Software Protection through Code Obfuscation

Dissertation

submitted in partial fulfillment of the requirements
for the degree of
Master of Technology, Computer Engineering
by

Aniket Kulkarni
Roll No: 121022016

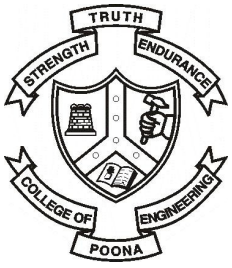
under the guidance of

Dr. Sachin Lodha (External Guide)

Tata Research Development and Design Centre, Pune

Prof. Shirish Gosavi (Internal Guide)

Department of Computer Engineering and Information Technology
College of Engineering, Pune
Pune - 411005.



JUNE 2012

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,
COLLEGE OF ENGINEERING, PUNE

CERTIFICATE

This is to certify that the dissertation titled

Software Protection through Code Obfuscation

has been successfully completed
By

Aniket Kulkarni
(121022016)

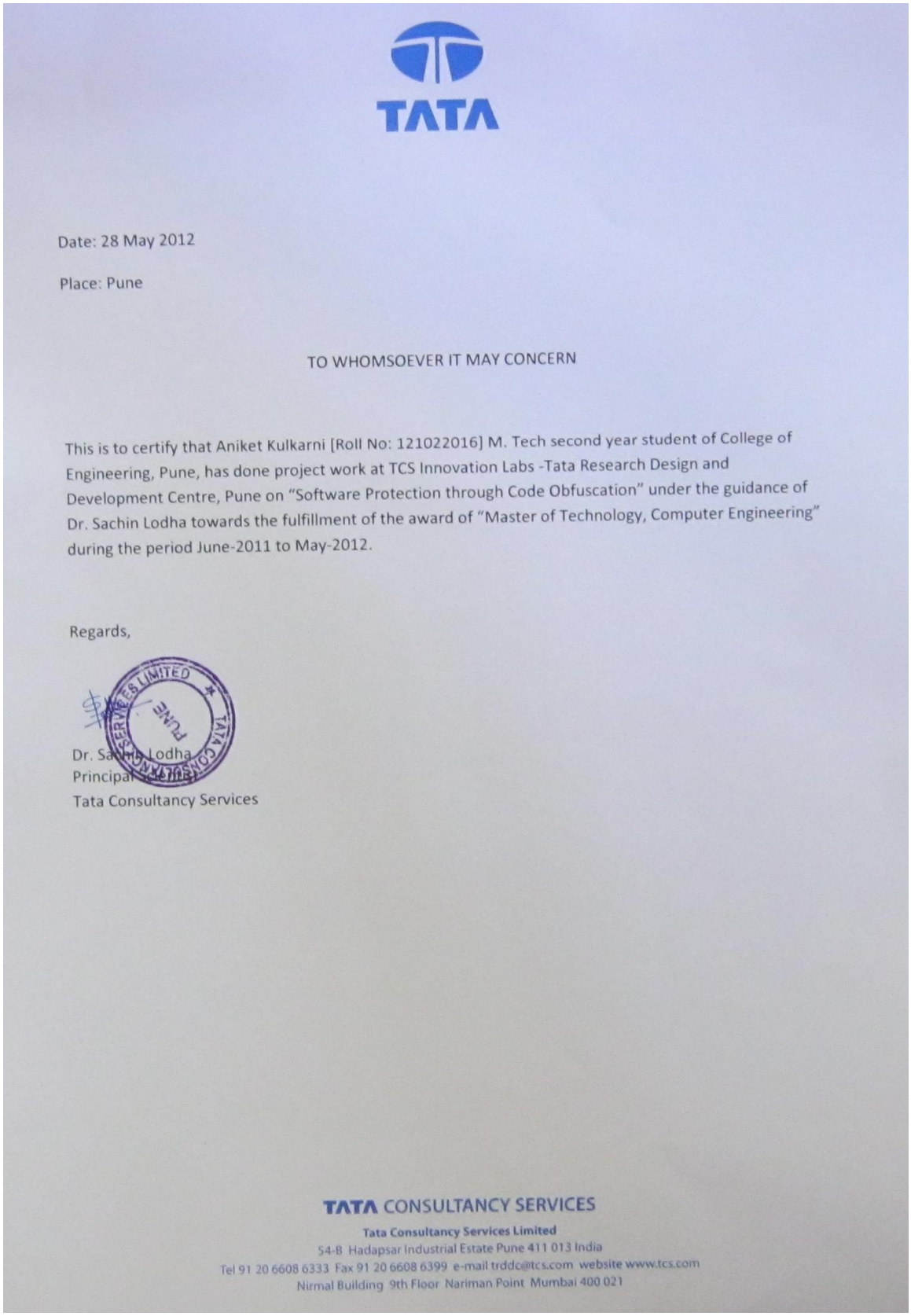
and is approved for the degree of

Master of Technology, Computer Engineering.

