g)
(c) set in FLAG[i] value 1, informing, that given instruction was already processed
(d) if processed instruction is unconditional branch, assign to i destination address
(e) if instruction is conditional branch, add 1 to nest, call procedure with destination
    address and current value of nest, assign to njmp maximum value of nesting level
    returned by call and current value of njmp
(f) if there is any next instruction, increase i by 1 and go to (b)
(g) return value of max(nest, njmp), as the result of measurement
```

3. Measure of flow $E_F$ – is a rational number, describing the average number of references to local memory[15] in basic block of a program (figure 4.2). Basic block is defined as continuous sequence of instructions laying between two nodes of control flow graph:

$$E_F(\mathcal{T}, P) = \frac{1}{M}\sum_{i=1}^{M} a_i$$

where $M$ is number of basic blocks in program, $a_i$ is number of references to local memory in block $i$. In practice this measure is also calculated with simple algorithm:

**Algorithm 4.2** *Algorithm uses two global variables: TOTSUM – holding number of references detected so far, and TOTNUM – holding number of basic blocks detected so far. After assigning zero to TOTSUM and one to TOTNUM, for every instruction in given program P following loop is executed:*

```
(a) if instruction references to local memory, increase TOTSUM by 1
(b) if instruction is a conditional jump, increase TOTNUM by 1 and skip step (c)
```

[15] Most ofter these are local variables and parameters of a function.

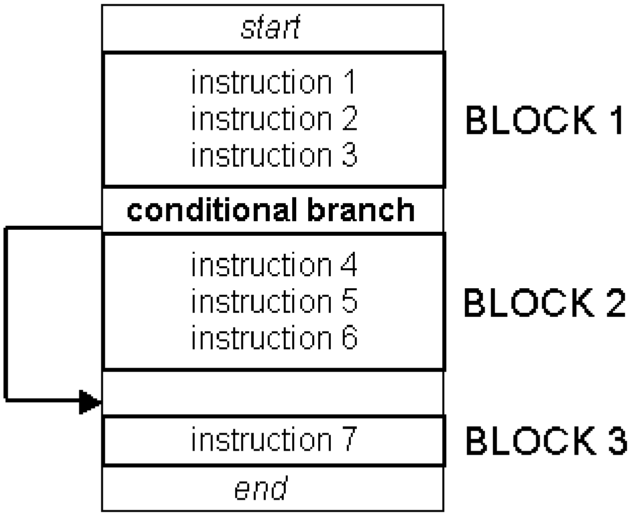


Figure 4.2: Basic blocks of typical program.

*(c) if instruction is destination of a branch(es), increase TOTNUM by* 1

As the final result we get in TOTSUM sum of all references to local memory and in TOTNUM number of basic blocks in program. Value of flow measure is calculated from the formula:

$$E_F = \frac{TOTSUM}{TOTNUM}$$

According to experiments (chapter 6.4.1), all three measures grow proportional in the obfuscation process. Efficient process of obfuscation should influence all three measures in the same way. The simplest method of concatenation of these measures – arithmetic average – seems to be rational choice.

To show potency in the form of single number we introduced average potency of obfuscation process. It is calculated as arithmetic average of potency calculated for three measures of complexity:

$$\Pi_A(\mathcal{T}, P) = \frac{1}{3}(\frac{E_L(\mathcal{T}(P))}{E_L(P)} + \frac{E_D(\mathcal{T}(P))}{E_D(P)} + \frac{E_F(\mathcal{T}(P))}{E_F(P)}) - 1$$

### 4.1.2 Resilience of transformation

Resilience of a transformation was defined in [25] as strenght of protection against automatic unobfuscation of the program. In practice resilience in described as combination of two measures:

1. Programmer's effort – time needed for programmer to write unobfuscating program.
2. Unobfuscator's effort – time of execution of such a program and amount of resources required for efficient execution.

It is hard to represent these measures in the form of numbers, because they have kind of statistical properties and depend on personal skills of a programmer and available system resources. Instead a descriptive scale was proposed ([25], [26]). Resilience $\Phi$ can be a value from the set:

$$\Phi \in \{\text{trivial, weak, strong, full, one-way}\}$$

**Definition 4.2** *Let $\mathcal{T}(P)$ be obfuscating transformation of a program $P$. $\Phi(\mathcal{T}, P)$ is the resilience of transformation $\mathcal{T}$ and can be a value:*

$$\Phi(\mathcal{T}, P) = \begin{cases} \textit{one-way} & \textit{if an information from } P \textit{ was removed} \\ \mathcal{F}(wpt, wpr) & \textit{in other case} \end{cases}$$

*where wpt is programmer's effort, wpr is program's effort and $\mathcal{F}$ is the function of resilience shown on figure 4.3.*

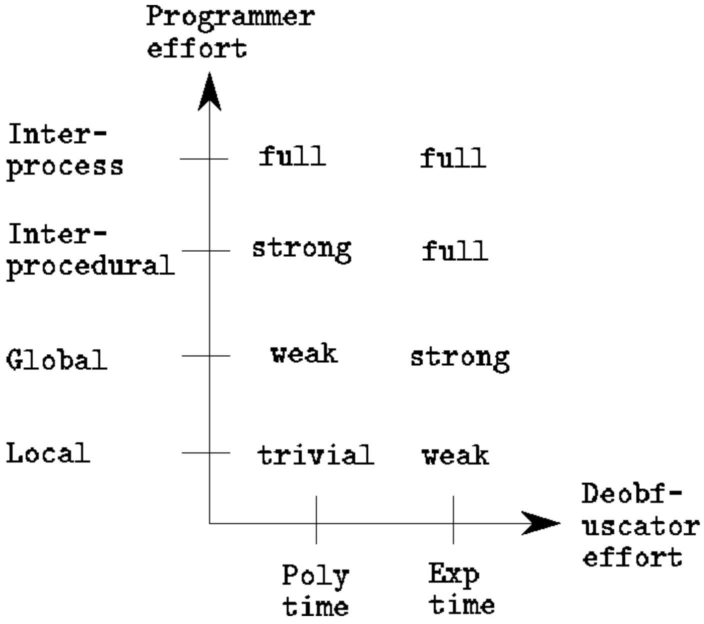


Figure 4.3: Resilience of a transformation as function of unobfuscating program's and programmer's effort, according to [25].

Program's effort was described as ponynomial or exponential, according to resources needed during the process of obfuscation. Programmer's effort was classified in the four levels:

- local – when local dependencies analysis is sufficient to unobfuscate
- global – when an additional global analysis of dependencies is required
- procedural – when there is a need to examine dependencies between procedures
- process based – when analyzed program is a parallel program and obfuscation concerns more than one process

If we add to above proposal the fifth level: random, when a programmer must guess correct form of obfuscated program (one-way resilience) and if we add two more levels to program's effort: NP-complete and infinite, then the function of resilience $\mathcal{F}(wpt, wpr)$ can be approximated by function $f(wpt, wpr) = \max(wpt, wpr)$ (figure 4.4), where appropriate levels of programmer's and program's effort are assigned to numbers $1, 2, ..., 5$. In result we get numbers assigned to ordered values of function $\Phi(\mathcal{T}, P)$.

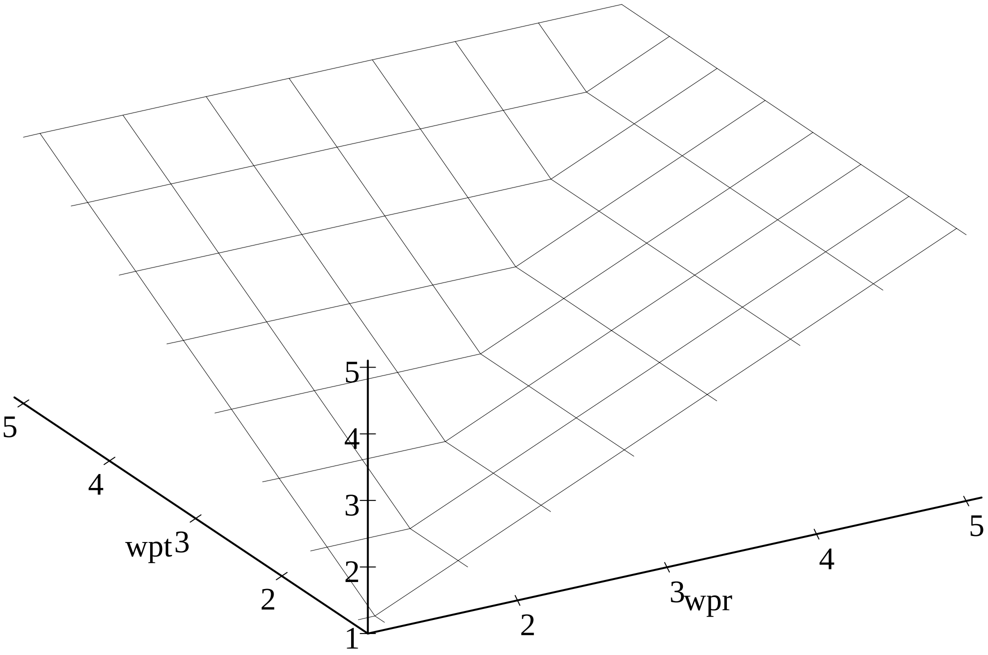


Figure 4.4: Resilience of obfuscating transformation in the form of function $\max(wpt, wpr)$.

Proposed approach is sufficient for practical applications, even we don't get a number after the measurement. Precise description of required computational complexity would have been based on complicated analysis, equally complex to the unobfuscation process. It can be seen on the example of algorithm of full analysis of interprocedural calls dependencies, given in [2] (p.658). In general, complexity of this algorithm is known (and is high), while checking if for a particular case the full analysis is possible with much lower complexity, is not an easy task.

### 4.1.3 Cost of transformation

Cost of a transformation is a penalty for obfuscating a program. In practice all methods of obfuscation make programs more complex. Fortunately in most cases it does not change the class of complexity (ex. from $O(n)$ to $O(n^2)$). In [25] the following definition of cost of obfuscating transformation was given:

**Definition 4.3** *Let $\mathcal{T}(P)$ be obfuscating transformation of a program $P$. $\Psi(\mathcal{T}, P)$ is the cost* *of transformation $\mathcal{T}$ and can be a value:*

$$\Psi(\mathcal{T}, P) = \begin{cases} \textit{dear} & \textit{when execution of } \mathcal{T}(P) \textit{ requires exponentially more resources than } P \\ \textit{costly} & \textit{when execution of } \mathcal{T}(P) \textit{ requires } O(n^p),\ p > 1 \textit{ more resources than } P \\ \textit{cheap} & \textit{when execution of } \mathcal{T}(P) \textit{ requires } O(n) \textit{ more resources than } P \\ \textit{free} & \textit{when execution of } \mathcal{T}(P) \textit{ requires } O(1) \textit{ more resources than } P \end{cases}$$

It is hard to find an universal method of static analysis, giving for a program exact number representing cost of an obfuscating transformation. In practice dynamic research must be performed with use of some software tools, because of huge complexity of such analysis. Moreover, the cost of transformation can be very different depending on obfuscating program. Insertion of multiplication instruction into a loop executing single addition in most processors will be classified as a cheap transformation. Insertion of the same instruction into the main loop of Fast Fourier Transform will give free cost.

Empirically determined cost of an obfuscating transformation can be given as the average factor of increase of execution time for the program after obfuscation, calculated for different input data. Representing by $t(P(d_i))$ time of execution of program $P$, obtained for a set of input data $d_i$, given $N$ different sets of input data, empirically calculated cost of transformation $\mathcal{T}$ can be found from the formula:

$$C(\mathcal{T}, P) = \frac{1}{N} \sum_{i=1}^{N} \frac{t(\mathcal{T}(P)(d_i))}{t(P(d_i))}$$

### 4.1.4 General measure

In [25] a simple definition of a general measure of an obfuscating transfomation was given. It was defined as a combination of three basic measures: potency, resilience and cost. It is hard to entangle these measures in the way allowing to obtain single value, especially because indeed they are not orthogonal. According to general analysis, based only on definitions of these measures, following dependencies can be concluded, shown in table 4.2. Growth of potency does not have to cause growth of resilience, because general complexity of program does not have to be connected with complexity of unobfuscatio process. In opposite way, growth of resilience must cause growth of potency, because general complexity of a program grows always. Cost of a transformation grows more or less, when resilience and/or potency grows, but growth of cost does not have to cause the growth of neither potency nor resilience, because a complexity can remain the same.

Table 4.2: Dependencies between measures of obfuscating transformation.

| **Growth** | **Potency** | **Resilience** | **Cost** |
|---|---|---|---|
| **Potency** | | variously | grows |
| **Resilience** | grows | | grows |
| **Cost** | grows | variously | |

In final decision, we decided to show results of quality measures of obfuscating transformations in the form of four objects: numerical value of an average potency of transformation, empirically determined cost and two descriptive measures: resilience and cost.

From the presented material we can see, that all analytical measures have rather relative meaning and should be used in the group of results obtained for different programs, to provide a broad view for comparison. Giving the analytical measures only for single program can be a false information. In practice the same program is measured before and after obfuscation, which gives quite good comparison.

## 4.2 Empirical methods

Analytical methods of obfuscating transformations quality measurements give set of informations about applied obfuscation process. Yet they do not answer the basic question: how efficient given obfuscating transformation protected a program from unauthorized analysis? To obtain answer to this question we must perform empirical methods of measurement, which in our case will be research done on some groups of people.

The main task of programs' obfuscation is protection against unauthorized analysis of a program. The last step of the process of analysis belongs always to a human, so to measure real efficiency of the obfuscation process, we must conduct appropriate empirical research.

The correct empirical test requires good choice of test group, program, algorithm of obfuscation and its parameters and statistical processing of obtained results. Possible persons, who can try to analize the obfuscated program, were classified into three groups according to their skills:

- students – in most cases of computer science or closely related subjects, having sufficient knowledge of assebler languages, but with low programming experience

- engineers – persons with high programming experience, not always having full knowledge about assembler programming environment

- crackers – representatives of the sub-culture, which main goal is free distribution of any information, including computer programs, as well as breaking protections present in software and information; persons having very good knowledge of assembler environment, tracing, decoding and transforming tools; persons having most often high programming experience[16]

A single function was selected for testing. The function decodes and checks correctness of a memory fragment, very often used in program's protection. As the result of test with get the minimum time needed in given group for correct understanding of meaning of obfuscated program. The fastest answer is the very important result, because security is the main goal of our process.[17] Examples of performed tests of this kind are included in chapter 6.4.3.

[16] Sources of information about this sub-culture are included in appendix D.

[17] We do not assume for an example: average strength of a rope when we design an elevator.

# Section 5

# Obfuscating Transformations

THE basic element of an obfuscating algorithm is obfuscating transformation. The quality of obfuscation process strongly depends on quality and types of used transformations. Because of the low level approach we decided to introduce our own, simplified classification of obfuscating transformations.

## 5.1 Classification

In opposition to Collberg's classification ([25]), based mainly on the structure of typical object oriented language, a taxonomy much closer to the theoretical background from the chapter 3 was designed and it is close to low level programming (on hardware level). Such approach allowed to simplify significantly method of obfuscation and in practice reduced total number of obfuscating transformations, which should have been taken into detailed analysis (figure 5.1).

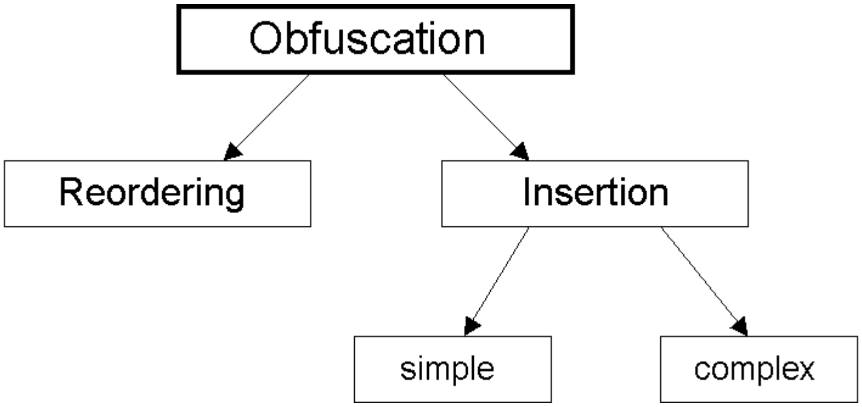


Figure 5.1: Classification of obfuscating transformations.

It is easy to see that there is no classification of control and data obfuscation. We removed it, to obtain fastest possible algorithm, which on the base of presented theory requires obfuscation of small fragments of programs. Large fragments would require analysis of larger context, including all used data structures. Yet it does not mean, that data obfuscation in the algorithm designed in sucha way does not exist. It is strongly entangled with control obfuscation, which is shown with details in section 5.5.

As the basic criterion of classification was taken type of information, used by the algorithm to make decision about inserted instruction. If the information consists only of machine context, it is simple insertion. Complex insertion uses more information, analyzing larger fragment of program.

Treating program only as a sequence of instructions a simple classification of obfuscating transformations can be constructed. On the given program $P$, split on two parts $P = P_1|P_2$, following obfuscating transformations can be applied:

- insertion of additional instructions:

  $$P = P_1|P_2 \Longrightarrow \mathcal{T}(P) = P_1|P_I|P_2$$

  while $P_I$ must satisfy complex conditions shown in section 5.3

- change of fragment of program $P$, ex. $P_1$ on $P_C$:

  $$P = P_1|P_2 \Longrightarrow \mathcal{T}(P) = P_C|P_2$$

  while following conditions are satisfied for $P_1(\mathbf{v}) = \mathbf{v}_1$ and $P_C(\mathbf{v}_C) = \mathbf{v}_3$:

  $$S \Cap S_C = S \quad \wedge \quad S_1 \Cap S_3 = S_1 \quad \wedge \quad P_C \equiv P_1(\mathbf{v}) = \mathbf{v}_1$$

- reordering of instructions of program $P$:

  $$P = P_1|P_2 \Longrightarrow \mathcal{T}(P) = P_2|P_1$$

  while following condition is satisfied[18]:

  $$(P_1(\mathbf{v}) = \mathbf{v}_1 \quad \wedge \quad P_2(\mathbf{v}_{12}) = \mathbf{v}_2) \Longrightarrow S_1 \Cap S_{12} = \{\bar{\alpha}\}$$

- reordering of blocks of program $P$:

  $$P = P_1|P_2 \Longrightarrow \mathcal{T}(P) = J_1|P_2|J_2|P_1|J_3$$

  where $J_1$, $J_2$, $J_3$ are branches preserving correct order of execution of $P_1$ and $P_2$

## 5.2 Properties of Programs

In case of insertion of an additional code the natural problem arises: what and how to insert? The easiest solution would be insertion of random instructions, to provide high resilience the randomness should be parametrized with some empirical values obtained for given model from real programs.

In further part we use two properties of typical programs of todays machines. The main source of these properties are dependencies, occurring between instructions placed in the program with given distance. Such dependency happens, when current instructions uses elements of context changed by earlier instruction.

[18] In practice the condition is satistied most often, when $P_1$ and $P_2$ are instructions of the given machine. That's why this type of reordering is called reordering of instructions.

**Definition 5.1** *Given program $P = (I_1, I_2, ..., I_N)$ which is $N$ instructions long. Let $S_i$ be input context for instruction $i$ and $S_i^*$ its output context. Probability of dependency between instructions distant d instructions in program $P$ can be calculated from the formula:*

$$p(P,d) = \frac{1}{N-d}\sum_{i=1}^{N-d} c_i \quad for \quad c_i = \begin{cases} 1 \text{ when } S_i^* \Cap S_{i+d} \neq \{\bar{\alpha}\} \text{ and } \bigwedge_{j=1,2,...,d-1} S_{i+j}^* \Cap S_{i+d} = \{\bar{\alpha}\} \\ 0 \text{ in other case} \end{cases}$$

Dependencies were calculated in context [**registers, local variables**[19]**, global memory**], according to the following algorithm:

**Algorithm 5.1** *Given program $P$, which is $N$ instructions long and distance $d = 1, 2, 3, ..., N-1$.*

1. *Set counter of dependencies $k = 0$.*
2. *For every pair of instructions from $P$, which has $d-1$ instructions between, increase $k$ by 1, if instruction earlier in the pair changes context of the input context of further instruction and no instruction between them modifies the changed part of context.*
3. *Return $\frac{k}{N-d}$ as the result of propability of dependency.*

The first property is shown on the figure 5.2. From the empirical research we determined, that for typical machines and programs probability of dependecy between instructions falls with distance between them and for machines with small number of registers (ex. Intel x86), stabilizes on some level[20]. It makes anaylisis of program simle, but on the other hand automatic detection of absence of dependencies (easy to implement) can indicate a non-real program or a fragment (when data are significantly different from shown on the figure 5.2).

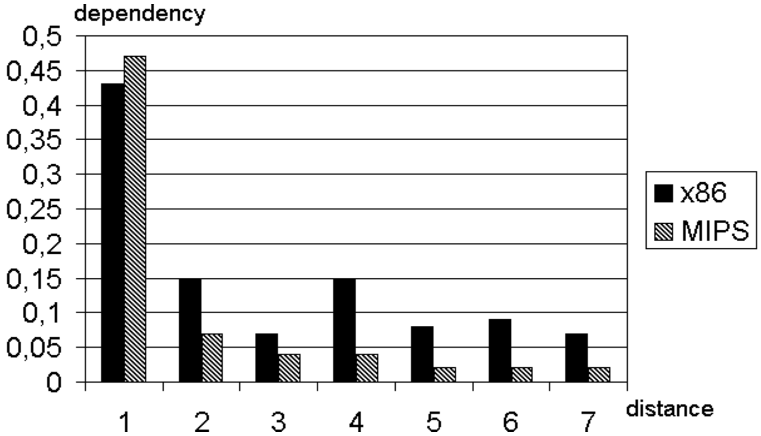


Figure 5.2: Probability of dependency between instructions in function of distance between them.

[19] Parameters of function and local variables. Most often it is a reserved stack area.

[20] Bump of the probability value for $d = 4$ in case of Intel x86 processor comes from the recurring pattern of registers usage generated by choosen compiler.

The second property was determined from an experiment, executed on two models of random programs generation.. Figure 5.3 shows values of probability, that $n$ random instructions taken from all operations and branches of given processor[21] would make a real looking program. The basic criterion of "reality" was the same algorithm of measurement of dependencies, occurring between instructions. It can be seen, that there is rather a little chance to obtain a real looking program only from random instructions.

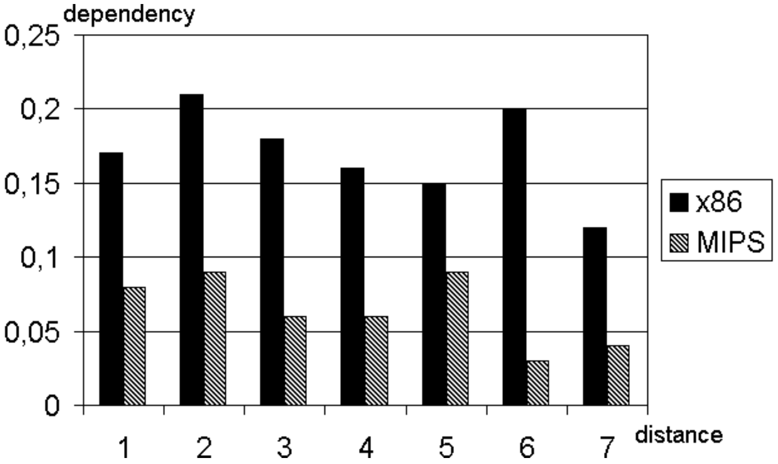


Figure 5.3: Probability of construction a real looking program from $n$ randomly selected instructions.

Much better results were obtained in a little more complicated model: a condition of dependency between neighbourhood instructions was added to randomness. The next random instructions will be accepted only, if its execution depends (with dependecy defined by the presented algorithm) on the previous instruction[22]. Figure 5.4 shows, that in such an easy way it is possible to create "real looking" programs.

The methodology and parameters of experiments are described with details in appendix B.

Results of above analysis in the case of real machines depend hardly on total number of instructions and number of registers. For not typical architectures above figures can degenerate a lot (ex. very high probability for random programs).

## 5.3 Insertion

The easiest way of obfuscation is addition of new instructions, covering the view of the real control path. These instructions should in some safe way entangled with obfuscated program, leaving impression of a real program, and they should be generated in possibly fast way.

It can be known from practice (see appendix C), that most of programs do not use in any moment of execution all available context of a machine. It gives some additional possibilites during contruction of the general method of insertion. According to theorem 3.3 following instructions can be inserted:

- any instructions not changing the context in obfuscated program

[21] Special instructions were ignored, because they are less frequent in real programs.

[22] In case when creation of dependecy is impossible, the instruction is accepted in the original form.

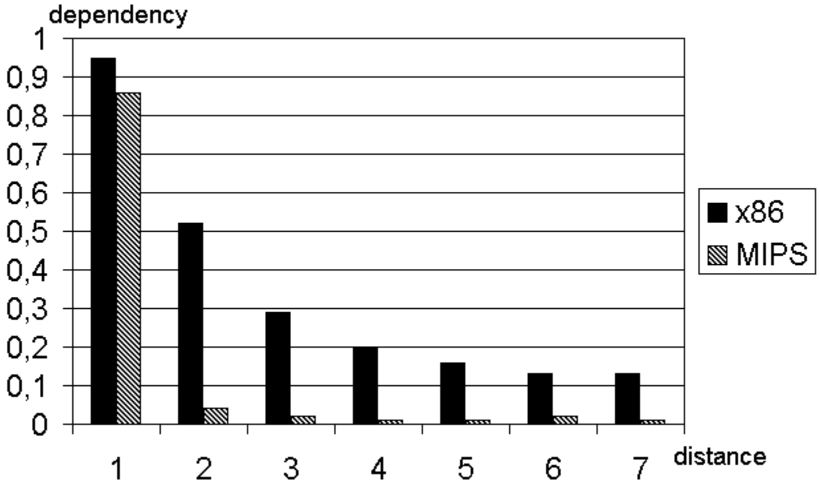


Figure 5.4: Probability of creation of "real looking" program from $n$ random and dependent instructions.

- reversible operations changing the used part of context in obfuscated program

"Any" instructions include any operations, changing not used part of machine context. Property of using/not using a part of context has local scope, that's why before we use insertion a full data flow analysis is required for given part of whole machine context.

**Example 5.1** *A machine program in assembler of microprocessor Motorola 68000 is considered in context* $\{(d0, d1, d2, d3)\}$*:*