Prof. Shirish Gosavi,
Guide,
Department of Computer Engineering and Information Technology,
College of Engineering, Pune,
Shivaji Nagar, Pune-411005.

Dr. Jibi Abraham,
Head,
Department of Computer Engineering and Information Technology,
College of Engineering, Pune,
Shivaji Nagar, Pune-411005.

Date : June 2012



TATA

Date: 28 May 2012

Place: Pune

TO WHOMSOEVER IT MAY CONCERN

This is to certify that Aniket Kulkarni [Roll No: 121022016] M. Tech second year student of College of Engineering, Pune, has done project work at TCS Innovation Labs -Tata Research Design and Development Centre, Pune on "Software Protection through Code Obfuscation" under the guidance of Dr. Sachin Lodha towards the fulfillment of the award of "Master of Technology, Computer Engineering" during the period June-2011 to May-2012.

Regards,

Dr. Sachin Lodha
Principal Scientist
Tata Consultancy Services

TATA CONSULTANCY SERVICES
Tata Consultancy Services Limited
54-B Hadapsar Industrial Estate Pune 411 013 India
Tel 91 20 6608 6333 Fax 91 20 6608 6399 e-mail trddc@tcs.com website www.tcs.com
Nirmal Building 9th Floor Nariman Point Mumbai 400 021

ACKNOWLEDGEMENT

First and foremost I would like to express deepest gratitude to my External Guide **Dr. Sachin Lodha** (TRDDC, Pune) who has encouraged and supported me during the project work. I take this opportunity to thank our Head of Department **Prof. Dr. Jibi Abraham** and my Internal Guide **Prof. Shirish Gosavi**, for their able guidance and suggestions which were indispensable in the completion of this project. Finally, I am also grateful to my family, friends, and colleagues at **TRDDC** for their valuable help and support during the entire period.

Aniket Kulkarni
SY M.Tech (Computer Engg)

Abstract

Software security is an important concern in IT industry due to its huge financial losses (a few tens of billions of dollars). Software is prone to various security attacks, such as malicious reverse engineering. In this thesis, we examine software protection through code obfuscation technique which resists reverse engineering attacks. Code obfuscation is code transformation technique in which functionality of original code is maintained while obfuscated code is made difficult to reverse engineer. In the literature, various sets of criteria are described to measure effectiveness of code obfuscation such as potency (difficulty for human to understand code), resistance to automated attacks (such as static and dynamic analysis) and increase in state space of program. Most of the existing obfuscation techniques and schemes satisfy only a few of these criteria. In this thesis, we present a novel code obfuscation scheme developed for protecting proprietary code. The basic idea is to transform original code to obfuscated code having exploded state space. This is achieved by constructing obfuscated non trivial code clones for logical code fragments present in original code. These code clone fragments are linked using dynamic predicate variables to introduce valid control flow paths. We show that obfuscated code constructed by applying the scheme satisfies the existing criteria used to measure effectiveness of code obfuscation. We perform experimentation on a few programs (such as sequential data processing code) to evaluate our scheme. We show that software complexity (for example, cyclomatic complexity) of obfuscated code is higher than that of original code and corresponding to single execution path present in original code, exponential number of valid paths are introduced in obfuscated code. Any path is selected randomly to perform computation during execution of software. We also show that obfuscated code does not incur performance loss. Although the scheme increases cost of development of obfuscated code (due to construction of non trivial code clones for logical code fragments), we believe that the scheme is effective for protecting secret code such as license checking mechanism present in software.

Contents

List of Tables

List of Figures

1. Introduction

In this chapter, we introduce software protection. We mention a few security attacks possible on software. This is followed by introduction of protection techniques described in the literature. In the last subsection, we describe code obfuscation technique which is used for protecting software against malicious reverse engineering.

1.1. Software Protection

Software is a collection of computer programs and related data that provide the instructions telling a computer what to do. Software is usually developed in controlled and secured environments (for example, development centre of IT companies). But it is deployed in uncontrolled or less secure environments (for example, universities, research institutes and so on). As per Business Software Alliance (BSA) report [27], the commercial value of software piracy (a type of software security attack) is $58.8 billion in year 2010. Gartner report [28] mentions that use of interpreted languages such as Java increases attacker's ability to steal intellectual property (IP). Platform neutral languages such as Java are widely used in software development. These languages translate source program (for example, Java code) to intermediate format (for example, bytecode) which retains almost all the information (such as meaningful variable names) present in source code. Attacker can easily reconstruct source code from bytecode and attacker can extract secret information such as proprietary algorithms or cryptographic keys present in software.