```
move.w #23,d0    ; d0 = 23
add.w  d0,d1     ; d1 = d1 + d0
move.w d2,d3     ; d3 = d2
```

and was obfuscated by inserting instructions modifying not used part of machine context $\{(d4, d5, d6)\}$[23]:

```
move.w #23,d0    ; d0 = 23
sub.w  d0,d4     ; d4 = d4 - d0 ***
add.w  d0,d1     ; d1 = d1 + d0
move.w d1,d5     ; d5 = d1      ***
move.w d2,d3     ; d3 = d2
sub.w  d3,d6     ; d6 = d6 - d3 ***
```

Such a simple insertion make harder analysis of program by a human. The main reason is that during the analysis a human covers at a time only small pieces of code and only a few paths of data flow. Adding a lot paths without outlet slows down the process of analysis significantly.

All methods of insertion casue increase of potency of given obfuscation process, because complexity of obfuscated program grows. There is not such easy dependecy with resilience,

[23] New instructions are marked with ***.

ex. insertion of instructions using only not used part of context does not change resilience significantly, because these instructions can be removed quite easily using automatic tools.

Insertion of any instruction causes growth of length of obfuscated program. To represent this growth in the form of number we introduce the *rescaling factor* $S$, which can be calculated from the formula:

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

where |...| returns length of given program in number of instructions. The rescaling factor will be used during the construction of general algorithm of obfuscation.

### 5.3.1 Simple insertion

The simple insertion adds any instruction using current information about context in obfuscated program. Given program $P$ is split into two parts $P_1$ and $P_2$, between them an instruction $P_I$ is inserted:

$$\begin{aligned} P &= P_1|P_2 \\ \mathcal{T}(P) &= P_1|P_I|P_2 \end{aligned}$$

The programs transform the machine context as follows:

$$\begin{aligned} P_1(\mathbf{v}) &= \mathbf{v}_1 \\ P_2(\mathbf{v}_1) &= \mathbf{v}_2 \\ P_I(\mathbf{v}_I) &= \mathbf{v}_I^* \end{aligned}$$

If the following condition is satisfied:

$$S_1 \Cap S_I^* = \{\bar{\alpha}\}$$

then instruction $P_I$ changes not used part of context and can be any instruction of the machine. If:

$$S_1 \Cap S_I \Cap S_I^* \neq \{\bar{\alpha}\} \qquad \wedge \qquad \bigvee_{P^{-1}} P^{-1}(\mathbf{v}_I^*) = \mathbf{v}_I$$

then the instruction changes the used part of context of obfuscated program and is an reversible operation (must be, to make $\mathcal{T}$ valid obfuscating transformation). Given conditions can be obtained easily from the theorem 3.3

**Example 5.2** *Given machine program in assembler of microprocessor Intel 80386 considered in context* $\{(eax, ebx, edx)\}$*:*

```
mov  eax,10        ; eax = 10
add  ebx,eax       ; ebx = ebx + eax
sub  edx,eax       ; edx = edx - eax
mov  [esi],ebx     ; [esi] = ebx
```

is obfuscated with simple insertion:

```
mov  eax,10       ; eax = 10
xchg eax,ebx      ; eax <-> ebx      ***
add  eax,ebx      ; eax = eax + ebx
sub  edx,23       ; edx = edx - 23  ***
sub  edx,ebx      ; edx = edx - ebx
add  ebx,42       ; ebx = ebx + 42  ***
mov  [esi],eax    ; [esi] = ebx
```

After fast analysis it can be seen, that context was not changed, but values of output context were changed from $\{(eax, ebx, edx)\}$ to $\{(ebx, eax + 42, edx - 23)\}$. Using three operations we can restore the original values:

```
xchg eax,ebx      ; eax <-> ebx
sub  eax,42       ; eax = eax - 42
add  edx,23       ; edx = edx + 23
```

Insertion of reversible operations increases resilience of obfuscated program, but it is not a critial change for potencial autiomati unobfuscator. Removal of effects of simple insertions is made by detection in data flow graph a paris of operations with preseved tracing of control flow. The method is not complex computationally.

### 5.3.2 Complex insertion

In general into the group of transformations called complex insertion fall all transformations, which during the obfuscation process use not only information about context usage, but also use its semantic – what a particular instruction does mean. It is possible to create almost any amount of such transformations ([25], [72]), because information included in a program can be interpreted in many different ways. From our formal point of view complex insertion is a modification of instructions of obfuscated program or simple insertion of programs not changing not used part of context only. Marking as in the simple insertion:

$$P = P_1 | P_2$$
$$\mathcal{T}(P) = P_1 | P_I | P_2$$

and

$$P_1(\mathbf{v}) = \mathbf{v}_1$$
$$P_2(\mathbf{v}_1) = \mathbf{v}_2$$
$$P_I(\mathbf{v}_I) = \mathbf{v}_I^*$$

it is required the following condition to be satisfied:

$$S_1 \Cap S_I^* = \{\bar{\alpha}\}$$

while in case of complex insertion the program $P_I$ consists most often of few or a dozen of instructions.

Sample methods of complex insertion (taken from [25]) can be described with the following action:

- exchange of equivalent fragments of a program – requires a little bit more detailed analysis of the program and some knowledge database, containing equivalent sequences; during exchange the context can be modified only in a reversible way

- extension of conditions – in case of conditional branches it is possible to extend the conditional expression and add some redundant conditional jumps
- loop unrolling – a short loop with constant counter $n$ can be substituted with unrolled $n$-times body of the loop
- code specific insertion – ex. jump into the middle of an instruction code, which can be beginnig of a code of totally different instruction (possible in Intel 80386 processor's assembler)
- environment specific insertion – ex. a sequence of instructions detecting presence of a debugger into the system
- insertion of "virtual machines" – simple synchronous state automaton, which control sequence execute encoded program (implementation of automaton must be included)
- insertion of opaque constructs – special fragments of programs, which are computationally difficult to identify and remove (described with details in the next section)

The most efficient method of complex insertion, which has highest resilience, is insertion of opaque contructs, that's why it was described in the speparate section.

**Example 5.3** *Given machine program in assembler of Intel 80386 processor is considered in context* $\{(eax, ebx, ef)\}$*:*

```
mov  eax,0        ; eax = 0
add  ebx,[esi]    ; ebx = ebx + [esi]
cmp  eax,ebx      ; eax = ebx ???
jz   jump         ; yes => jump
```

was obfuscated with the following result:

```
sub  eax,eax      ; eax = 0
mov  edx,eax      ; edx = eax       ***
add  ebx,[esi]    ; ebx = ebx + [esi]
cmp  eax,ebx      ; eax = ebx ???
jz   jump         ; yes => jump
cmp  eax,edx      ; eax = ebx ???    ***
jnz  never        ; no => never    ***
```

which uses extended context $\{(eax, ebx, edx, ef)\}$. It can be seen, that obfuscation inserted the additional condition, which will never be satisfied, so the second jump will never occurr. A typical trick is also substitution of load of value zero, by substraction of the register from itself.

Depending on its type, complex insertion increases resilience of obfuscated program in different way. Efficiency of exchanging of equivalent fragments and loop unrolling is similar to efficiency of insertion of instructions changing not used part of context. Slightly better is in case of extending branches conditions, where full removal of inserted code required complex optimizing analysis.

**Example 5.4** *Given two instructions of Intel 80386 processor's assembler:*

```
1000   8B 06      mov  eax,[esi]     ; eax = [esi]
1002   83 C0 0A   add  eax,10        ; eax = eax + 10
```

were obfuscated by inserting instructions specific for the code of given machine:

```
1000   8B DA             mov  ebx,edx     ; ebx = edx      ***
1002   8B 06             mov  eax,[esi]   ; eax = [esi]
1004   43                inc  ebx         ; ebx = ebx + 1 ***
1005   3B DA             cmp  ebx,edx     ; ebx = edx ???  ***
1007   75 01             jnz  1010        ; no => 1010     ***
1009   E9 83 C0 0A 90    jmp  900AC083H   ; add  eax,10    ***
```

The second instruction of source program was put as the address of jump, which will never be executed (90H code is a „nop" instruction – do nothing, required to fill up the code of jump instruction). This is the simplest technique of hiding real instructions against disassebling programs.

Insertion showed in the above example is easy to detect by a human, becasue of the specific of inserted code. Yet the big advantage for these types of insertion if high resilience for automatic analysis. The resilience comes from the must of creation of special algorithms detecting particular type of inserted code.

### 5.3.3 Opaque Constructs

Unfortunately program obfuscated in the easy way, based only on reversible changes in the context and extensions of context, can be quite easy unobfuscated. The method can protect only against "manual" analysis. Yet using automatic analysis one can remove almost all redundant code. The solution improving efficiency of obfuscation is including of so called *opaque constructs*, a specific pieces of code, not entangled with the main context of obfuscated program (but optionally using this part of context). The specific property of these pieces is that automatic detection of them is possible only with use of computationally very complex algorithms.

The idea of opaque construct was introduced for the first time in [26], where the definition was given.

**Definition 5.2** *Opaque construct in the point $p$ of a program is the variable $V$ or a fragment of programu $P$, which value of result of calculations is well known during the time of obfuscation, but is very hard to determine after obfuscation*[24]*.*

If the variable $V$ in the point $p$ has always value 5, it is written as $V_p^{=5}$. If the value is unknown, as $V_p^?$.

**Example 5.5** *Into the given program of Intel x86 processor considered in the context $\{(eax, ebx)\}$:*

```
mov  eax,[esi]     ; eax = [esi]
add  eax,ebx       ; eax = eax + ebx
mov  [edi],ebx     ; [edi] = ebx
```

an opaque instruction was inserted:

[24] Hard in the sense of computational complexity.