1.2. Software Security Attacks

Software is prone to various security attacks (described in [2], [23]). These attacks include the following:

- Malicious reverse engineering of software to understand its functionality
- Stealth of software IP such as proprietary algorithms and code
- BORE (Break Once and Run Everywhere) attack
- Altering software functionality by tampering software

- Software hacking such as bypassing security checks (for example, license checking functionality)
- Software piracy to create unauthorized copies of software

These attacks are classified into three broad categories (described in [2], [3]):

- Malicious Reverse engineering
- Software Piracy
- Software Tampering

1.3. Software Protection Techniques

In this section, we take a look at protection techniques developed for protecting software namely obfuscation, watermarking and tamper-proofing described in the paper [2].

- Code Obfuscation:
  Code obfuscation is a technique used to protect software against malicious reverse engineering attacks. Code obfuscation is semantic preserving transformation which maintains functionality of original program while it makes understanding of obfuscated program harder for an attacker.

- Code Watermarking:
  Code watermarking is a technique used to protect software against software piracy attacks. In code watermarking, developer inserts a message called "watermark" into software (using secret key) to prove ownership of software.

- Tamper-proofing:
  Tamper-proofing is a technique used to protect software against tampering attacks. In tamper-proofing, tamper-proofing code such as parity bits is used to protect software against tampering attacks.

1.4. Software Protection through Code Obfuscation

In this section, we introduce code obfuscation technique in detail. Code obfuscation is defined by Collberg and others in paper [1] as follows:

Let P → P' be a transformation of a source program P into a target program P'.
P → P' is an obfuscating transformation, if P and P' have the same "**observable behavior**".
More precisely, in order for P → P' to be legal obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, the P' may or may not terminate
- Otherwise, P' must terminate and produce same output as P

A simple example of code obfuscating transformation is shown in the Figure 1.4.1. In the example, meaningful names from original code (for example, principle, rate and time) are replaced with random meaningless names (for example, a, b and c). Observable behavior of the obfuscated code is the same as that of the original code.

Original code

```
double computeInterest(double principal, double rate, long time)
{
        double interest = 0;
        interest = (principal * rate * time) /100;
        return interest;
}
```

Obfuscated code

```
double ci(double a, double b, long c)
{
        double d = 0;
        d = (a * b * c) /100;
        return d;
}
```

Figure 1.4.1: An Example of Layout Obfuscation

Now, we briefly introduce classification of code obfuscation techniques (described in paper [1]) namely layout obfuscation, control obfuscation, data obfuscation and preventive obfuscation.

- Layout obfuscation:

  In layout obfuscation, layout of program is changed. This includes the techniques such as:

  - Scrambling identifiers (program variables): Original identifiers are replaced with random meaningless identifiers.
  - Removing code comments present in program.

- Control obfuscation:

  In this technique, control flow of program is changed. This is achieved using techniques such as:

  - Aggregation:

    Inlining methods: Replacing method calls with method body.

    Outlining statements: Creating methods from program statements.

Clone methods: Creating duplicate methods with same functionality.

Unroll loops: Changing loop condition and duplicating loop body.

- Reordering:

  Statements: Order of statements in code is changed.

  Loop: Loop condition and loop body is altered

  Expression: Expressions are reordered preserving their semantics.

- Data obfuscation:

  In this technique, data present in program is obfuscated. This is achieved using techniques such as:

  - Storage Modification

    Split variables: Original variable is split into multiple variables. Functions performing operations on the variables are developed to preserve characteristic of original variable (for example, Boolean variable is split to multiple integer variables).

    Promoting scalar to Object: Scalar variables are promoted to object variables. For example, "int" variable in Java is changed to "Integer" object variable.

  - Encoding Modification

    Change variable lifetime: Variable lifetime is changed. For example, local variable is changed to global, so that it is active throughout program run.

- Preventive obfuscation:

  - Inherent:

    Inherent problems (for example, un-decidability of alias analysis [1]) with known de-obfuscation techniques (such as static analysis) are explored.