```
mov  ecx,32       ; ecx = 32
mov  eax,[esi]    ; eax = [esi]
add  eax,ebx      ; eax = eax + ebx
mov  edx,ecx      ; edx = ecx = 32
shl  ecx,3        ; ecx = 32<<3 = 256
and  ecx,edx      ; ecx = ecx AND edx ($V_{ecx}^{=0}$)
add  eax,ecx      ; eax = eax + ecx = eax
mov  [edi],ebx    ; [edi] = ebx
```

As the result of execution of the opaque construct, to the element *eax* value 0 will be always added. This type of opaque construct is of course trivial, but it still requires full static analysis of the obfuscated program to do removal.

Opaque construct can be used in very different ways. In the above example result of the operation in contruct became the additional argument of an expression. In the same way any expression or condition can be extended. Using construct as a independed condition a redundant branch can be inserted (jump when the condition is always true) or hide the real code behind a conditional jump (always false).

In the same paper ([26]) a general solution of creating of opaque construct was given. The constructs have high resilience and their removal requires use of exponentially complex algorithms (or even more complex). The solution uses weakness of present algorithms of pointer alias analysis problem ([32], [12]) and looks like this:

**Solution 5.1** *Creation and implementation of an efficient opaque construct:*

1. *Add to the obfuscated code a fragment, creating set of complex dynamic structures $S_1, S_2, ...$.*

2. *Store a set of pointers $p_1, p_2, ...$ in these structures.*

3. *Insert into the obfuscated program a code, updating from time to time these structures (modifying pointers, adding nodes, binding and spliting structures, etc.), but preserving some particular properties, conditions specific for the set of pointers $p_1, p_2, ...$, which have always the same logic value, ex. „$p_1$ never uses the same area of memory as $p_2$” or „there is always a path from $p_1$ to $p_2$”, etc.*

4. *Use these conditions, whenever you want to insert opaque construct, making appriopriate values $V_{variable}^{condition}$.*

The solution is very attractive because of the following reasons:

- inserted code does not differ from the code of most present programs (mass use of structures and pointers)

- it is easy to create methods of structure's updating, impossible to analysis for present algorithms, [26]

- it is easy to create simple (cheap) conditions (ex. comparison of two pointers)

The main disadvantage of this solution is high-level approach to any program (concept of structure) and use of system services during execution (operations on memory heap during dynamic management of structures). The way this solution works is shown in the below example ([26]):

**Example 5.6** *An object $O$ was created, which consists of three fields:*

| **Name of field** | token | ptr_1 | ptr_2 |
|---|---|---|---|
| **Descirption** | TRUE or FALSE | pointer to $O$ | pointer to $O$ |

On the beginning of the obfuscated program a fragment creating two separated groups of objects $O$: structures $G$ and $H$ (figure 5.5). Three pointers $f$, $g$ and $h$ were added to global variables, pointing to one of objects in structures: $f$, $g$ in $G$ and $h$ in $H$.

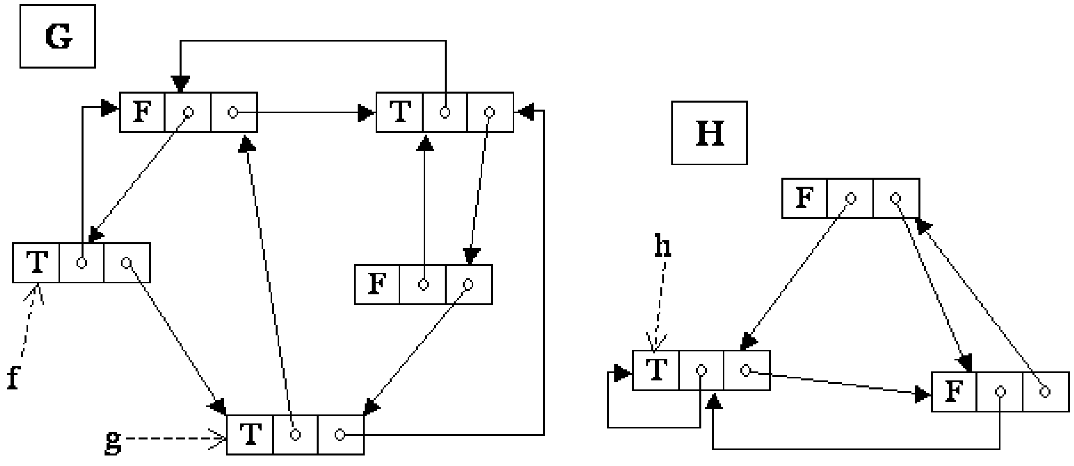


Figure 5.5: Example of binding structures od some objects, creating base for opaque constructs.

Three operations were defined on the structures $G$, $H$:

- `Join(f,g)` – joins two structures, pointed by $g$, $h$
- `f=Split(g)` – splits the structure pointed by $g$, returning to $f$ pointer to separated part
- `Insert(f)` – inserts a new object into the structure pointed by $f$, adding it to the present objects

and one operation on the pointers to $O$:

- `f=Move(f)` – move pointer $f$ to the object pointed by the field *ptr*_1

Using these operations and modifying field *token*, any number of opaque constructs can be created. Below three examples are shown.

1. Do any operations on the pointers $f$, $g$, $h$, except `Join(f,h)`and `Join(g,h)`. We obtain in such easy way two simple conditions: $(f \neq h)^T$ and $(g \neq h)^T$.
2. Modify the field *token* in the object pointed by $f$, ex. $f.token = TRUE$, next in the object pointed by $h$. These are always different objects, so the change through the pointer $h$ will never changed the value pointed by $f$. In this way we can use the condition: $(f.token = TRUE)^T$.

3. Call `Join(f,h)`, next do any operations on the pointers and fields *token*. We obtain in this way unlimited number of conditions of unknown value, ex. $(f = g)^?$. These conditions can be used to insert a neutral code or to copy the original code in two parallel control paths.

Analyzing examples from the papers [26] and [25], we created a more general methodology of creation of opaque constructs:

**Solution 5.2** *General method of creation and implementation of opaque constructs:*

1. *Create a list of typical constructions present in the source program.*
2. *Select from the list such contructs, which analysis is most computationally complex during the process of unobfuscation.*
3. *Add to the program a new elements, allowing to create opaque constructs.*
4. *Insert into the obfuscated code opaque constructs according to your needs and constructs found on the list.*

Using this method once can create opaque constructs staying on the low-level of programming. It is demostrated in the following example:

**Example 5.7** *Given program was examined and it was determined, that some of its functions are called very often during the execution by lot of other functions, with different values of input parameters, but with known ranges.*

| **Function** | **Ranges of parameters** |
|---|---|
| `Func1(a,b)` | $a = 1, 2, ..., 255 \quad b = 0, 1, ..., 5$ |
| `Func2(a,b,c)` | $a = 1, 2 \quad b = 32, 33, ..., 127 \quad c = 0, 1, ..., 65535$ |
| `Func3(a)` | $a = 0, 2, 3, 6$ |

Adding few global variables $aa$, $bb$, $cc$, ..., we can make new resilient opaque constructs.[25] 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. Much stronger resilience can be achieved by inserting some dependencies between functions:

$aa = bb\ OR\ b\ OR\ a$ – into the function `Func1(a,b)`
$bb = ((cc + a) - (b * c))\ AND\ aa$ – into the function `Func2(a,b,c)`
$cc = a + (aa + bb) * 2$ – into the function `Func3(a)`

Assuming, that all global variables will be initialized with value 0, we can calculate, that variables $aa$ and $bb$ will never be greater than 255 and variable $cc$ will be never greater than 1030[26]. Knowing the rule of creation of such expressions one can create any number of them without any problems.

If we make an automatic method of generating opaque constructs of some type (like in the example above), then having algorithm of creation, it would be easy to write specific algorithm detecting only these type of opaque constructs. It can be seen, that opaque constructs are

[25] Global variables in this case add requirement of full analysis of interprocedural dependencies for data and control paths.

[26] We assume, that we have full control of the values of variables, even in case of invalid input data.

encryption keys of obfuscating algorithms. To provide most efficient method of obfuscation, the key and methods of its generation should be kept secret and be as individual as possible.

High resilience of opaque constructs on the methods of static program analysis does not guarantee high resilience on methods of dynamic analysis. A simple check, that a condition in program during multiple execution has always a constant value, can suggest existence of an opaque construct in this place. This problem can be solven in a few ways. They were described in the paper [26]. In general adding dependencies between different opaque constructs can be very helpful in this case. Appropriate dependencies will require, that proper run of obfuscated program could be possible only after removal of all inserted opaque constructs in the single step.

The important factor, increasing the quality of opaque construct, is its uniqueness. In the perfect case such a construct should be created by the programmer, author of obfuscated program. The nature of opaque constructs allows to satisfy this requirements, because the only limitations to the form of an opaque construct are in the technical parameters of the machine or used software environment[27] and in the imagination of creator. To create an efficient algorithm of obfuscation it would be good to create a database of opaque constructs, which could be used by the algorithm in some given way.

**Example 5.8** *Given program in the assembler of Motorola 68000 processor considered in the context* $\{(d0, d1, d2, d3, CCR)\}$*:*

```
move.w d0,d1    ; d1 = d0
add.w  d2,d1    ; d1 = d1 + d2
beq    jump     ; d1 = 0 ?, yes => jump
sub.w  d3,d1    ; d1 = d1 - d3
```

and the opaque construct defined by the formula:

$$aa = (aa + bb - X)\ AND\ cc$$

where :

$aa, bb, cc$ – global variables of integer type

$X$ – any integer number

and :

$aa > 0$ is always true

which was inserted into the obfuscated program. As the result following machine program was obrained:

```
move.w d0,d1    ; d1 = d0
move.w aa,d4    ; d4 = aa              ***
sub.w  d1,d4    ; d4 = d4 - d1         ***
add.w  d2,d1    ; d1 = d1 - d2
beq    jump     ; d2 = 0 ?, yes => jump
add.w  bb,d4    ; d4 = d4 + bb         ***
sub.w  d3,d1    ; d1 = d1 - d3
and.w  cc,d4    ; d4 = d4 AND cc       ***
add.w  #31,d4   ; d4 = d4 + 31         ***
blt    jump     ; d4 < 0 ?, yes => jump ***
```

[27] In most cases highly redundat comparing to the needs.

working in the extended context $\{(d0, d1, d2, d3, d4, CCR)\}$, in which the jump „`blt jump`" will never be executed. In practice the inserted opaque construct is also obfuscated by use of other techniques.

Opaque constructs are the main tool to increase resilience of an obfuscated program (chapter 5.3.3). It is implied from their constructions and high computational complexity of removing programs. Similar to other methods of insertion we observe increase of potency of obfuscated program, propotional to the number of inserted instructions.

## 5.4 Reordering

The second simplest method of code obfuscation is the reordering of program's fragments. In opposition to insertion, the goal of reordering is not to change the context of a program, but to disturb original sequence of execution. If reordered fragments are short, independent from themselves, no more changes are inserted into the program. In case of longer fragments, addidion of jumps organizing correct sequence of execution is required. The best effect is achieved, when this jumps are conditional, with constant control flow.

**Example 5.9** *Given program in the assembler of Motorola 68000 processor:*

```
move.w #10,d0      ; d0 = 10
move.w (a0),d1     ; d1 = (a0)
add.w  d1,d2       ; d2 = d2 + d1
add.w  d0,d3       ; d3 = d3 + d0
```

was obfuscated by instructions reordering:

```
move.w (a0),d1     ; d1 = (a0)
move.w #10,d0      ; d0 = 10
add.w  d0,d3       ; d3 = d3 + d0
add.w  d1,d2       ; d2 = d2 + d1
```

which did not change the overall look of the program. Advantages of this method can be seen by longer programs, when distance between exchanged instructions are longer.

**Example 5.10** *Given program in the assembler of Motorola 68000 processor:*

```
move.w (a0),d0     ; d0 = (a0)
move.w (a1),d1     ; d1 = (a1)
add.w  d2,d0       ; d0 = d0 + d2
add.w  d3,d1       ; d1 = d1 + d3
muls   d4,d0       ; d0 = d0 * d4
muls   d5,d1       ; d1 = d1 * d5
```

was obfuscated with blocks reordering:

```
     bra   lab1           ; => lab1
lab2:
     add.w  d3,d1         ; d1 = d1 + d3
lab4:
     muls   d4,d0         ; d0 = d0 * d4
     muls   d5,d1         ; d1 = d1 * d5
     bra    lab3          ; => lab3
lab1:
     move.w (a0),d0       ; d0 = (a0)
     move.w (a1),d1       ; d1 = (a1)
     add.w  d2,d0         ; d0 = d0 + d2
     bne    lab2          ; d0 = d2 ?, no => lab2
     add.w  d3,d1         ; d1 = d1 + d3
     bra    lab4          ; => lab4
lab3:
```

which slowed down the process of analysis of this program. Together with insertion of jumps a local code replication is used. Use of this technique with different methods of insertion, by increasing the size of obfuscated program few times, can make analysis of correct path of execution very difficult.

In the formal way blocks reordering is described by transformation:

$$P = P_1|P_2 \Longrightarrow \mathcal{T}(P) = J_1|P_2|J_2|P_1|J_3$$

where instructions $J_1$, $J_2$, $J_3$ are unconditional branches or empty programs, if $P_1$ i $P_2$ satisfy conditions given in the section 5.1. Applying $\mathcal{T}(P)$ on the obtained programs we get:

$$\mathcal{T}_R(P) = \mathcal{T}(\mathcal{T}(...\mathcal{T}(P)))$$

which is the final reordering transformation. Knowing the number of instructions inserted during reordering we can calulate the empirical frequency of blocks reordering $R_B$:

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

where by $|...|_R$ we mean the number of instructions inserted during the reordering. For an example $R_B = 0.1$ means, that on average after every tenth instruction of program $P$ there is reordering instruction in the program $\mathcal{T}(P)$. Coefficient $R_B$ will be used as the input parameter of the general algorithm of obfuscation.

Reordering does not change significantly analytical measures of quality of an obfuscation process. With use of automatic analysis it is possible to remove all inserted jumps in an easy way. Yet a special software is still required (no common tool is known so far).

## 5.5 Data Obfuscation

Data obfuscation as presented in papers [25], [26], does not exist in the proposed formal analysis. Yet elements of data obfuscation are present in the implementation in a limited scope.

**Example 5.11** *Given program in the assembler of Intel 80386 processor, which is a fragment of bigger procedure, having some variables stored on the local stack, pointed by the value of register ebp:*

```
mov  eax,[ebp+4]         ; eax = [ebp+4]
add  eax,[ebp]           ; eax = eax + [ebp]
mov  [ebp+8],eax         ; [ebp+8] = eax
inc  dword ptr [ebp+4]   ; [ebp+4] = [ebp+4] + 1
```

The size of all variables is equal to the size of register *eax* (4 bytes). Given program was obfuscated in the context $\{(eax, ebp, [ebp], [ebp+4], [ebp+8])\}$:

```
xor  eax,eax             ; eax = 0            ***
or   eax,[ebp]           ; eax = eax OR [ebp]
xchg eax,[ebp+4]         ; eax <-> [ebp+4]     ***
mov  [ebp],eax           ; [ebp] = eax        ***
add  eax,[ebp+4]         ; eax = eax + [ebp+4]
mov  [ebp+12],eax        ; [ebp+12] = eax
inc  dword ptr [ebp+4]   ; [ebp+4] = [ebp+4] + 1
```

After obfuscation the context was changed to $\{(eax, ebp, [ebp+4], [ebp], [ebp+12])\}$, so the typical technique of obfuscation was used: data reordering. The data obfuscation was gained through addition to the context some fields from the local memory. Because of the technical reasons only addition of local memory makes sense (most often it is stack), which holds usually arguments of call and local variables.

In opposition to registers of a processor, which are purely static resources, local memory can be extended with some reasonable limits, adding new elements of context allowing to efficiently obfuscate and manage data.

## 5.6 Summary of Theoretical Background

From the present theoretical background the following conclusions can be drawn:

- adding partially chaotic code into a program and changing a context, we get good method of protection of program code against analysis by human, but less resistant to analysis with automatic methods
- insertion of opaque constructs into the obfuscated code protects it well against automatic analysis, but not always protects against analysis by human
- efficient method of obfuscation must use at the same time modification of context and insertion of opaque constructs

The way of concatenation of changing context and inserting opaque constructs is not determined. It can be done in some different ways. It is quite sure, that an obfuscation process without one of these elements, would give a weak protection of programs.

# Section 6

# Algorithm of Obfuscation

KNOWING the two basic elements: obfuscating transformations and results of research on the stucture of typical programs, an universal method of creation of algorithms of obfuscation working on the machine code level can be given. Using this method a sample algorithm of obfuscation was designed and implemented for two typical machines having different architecture. It was used next for quality tests and analytical measures.

## 6.1 Algorithm Creation Method

Summarizing the analysis of problem of code obfuscation on the machine level we can assume, that the whole process can be reduced to the four activities (chapter 5.1):

- reordering of program's instructions
- reordering of blocks of the program
- exchange of equivalent fragments of the program
- inserting of an additional code

All presented activities are mutually independent, which does not mean, that change of execution order can give the same final result in the form of identical obfuscated code. Yet it means, that as the result of any activity we get a program, we can be input for any one of them.

If we mark all activities with capital letters (table 6.1), then a sequence of such letters will describe scenario of obfuscation. For an example the scenario **S.C.I.R.** means obfuscation with all four activities and scenario **I.I.I.** triple obfuscation with insetion of an additional code only (not always with the same parameters describing the internal work of the activity).

Table 6.1: Symbols of activities of code obfuscation.

| **Activity** | **Description** | |
|---|---|---|
| insructions reordering | **S** | `swap` |
| exchange of fragments | **C** | `change` |
| insertion of additional code | **I** | `insert` |
| blocks reordering | **R** | `reorder` |

### 6.1.1 Instructions Reordering

Instructions reordering is an easy technique of obfuscation, which changes the sequence of execution of instructions not depending on eachother. Such general definition allows to move even the whole groups of instructions, that's why it is assumed additionally, that this kind of reordering cannot insert any instruction into the obfuscated program (chapter 5.4).

**Example 6.1** *Given program in the assembler of Intel 80386 processor:*

```
     mov  ecx,[esi]    ; ecx = [esi]
     cmp  eax,ebx      ; eax = ebx ???
     jnz  jump         ; no => jump
     mov  edx,1        ; edx = 1
     jmp  cont        ; => cont
jump:
     mov  edx,2        ; edx = 2
cont:
```

was obfuscated with instructions reordering:

```
     cmp  eax,ebx      ; eax = ebx ???
     mov  ecx,[esi]    ; ecx = [esi]
     mov  edx,2        ; edx = 2
     jz   jump         ; tak => jump
     jmp  cont        ; => cont
jump:
     mov  edx,1        ; edx = 1
cont:
```

It can be seen, that reordering allows to modify some instructions (the conditional jump), but the total number of instructions remained unchanged.

The above example shows, that in general instructions reordering is not quite trivial. The easiest implementation of this part of obfuscating algorithm is to find such pairs of instructions, between which (on the base of condition from chapter 5.1):

- does not occurr dependencies of type „previous writes - next reads"[28]
- does not occurr branches
- there is no instruction, being destination of a jump[29]

More complex algorithms should consider dependencies between groups of instructions (not only pairs) and possibilities of reordering and changing of conditional and unconditional branches.

[28] An example of such dependency on the register eax is pair of instructions:

```
mov  eax,[esi]
add  ebx,eax
```

[29] It simplifies the algorithm a lot.

### 6.1.2 Blocks Reordering

In opposition to the instructions reordering, in blocks reordering we allow to insert some additional jumps into the obfuscated program. The goal of this type of reordering is to make analysis of program by human more difficult. It is well known from research [31], that a jump causes temporal stop of analysis, break of so called *flow state* in the brain of analyzing person and requires to move the program to the destination of a jump. In the final result overall performance of code analysis drops significantly.

More advanced methods could use insertion of conditional jumps, which do not insert a new meaning to the obfuscated program, because every replicated branch contains exactly the same part of the program. Obfuscating both replicated branches with different methods and adding nested jumps can also make analysis much difficult.

**Example 6.2** *Given program in the assembler of Motorola 68000 processor:*

```
     move.w d1,d2      ; d2 = d1
     add.w  d3,d2      ; d2 = d2 + d3
     beq    jump       ; d2 = 0 ?, yes => jump
     move.w d0,d2      ; d2 = d0
jump:
     move.w d2,(a1)    ; (a1) = d2
```

was obfuscated with blocks reordering:

```
     bra    jump1      ; => jump1
jump:
     move.w d2,(a1)    ; (a1) = d2
     bra    cont      ; => cont
jump2:
     add.w  d3,d2      ; d2 = d2 + d3
     beq    jump       ; d2 = 0 ?, yes => jump
     bra    jump3      ; => jump3
jump1:
     move.w d1,d2      ; d2 = d1
     bne    jump2      ; d2 = 0 ?, no => jump2
     add.w  d3,d2      ; d2 = d2 + d3
     beq    jump       ; d2 = 0 ?, yes => jump
jump3:
     move.w d0,d2      ; d2 = d0
     bra    jump       ; => jump
cont:
```

After the obfuscation the program became two times longer and the number of nodes in control flow graph increased sinificantly. The additional instruction of conditional branch „`bne skok2`" introduces a fictitious condition, which after further obfuscation with different methods cannot be removed easily even using automatic methods.

### 6.1.3 Exchange of Fragments

There is a huge number of algorithms, which can be implemented in many different ways (table 6.2). Such diversity of implementation comes very often even from the mathematic transformations (examples in table 6.3). Identifying known fragmens one can exchange them on an alternative programs. If there are many possible solutions for one often occurring fragment of an algorithm, a random replacement will introduce some difficulties into the analysis of the obfuscated program. The full reserve transformation of such inserted fragments requires to create exactly the same database, like the one used during obfuscation.

Table 6.2: Examples of fragments of programs in the assembler of Intel 80386 processor, which can be exchanged.

| **Program** | **Alternative** |
|---|---|
| `mov  eax,0` | `xor  eax,eax` |
| `xchg eax,ebx` | `mov  edx,eax`<br>`mov  eax,ebx`<br>`mov  ebx,edx` |
| `    cmp  eax,0`<br>`    jge  skok`<br>`    neg  eax`<br>`skok:` | `cdq`<br>`xor  eax,edx`<br>`sub  eax,edx` |

Table 6.3: Examples of expressions and their alternatives.

| **Expression** | **Alternative 1** | **Alternative 2** |
|---|---|---|
| $ab+b\overline{c}$ | $\overline{a}+\overline{b}+b\overline{c}$ | $\overline{a}+\overline{b}+c$ |
| $(a+b)*c-a*b$ | $(c-b)*a+b*c$ | $(c-a)*b+a*c$ |
| $\sin(\alpha-\pi)$ | $-\sin(\alpha)$ | $\cos(\alpha-\frac{\pi}{2})$ |

**Example 6.3** *Given fragment of a program in the assemblerze of Intel 80386 processor:*

```
mov  eax,ebx    ; eax = ebx
add  eax,ecx    ; eax = eax + ecx
imul eax,edx    ; eax = eax * edx
imul ebx,ecx    ; ebx = ebx * ecx
sub  eax,ebx    ; eax = eax - ebx
```

was obfuscated exchanging the way of calculation of detected expression (we assume that full overflow occuring analysis was done):

```
mov  eax,edx    ; eax = edx
sub  eax,ecx    ; eax = eax - ecx
imul eax,ebx    ; eax = eax * ebx
imul ecx,edx    ; ecx = ecx * edx
add  eax,ecx    ; eax = eax + ecx
```

getting the fragment returing in the register *eax* always the same result.

Realization of exchanging fragments of program requires performing the full control and data flow analysis, optimizing of obtained structures, next comparing with strucures prepared on the base of created database. Insertion of changed instructions is a reverse process, allowing to generate program on the base of matched structures. In addition to matching, in case of arithmetic operations, obfuscating program must make full analysis of possibilities of overflow occurences. Also lot of fragments require to satisfy some conditions, applying to the state of context on the end of obfuscated fragment (chapter 5.1).

### 6.1.4 Insertion of Code

To the group of activities, described with single name ”insertion of code”, we classified three methods inserting code into the obfuscated program (chapter 5.3):

- insertion of any instructions changing not used locally part of context
- insertion of reversible operations changing the used part of context
- insertion of opaque constructs

The main property of these methods is ability of making decision about the insertion only on the base of information about current data flow (context) and analysis of small program fragment (2-3 instructions). Information about local context is required, because inserted instructions cannot damage the source program.

The problem of efficient analysis of data flow was described in many books about construction of compilers and the example of creation of fast algorithm can be found in [61], page 244. Implementing such an algorithm we get full information about local context in the obfuscated program.

**Example 6.4** *Given program in the assembler of Intel 80386 processor, was given to data flow analysis to get information about local context context* $\{(eax, ebx, ecx, edx, ef)\}$*:*

| *program* | *eax* | *ebx* | *ecx* | *edx* | *ef* |
|---|---|---|---|---|---|
| `mov eax,10` | O | | X | X | |
| `mov ebx,20` | X | O | X | X | |
| `add eax,ecx` | X | X | X | X | O |
| `sub ebx,edx` | X | X | | X | O |
| `cmp eax,ebx` | X | X | | | O |
| `jnz jump` | X | | | | X |
| `add eax,30` | X | | | | O |
| `jump:` | | | | | |
| `mov ecx,eax` | X | | O | | |
| `mov edx,40` | X | | X | O | |
| `sub edx,ecx` | X | | X | X | O |

The fields descibing current state of an element of context have the following meaning:

- O – a new value is transferred into the element of context
- X – an element of context holds a value important for the program
- empty field – an element of context is not used currently by the program

Two variants of represention of information about local context are possible: state of context after of before execution of given instruction (example above). In the variant ”after” there are two possibilities of code insertion changing value of an element without a disturbance:

- if the element is described by an empty field, additional code can be inserted before and after current instruction
- if the element is descirbed by the field „O”, additional code can be inserted only before current instruction

**Example 6.5** *Into the fragment of program from the previous example:*

```
mov  eax,10     ; eax = 10
mov  ebx,20     ; ebx = 20
add  eax,ecx    ; eax = eax + ecx
```

some additional instructions chaning not used part of context were inserted:

```
add  eax,ecx    ; eax = eax + ecx ***
add  ebx,ecx    ; ebx = ebx + ecx ***
mov  eax,10     ; eax = 10
sub  ebx,eax    ; ebx = ebx - eax ***
mov  ebx,20     ; ebx = 20
add  eax,ecx    ; eax = eax + ecx
```

The final result of such insertion is frequent disturbance in attention of a person analyzing the correct control path of a program, which causes significant slow down of the analysis process.

In the similar way, using information about local context usage, we implement insertion of reversible operations on the used elements and insertion of opaque constructs. The general methodology of implementation of such methods was described in the previous chapter.

## 6.2 Sample Algorithm of Obfuscation

On the base of presented method a sample algorithm of obfuscation was created, which is designed to work on COSH machines (in Treleaven's classification), which are most of present computers.Two methods of obfuscation were selected for direct implementation: insertion **I** and blocks reordering **R**, because these two have the greatest influence on the quality of the obfuscation process.

Similarly like in [72] the whole program was treated as a set of procedures, where inter- and intraprocedural obfuscation can be distinguished. Sample algorithm is a solution for obfuscation of procedures only actually protecting small fragments of programs.

### 6.2.1 Entry Assumptions

To precisly describe the area of application of sample algorithm some entry assumptions were made:

- the input of the algorithm is a high-level programming language single function or subroutine of low-level programming language, given in the form of sequence of instructions[30]
- the obfuscating context consists of registers of processor and local stack
- for the obfuscated context a full data flow analysis was made (like in the example 6.4)

[30] It is not a critical limitation. Extension of the algorithm on many functions is mostly an engineering work.

- the influence of external calls on the context is known (if such a calls occurr in the function)
- obfuscated program does not contain instructions using elements out of obfuscated context
- only static jumps are present in the program (with hardcoded address)
- output context of the program is a single distinguished register or empty set
- the program does not contain indirect references (using pointers) to the local data, stored on the stack
- the following parameters of the algorithm are given:
  1. Rescaling factor – how much the output code must be longer than input code[31]:
  $$S > 1$$
  2. Frequency of reordering – how often a block reordering must be done, $R_B$:
  $$0 \leqslant R_B < 1$$
  3. Logic flags of code insertion: instructions changing not used part of context $I_N$, reversible operations $I_D$ and opaque constructs $I_O$ (zero means no insertion):
  $$I_N, I_D, I_O \in \{0, 1\}$$
  4. Flag of occurrence of dependecies in instructions changing not used part of context:
  $$I_Z \in \{0, 1\}$$

Parameters $S$ and $R_B$ come from the theoretical analysis of insertion (section 5.3) and reordering of instructions (section 5.4). Remaining parameters were added for additional experiments. The ability of switching on/off particular types of code insertion allowed to empirically check the influence of given method on the final shape of the obfuscated program (chapter 6.4.1).

If the input of the algorithm will be a sequence of instructions $d_I$ instructions long, then after obfuscation we should get a sequence long about $d_O = S \times d_I$ instructions (inaccuracy comes from the integrity of opaque constructs, which usually are longer then one instruction).

### 6.2.2 Basic Elements

The table 6.4 shows the global objects used by the sample algorithm, required by the way it works.

The algorithm itself consists of the initializing part, main loop, finishing part and a set of auxiliary functions. The most importatnt auxiliary function in a subroutine doing full analysis of context (already mentioned data flow analysis) and a procedure balancing context according to given reversible operations.

**Example 6.6** *Given context of the place of branch $\{(ebx, eax-10, ecx+ebx, edx+20)\}$ and the instruction „`add ebx,ecx`”, required the original context $\{(ebx, ecx)\}$. The procedure of context balancing returned the following program:*

[31] The upper limit depends on the implementation (available resources).

Table 6.4: Global objects used by the algorithm of obfuscation.

| Object | Type | Contents |
|---|---|---|
| $L_S$ | list | addresses of jumps "forward" |
| $L_O$ | list | operations executed on the current context |
| $N$ | int | number of instructions in the source program |
| $N_B$ | int | number of reordered blocks |
| $M$ | int | current number of instructions in the obfuscated program |
| $N_I$ | int | current number of obfuscated instructions of source program |