1.5. Organization of the thesis

The thesis is organized as follows. In this chapter, we have introduced software protection and key software protection techniques described in the literature. In the second chapter, we review current state of the art of code obfuscation technology. We

also look at various concepts and ideas described in the literature for protecting software against malicious reverse engineering. In the third chapter, we define goal of our project, that is, to develop a novel code obfuscation scheme for protecting secret algorithm present in software. We evaluate existing code obfuscation schemes on set of criteria defined to measure effectiveness of code obfuscation. In the fourth chapter, we describe design of our obfuscation scheme. We also evaluate the scheme on the set of criteria. In the fifth chapter, we apply our scheme to a few sample codes and we perform experimentation on original and obfuscated code. In the sixth chapter, we develop an architecture and design of code obfuscation tool which implements the obfuscation scheme. In the seventh chapter, we provide conclusions and future work on the obfuscation scheme. In the eighth chapter, we list references. In the ninth chapter, we provide an end to end example of the obfuscation scheme for a sequential data processing program.

2. Literature Survey

This chapter provides detailed literature survey on code obfuscation techniques and schemes.

2.1. Literature Review

We categorize the contents of the literature survey into the following subsections. The first subsection provides literature on key software protection techniques. In the second subsection, we take a look at literature specific to code obfuscation techniques and schemes. The third subsection presents literature on criteria used to measure effectiveness of code obfuscation and literature on de-obfuscation techniques.

2.1.1. Software Protection Techniques

Collberg and Thomborson [2] introduce techniques for protecting software, namely obfuscation, watermarking and tamper-proofing for defending software against reverse engineering, software piracy and tampering attacks respectively.

The book by Collberg and Nagra [3] compiles all the existing literature on software protection. The book includes topics such as program analysis, static and dynamic code obfuscation, obfuscation theory, software tamper proofing, static and dynamic software watermarking and so on.

Collberg and others [1] present taxonomy of code obfuscation techniques (for example, layout obfuscation) along with examples of each type of obfuscation techniques. This paper gives detailed scheme for control flow obfuscation using opaque predicates (defined in the section [2.1.2.2]) and aliased variables to protect software against static analysis attacks.

Oorschot [4] describes code obfuscation and other selective software protection techniques. This paper mentions that in code obfuscation, security can be achieved by transforming original code to obfuscated code, such that space of possible transformations is very large. This is similar to security provided by cryptography, that is, security strength depends on key-size (that is, possible encryption choices). This paper also describes other software protection techniques such as software diversity. In

software diversity, protection is achieved by creating multiple versions of software which are functionally equivalent.

2.1.2. Code Obfuscation Techniques and Schemes

This section presents literature on code obfuscation techniques and schemes. The first subsection presents techniques specific to identifier renaming (a type of layout obfuscation). The second subsection focuses on techniques specific to control flow transformation. The last subsection presents various obfuscation schemes described in the literature in which code obfuscation is combined with additional mechanisms (such as time sensitive codes).

2.1.2.1. Identifier Renaming Obfuscation

In identifier renaming (or scrambling identifiers) technique original meaningful names are replaced with random meaningless names. It is one way transformation (as original meaningful names cannot be recovered by attacker) without any cost overhead. It is widely supported by commercial (such as DashO [32]) and open source (such as ProGuard [33]) code obfuscators.

Ceccato and others [7] present results of experimental assessment of source code obfuscation performed by applying identifier renaming technique. They perform experimentation on two types of software systems (car and chat software) with two types of attackers (naive and experienced attackers) and on two types of attack tasks (program comprehension and modification). They conclude that if code is obfuscated using identifier renaming, attacker's efforts are increased to perform successful attack. This paper also shows that even for experienced attacker, it requires at least double time to complete attack.

Chan and Yang [8] use intelligent identifier renaming for layout obfuscation. They exploit the gap between definition of legal identifiers for Java compiler and definition of legal identifier for Java Virtual Machine (JVM). For example, JVM allows keywords (such as "false", "int") as valid identifiers while Java compiler does not allow these names as valid identifiers. Syntactic and semantic errors are introduced at java byte code

level. Thus, if the obfuscated byte code is decompiled, the generated code cannot be recompiled due to the errors introduced.

2.1.2.2. Control Flow Obfuscation

This subsection presents literature survey on code obfuscation techniques based on transformation of control flow of a program.

In thesis report [10], Low focuses on control flow obfuscation technique for software protection. Control flow transformations hide algorithms used in programs by introducing new fake control flows and by creating features (for example, unstructured control flow graph) at object level which have no source code equivalent. This paper describes control flow obfuscation techniques such as construction of opaque predicates (described in the next paragraph), aggregate obfuscation, ordering obfuscation and so on.

Collberg and others [9] describe a method for creating cheap, resilient and stealthy static opaque constructs to protect code against static analysis. Opaque predicates are boolean valued expressions whose outcome is known to obfuscator, but it is difficult for automatic de-obfuscator to deduce the outcome. Their basic idea is to create and manipulate dynamic data structure (such as graphs) containing alias variables maintaining certain conditions. These conditions are used to manufacture opaque predicates when needed. Their approach produces cheap (less computation overhead), resilient (to static analysis) and stealthy opaque predicates.