```
xchg eax,ebx    ; eax <-> ebx
sub  ecx,ebx    ; ecx = ecx - ebx
add  ebx,ecx    ; ebx = ebx + ecx
```

The easiest method to balance a context is to reverse all operations executed on the source context and insert all operations performed on the destination context. It is not an optimal solution, because it is not always required to reverse all operations and it is sometimes possible to bind tow or more operations or even ignore some of them, depending of the instruction requirements.

### 6.2.3 The Structure of the Algorithm

The initialization of the code obfuscation algorithm consists of the following steps:

- assignment of the start values to the gloal objects: to numbers $N_B$, $M$, $N_I$, value 0; to lists $L_S$, $L_O$ meaning "an empty list"; to number $N$ value equal to the number of all instructions in the source program
- obtaining the base address and allocation of some additional local variables (data obfuscation)
- allocation of memory for the obfuscated code
- execution of the data flow analysis (calculation of the local context "usage" for every instruction in the source program)
- starting the main loop of the algorithm

The construction of the obfuscation algorithm's main loop is base on the theorem 3.4, allowing to obfuscate the source program step by step. There are some actions in the algorithm which need a comment:

1. The parameter $S$ says how many additional instruction should be inserted. In the algorithm, which processes a single instruction at a time, the value of this parameter is translated onto a condition of insertion, saying if we should or should not insertd in the given moment of obfuscation process. The condition, marked as $C_I$, has the form:

   $$C_I = (M < N_I \times S)$$

   As long as $C_I$ is true the algorithm uses insertion. Every inserted instruction increases the value of variable $M$. If the inserted instruction is a reversible operation, information about the insertion is added to the list $L_O$, to make reverse action possible.

2. Usage of blocks reordering in the algorithm of obfuscation working instruction by instruction requires to create a condition, saying if in the given step a reordering must be done or not. In the simplest case the condition uses the current number of inserted reorderings and has the form:

   $$C_R = (\frac{N_B}{N_I} < R_B)$$

   During execution of reordering at least one instruction is inserted (a jump), that's why in addition the condition $C_I$ should be checked as well. As long as $C_I$ and $C_R$ are true, the algorithm should make reordering.

3. During the obfuscation of the source program there are moments, when two control paths gather in the single instruction. Because of separate obfuscation of these two paths, they have different output contexts. Using the context balance procedure will make contexts the same, making correct input context for the next instruction of the obfuscated program (see example 6.6). Without context balancing we would observe the effect shown in the example 3.7. In case of machine programs obfuscation gathering of two control paths can occurr when:

   - a jump occurred from the part already obfuscated to currently obfuscated instruction
   - reversible operations were made on the input context of current intruction
   - current instruction is a jump into already obfuscated part

4. It can be implied from the previous point, that the obfuscating algorithm must manage all jumps occurring in the source program, heading from the processed part into already obfuscated part. To store these jumps we created the list $L_S$. Unconditional jumps must be treated in a special way, because they do not pass context into the next obfuscated instruction (all other instructions do it). In the final effect the whole obfuscated context must be canceled, what is done be emptying of the list $L_O$.

**Algorithm 6.1** *Given source program and structures prepared during initialization execute all steps of the algorithm's main loop, shown on the figure 6.1.*

1. *If conditions $C_R$ and $C_I$ are true, do blocks reordering, for an example changing randomly the destination address to random number form the range $0 ... N \times S$.*

2. *If there are any jumps on the list $L_S$ to the current instruction, for every jump generate on the base of list $L_O$ a code balancing the context from the place of jump, to the current context, then remove the jump from the list $L_S$.*[32]

3. *Balance context to the original shape (if obfuscated instruction needs it), by reversing all operations from the list $L_O$, which are connected with the context of the instruction.*

4. *If the current instruction is a jump "backward"*[33]*, balance context from the form made by operations from the list $L_O$, to the shape stored in the destination place of the jump.*

5. *If the current instruction is a jump "forward"*[34]*, add it to the list $L_S$.*

[32] The optimal algorithm of context balancing depends on implementation.

[33] To the already processed address.

[34] To the address, which is not processed yet.

6. *Store current contents of the list $L_O$, to make possible context balancing in the next steps of obfuscation.*

7. *Copy current instruction to the destination program.*

8. *As long as the condition $C_I$ is true, insert additional instructions into the destination program (described in the next section).*

9. *If the copied instruction was unconditional jump, remove all operations from the list $L_O$.*

More precise description of steps executed by the algorithm requires some references to properties of used architecture and a choosen implementation of the algorithm. Because of low scientific value such description was omitted.

The main loop of the algorithm is executed sequentially for every instruction of the source program. The algorithm finishes by execution of the final part, consisted of the following steps:

- calculation of the real size of obfuscated program
- insertion in the empty cells of the destination program, which remained empty between reordered blocks, a randomly generated code, to make direct analysis after a dissassembly more difficult

The correctness of the algorithm was prooved in the following theorem:

**Theorem 6.1** *Transformation of a program $P_S$ into a program $P_D$, made by the algorithm 6.1 is an obfuscating transformation.*

**Proof.** On the input of the main loop the program $P_S$ can be treated as a concatenation of the already obfuscated part $P_{S1}$ and not yet obfuscated part $P_{S2} = (I_n, I_{n+1}, I_{n+2}, ..., I_N)$:

$$P_S = P_{S1}|P_{S2}$$

Let program $P_D$ be current result of the obfuscation process (obfuscated part of $P_{S1}$)[35]. Let program $P_O$ contain all added so far reversible operations, stored on the list $L_O$, and program $P'_O$ operations reversing the effects of insertions of instructions from $P_O$. Let $\mathcal{T}$ be an obfuscating transformations representing transformation from the algorithm 6.1. Thus in the given moment of obfuscation:

$$\mathcal{T}(P_{S1}) = P_D$$

so changes in program $P_D$ must satisfy conditions of definition of obfuscating transformation. The algorithm was divided on many steps, which are separate obfuscating transformations[36]. The steps of the algorithm do following transformations of the program $P_D$:

1. The block reordering in the proposed solution consists of insertion of an unconditional jump:

$$P_D := P_D|J_D$$

[35] On the beginning of the obfuscation process $P_{S1}$ and $P_D$ are empty programs.

[36] Because they take a program on the input, and put an obfuscated program on the output.

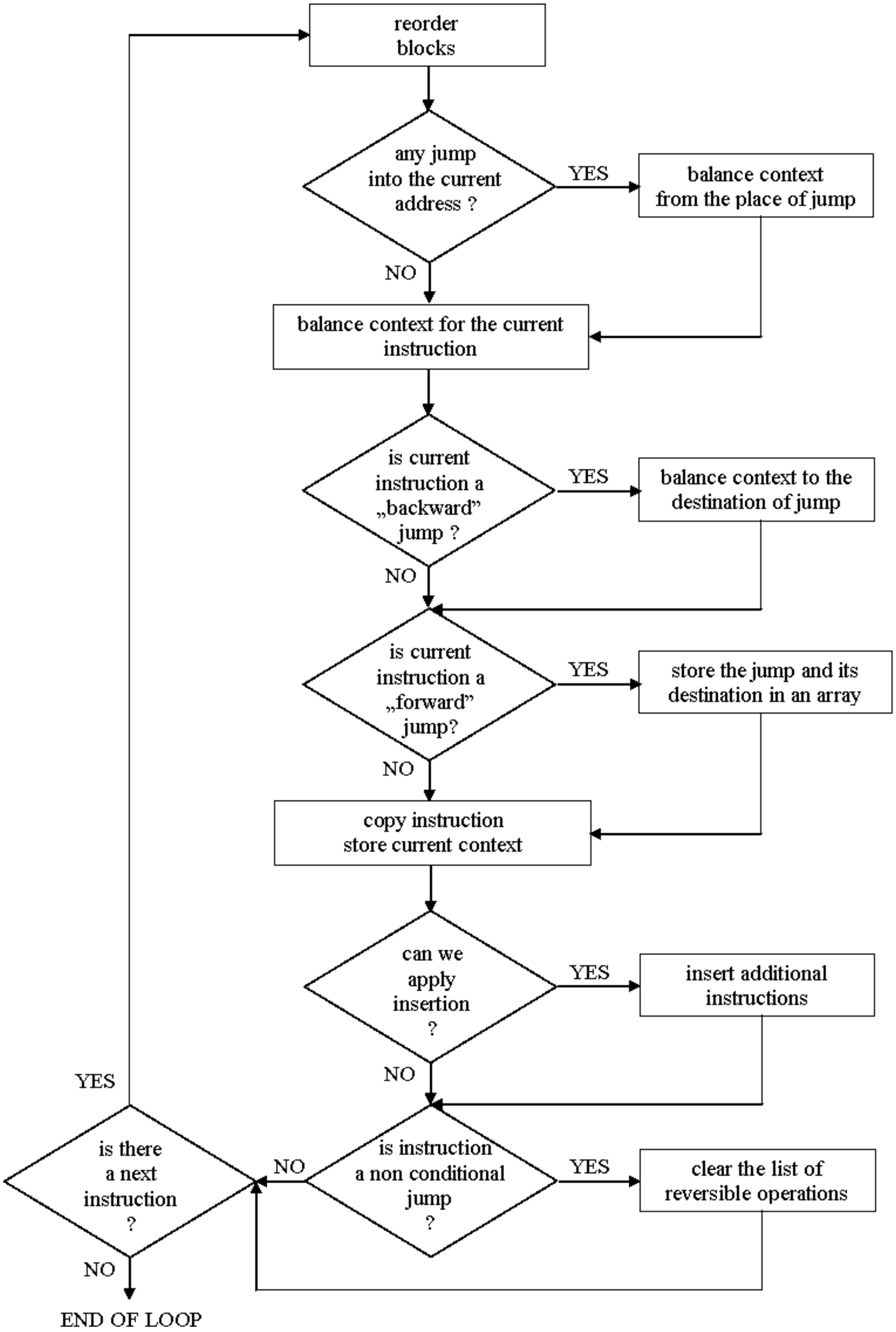

reorder blocks
any jump into the current address ?
YES
balance context from the place of jump
NO
balance context for the current instruction
is current instruction a „backward" jump ?
YES
balance context to the destination of jump
NO
is current instruction a „forward" jump?
YES
store the jump and its destination in an array
NO
copy instruction store current context
can we apply insertion ?
YES
insert additional instructions
NO
YES
is there a next instruction ?
NO
is instruction a non conditional jump ?
YES
clear the list of reversible operations
NO
END OF LOOP

Figure 6.1: The main loop of the algorithm of code obfuscation.

and optionally any program, skipped by the inserted jump:

$$P_D := P_D|P$$

The inserted jump does not change the context of obfuscated program, so the transformation according to the theorem 3.3 is obfuscating transformation.

2. Balancing of context from the place of jump from the list $L_S$ in the simplest implementation (shown in the example 6.6) is the addition of program $P'_O$ and program $P_S$, which is the program $P_O$ stored in the place of jump:

   $$P_D := P_D|P'_O|P_S$$

   and after the addition:

   $$P_O := P_S$$

   because all operations from the original $P_O$ were canceled. According to definition 3.11 and lemma **??** given transformation is obfuscating transformation (the context is balanced and equivalence is preserved).

3. Balancing of the current context to the input context of $P_{S2}$ in the general case can be reversing of all inserted reversible operations:

   $$P_D := P_D|P'_O$$

   Yet in practice we insert only these operations, which modify the part of context used by the first instruction of program $P_{S2}$ (instruction $I_n$). Every inserted operation is removed from the program $P_O$. The step uses conclusions from the theorem 3.4, described in the example 3.7.

4. The execution of this step is the same as execution of step number 2.

5. The program $P_D$ is not changed, only the program $P_O$ is stored and the destination address on the list $L_S$.

6. Step does not change the program $P_D$.

7. After execution of steps 2 – 4 the input context of program $P_{S2}$ was balanced, so in the simplest case in this step we add instruction $I_n$ to the program $P_D$:

   $$P_D := P_D|I_n$$

   Obfuscated part of the context preserves conditions, allowing to treat this step as an obfuscating transformation (thanks to the remaining part of the program $P_O$).

8. Independently from the kind of used insertion (details in the next section) this transformation is only an addition of some instructions:

   $$P_D := P_D|P_I$$

   If a reversible operation is inverted, it is added to the program $P_O$ as well:

   $$P_O := P_O|P_I$$

   Correctness of such transformation was shown in the chapter 5.3.

9. Does not change the program $P_D$. The only effect of this step is removal of all instructions from the program $P_O$.

After finishing of the obfuscation process $P_{S1} = P_S$, thus we have:

$$\mathcal{T}(P_S) = P_D$$

which means that the program $P_S$ was obfuscated to the form $P_D$ according to conditions of definition of obfuscating transformation. ■

### 6.2.4 Insertion of Instructions

Before insertion of instruction we must choose, which kind of insertion should be applied. From the different variants of choosing we propsed a random drawing, which can make even distribution of all methods of insertion in the obfuscated program. The particular algorithms of insertion consist of the following steps:

1. Insertion on the free elements of context – done, if $I_N = 1$.
    - draw a not used element of context
    - draw an instruciton to insert
    - draw an additional elements from the whole context (if required)
    - if $I_Z = 1$, then add dependency between last inserted instruction of this kind and currently inserted (if possible)
    - add instruction to the program
2. Insertion of reversible operations – done, if $I_D = 1$.
    - draw an used element of context
    - draw an operation to insert
    - draw an additional elements from the whole context (if required)
    - add the operation to the list $L_O$
    - add instruction to the program
3. Insertion of the opaque constructs – done, if $I_O = 1$.
    - draw a not used elements of context for execution basis
    - draw a place of jump from the opaque construct
    - draw type of the opaque construct
    - insert instructions of the construct, using drawn elements of the context

## 6.3 Implementation of the Algorithm

The implementation of presented algorithm is a more time-consuming task then a difficult one. Writing of machine code management procedures (loading and saving), procedures analyzing data flow, measuring procedures, etc. is required. To obtain good performance we designed proper data structures, storing all informations needed for algorithm's run.

### 6.3.1 Data Structures

In the table 6.5 we put the basic structures used by the implemented algorithm of obfuscation. The structure INSTR gives all information about instructions, which can occurr in the obfuscated program. The structure CONTEL stores temporal data about state and contents of a context element. The field *state* can have the following values:

- NOT_USED – when given element was not used yet
- VALUE – when given element holds a numeric value (ex. after loading instruction); the value is present in the *param* field
- USED – when the element was used and holds an unknown value
- POINTER – the element was used as a pointer to the external memory, not considered in the analyzed context[37]

The structure CONTEL gives information about the state of a context element and the list of structures OPER contains operations, which caused obfuscation of the original context in the given step of obfuscation. Every operation has its type and arguments, which are usually numbers or indexes to the table of context elements. The structure ADRJUMP is needed to analyze jumps ”forward” and the structure ADRMAP is the main structure mapping instructions from the source program onto places in the obfuscated program and holding data about their input context.

Table 6.5: Data structures used by the algorithm of obfuscation.

| **Name** | **Field** | **Contents** |
|---|---|---|
| INSTR | type<br>opcode<br>data | type of instruction (operation, branch, special)<br>code of instruction<br>arguments of instruction |
| CONTEL | type<br>state<br>param | type of context element (register or memory)<br>temporal state of the element<br>parameter of the state (ex. value) |
| OPER | type<br>data | type of operation executed on context<br>arguments of operation |
| ADRJUMP | jump<br>address | address of jump in obfuscated code<br>destination address in source code |
| ADRMAP | original<br>new<br>oplist<br>context | oryginal address of instruction<br>new address of instruction (after obfuscation)<br>current list of operations executed on context<br>current contents of context |

The source and obfuscated program are stores as the tables of objects INSTR. Lists $L_S$ and $L_O$ were substituted with regular tables too, mainly because of simplicity of data manipulation during the development phase of the algorithm.

[37] We distinguish pointers to avoid using them in reversible operations, which could uncover them easier.

### 6.3.2 Structure of The Program

The complete program of code obfuscation consists of seven independent modules:

1. Loading of the source program – transformation from the binary or text form into an array of structures INSTR.

2. Basic analysis of the program – check of parameters and local variables number and searching for addresses of jumps.

3. Data flow analysis – calculation of the local context for every instruction (array of structures CONTEL).

4. Optimization of data flow – calculation of the free parts of context (second pass of the data flow analysis algorithm).

5. Obfuscation of the program – creation of obfuscated program by sequential scanning of the source program.

6. Analytical measures of the obfuscated program – calculation of complexity measures (potency of obfuscation).

7. Saving of the obfuscated program – transformation of the program from the array of structures INSTR into binary or text form.

The obfuscation module has the auxiliary procedure of context balancing. This procedure inserts in the reverse order all reversible operations, which were performed on the given element of context's vector, taking into account dependencies from the other elements. The balancing of context was implemented by execution of this procedure for every element of the context's vector and insertion of all reversible operation from the appropriate list.

### 6.3.3 Comments to the Algorithm

The basic unit processed in the algorithm is an instruction. The natural unit of information for present computer is a byte, recently mostly a 32-bit long word. In the consequence, rescaling of the obfuscated program counted in bytes could not be right with the value of parameter $S$, given on the input of the obfuscation algorithm.

It can be seen in the implementation, that both source and obfuscated program are stored in the form of array of structures INSTR. Thus it is possible to obfuscate already obfuscated program. As the result the program can be obfuscated in the two ways:

- one time, with given values of parameters of obfuscation

- iteratively, with constant or variable values of parameters of obfuscation for every iteration

It is hard to score *a priori*, which from the possible ways can return a more efficient obfuscated program (for the approximately same value of the rescaling factor $S$). Thanks to multiple obfuscation of control flow it should be the iterative method, but for sure it is computationally more complex, because every time we must perform full analysis of data flow.

## 6.4 Efficiency of the Algorithm

The most interesting parameter of the algorithm of obfuscation is its efficiency – influence of obfuscation process on the ability to read the correct meaning of obfuscated program. Precisely this parameter can be measured only with appropriate empirical research (chapter 6.4.3), but analytical measures add also some information about quality of used obfuscation process.

### 6.4.1 Reference for Obfuscation Quality Tests

All test programs, presented in the appendix A were measured with three analytical measures defined in chapter 4. The results are reference base for changes introduced by an obfuscation process into a program structure. Results of these measurements are shown in table 6.6. Particular programs have very individual internal structure, which can be seen especially from the values of measures of **length** $E_L$ and **flow** $E_F$. The values of measure of **length** $E_L$ were normalized by division of a defined value by length of the measured program, given as a number of instructions.

Table 6.6: Values of complexity measures of test programs for Intel x86 and MIPS R4000 processors.

| **Program** | **x86**<br>$E_L$ | **x86**<br>$E_D$ | **x86**<br>$E_F$ | **MIPS**<br>$E_L$ | **MIPS**<br>$E_D$ | **MIPS**<br>$E_F$ |
|---|---|---|---|---|---|---|
| HASH | 0.88 | 1 | 3.0 | 0.71 | 2 | 1.0 |
| MATRIX | 0.87 | 3 | 4.0 | 0.74 | 4 | 1.9 |
| INSORT | 0.88 | 3 | 3.6 | 0.71 | 3 | 2.0 |
| BUBSORT | 0.90 | 3 | 4.7 | 0.72 | 4 | 2.4 |
| MAXARRAY | 0.92 | 4 | 4.5 | 0.62 | 4 | 3.5 |
| QSORT | 0.94 | 6 | 6.7 | 0.71 | 7 | 4.4 |
| SIMPROC | 0.88 | 11 | 2.4 | 0.74 | 14 | 1.2 |
| IDCT | 0.88 | 6 | 3.5 | 0.72 | 8 | 4.0 |
| CODETEST | 0.90 | 4 | 3.7 | 0.76 | 5 | 1.8 |
| DECODE | 0.93 | 1 | 7.3 | 0.74 | 2 | 4.3 |
| Average | **0.90** | **4** | **4.3** | **0.72** | **5** | **2.7** |

In case of empirical research as the reference base the average time of analysis by human of an not obfuscated program was taken. For final research only shorter test programs were selected. Results of reference data measures are shown in table 6.7. The measures were done only for programs compiled for Intel x86 processor, because of low popularity of assembler of MIPS R4000 processor.

### 6.4.2 Analytical Test Results

Given test programs were obfuscated with three methods, having different values of parameters controlling the algorithm. As the final result the average potency $\Pi_S$ for given process of obfuscation was calculated and shown in tables.

1. In the first method all implemented techniques of code insertion were used:
   - instructions changing not used part of context

Table 6.7: An average time of analysis of not obfuscated program by different groups of people.

| **Program** | **student** [**h**] | **engineer** [**h**] | **cracker** [**h**] |
|---|---|---|---|
| HASH | 0.21 | 0.20 | 0.15 |
| MATRIX | 0.33 | 0.31 | 0.18 |
| INSORT | 0.29 | 0.30 | 0.16 |
| BUBSORT | 0.26 | 0.24 | 0.17 |
| CODETEST | 0.27 | 0.29 | 0.16 |
| DECODE | 0.22 | 0.19 | 0.14 |
| Average | 0.26 | 0.25 | 0.16 |

- reversible operations changing used part of context
- opaque constructs

and following values of controlling parameters of the algorithm were set:

- rescaling factor – 10
- frequency of blocks reordering – 0.2
- dependencies in instructions changing not used part of context – yes

After obfuscation (table 6.8) test programs became more uniform, what can be seen by less differentiated values of measures of **length** $E_L$ and **flow** $E_F$, and depth level has increased in proportion to length of a program (measure of **depth** $E_D$). Average potency of programs grew twice less, than their length.

Table 6.8: Values of complexity measures of test programs after obfuscation with method 1.

| **Program** | **x86** $E_L$ | **x86** $E_D$ | **x86** $E_F$ | **MIPS** $E_L$ | **MIPS** $E_D$ | **MIPS** $E_F$ | **x86** $\Pi_S$ | **MIPS** $\Pi_S$ |
|---|---|---|---|---|---|---|---|---|
| HASH | 8.4 | 11 | 3.1 | 9.1 | 9 | 1.0 | 6.19 | 5.11 |
| MATRIX | 8.3 | 21 | 3.9 | 9.2 | 17 | 2.1 | 4.84 | 4.93 |
| INSORT | 8.1 | 14 | 4.2 | 9.2 | 10 | 1.7 | 4.01 | 4.71 |
| BUBSORT | 8.2 | 18 | 3.5 | 9.0 | 16 | 2.1 | 4.29 | 4.79 |
| MAXARRAY | 8.2 | 20 | 4.4 | 8.9 | 23 | 2.7 | 3.96 | 5.96 |
| QSORT | 8.2 | 80 | 3.4 | 9.0 | 73 | 1.9 | 6.52 | 6.85 |
| SIMPROC | 8.2 | 67 | 3.8 | 9.1 | 52 | 2.0 | 4.66 | 4.89 |
| IDCT | 8.1 | 42 | 3.6 | 9.0 | 37 | 2.0 | 4.74 | 4.88 |
| CODETEST | 8.4 | 28 | 3.4 | 9.1 | 20 | 2.0 | 4.75 | 4.69 |
| DECODE | 8.5 | 19 | 3.2 | 9.1 | 15 | 2.1 | 8.53 | 5.76 |
| Average | **8.3** | **32** | **3.7** | **9.1** | **27** | **2.0** | **4.89** | **5.17** |

2. In the second method insertion of opaque constructs was skipped, while other methods of insertion remained:

   - instructions changing not used part of context

- reversible operations changing used part of context

and following parameters controlling the algorithm were set:

- rescaling factor – 10
- frequency of blocks reordering – 0.0
- dependencies in instructions changing not used part of context – yes

After obfuscation (table 6.9) test programs became more uniform (relatively to measure of **length** $E_L$), but because of the same number of blocks in the control flow graph, inserted references to the local data have increased the values of **flow** $E_F$ measure in proportion to growth of length of the program. Absence of branches caused, that the depth level (measure of **depth** $E_D$) remained unchanged.

Table 6.9: Values of complexity measures of test programs after obfuscation with method 2.

| | **x86** | **x86** | **x86** | **MIPS** | **MIPS** | **MIPS** | **x86** | **MIPS** |
|---|---|---|---|---|---|---|---|---|
| **Program** | $E_L$ | $E_D$ | $E_F$ | $E_L$ | $E_D$ | $E_F$ | $\Pi_S$ | $\Pi_S$ |
| HASH | 7.9 | 1 | 28.0 | 9.4 | 2 | 5.5 | 5.44 | 5.58 |
| MATRIX | 8.1 | 3 | 36.9 | 9.3 | 4 | 12.4 | 5.51 | 5.70 |
| INSORT | 8.0 | 3 | 29.0 | 9.3 | 3 | 11.7 | 5.05 | 5.65 |
| BUBSORT | 8.0 | 3 | 40.3 | 9.3 | 4 | 13.0 | 5.15 | 5.44 |
| MAXARRAY | 8.1 | 4 | 35.6 | 9.0 | 4 | 16.9 | 4.91 | 5.78 |
| QSORT | 8.1 | 6 | 60.4 | 9.2 | 7 | 25.9 | 5.21 | 5.61 |
| SIMPROC | 8.2 | 11 | 21.8 | 9.3 | 14 | 8.3 | 5.47 | 5.83 |
| IDCT | 8.0 | 6 | 38.7 | 9.2 | 8 | 22.6 | 6.05 | 5.48 |
| CODETEST | 7.8 | 4 | 29.5 | 9.3 | 5 | 10.3 | 4.88 | 5.32 |
| DECODE | 8.2 | 1 | 52.7 | 9.2 | 2 | 28.0 | 4.68 | 5.65 |
| Average | **8.0** | **4** | **37.3** | **9.3** | **5** | **15.5** | **5.18** | **5.58** |

3. In the third method only insertion of instructions changing not used part of context was left and following controlling parameters of the algorithm were set:

- rescaling factor – 10
- frequency of blocks reordering – 0.0
- dependencies in instructions changing not used part of context – no

Results of obfuscation with this method (table 6.10) are not significantly different from the results of previous method. Even bigger uniformity (measure of **length** $E_L$) is caused by the high amount (about 90%) of instructions generated by the same algorithm. Smaller values of **flow** $E_F$, comparing to the method 2, are result of drawing of arguments in the inserted instructions, while in the source program local data are usually used more often than registers.

Resilience of obfuscated programs depends strongly on the types of used opaque constructs, which was shown on the examples in the appendix C. Cost of shown transformations in general is classified as cheap and empirical measurements determined, that increase of time of program's execution after obfuscation is proportional to the value set in rescaling factor.

Table 6.10: Values of complexity measures of test programs after obfuscation with method 3.

| Program | x86 $E_L$ | x86 $E_D$ | x86 $E_F$ | MIPS $E_L$ | MIPS $E_D$ | MIPS $E_F$ | x86 $\Pi_S$ | MIPS $\Pi_S$ |
|---|---|---|---|---|---|---|---|---|
| HASH | 8.7 | 1 | 11.7 | 9.1 | 2 | 3.0 | 3.93 | 4.61 |
| MATRIX | 8.7 | 3 | 22.1 | 8.6 | 4 | 8.7 | 4.51 | 4.73 |
| INSORT | 8.6 | 3 | 19.7 | 8.7 | 3 | 9.5 | 4.41 | 5.00 |
| BUBSORT | 8.7 | 3 | 25.1 | 8.6 | 4 | 10.4 | 4.34 | 4.76 |
| MAXARRAY | 8.6 | 4 | 28.8 | 8.4 | 4 | 11.0 | 4.58 | 4.90 |
| QSORT | 8.7 | 6 | 51.4 | 8.6 | 7 | 19.7 | 4.98 | 4.86 |
| SIMPROC | 8.6 | 11 | 17.8 | 8.6 | 14 | 7.3 | 5.06 | 5.23 |
| IDCT | 8.8 | 6 | 22.4 | 8.5 | 8 | 18.2 | 4.80 | 4.79 |
| CODETEST | 8.8 | 4 | 9.9 | 8.6 | 5 | 8.4 | 3.48 | 4.66 |
| DECODE | 8.7 | 1 | 27.0 | 8.7 | 2 | 19.3 | 3.68 | 4.75 |
| Average | **8.7** | **4** | **23.6** | **8.6** | **5** | **11.6** | **4.37** | **4.80** |

### 6.4.3 Empirical Test Results

Empirical quality tests of the sample algorithm were performed only on the test program DECODE, obfuscated with two different methods. In the first method program was obfuscated using blocks reordering (with frequency $R_B = 0.2$) and insertion of instructions changing not used part of context only (with dependencies between them switched off), with rescaling factore set to $S = 5$. Due to obfuscation the time of analysis increased few times (table 6.11), but still for most people from the group of crackers reading of the correct meaning of the program was not a difficult task. Partial results obtained by students could not be classified as correct.

Table 6.11: Summary of empirical research of the quality of code obfuscation for metod 1.

| Group | Number of people | Best time of answer [h] | Correct answers |
|---|---|---|---|
| students | 46 | — | — |
| engineers | 9 | 3 | 1 |
| crackers | 7 | 1 | 5 |

In the second method the full implementation of the sample algorithm was used. A simple opaque constructs were applied, based on logic and arithmetic operations and additional input parameters of obfuscated function (like in examples in appendix C). The same values of frequency of blocks reordering and rescaling like in the first method were set. The final program turned out to be much more resilient (table 6.12). No correct answer was obtained from students and engineers, although especially students put a lot of work into analysis and obtained quite significant partial results. Using ”manual” method *brute force* crackers analyzed the obfuscated program in quite a short time without bigger problems.

### 6.4.4 Summary of Quality Test Results

After carrying out of empirical tests it turned out, that an experienced person can unobfuscate the test program rescaled few times ($S = 5...15$) without any problems. In case of programs obfuscated with bigger value of rescaling factor $S$ participants proposed similar algorithms of unobfuscation, all partially automatic. General pattern of such an algorithm looks like this:

Table 6.12: Summary of empirical research of the quality of code obfuscation for metod 2.

| Group | Number of people | Best time of answer [h] | Correct answers |
|---|---|---|---|
| students | 14 | — | — |
| engineers | 7 | — | — |
| crackers | 5 | 2 | 3 |

- do a full data flow analysis – build control flow graph, then data flow graph and analyze local context for every instruction
- optimize the data flow graph – causes removal of some reversible operations and instructions changing not used part of context
- scan control flow graph to look for opaque constructs – partially automatic step, because decision about removal of a node of graph may be made efficiently only by human

In practice the unobfuscation process must be executed in an iterative way, because the presence of opaque constructs make single-step removal of all additional instructions impossible. It implies, that it is possible to create a good protection by insertion of high number of different opaque constructs into a program being rescaled as much as possible ($S \gg 1$). An example of such solution can be concatenation of method of control flow graph flattening (strenghtened with data aliasing [72]) and insertion of reversible operations and instructions changing not used part of context. Great advantage of such solution would be theoretically proven high resilience, disadvantage – small differentiation of type of used opaque constructs.

The results of empirical research can be compared with results of analytical measures. Making empirical research takes a lot of time, so finding general conclusions from such comparison could allow to make a quick estimations of efficiency of tested algorithm only on the base of analytic measures. In the table 6.13 are shown results of obufscation with two metods, made on the program DECODE. The simplified method is method 3 from the chapter 6.4.1 and method full is method 1. Efficiency was calculated by division of the best time of correct answer by the time needed for analysis before obfuscation of the program. This example shows, that a good measure of real efficiency is similar to the average potency of obfuscated program. Yet precise analysis of tables 6.8, 6.9 and 6.10 shows, that dependency between average potency and efficiency has more non-linear character. From the performed experiments it can be seen, that growth of potency of resilience of obfuscated program causes growth of efficienty of the obfuscation process.

Table 6.13: Summary of results of obfuscation algorithm quality measures for program DECODE.

| Method of obfuscation | Average potency | Resilience | Efficiency |
|---|---|---|---|
| simplified | 3.68 | weak | 14 |
| full | 8.53 | full | 28 |

It is hard to compare obtained results of empirical research to any research made so far. In present publications only pure theoretical analysis of efficiency of obfuscation algorithms was taken into account ([25]) or only empirical measures of cost of obfuscated transformation were made ([72]). From the theoretical background (chapter 5.3.3) efficiency of presented algorithm is not different from efficiency of the other algorithms ([26]).

# Section 7

# Summary and Conclusions

After the implementation of sample algorithm for different microprocessor architectures: Intel x86 and MIPS, we performed analytical and empirical measures to determine quality of the obtained process of obfuscation.

## 7.1 Summary

In comparison with other known algorithms of obfuscation, the proposed approach looks very promising (table 7.1). Its low complexity comes from easiness of semantic analysis of machine languages and simplicity of implementation of data flow analysis on the low level of programming. Other areas of comparison have the following meaning:

- 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

Collberg's algorithm is not very portable, because it was designed especially for use with the Java Virtual Machine. Proposed algorithm is most flexible being most isolated from high level programming languages structures. Chenxi Wang's algorithm uses very specific opaque constructs, making him not very scalable.

Table 7.1: Comparison of three algorithms of code obfuscation.

| **Property** | **Collberg** | **Chenxi Wang** | **Wróblewski** |
|---|---|---|---|
| **Complexity** | high | medium | low |
| **Portability** | no | yes | yes |
| **Flexibility** | medium | medium | high |
| **Scalability** | high | low | high |

In addition we should point in the compared algortihms on the following properties:

- Chenxi Wang – very high resilience (proven theoretically), but specific opaque constructs are not protected at all (ex. no reversible operations are inserted), so they are easy to catch by a human

- Collberg, Chenxi Wang – after obfuscation the program cannot be obfuscated once again (one-way change on control flow)
- Collberg – high number of parameters controlling the algorithm makes empirical research almost impossible

Removal of code inserted by three compared algorithms requires a construction of special optimizer for given type of processor. It can be read from long and often discussion on the discussion group `comp.compilers`, that creation of such a tool is not very realistic.

Proposed algorithm allows to obfuscate already obfuscated programs. Programs obufscated 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.

Thanks to low complexity of the algortihm, presented method can be applied in software watermarking and in computer viruses. Every next copy of virus can be obfuscated in a different way, covering characteristic places of a program.

The main drawback of all developed algorithms of obfuscation so far is the fact, that they remove all effects of code optimization, done by compliers. In modern processor it causes very ofter breaking of data processing stream, which slows down execution. That's way critical loops and highly optimized fragments should not be obfuscated.

## 7.2 Future Work

To develop presented method of obfuscation, we propose following paths of research:

- searching for methods of obfuscation of context between procedures, which would allow to obfuscate the whole programs
- research on influence of types of opaque constructs on resilience of obfuscated programs
- performing more detailed empirical research, to penetrate abilities of crackers environment
- adaptation of the algortihm for a VLIW type machine (*Very Long Instruction Word*), in which obfuscation has especially great impact on performance of obfuscated programs

Development of proposed method in showed directions would allow to obatin even more general and efficient algorithms, obfuscating whole programs in a complex way.

## 7.3 Conclusions

The proposed 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 specific properties of its architecture (like special instructions)[38].

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. Results of this research, made first time, add some information impossible to obtain only from pure analytical measures ([26], [72]). To estimate a work required to read the correct meaning of an obfuscated program, we made following assumptions (chapter 6.4.3):

[38] Convertion of implemented version for x86 processor on the MIPS processor took few hours of work only.

1. 10 to 100 experienced crackers can work simultaneously.
2. They can work about one year (350 days).
3. Obfuscated software includes different opaque constructs.
4. Each cracker works 4 hours daily (on average).
5. Average performance of a cracker in removing of opaque constructs is 10 constructs per hour.
6. The average opaque construct is 5 instructions long.
7. Additional code protecting opaque constructs makes final code 2 to 10 times longer.

After multiplication of given values, the estimated length of an obfuscated short program is from $1,400,000$ to $70,000,000$ instructions. Above example is extreme, but it ensures almost full protection of hidden code. Even such a long piece of code would execute quickly on present machines.

The main goal of the work, creation of equally 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 results.

# References

[1] F.B. Abreu, *Metrics for Object-Oriented Environment*, Proceedings of the Third International Conference on Software Quality, Lake Tahoe, Nevada, October 4-6, 1993, pp. 67-75

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988

[3] D.J. Albert, S.P. Morse, *Combating software piracy by encryption and key management*, IEEE Computer, April 1982

[4] F.E. Allen, *Control flow analysis*, SIGPLAN Notices 5(7):1-19, July 1970

[5] Ross J. Anderson, Fabien A. Petitcolas, *On The Limits of Steganography*, IEEE Journal of Selected Areas in Communications, 16(4):474-481, May 1998

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

[7] C.T. Bailey, W.L. Dingee, *A Software Study Using Halstead Metrics*, Bell Laboratories Denver, CO. 80234, 1981

[8] B.S. Baker, *An algorithm for structuring flowgraphs*, Journal of the ACM, 24(1):98-120, January 1977

[9] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, Ke Yang, *On the (Im)possibility of Obfuscating Programs*, Advances in Cryptology — CRYPTO'01, Springer Lecture Notes in Computer Science vol. 2139, pp. 1-18, Santa Barbara, CA, November 2001

[10] 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

[11] Manuel Blum, Sampath Kannan, *Designing programs that check their work*, Journal of the ACM, 42(1):269-291, January 1995

[12] David R. Chase, Mark Wegman, F. Kenneth Zadeck, *Analysis of pointers and structures*, ACM SIGPLAN Notices 25(6):296-310, June 1990

[13] Shyam R. Chidamber, Chris F. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, 20(6):476-493, June 1994

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

[15] Cristina Cifuentes, *The Impact of Copyright on the Development of Cutting-Edge Reverse Engineering Technology*, Proceedings of the Sixth Working Conference on Reverse Engineering, Atlanta, USA, October 1999, IEEE-CS Press, pp. 66-76

[16] Cristina Cifuentes, Doug Simon, *Procedural Abstraction Recovery from Binary Code*, Technical Report 448, Department of Computer Science and Electrical Engineering, The University of Queensland, September 1999

[17] Cristina Cifuentes, Antoine Fraboulet, *Interprocedural Data Flow Recovery of High-level Language Code from Assembly*, Technical Report 421, Department of Computer Science and Electrical Engineering, The University of Queensland, December 1997

[18] Cristina Cifuentes, Antoine Fraboulet, *Intraprocedural Static Slicing of Binary Executables*, Proceedings of the International Conference on Software Maintenance, Bari, Italy, October 1997, pp. 188-195, IEEE-CS Press

[19] Cristina Cifuentes, *Interprocedural Data Flow Decompilation*, Journal of Programming Languages, Vol 4(2), June 1996, pp. 77-99

[20] Cristina Cifuentes, *An Environment for the Reverse Engineering of Executable Programs*, Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society Press. Brisbane, Australia, December 1995, pp. 410-419

[21] Cristina Cifuentes, K.John Gough, *Decompilation of Binary Programs*, Software - Practice & Experience, Vol 25(7), July 1995, pp. 811-829

[22] Cristina Cifuentes, K.John Gough, *A Methodology for Decompilation*, Proceedings of the XIX Conferencia Latinoamericana de Informatica, Buenos Aires, Argentina, August 1993, pp. 257-266

[23] Cristina Cifuentes, Doug Simon, Antoine Fraboulet, *Assembly to High Level Language Translation*, Technical Report 439, Department of Computer Science and Electrical Engineering, The University of Queensland, August 1998

[24] Frederick B. Cohen, *Operating System Protection Through Program Evolution*, 1992

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

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

[27] Christian Collberg, Clark Thomborson, Douglas Low, *Breaking Abstractions and Unstructuring Data Structures*, IEEE International Conference on Computer Languages, ICCL'98, Chicago, IL, May 1998

[28] Christian Collberg, Clark Thomborson, *On the Limits of Software Watermarking*, Technical Report #164, Department of Computer Science, The University of Auckland (1998)

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

[30] 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)

[31] Tom DeMarco, Timothy Lister, *Peopleware: Productive Projects and Teams*, 2nd. ed., Dorset House Publishing Company, New York, 1999

[32] A. Deutsch, *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, SIGPLAN PLDI'94, Orlando, FL, ACM SIGPLAN Notices 29(6), June 1994, pp. 230-241

[33] J. Domingo-Ferrer, *Software run-time protection: A cryptographic issue*, Advances in Cryptology: EUROCRYPT'90, May 1990, Springer Lecture Notes in Computer Science vol. 473, 1991, pp. 474-480

[34] Rex Jaeschke, *Encrypting C source for distribution*, Journal of C Language Translation, 2(1), 1990

[35] Neil F. Johnson, Sushil Jajodia, *Computing practices: Exploring steganography: Seeing the unseen*, Computer, 31(2):26–34, February 1998

[36] James R. Gosler, *Software protection: Myth or reality?*, CRYPTO'85 — Advances in Cryptology, August 1985, pp. 140–157

[37] Satoshi Hada, *Zero-knowledge and Code Obfuscation*, ASIACRYPT'2000 — Advances in Cryptology, International Association for Cryptologic Research, Kyoto, Japan, 2000

[38] M.H. Halstead, *Elements of Software Science*, Elsevier North-Holland, 1977

[39] Warren A. Harrison, Kenneth I. Magel, *A complexity measure based on nesting level*, SIGPLAN Notices 16(3):63-74, 1981

[40] M. Hecht, J. Ullman, *A simple algorithm for global data flow analysis problems*, SIAM Journal of Computing, vol. 4, December 1975, pp. 519-532

[41] Amir Herzberg, Shlomit S. Pinter, *Public protection of software*, ACM Transactions on Computer Systems, 5(4):371-393, November 1987

[42] A. Herzberg, G. Karmi, *On software protection*, 4th Jerusalem Conference on Information Technology, Jerusalem, Israel, April 1984

[43] G.L. Hopwood, *Decompilation*, PhD Dissertation, University of California, Irvine, Computer Science, 1978

[44] B.C. Housel, *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*, PhD Dissertation, Purdue University, Computer Science, August 1973

[45] William Landi, Barbara Ryders, *A Safe Approximate Algorithm for Interprocedural Pointer Analysis*, Technical Report, Department of Computer Science, Rutgers University, New Brunswick, 1991

[46] William Landi, *Interprocedural Aliasing in the Presence of Pointers*, PhD Dissertation, Department of Computer Science, Rutgers University, New Brunswick, January 1992

[47] U. Lichtblau, *Decompilation of control structures by means of graph transformations*, Proceeding of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin, 1985

[48] Jean-Paul M.G. Linnartz, A.A.C. Kalker, G.F.G. Depovere, R.A. Beuker, *A reliability model for the detection of electronic watermarks in digital images*, Philips Research, Eindhoven, the Netherlands

[49] Douglas Low, *Java Control Flow Obfuscation*, Master of Science Thesis, Department of Computer Science, The University of Auckland (1998)

[50] Douglas Low, *Protecting Java code via code obfuscation*, ACM Crossroads, Spring issue 1998

[51] Josh MacDonald, *On Program Security and Obfuscation*, Technical report, Department of Computer Science, University of California, Berkeley, December 1998

[52] Stavros Macrakis, *Protecting source code with ANDF*, January 1993, `www.andf.org`

[53] Antonio Maña, Ernesto Pimentel, *An Efficient Software Protection Scheme*, IFIP Sixteenth Annual Conference on Information Security, Paris, France, June 2001

[54] Thomas J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering, 2(4):308-320, December 1976

[55] Siba N. Mohanty, *Entropy Metrics for Software Design Evaluation*, The Journal of Systems and Software, No. 2, 1981, pp. 39-46

[56] John. C. Munson, Taghi M. Kohshgoftaar, *Measurement of data structure complexity*, Journal of Systems Software, 20:217-225, 1993

[57] Jasvir Nagra, Clark Thomborson, Christian Collberg, *A Functional Taxonomy for Software Watermarking*, Australasian Computer Science Conference (ACSC2002), Melbourne, Australia, August 2002

[58] R. Ostrovsky, *An efficient software protection scheme*, Advances in Cryptology: CRYPTO'89, Berlin, August 1990, Springer, pp. 610-611

[59] Enrique I. Oviedo, *Control Flow, Data Flow and Programmers Complexity*, Proceedings of COMPSAC 80, Chicago IL, 1980, pp.146-152

[60] Fabien A. Petitcolas, Ross J. Anderson, Markus G. Kuhn, *Attacks on Copyright Marking Systems*, Second workshop on information hiding, Portland, Oregon, April 1998, pp. 218-238

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

[62] Henry Sallie, Dennis Kafura, *Software structure metrics based on information flow*, IEEE Transactions of Software Engineering, 7(5):510-518, September 1981

[63] Pamela Samuelson, *Reverse-engineering someone else's software: Is it legal?*, IEEE Software, January 1990, pp. 90-96

[64] T. Sander, Chr. Tschudin, *On Software Protection via Function Hiding*, Proceedings of the Second Workshop on Information Hiding, Springer Lecture Notes in Computer Science

[65] Luis Sarmenta, *Protecting Programs from Hostile Environments: Encrypted Computation, Obfuscation, and Other Techniques*,

[66] R. Sedgewick, *Implementing Quicksort programs*, Comm. ACM 21, pp. 847-857

[67] Doug Simon, *Structuring assembly programs*, Honours thesis, The University of Queensland, Department of Computer Science and Electrical Engineering, 1997

[68] Jan Stanisławski, *Wielki Słownik Angielsko-Polski*, Wiedza Powszechna, Warszawa 1966

[69] Frank Tip, *A survey of program slicing techniques*, Journal of Programming Languages, 3(3):121–189, September 1995

[70] 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

[71] Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson, *Protection of Software-based Survivability Mechanisms*, International Conference of Dependable Systems and Networks, Göteborg, Sweden, July 2001

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

[73] Hal Wasserman, Manuel Blum, *Software reliability via run-time result-checking*, Journal of the ACM, 44(6):826-849, November 1997

[74] S.P. Weisband, Seymour E. Goodman, *International software piracy*, Computer, 92(11):87-90, November 1992

[75] Elaine J. Weyuker, *Evaluating Software Complexity Measures*, IEEE Transactions of Software Engineering Volume 14, No. 9, September 1988

[76] Grzegorz Wróblewski, *General Method of Program Code Obfuscation*, Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2002, Las Vegas, USA, June 2002, pp. 153-159

[77] Shikun Zhou, Hongji Yang, William C. Chu, *Reverse Engineering Metrics: the Sixth Element*, Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2002, Las Vegas, USA, June 2002, pp. 160-167

# Appendix A

# Test Programs

For practical research and tests we have choosen 10 simple programs doing different tasks (table A.1). Programs were written in C programming language and compiled with typical compiler for given processor. Obtained executable code was obfuscated and measured. Below we included listings and short descriptions of all programs.

Table A.1: Programs used for research and tests of algorithm of obfuscation.

| **Name** | **Description** |
|---|---|
| HASH | calculation of 32-bit hash code from a string |
| MATRIX | multiplication of matrices |
| INSORT | sorting by simple insertion |
| BUBSORT | bubble sorting |
| MAXARRAY | searching of sub-array with max sum of elements |
| QSORT | quick sorting |
| SIMPROC | simulation of simple processor |
| IDCT | reverse discrete cosine transform |
| CODETEST | self-check of code by calculating a checksum |
| DECODE | decoding of a fragment of program |

## A.1 Program HASH

The program calculates a 32-bit hash code of a string of characters, ending with character of value 0. Algorithm of calculation uses one rotation and one xor of final sum and next character in the string.

```
long hash(unsigned char *str)
{
    unsigned long hash=0x12345678;

    while(*str!=0)
        hash=((hash<<9)|(hash>>23))^*str++;
    return hash;
}
```

## A.2 Program MATRIX

Program multiplies two marices: *tab*1 and *tab*2, which have dimension *m*. Result is put into matrix *tab*3.

```
void matrix(long **tab1,long **tab2,long **tab3,long m)
{
    long i,j,k,sum;

    for(i=0;i<m;i++)
        for(j=0;j<m;tab3[i][j]=sum,j++)
            for(k=0,sum=0;k<m;k++)
                sum+=tab1[i][k]*tab2[k][j];
}
```

## A.3 Program BUBSORT

It is a simple implementation of bubble sorting. A table of numbers of type *long* is sorted, which has length *len*.

```
void bubsort(long *tab,long len)
{
    long i,tmp,flag;

    len--;
    for(flag=1;flag!=0;)
    {
        flag=0;
        for(i=0;i<len;i++)
            if(tab[i]>tab[i+1])
            {
                tmp=tab[i];
                tab[i]=tab[i+1];
                tab[i]=tmp;
                flag=1;
            }
    }
}
```

## A.4 Program INSORT

It is an example of sorting by simple insertion. A table of numbers of type *long* is sorted, which has length *len*.

```
void insort(long *tab,long len)
{
    long i,j,k,pos;

    len--;
    for(i=0;i<len;i++)
    {
        for(pos=i,k=tab[i],j=i+1;j<len;j++)
            if(tab[j]<k)
            {
                k=tab[j];
                pos=j;
            }
        tab[pos]=tab[i];
        tab[i]=k;
    }
}
```

## A.5 Program MAXARRAY

The program finds in the array of integer numbers a continuous sub-array, which sum of number is maximum. The position of the table is returned by program and its size – by pointer *size*.

```
long maxarray(long *tab,long len,long *size)
{
    long i,sum,tmp,best,mpos,min,tsum,bpos;

    best=0; sum=0; *size=0; min=0; mpos=0;
    for(i=0;i<len;i++)
    {
        tmp=sum+tab[i];
        if(sum>=tmp)
        {
            tsum=sum-min;
            if(tsum>best)
            {
                best=tsum;
                bpos=mpos;
                *size=i-mpos;
            }
            if(tmp<min)
            {
                min=tmp;
                mpos=i;
            }
        }
        sum=tmp;
    }
    if((sum-min)>best)
    {
        bpos=mpos;
        *size=i-mpos;
    }
    return bpos;
}
```

## A.6 Program QSORT

It is a typical implementation of a quick sorting algorithm, based on [66]. The array *stack* holds arguments to avoid recursive calls.

```
void qsort(long *tab,long len,long *stack)
{
    long i,r=len/2,sp=0,l=len,v,j,tmp;

    stack[sp++]=l; stack[sp++]=r;
    while (sp!=0)
    {
        r=stack[--sp]; l=stack[--sp];
        if(r<=l)
            continue;
        v=tab[r]; i=l-1; j=r;
        for (;;)
        {
            while(tab[++j]>v)
                if(j==l)
                    break;
            while(v>tab[--i])
                if(i==0)
                    break;
            if(i>=j)
                break;
            tmp=tab[i];
            tab[i]=tab[j];
            tab[j]=tmp;
        }
        tmp=tab[i];
        tab[i]=tab[r];
        tab[r]=tmp;
        if(i-l>r-i)
        {
            stack[sp++]=l; stack[sp++]=i-1;
            stack[sp++]=i+1; stack[sp++]=r;
        } else {
            stack[sp++]=i+1; stack[sp++]=r;
            stack[sp++]=l; stack[sp++]=i-1;
        }
    }
}
```

## A.7 Program SIMPROC

Program is a simple simulation of a virtual processor. It executes code from an array of given length.

```
long simproc(long *code,long len)
{
    long ip=0,i,j,k;
    long reg[4];

    while(ip!=len)
    {
        switch(code[ip++])
        {
        case 100:
            i=code[ip++];
            if(i<0 ||i>3)
                return 0;
            reg[i]=code[ip++];
            break;
        case 200:
            i=code[ip++]; j=code[ip++]; k=code[ip++];
            if(i<0 ||i>3 ||j<0 ||j>3 ||k<0 ||k>3)
                return 0;
            reg[i]=reg[j]-reg[k];
            break;
        case 300:
            i=code[ip++]; j=code[ip++];
            if(i<0 ||i>3 ||j<0 ||j>=len)
                return 0;
            if(reg[i]==0)
                ip=j;
            break;
        case 400:
            i=code[ip++]; j=code[ip++];
            if(i<0 ||i>3 ||j<0 ||j>=len)
                return 0;
            if(reg[i]>0)
                ip=j;
            break;
        case 500:
            i=code[ip++]; j=code[ip++]; k=code[ip++];
            if(i<0 ||i>3 ||j<0 ||j>3 ||k<0 ||k>3)
                return 0;
            reg[i]=reg[j]*reg[k];
            break;
        case 600:
            i=code[ip++]; j=code[ip++]; k=code[ip++];
            if(i<0 ||i>3 ||j<0 ||j>3 ||k<0 ||k>3)
                return 0;
            if(reg[k]!=0)
                reg[i]=reg[j]/reg[k];
            else
                reg[i]=0;
            break;
        case 700:
            ip=len;
            break;
        default:
            return 0;
        }
    }
    return reg[0];
}
```

## A.8 Program IDCT

Program calculates two-dimensional (8x8) reverse cosine transform for integer values. Array *block* contains input and output data, array $c$ contains coefficients of transformation (fixed point was set on position $2^{12}$).

```
void idct(long *block,long **c,long *tmp)
{
    int i, j, k, v;
    long partial_product;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
        {
            partial_product = 0;
            for (k=0; k<8; k++)
                partial_product+= (c[k][j]*block[8*i+k])>>12;
            tmp[8*i+j] = partial_product;
        }
    for (j=0; j<8; j++)
        for (i=0; i<8; i++)
        {
            partial_product = 0;
            for (k=0; k<8; k++)
                partial_product+= (c[k][i]*tmp[8*k+j])>>12;
            v = partial_product;
            block[8*i+j] = (v<-256) ?  -256 :  ((v>255) ?  255 :  v);
        }
}
```

## A.9 Program CODETEST

The program calculates a control sum for given array and then enters an infinite loop if the calculated value is different from hardcoded one. To avoid detection of the moment of checking a non-trivial infinite loop was created and an extra loop on the end of procedure was inserted.