Madou and others [18] list three approaches to code obfuscation, namely source code obfuscation, bytecode obfuscation and binary obfuscation. They evaluate effectiveness of source code level transformations. They suggest that control flow flattening, insertion of opaque predicates and conversion of reducible control flow graph to irreducible control flow graph are the most effective source code obfuscations (as compiler cannot remove these obfuscations during its optimization phase) to hide control flow of a software.

Wang and others [11] present an obfuscation technique in which program control flow is flattened and alias variables (manipulated using global arrays and functions) are introduced into program to decide control flow. This technique transforms high level control transfers to indirect addressing (of dispatcher variable) through aliased pointers. Theoretical basis for difficulty of understanding obfuscated code is un-decidability of

aliases using static analysis. This paper concludes that attacker needs to perform dynamic analysis to analyze obfuscated code.

Chow and others [12] present obfuscation of control flow of program by embedding an instance "I" of hard combinatorial problem "C" by applying semantic preserving transformations. Solution to the instance is key "K" which is known to obfuscator but de-obfuscator needs to detect it. It is hard to deduce the key by static analysis of obfuscated code. This paper extends earlier research on control flow flattening code obfuscation technique.

Schrittwieser and Katzenbeisser [19] mention that most of the existing obfuscation techniques resist static reverse engineering attacks only but these fail against dynamic attacks. This paper introduces obfuscation scheme that applies the concept of software diversification to control flow graph to enhance its complexity. Control flow of obfuscated code depends upon program's input data and thus attacker's efforts are shifted from static analysis to dynamic analysis. Their approach makes dynamic analysis harder as attacker needed to execute program multiple times on different inputs to generate traces to obtain complete view of the program.

2.1.2.3. Obfuscation Schemes Using Additional Mechanisms

This subsection presents obfuscation schemes developed using combination of existing obfuscation techniques and additional mechanisms, such as use of self-modification, time sensitive codes and use of distributed system.

Kanzaki and Monden [21] propose a systematic method for protecting software against dynamic analysis attack. A program is protected by a method containing many time sensitive codes which are overwritten by dummy (fake) codes using self-modifying mechanism. If execution time of a guard code block is within predetermined range, time sensitive code becomes original one. If execution time is out of range, time sensitive code becomes fake one. This approach resists dynamic reverse engineering attacks but it requires accurate profiling information (such as estimation of time taken by guard code) of original code before obfuscating it.

Falcarin and others [20] present an obfuscation approach based on deployment of incomplete application. Code of application arrives from trusted network entity as a flow

of mobile code blocks which are arranged in memory with different customized layout. This approach defeats static and dynamic attacks due to deployment of incomplete application and code mobility respectively.

2.1.3. Measuring Effectiveness of Code Obfuscation and De-obfuscation Techniques

Collberg and others [1] describe measures such as potency, resilience and cost to evaluate quality of code obfuscation transformations. These criteria are described in detail (in section [3.2]). These measures are based on software complexity metrics (such as cyclomatic complexity described by McCabe [26]).

Madou and others [17] discuss potential of dynamic and hybrid (static-dynamic) attacks on software. They propose a realistic attack model where attacker can inspect each instruction and data value at every program point. In this attack approach, dynamic instrumentor is used to trace execution of original program and instrumentation information is gathered for attacking software. They validate this approach through a case study on software watermarking algorithm.

Anckaert and others [16] evaluate control flow obfuscating transformations on four software complexity metrics: control, control flow, data and data flow. They evaluate three transformations: control flow flattening, static disassembly thwarting and transformations based on opaque predicates. For example, they show that control flow flattening increases cyclomatic number (a software complexity metric) as each basic block is considered for flattening. A basic block is a sequence of code statements having single entry and exit point.

Karnick and others [15] provide a method to measure program obfuscation using measures (such as potency, cost). These measures are broken down into sub measures (for example, cost is broken into sub measures - memory, storage and runtime). Analytical metrics are developed to quantify performance of code obfuscation. For example, obfuscation measure is function of metric parameters: potency, resilience and cost. This paper also empirically evaluates various commercial obfuscators (namely DashO-Pro, Smokescreen, KlassMaster and Allatori) on these measures.

D'Anna and others [23] investigate use of software obfuscation in building self-protecting mobile agents (SPMA). This paper concludes that there is no general obfuscation

problem and authors believe that all automated obfuscation is merely emulation. They conclude that software obfuscation is useful only if it is employed for a specific purpose, such as hiding secret algorithm. This paper describes various reverse engineering attacks on software (for example, static-dynamic attacks, re-run attack and so on) too.

Heffner and Collberg [24] examine problems in constructing obfuscation executive to obfuscate application. Obfuscation executive applies a series of transformation to obfuscate code in which transformation dependency graph (modeled as finite state automata) is constructed. Authors conclude that while each individual obfuscation may add trivial amount of confusion, layering and interaction between different transformations can result in highly obfuscated application.

Collberg in paper [22] mentions that defenses (such as code obfuscation) that dynamically adjust themselves to new attack scenarios will survive longer than static defenses. This paper proposes two meta-level software protection principles:

- Diversity: It ensures that every asset (for example, software binary) is different.
- Defense in depth: It increases strength of defense by layering multiple defenses.

2.1.4. Code Obfuscation Theory

In this section, papers providing theoretical contributions on code obfuscation technology are highlighted. On negative side, Barak and others [5] give results on impossibility of generalized obfuscator which can obfuscate all programs for all program properties. On positive side, Lynn and others [6] give results of obfuscation of point functions (a function which evaluates to true on a particular input, and evaluates false on other inputs).

3. Goal of the Project

Goal of the project is to protect secret algorithm present in software by developing a novel code obfuscation scheme. Obfuscated code constructed by applying the scheme should satisfy most of the existing criteria used for measuring effectiveness of code obfuscation.

3.1. Problem Statement

This section describes the problem of protecting secret algorithm present in software. As described by D'Anna and others [23] code obfuscation is useful only if it is applied for specific purpose. In this project, we focus on the specific problem of protecting secret algorithm (such as license checking mechanism) present in software against malicious reverse engineering. Existing obfuscation schemes satisfy only a few criteria (described in the subsections [3.2] and [3.3]) used to measure effectiveness of code obfuscation. The subsection [3.4] describes limitations of existing control flow based transformation techniques and schemes. There is a need for development of novel code obfuscation scheme to protect secret code such that obfuscated code satisfies most of the existing criteria. These sets of criteria are described in the subsection [3.2]. Attacker can also employ various reverse engineering attacks (described in the subsection [3.3]) on software.

3.2. Criteria to Measure Effectiveness of Code Obfuscation

The three sets of criteria (C-1, C-2 and C-3) are as described in the subsections below.

3.2.1. C-1: Potency, Resilience, Cost

Collberg and others in paper [1] describe the following three measures to evaluate effectiveness of code obfuscation techniques:

- Potency: It is degree to which a human reader is confused with obfuscated code.
- Resilience: It is degree to which automated de-obfuscated attacks are resisted.
- Cost: It indicates overhead added to source application due to obfuscation.

The above criteria are related to software complexity metrics (for example, cyclomatic number by McCabe described in [26]). Effectiveness of code obfuscation is measured in terms of increase in values of these complexity measures. Higher the values of software complexity metrics of obfuscated code (obtained by applying obfuscation technique), more effective the code obfuscation technique is.

In paper [9], Collberg and others describe a measure called "stealth". Stealth is degree to which obfuscated code look similar to un-obfuscated code. We believe that there is trade of tradeoff between potency and stealth measures. For example, replacing original meaningful names with random meaningless names increase potency measure. But random meaningless names are not similar to original meaningful names, thus stealth measure is decreased. Hence, for sake of clarity, we restrict our self to measures - potency, resilience and cost (as described in paper [1]) for evaluating effectiveness of our obfuscation scheme.

3.2.2. C-2: Resistance to Static and Dynamic Attacks

Madou and others [17] describe static and dynamic attacks carried out on software. Static Attack is solely based on static information. It is obtained by examining and analyzing program without executing it. Dynamic Attack is solely based on dynamic information. It is obtained by executing program and observing execution traces. They measure effectiveness of code obfuscation based on resistance of obfuscated code to static and dynamic attacks. Sebastian and others [19] also describe effectiveness of code obfuscation based on resistance to static and dynamic reverse engineering attacks.

3.2.3. C-3: Increase in Program State Space

In paper [12], Chow and others describe obfuscation of control flow of program by expanding state space of program. This is achieved by embedding an instance "I" of hard combinatorial problem "C" into code of program. It is necessary to find the solution ("K") to the instance (by static analysis) which is needed to detect essential property "P" of code. This property is required to comprehend program. This obfuscation technique expands state space of program (called dispatcher code) and this paper shows that it is not

possible to minimize its state space. Thus, if state space of obfuscated program is larger than original program, reverse engineering efforts by attacker are increased. Hence such code obfuscation technique is better that technique which does not increase state space of program (for example, identifier renaming technique shown in the Figure 1.4.1).