```
void codetest(long *code,long len)
{
    long i,sum=0;

    for(i=0;i<len;i++)
        sum=(sum+code[i])^code[i];

    if(sum!=0x12345678)
        for(i=0;i<len;i++)
        {
            sum+=0x12345678^i+code[i];
            i--;
        }

    for(i=0;i<len-1;i++)
        if(sum>0x12345678)
            code[i]=sum^i-code[i+1];
}
```

## A.10 Program DECODE

The program decodes a fragment of memory, given in the form of array of integer numbers without sign. The coding key is value given in parameter *key* and the decoded data. The decoding looks almost like obfuscated code, so it should be an easy object to hide in the process of obfuscation.

```
void decode(unsigned long *code,long len,unsigned long key)
{
    unsigned long i,j=0x87654321;

    for(i=0;i<len;i++)
    {
        j^=(code[i]^key)>>7;
        j^=(code[i]^key)<<25;
        key=j^key^0x12345678;
        code[i]=j;
    }
}
```

# Appendix B

# Research on Properties of Programs

Programs of most todays computers contain dependencies between neighbourhood instructions. These dependencies are present in data sources processed by given pair of instructions. Introduction of such model: instructions plus dependencies, allowed to obtain a simple algorithm of generation of obfuscated code. Yet numeric parameters of this model are probabilities of occurrences of these dependencies, which were determined empirically.

## B.1 Dependencies Between Instructions

To determine dependecies between instructions in a typical program we used executable code of test programs described in appendix A. Frequency of occurence of a dependency was examined in the function of distance between two instructions, marked as $d$. Value $d = 1$ means adjoining instructions. Test programs were compiled for two types of processors: Intel x86 and MIPS R4000. Results of research for Intel processor are shown in table B.1 and for MIPS processor – table B.2.

Table B.1: Dependencies between instructions in test programs for Intel x86 processor.

| **Program** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| HASH | 0.46 | 0.09 | 0.09 | 0.14 | 0.10 | 0.05 | 0.11 |
| MATRIX | 0.43 | 0.13 | 0.12 | 0.1 | 0.02 | 0.04 | 0.02 |
| INSORT | 0.32 | 0.11 | 0.00 | 0.18 | 0.05 | 0.14 | 0.02 |
| BUBSORT | 0.4 | 0.13 | 0.00 | 0.19 | 0.09 | 0.13 | 0.02 |
| MAXARRAY | 0.44 | 0.11 | 0.02 | 0.10 | 0.07 | 0.12 | 0.14 |
| QSORT | 0.45 | 0.10 | 0.08 | 0.20 | 0.08 | 0.13 | 0.02 |
| SIMPROC | 0.45 | 0.19 | 0.15 | 0.16 | 0.15 | 0.11 | 0.11 |
| IDCT | 0.38 | 0.25 | 0.15 | 0.06 | 0.05 | 0.05 | 0.01 |
| CODETEST | 0.41 | 0.17 | 0.03 | 0.24 | 0.10 | 0.05 | 0.12 |
| DECODE | 0.51 | 0.24 | 0.03 | 0.08 | 0.11 | 0.09 | 0.12 |
| Average | **0.43** | **0.15** | **0.07** | **0.15** | **0.08** | **0.09** | **0.07** |

Averaged arithmetically result for each processor is shown in the form of graph on the figures B.1 and B.2. It can be seen that in case of Intel processor dependencies between adjoining instructions occurr most often and the average value for larger distances is $\approx 0,21$. It means, that dependency occurrs in one of five instructions (on average).

Table B.2: Dependencies between instructions in test programs for MIPS R4000 processor.

| **Program** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| HASH | 0.44 | 0.07 | 0.07 | 0.08 | 0.00 | 0.00 | 0.00 |
| MATRIX | 0.40 | 0.07 | 0.02 | 0.02 | 0.00 | 0.00 | 0.05 |
| INSORT | 0.38 | 0.09 | 0.09 | 0.03 | 0.03 | 0.00 | 0.04 |
| BUBSORT | 0.46 | 0.16 | 0.00 | 0.05 | 0.00 | 0.02 | 0.00 |
| MAXARRAY | 0.41 | 0.04 | 0.02 | 0.02 | 0.04 | 0.02 | 0.00 |
| QSORT | 0.50 | 0.03 | 0.07 | 0.06 | 0.02 | 0.04 | 0.02 |
| SIMPROC | 0.43 | 0.10 | 0.02 | 0.02 | 0.04 | 0.06 | 0.00 |
| IDCT | 0.53 | 0.06 | 0.01 | 0.02 | 0.00 | 0.02 | 0.04 |
| CODETEST | 0.49 | 0.09 | 0.04 | 0.06 | 0.02 | 0.02 | 0.00 |
| DECODE | 0.62 | 0.03 | 0.06 | 0.06 | 0.00 | 0.00 | 0.00 |
| Average | **0.47** | **0.07** | **0.04** | **0.04** | **0.02** | **0.02** | **0.02** |

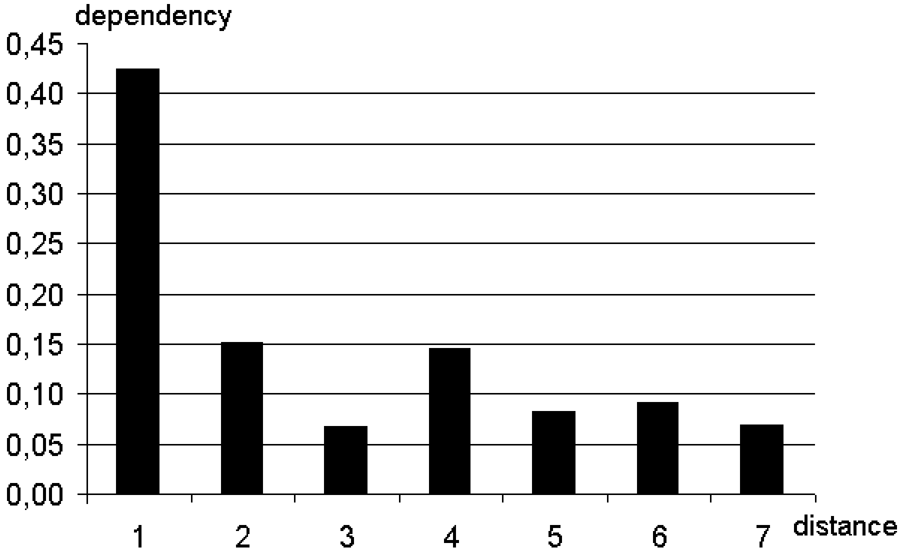


Figure B.1: Probability of dependency between instructions of program for Intel x86 processor in the function of distance between them.

Joined results from two processors were used in the dissertation (figure 5.2). We think, that result obtained in such a way is quite general, because processors Intel x86 and MIPS represent two extreme different architectures (CISC and RISC) with very different instruction set.

## B.2 Random Programs

Calculation of probabilities of dependencies in case of generated random programs was also very important. Tests were performed on two models: pure random (every instruction is drawn independently and added unconditionally to program) and random with dependencies (an instruction is drawn and an attempt to add some dependencies between the drawn instruction

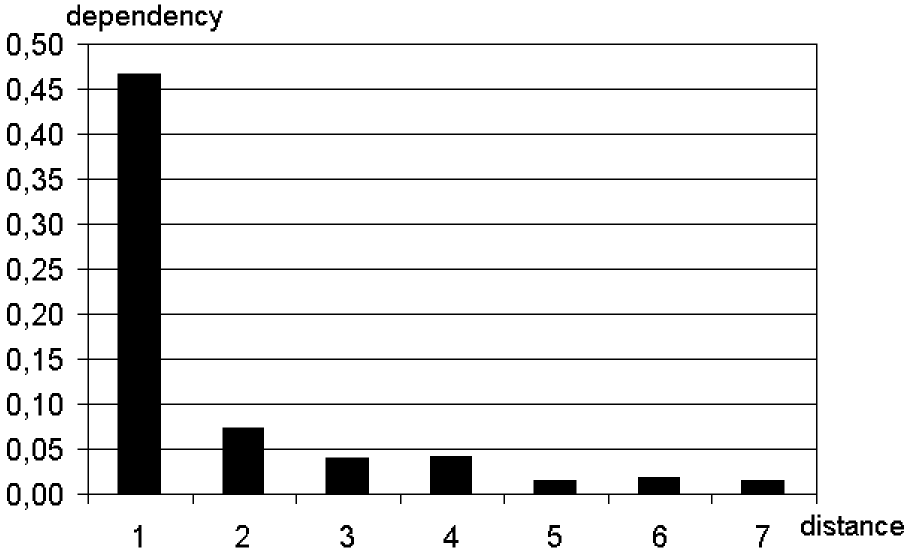


Figure B.2: Probability of dependency between instructions of program for MIPS R4000 processor in the function of distance between them.

and previous one is made[39]).

Tests were performed for two processors: Intel x86 and MIPS. For each processor we get two groups of tests:

1. Random programs (tables B.3 and B.5) – tests for programs hacing different length: 100 and 1000 instructions, performed two times for different values of random numbers generator seed (RANDOM1 and RANDOM2).
2. Random programs with dependencies (tables B.4 and B.6) – tests for programs having different length: 100 and 1000 instructions, performed two times for different values of random numbers generator seed (RANDOM1 and RANDOM2).

Table B.3: Dependencies between instructions in random programs for processor Intel x86.

| **Program (length)** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| RANDOM1 (100) | 0.17 | 0.22 | 0.16 | 0.14 | 0.14 | 0.22 | 0.11 |
| RANDOM1 (1000) | 0.16 | 0.19 | 0.19 | 0.18 | 0.15 | 0.19 | 0.13 |
| RANDOM2 (100) | 0.18 | 0.22 | 0.16 | 0.13 | 0.14 | 0.22 | 0.11 |
| RANDOM2 (1000) | 0.16 | 0.19 | 0.19 | 0.18 | 0.15 | 0.18 | 0.13 |
| Average | **0.17** | **0.21** | **0.18** | **0.16** | **0.15** | **0.20** | **0.12** |

Joined results for each processor are presented in the form of graphs on the figures B.3 and B.4. It can be seen, that the model with dependencies is good enough for generation of a real programs.

Joined results from two processors were used in the dissertation (figures 5.3 and 5.4).

[39] In general case it is not always possible, because instructions need not to have a common argument.

Table B.4: Dependencies between instructions in random programs with dependencies for processor Intel x86.

| **Program (length)** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| RANDOM1 (100) | 0.98 | 0.55 | 0.29 | 0.22 | 0.18 | 0.12 | 0.10 |
| RANDOM1 (1000) | 0.92 | 0.48 | 0.28 | 0.18 | 0.14 | 0.14 | 0.16 |
| RANDOM2 (100) | 0.98 | 0.55 | 0.29 | 0.22 | 0.18 | 0.12 | 0.09 |
| RANDOM2 (1000) | 0.92 | 0.48 | 0.28 | 0.18 | 0.14 | 0.14 | 0.16 |
| Average | **0.95** | **0.52** | **0.29** | **0.20** | **0.16** | **0.13** | **0.13** |

Table B.5: Dependencies between instructions in random programs for processor MIPS R4000.

| **Program (length)** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| RANDOM1 (100) | 0.06 | 0.11 | 0.04 | 0.06 | 0.12 | 0.01 | 0.03 |
| RANDOM1 (1000) | 0.09 | 0.07 | 0.07 | 0.06 | 0.06 | 0.04 | 0.04 |
| RANDOM2 (100) | 0.06 | 0.11 | 0.04 | 0.06 | 0.12 | 0.01 | 0.03 |
| RANDOM2 (1000) | 0.09 | 0.07 | 0.07 | 0.06 | 0.06 | 0.04 | 0.04 |
| Average | **0.08** | **0.09** | **0.06** | **0.06** | **0.09** | **0.03** | **0.04** |

Table B.6: Dependencies between instructions in random programs with dependencies for processor MIPS R4000.

| **Program (length)** | **d=1** | **d=2** | **d=3** | **d=4** | **d=5** | **d=6** | **d=7** |
|---|---|---|---|---|---|---|---|
| RANDOM1 (100) | 0.90 | 0.02 | 0.02 | 0.00 | 0.01 | 0.02 | 0.01 |
| RANDOM1 (1000) | 0.81 | 0.05 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| RANDOM2 (100) | 0.90 | 0.02 | 0.02 | 0.00 | 0.01 | 0.02 | 0.01 |
| RANDOM2 (1000) | 0.81 | 0.05 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| Average | **0.86** | **0.04** | **0.02** | **0.01** | **0.01** | **0.02** | **0.01** |

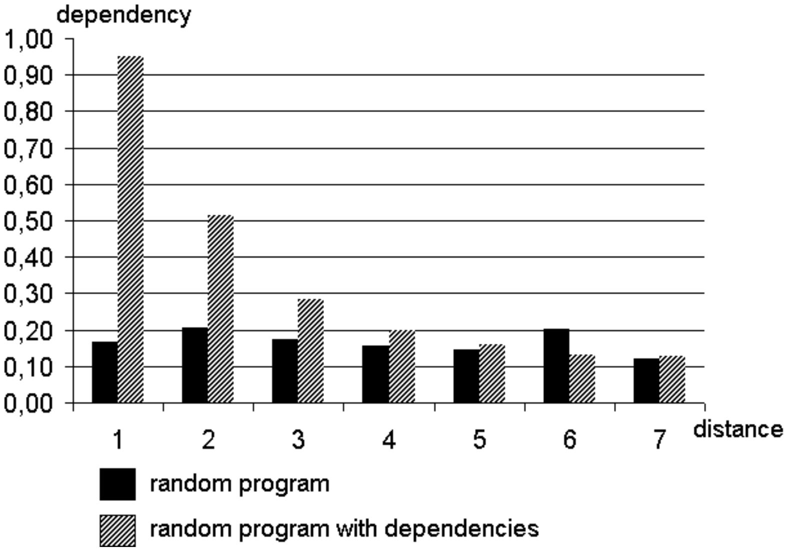


Figure B.3: Probability of occurrence of dependency between instructions of random programs for Intel x86 processor.

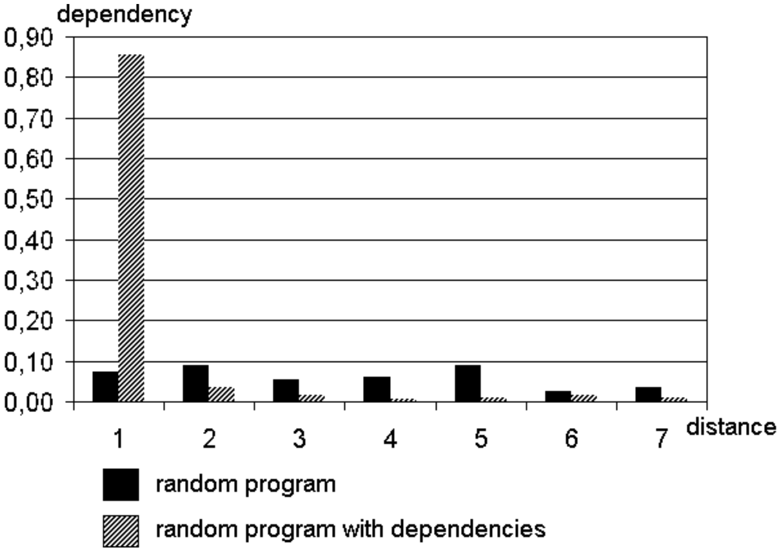


Figure B.4: Probability of occurrence of dependency between instructions of random programs for MIPS R4000 processor.

# Appendix C

# Examples of Obfuscated Programs

To demonstrate the implemented sample algorithm of obfuscation in action, we selected from the test set three programs: sorting program BUBSORT, calculating checksum CODETEST and decoding DECODE. The programs were obfuscated with different parameters, which showed in the final code different techniques of obfuscation. Listings in the assembler of microprocessor MIPS were cut to only beginning fragments, because of its small popularity.

In the presented listings the obfuscated code was printed with larger font, reversible operations with normal font, instructions using not used part of context with italic font and opaque constructs were underlined (see examples in the table C.1).

## C.1 Sorting Program

The program was obfuscated using all three methods of code insertion. Frequency of blocks reordering was set on 0.0 (no reordering). Occurrence of dependencies between instructions changing not used part of context was turned on ($I_Z = 1$). Rescaling factor $S = 3$, made that the program's length grew three times.

### C.1.1 Version for x86 processor

Thanks to switch off reordering in the obfuscated program can be seen the original control flow path. Inserted instructions changing not used registers efficiently slow down the analysis process. Opaque constructions quite nice integrated themselves with the rest of code. According to analytical measures the transformation has following properties (more information about analytical measures – see chapter 6.4.1):

- potency: length $E_L = 2.55$; depth $E_D = 11$; flow $E_F = 2.3$; average potency $\Pi_S = 1.33$
- resilience: full

Table C.1: Examples of formatting of listings of obfuscated programs.

| **Example** | **Description** |
|---|---|
| `mov [ebp+12],eax` | original instruction |
| *`not esi`* | instruction with not used part of context |
| `sub [ebp+12],ecx` | reversible operation |
| `mov  esi,[ebp+16]` | opaque construct |

- cost: cheap

```
    ; tmp=ebp-4, flag=ebp-8, i=ebp-12
    ; tab=ebp+8, len=ebp+12
    ; x1=ebp+16, x2=ebp+20, x3=ebp+24
    push   ebp
$L001:
    mov    ebp,esp
    dec    DWORD PTR [ebp+12]
    sub    esp,12          ; allocation of local variables
    add    esi,ebp
    not    esi
    push   esi
    add    ecx,esi
    sub    [ebp+12],ecx
    add    [ebp+12],ecx
$L010:
    inc    DWORD PTR [ebp+12]
    mov    eax,[ebp+12]    ; len=len-1
    sub    eax,1
    sar    ecx,22
    add    ebx,ecx
    mov    [ebp+12],eax
$L016:
    mov    ebx,esi
    sar    ebx,30
    mov    DWORD PTR [ebp-8],1 ; flag=1
    neg    DWORD PTR [ebp-8]
    mov    esi,[ebp+16]
    shl    esi,3
    add    esi,160
    jl     $L088
$L025:
    dec    ebx
    neg    DWORD PTR [ebp-8]
    cmp    DWORD PTR [ebp-8],0 ; for(...;flag!=0;)
    je     $L129
    mov    edi,[ebp+20]
    mov    DWORD PTR [ebp-8],0 ; flag=0
    add    edi,550
    jl     $L001
    mov    DWORD PTR [ebp-12],0 ; i=0
    mov    esi,[ebp+16]
    add    esi,910
    jg     $L116
    jmp    $L050
    mov    edx,[ebp+20]
    add    edx,240
    jg     $L010
$L041:
    mov    ecx,[ebp-12]     ; i++
    add    ecx,1
    rol    ecx,5
    mov    edi,[ebp+24]
    shr    edi,11
    sub    edi,5913
    jg     $L016
    ror    ecx,5
    mov    [ebp-12],ecx
$L050:
    mov    edx,[ebp-12]     ; for(...;i<len;...)
    cmp    edx,[ebp+12]
    jg     $L127
    xchg   edi,eax
    ror    edx,6
    not    DWORD PTR [ebp+8]
    add    edx,[ebp+8]
    mov    eax,[ebp-12]
    rol    edi,19
    add    esi,edi
    not    DWORD PTR [ebp+8]
    mov    ecx,[ebp+8]
    inc    ecx
    mov    edx,[ebp-12]
    and    ebx,esi
    inc    ecx
    mov    esi,[ebp+8]
    xor    ebx,eax
    ror    eax,18
    ror    eax,14
    mov    eax,DWORD PTR [ecx+eax*4]
    ror    edx,31
    cmp    eax,DWORD PTR [esi+edx*4+4]
    jl     $L123            ; if(tab[i]>tab[i+1])
$L073:
    mov    ecx,[ebp+24]
    sub    ecx,148
$L075:
    jg     $L134
    inc    ebx
    dec    ecx
    dec    ecx
    mov    ecx,[ebp-12]
    rol    edx,31
    mov    edx,[ebp+8]
    neg     ebx
     mov      eax,DWORD PTR [edx+ecx*4]
    mov     esi,[ebp+16]
    add     esi,690
    jl      $L129
$L088:
     mov      [ebp-4],eax     ; tmp=tab[i]
    inc     DWORD PTR [ebp-12]
    dec     DWORD PTR [ebp-12]
     mov      ecx,[ebp-12]
    ror     ecx,7
     mov      edx,[ebp+8]
    xor     ebx,ebx
    ror     ebx,23
     mov      eax,[ebp-12]
    sub     ecx,[ebp-8]
    sar     esi,22
     mov      esi,[ebp+8]
    mov     ebx,[ebp+24]
    sub     ebx,229
    jg      $L131
     mov      eax,DWORD PTR [esi+eax*4+4]
    and     esi,edx
     mov      DWORD PTR [edx+ecx*4],eax ; tab[i]=tab[i+1]
    shl     edx,15
    sar     edx,4
    add     ecx,[ebp-8]
    rol     ecx,7
     mov      ecx,[ebp-12]
     mov      edx,[ebp+8]
    add     esi,edx
    rol     eax,25
     mov      eax,[ebp-4]
    add     edi,eax
$L116:
    neg     DWORD PTR [ebp-8]
     mov      DWORD PTR [edx+ecx*4],eax ; tab[i+1]=tmp
    neg     DWORD PTR [ebp-8]
    mov     ecx,[ebp+20]
    xor     ecx,6447
    je      $L073
     mov      DWORD PTR [ebp-8],1    ; flag=1
$L123:
    rol     edx,31
    dec     ecx
    dec     ecx
     jmp      $L041
$L127:
    jmp     $L025
    inc     DWORD PTR [ebp+8]
$L129:
    pop     esi
    mov     edx,[ebp+16]
$L131:
    shr     edx,11
    sub     edx,8055
    jg      $L075
$L134:
    mov     esp,ebp
    pop     ebp
    ror     eax,5
    add     [ebp-8],eax
    ret     0
```

### C.1.2 Version for MIPS processor

The program has similar properties like program for x86 processor. Higher number of registers allowed for greater dissipation of data flow, which makes analysis of program even more difficult. Result of analytical measures:

- potency: length $E_L = 2.71$; depth $E_D = 4$; flow $E_F = 2.6$; average potency $\Pi_S = 0.95$
- resilience: full
- cost: cheap