3.3.Reverse Engineering Attacks

This section presents a few reverse engineering attacks (described in papers [17], [19] and [23]). Attacker can employ these attacks to comprehend functionality of software.

3.3.1. Static Analysis Attack

In this attack, attacker builds Control Flow Graph (CFG is a higher level representation of code) of software. Control flow graph contains a set of nodes, representing instructions in code and a set of edges between nodes representing possible control paths. CFG of program helps attacker to comprehend functionality of software.

3.3.2. Dynamic Analysis Attack

In this attack, attacker executes software on a set of inputs and studies output of software by generating execution traces. Attacker can also profile software to reveal actual paths selected for program execution. Dynamic analysis provides considerable leverage in locating secret information (for example, cryptographic keys) present in software. Dynamic analysis is harder than static analysis as software needs to be run on different inputs.

3.3.3. Code Clone Detection Attack

In this attack, attacker detects and removes code clones present in program to comprehend functionality of software. Attacker can employ existing code clone detection techniques such as matching Abstract Syntax Tree (AST) (described by Baxter and others [25]) to detect and remove code clones.

3.4.Evaluation of Existing Code Obfuscation Schemes

This section examines existing obfuscation techniques and schemes for their effectiveness in terms of sets of criteria described in the sections above.

Control flow obfuscation technique described by Chow and others in paper [12], makes static analysis of obfuscated program difficult. In this technique, obfuscation is achieved by embedding an instance of hard problem into program. They view program (dispatcher code) as finite automaton. They propose to expand state space of automata by incorporating numerous dummy states into dispatcher code. This obfuscation scheme satisfies set of criteria (C-1 and C-3). Criterion C-1 is satisfied as software complexity of obfuscated code is higher than that of original code due to embedding an instance of hard problem. Criterion C-3 is satisfied as program state space of obfuscated code is higher than that of original code and authors show that minimization of state space of obfuscated code is not possible. But this obfuscation scheme does not satisfy criterion C-2 as the obfuscated code resists static analysis attacks only. Solution to the instance of hard problem can be obtained by performing dynamic analysis (for example, by executing the program on a set of inputs). It has been shown by Madou and others [17] that attacker can trace program execution to understand actual path selected for execution.

Schrittwieser and Katzenbeisser [19] describe an obfuscation scheme using the concept of control flow diversification. In this scheme, software binary is divided into fragments (called gadgets) and additional valid control flow paths are introduced in software. For different set of inputs, different program paths are executed by software. This obfuscation scheme satisfies criteria C-1 and C-2. Software complexity of obfuscated code is higher than that of original code due to increase in control flow paths. Thus, criterion C-1 is satisfied. This obfuscation scheme uses concept of branching function to resists static analysis attacks. In branching function, target of branch statement is decided dynamically (as described by Wang and others in paper [11]). This obfuscation scheme resists dynamic analysis attacks by introduction of valid control flow paths in obfuscated code. Thus, criterion C-2 is satisfied too. But this scheme does not satisfy criterion C-3. As this scheme increases program state space by adding syntactically different but semantically same gadgets to obfuscate code. The concept of gadget is similar to that of basic block (that is, a sequence of instructions having single entry and exit point). Authors mention that such cloned gadgets provide weak security against automated gadget matching algorithm attacks. Hence, it is possible to minimize state space of obfuscated program.

Authors propose to enhance their scheme by developing inter-gadget diversification techniques in future. Thus, this scheme does not satisfy criterion C-3.

Thus, to the best of our knowledge, there does not exist code obfuscation scheme that satisfies the above criteria C-1, C-2 and C-3 (which are used to measure effectiveness of code obfuscation) and resists above reverse engineering attacks (namely static, dynamic and clone detection attacks). Hence, there is a need for development of novel code obfuscation scheme which satisfies these sets of criteria.

4. Design of Obfuscation Scheme

In this section, design of novel code obfuscation scheme is presented. Obfuscated code constructed by applying our scheme satisfies the criteria (described in the section [3.2]) and resists reverse engineering attacks (described in the section [3.3]) used for measuring effectiveness of code obfuscation. The basic idea is to transform original code to obfuscated code having exploded state space. State space of obfuscated code is expanded by inserting non trivial code clones constructed for logical code fragments present in original code. These non trivial code clone fragments are linked using dynamic predicate variables. An identifier renaming technique is applied to increase attacker's effort to understand obfuscated code. Corresponding to single path present in original code, exponential number of valid paths is introduced in obfuscated code. Any execution path is randomly selected to perform computation such as license checking mechanism. Attacker needs to execute software with obfuscated code multiple times to generate execution traces to understand functionality of software.