```
    ; tab=$4, len=$5, x1=$6, x2=$7
    ; tmp=4($29), i=8($29), flag=12($29)
    subu    $29,16          ; allocation of local variables
    addu    $5,$5,-1        ; len=len-1
    addu    $5,$5,$31
    addu    $14,$0,1
    sw      $14,12($29)     ; flag=1
$L004:
    lw      $14,12($29)
    sw      $9,0($29)
    beq     $14,0,$L111     ; for(...;flag!=0;)
    and     $23,$15,$23
    sw      $0,8($29)       ; i=0
    xor     $4,$4,$12
    lw      $14,0($29)
    sw      $0,12($29)      ; flag=0
    subu    $5,$5,$31
    ble     $5,0,$L111      ; for(...;i<len;...)
    subu    $4,$4,$9
$L012:
 lw      $14,8($29)
srl     $1,$14,30
sll     $27,$1,17
 mul     $15,$14,4
subu    $14,$27,$24
xor     $5,$5,$2
addu    $4,$4,$9
xor     $4,$4,$12
 addu    $24,$4,$15
 lw      $25,0($24)
move    $28,$7
addu    $28,$28,160
sll     $28,$28,3
ble     $28,5313,$L034
 addu    $8,$4,$15
 lw      $9,4($8)
subu    $5,$5,$4
xor     $14,4($29),$2
 ble     $25,$9,$L087 ; if(tab[i]<tab[i+1])
xor     $5,$5,$23
xor     $5,$5,44
 ...
```

## C.2 Program Calculating a Checksum

Program was obfuscated using insertion of instruction changing not used part of context only, with occurrence of dependecies between them turned off ($I_Z = 1$). Frequency of reordering of blocks was set to 0.0 (no reordering). Rescaling factor $S = 2$, caused the final program to grow twice.

### C.2.1 Version for x86 processor

The obfuscated program remained quite readible, because the simples method of obfuscation was used and small rescaling factor value. On the example of this program we can trace the method of insertion of instructions changing not used part of context. Analytical measures of the transformation are as follows:

- potency: length $E_L = 1.78$; depth $E_D = 4$; flow $E_F = 4.1$; average potency $\Pi_S = 0.36$
- resilience: weak
- cost: cheap

```
; i=ebp-8, sum=ebp-4
; code=ebp+8, len=ebp+12
push   ebp
mov    ebp,esp
sub    esp,8        ; allocation of local variables
```

```
    xchg  ecx,eax
     mov    DWORD PTR [ebp-4],0   ; sum=0
    sal   ecx,19
     mov    DWORD PTR [ebp-8],0   ; i=0
    shr   ecx,24
     jmp    $L016
    and   ecx,ebx
$L010:
     mov    eax,[ebp-8]
    neg   ebx
     add    eax,1
    add   esi,ebx
     mov    [ebp-8],eax           ; i++
    mov   esi,eax
$L016:
     mov    ecx,[ebp-8]
    sar   esi,30
     cmp    ecx,[ebp+12]
    xchg  esi,eax
     jg     $L040                 ; for(...;i<len;...)
    inc   edx
     mov    edx,[ebp-8]
    neg   edi
     mov    eax,[ebp+8]
    sub   edi,ebp
     mov    ecx,[ebp-4]
    inc   edi
     add    ecx,DWORD PTR [eax+edx*4] ; sum+code[i]
    mov   edi,edi
     mov    edx,[ebp-8]
    sar   edi,7
     mov    eax,[ebp+8]
    rol   edi,10
     xor    ecx,DWORD PTR [eax+edx*4] ; (sum+code[i])^code[i]
    sal   edi,29
     mov    [ebp-4],ecx  ; sum=(sum+code[i])^code[i]
    xchg  edi,eax
     jmp    $L010
    add   edi,236
$L040:
     cmp    DWORD PTR [ebp-4],305419896
    mov   eax,edi
     je     $L084                ; if(sum!=0x12345678)
    and   esi,eax
     mov    DWORD PTR [ebp-8],0  ; i=0
    neg   esi
     jmp    $L054
    xor   esi,esi
```

```
$L048:
     mov    ecx,[ebp-8]
    add    esi,edx
     add    ecx,1
    sar    edx,29
     mov    [ebp-8],ecx            ; i++
    inc    edx
$L054:
     mov    edx,[ebp-8]
    sub    edi,edx
     cmp    edx,[ebp+12]
    xchg   edx,eax
     jg     $L000                  ; for(...;i<len;...)
    rol    edx,8
     mov    eax,[ebp-8]
    xchg   ebx,ecx
     mov    ecx,[ebp+8]
    xor    ebx,ebx
     mov    edx,[ebp-8]
    ror    edi,23
     add    edx,DWORD PTR [ecx+eax*4] ; i+code[i]
    and    ebx,edi
     xor    edx,305419896          ; (i+code[i])^0x12345678
    and    ebx,ebx
     mov    eax,[ebp-4]
    shl    edi,15
     add    eax,edx
    ror    esi,4
     mov    [ebp-4],eax            ; sum+=0x12345678^i+code[i]
    sub    esi,ebp
     mov    ecx,[ebp-8]
    rol    eax,25
     sub    ecx,1
    dec    edx
     mov    [ebp-8],ecx            ; i=i-1
    neg    ecx
     jmp    $L048
    add    ecx,ebx
$L084:
     mov    DWORD PTR [ebp-8],0    ; i=0
    inc    edx
     jmp    $L094
    neg    edx
$L088:
     mov    edx,[ebp-8]
    and    edi,edx
     add    edx,1
    add    edi,edi
```

```
      mov    [ebp-8],edx            ; i++
      rol    edi,21
$L094:
      mov    eax,[ebp+12]
      not    ebx
      sub    eax,1
      shl    ebx,1
      cmp    [ebp-8],eax
      xchg   ecx,ebx
      jg     $L126                  ; for(...;i<len-1;...)
      inc    ecx
      cmp    DWORD PTR [ebp-4],305419896
      mov    ecx,[ebp+12]
      jl     $L124                  ; if(sum>0x12345678)
      not    edx
      mov    ecx,[ebp-8]
      sub    edx,eax
      mov    edx,[ebp+8]
      xor    ebx,edx
      mov    eax,[ebp-8]
      sub    ebx,[ebp-8]
      sub    eax,DWORD PTR [edx+ecx*4+4]
      shr    ecx,18
      mov    ecx,[ebp-4]
      neg    ebx
      xor    ecx,eax                ; sum^i-code[i+1]
      rol    ebx,20
      mov    edx,[ebp-8]
      rol    ebx,5
      mov    eax,[ebp+8]
      or     ebx,ebp
      mov    DWORD PTR [eax+edx*4],ecx ; code[i]=sum^i-code[i+1]
      sub    eax,ebx
$L124:
      jmp    $L088
      mov    eax,[ebp-8]
$L126:
      mov    esp,ebp
      sar    esi,10
      pop    ebp
      add    [ebp-8],esi
      ret    0
```

### C.2.2 Version for MIPS processor

Program has properties similar to version for x86 processor. Result of analytical measures are:

- potency: length $E_L = 1.7$; depth $E_D = 5$; flow $E_F = 2.7$; average potency $\Pi_S = 0.58$

- resilience: weak
- cost: cheap

```
$L000:
    ; code=$4, len=$5
    ; sum=4($29), i=0($29)
    subu   $29,8               ; allocation of local variables
    sw     $0,4($29)           ; sum=0
    sw     $0,0($29)           ; i=0
    subu   $15,$21,$10
    ble    $5,0,$L032          ; for(...;i<len;...)
    subu   $23,$15,$4
$L006:
    lw     $14,4($29)
    xor    $16,$23,$23
    lw     $15,0($29)
    move   $27,$16
    mul    $24,$15,4
    srl    $1,$27,30
    addu   $25,$4,$24
    sll    $17,$1,17
    lw     $8,0($25)           ; code[i]
    subu   $22,$17,$24
    addu   $9,$14,$8
    lw     $25,4($29)
    addu   $10,$4,$24          ; sum+code[i]
    xor    $25,4($29),$2
    lw     $11,0($10)
    and    $1,$25,$3
    xor    $12,$9,$11          ; (sum+code[i])^code[i]
    lw     $19,4($29)
    sw     $12,4($29)          ; sum=(sum+code[i])^code[i]
    srl    $19,0($29),0
    addu   $13,$15,1
    sla    $19,0($29),0
    sw     $13,0($29)          ; i++
    and    $17,$19,$24
    blt    $13,$5,$L006
    sll    $16,9
    ...
```

## C.3 Decoding Program

Program was obfuscated using all methods of obfuscation, except of opaque constructs. Frequency of blocks reordering was set on 0.2 (one branch every five instructions). Occurence of dependencies between instructions changing not used part of context was turned on ($I_Z = 1$). The rescaling factor $S = 3$, caused triple growth of final program's length.

### C.3.1 Version for x86 processor

This example shows concatenation of simple methods of insertion with blocks reordering. Analysis of program was slowed down significantly, because of many branches. The transformation has following efficiency according to analytical measures:

- potency: length $E_L = 2.52$; depth $E_D = 1$; flow $E_F = 18.0$; average potency $\Pi_S = 1.06$
- resilience: weak
- cost: cheap

```
     ; i=ebp-4, j=ebp-8
     ; code=ebp+8, len=ebp+12, key=ebp+16
     push   ebp
     mov    ebp,esp
    dec     DWORD PTR [ebp+8]
     sub    esp,8             ; allocation of local variables
    add     [ebp+8],[ebp-8]
    rol     ecx,9
    sub     [ebp+8],[ebp-8]
     mov    DWORD PTR [ebp-8],-2023406815 ; j=0x87654321
    add     eax,ecx
     mov    DWORD PTR [ebp-4],0   ; i=0
    sub     [ebp-8],ecx
    sar     eax,22
     jmp    $L125
    add     eax,eax
    mov     eax,esi
$L015:
    jmp     $L117
$L020:
     mov    eax,[ebp+8]
     mov    ecx,[ebp-8]
    sub     [ebp+16],ebx
    and     esi,[ebp-8]
     mov    DWORD PTR [eax+edx*4],ecx  ; code[i]=j
    sub     ebx,105
    neg     esi
    jmp     $L100
$L030:
     mov    [ebp+16],ecx   ; key=j^key^0x12345678
    rol     edi,16
    xor     [ebp-4],edi
     mov    edx,[ebp-4]
    jmp     $L020
$L035:
     mov    [ebp-8],eax    ; j^=(code[i]^key)<<25;
    xor     ebx,ebx
    ror    edi,16
    mov    ecx,[ebp-8]
    ror    ecx,21
    neg    ebx
    rol    ecx,21
    xor    ecx,[ebp+16]    ; j^key
    jmp    $L073
$L044:
    mov    ecx,[ebp+8]
    sub    [ebp-8],ebx
    mov    edx,DWORD PTR [ecx+eax*4]  ; code[i]
    xor    [ebp-4],edi
    add    edx,esi
    not    DWORD PTR [ebp+16]
    sub    edx,esi
    xor    edx,[ebp+16]    ; code[i]^key
    shl    edx,25          ; (code[i]^key)<<25
    sub    [ebp-8],ecx
    dec    DWORD PTR [ebp+12]
    add    [ebp-8],ecx
    add    [ebp-8],ebx
    mov    eax,[ebp-8]
    jmp    $L067
$L059:
    inc    DWORD PTR [ebp+12]
    inc    DWORD PTR [ebp+12]
    jmp    $L015
$L062:
    xor    [ebp+16],eax
    neg    DWORD PTR [ebp+8]
    mov    esp,ebp
    pop    ebp
    jmp    $L095
$L067:
    xor    eax,edx
    jmp    $L035
$L069:
    jmp    $L044
$L073:
    xor    ecx,305419896   ; j^key^0x12345678
    dec    esi
    rol    ecx,26
    jmp    $L103
$L079:
    dec    ecx
    shr    ecx,7           ; (code[i]^key)>>7
    mov    edx,[ebp-8]
    mov    edx,esi
    rol    ecx,31
    ror    ecx,31
    xor    edx,ecx          ; j^((code[i]^key)>>7)
    inc    edx
    dec    edx
    mov    [ebp-8],edx      ; j^=(code[i]^key)>>7
    xchg   ebx,eax
    mov    eax,[ebp-4]
    not    DWORD PTR [ebp+16]
    jmp    $L069
$L093:
    jmp    $L079
$L095:
    ret    0
$L100:
    add    ebx,105
    add    [ebp+16],ebx
    jmp    $L059
$L103:
    ror    ecx,26
    jmp    $L030
$L105:
    ja     $L062            ; for(...;i<len;...)
    mov    edx,[ebp-4]
    dec    DWORD PTR [ebp+12]
    neg    DWORD PTR [ebp+8]
    xor    [ebp+16],eax
    mov    eax,[ebp+8]
    mov    ecx,DWORD PTR [eax+edx*4] ; code[i]
    not    eax
    or     edx,eax
    xor    ecx,[ebp+16]     ; code[i]^key
    inc    ecx
    jmp    $L093
$L117:
    mov    eax,[ebp-4]
    neg    DWORD PTR [ebp+8]
    add    eax,1
    dec    eax
    neg    eax
    mov    [ebp-4],eax      ; i++
    xor    [ebp+16],eax
    add    [ebp+12],esi
$L125:
    add    [ebp-8],ecx
    inc    DWORD PTR [ebp+8]
    neg    DWORD PTR [ebp+8]
    xor    [ebp+16],eax
```

```
        add     [ebp+12],esi
         mov    ecx,[ebp-4]
        sub     [ebp+12],esi
         cmp    ecx,[ebp+12]   ; i<len ???
        jmp     $L105
```

### C.3.2 Version for MIPS processor

Program has properties similar to version for x86 processor. Results of analytical measures are:

- potency: length $E_L = 2.58$; depth $E_D = 2$; flow $E_F = 6.3$; average potency $\Pi_S = 0.98$
- resilience: weak
- cost: cheap

```
         ; code=$4, len=$5, key=$6
         ; i=0($29), j=4($29)
    $L000:
         subu   $29,8           ; allocation of local variables
         li     $14,-2023406815
        addu    $14,$14,$31
        subu    $14,$14,$31
         sw     $14,4($29)      ; j=0x87654321
        sw      $10,0($29)
         sw     $0,0($29)       ; i=0
        subu    $22,$16,$4
        xor     $4,$4,$12
        xor     $4,$4,$12
         bleu   $5,0,$L000      ; for(...;i<len;...)
        sll     $14,$22,26
    $L012:
         lw     $15,4($29)
        addu    $4,$4,$21
        subu    $15,$15,140
        b       30
    $L022:
         xor    $15,$13,$10
        addu    $15,$15,$28
        subu    $15,$15,$28
         sw     $15,4($29)      ; j^=(code[i]^key)<<25
        xor     $17,$17
         lw     $11,4($29)
        xor     $4,$4,$1
        b       69
    $L030:
         lw     $24,0($29)
        sll     $17,$14,17
    mul     $25,$24,4
    subu    $22,$17,$24
    xor     $25,$25,$2
    xor     $25,$25,$2
    subu    $4,$4,$21
    addu    $8,$4,$25
    lw      $9,0($8)        ; code[i]
    subu    $15,$15,213
    subu    $6,$6,$4
    addu    $6,$6,$4
    xor     $10,$9,$6       ; code[i]^key
    xor     $22,4($29),$2
    b       49
$L045:
    mul     $24,$14,4
    addu    $18,4($29),$19
    subu    $17,$17,$21
    b       84
$L049:
    srl     $11,$10,7       ; (code[i]^key)>>7
    xor     $15,$15,44
    xor     $15,$15,44
    addu    $15,$15,213
    addu    $15,$15,140
    xor     $12,$15,$11
    sw      $12,4($29)      ; j^=(code[i]^key)>>7
    subu    $4,$4,$17
    lw      $13,4($29)
    addu    $17,$17,$21
    sla     $18,$22,26
    ...
```

# Appendix D

# Remaining Sources of Information

As in the case of any new problem, the basic source of information about code obfuscation is Internet. There are about 29000 web pages containing phrase *code obfuscation*. These pages can be classified on four useful categories according to they subject:

1. Educational pages.
    - `www.ioccc.org` – methods of C source code obfuscation
    - `www.wikipedia.com/wiki/obfuscated+code` – general description of the idea of "code obfuscation"
    - `www.softpanorama.org/SE/reverse_engineering_links.shtml` – set of links to pages about reverse engineering
2. Scientific pages containing publications in different form.
    - `www.cs.arizona.edu/~collberg` – Christian Collberg's home page
    - `www.csee.uq.edu.au/~cristina` – Cristina Ciffuentes' home page
    - `www.cs.auckland.ac.nz/~cthombor` – Clark Thomborson's home page
    - `www.cs.washington.edu/homes/douglas` – Douglas Low's home page
    - `www.research.microsoft.com/~toddpro` – Todd Proebsting's home page
3. Pages containing different software for code obfuscation.
    - `www.vegatech.net/jzipper` – JZipper: Java VM code obfuscation
    - `www.kotovnik.com/~avg` – cshred.c: obfuscation of C source code
    - `www.syncfusion.com` – code obfuscation of .NET VM
4. Pages of the *crackers'* subculture, containing mainly information about removing protection from programs.
    - `www.searchlores.org` – Fravia: Mekka of crackers
    - `astalavista.box.sk` – knowledge database about programs' cracking
    - `www.cracking.pl` – Polish source of information about crackers
    - `www.reverser-course.de` – practical knowledge about *reverse engineering*

From the pages presented in the tables some are really useful:

- home page of Christian Collberg – many articles and links to pages about obfuscation and decompilation
- home page of Cristina Ciffuentes – source of knowledge about decompilation, especially of type „from assembler do C"
- `www.ioccc.org` – open competition of source code obfuscation of program in C language
- Fravia – set of tools, tutorials, descriptions and essays about software protection, safety and reverse engineering
- `www.reverser-course.de` – a course of reverse engineering: useful tools, basics, practical advices