4.1. Obfuscation Scheme

In this section, a novel code obfuscation scheme is presented with an example of data processing code. The obfuscation scheme consists of the following four steps.

4.1.1. Step-1: Construction of Logical Code Fragments

In the first step, code implementing secret algorithm is divided into logical code fragments. Each logical fragment implements specific functionality (for example, sorting) which known to developer. But attacker does not know specific functionality of these logical code fragments (as algorithm or code is not publicly available). There are two possible choices for creating logical code fragments. They are as follows:

- Use of Basic block:

  Basic block is a sequence of code instructions having single entry and exit points. Basic blocks cannot contain control flow instructions such as conditional statements.

- Use of Methods (or subroutines):

  Method (or subroutine) is higher level representation of code. Methods have rich computation structure which is represented using Control Flow Graph (CFG).

In the literature ([1], [10] and [19]), code obfuscation of basic blocks as a code fragment is described (for example, by insertion of dummy instructions). Basic blocks cannot contain language constructs such as loops or conditional statements. Hence, if logical code fragmentation is carried out at basic block level, limited program functionality can be implemented. The obfuscation scheme proposes creation of logical code fragments at higher level program representation such as methods (or subroutines). This rich computation structure allows developer to construct code fragments implementing specific functionality. Logical code fragments are constructed in such a way that they are executed in a sequence to perform computation.

For example, consider a data processing application. The application takes inputs, processes data and generates output as shown in the Figure 4.1.1.1.

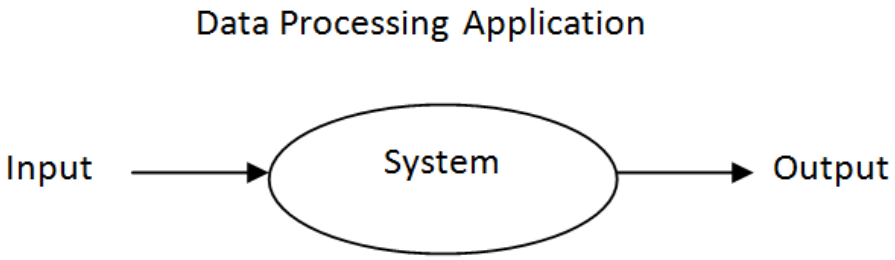


Figure 4.1.1.1: Simple Data Processing Application

Suppose that the data processing application is divided into four logical code fragments namely

1. Read Inputs
2. Sort Data
3. Search Data
4. Write Output

These four logical code fragments are executed in a sequence to process data. The Figure 4.1.1.2 shows the system after constructing logical code fragments.

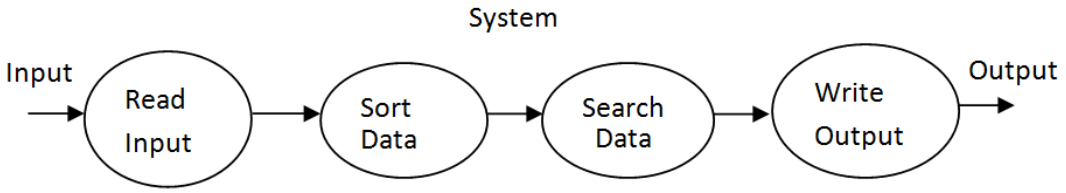


Figure 4.1.1.2: Logical fragments of Data Processing Application

4.1.2. Step-2: Construction of Non trivial Code Clones for Fragments

For each code fragment, non trivial code clones are constructed. Non trivial code clones are defined as a set of code fragments which perform same computation but they have different code structure such as Abstract Syntax Tree (AST). This is shown in the Figure 4.1.2.1with an example of variable swap functionality which is implemented using memory and XOR operation.

Swap using memory

```
Swap_1(int x,int y)
  int t; // temp variable
  t = x;
  x = y;
  y = t;
```

Swap using XOR

```
Swap_2(int x,int y)

 int u;  // temp variable
 u= x ^ y;
 x = u ^ x;
 y = u ^ y;
```

Figure 4.1.2.1: Non trivial code clones for variable Swap operation

Due to introduction of code clones, software complexity of obfuscated code is increased. In the literature [1], cloning method is described as a technique to increase reverse engineering efforts. The literature also describes code clone detection techniques such as matching Abstract Syntax Tree (described in paper [25]). These can be used to detect and remove code clones, so that reverse engineering efforts by attacker are reduced.

There are two possible choices for creating code clones to obfuscate code. They are as follows:

- Trivial code clones

  Trivial code clones can be created using identifier renaming obfuscation technique. For example, set of pseudo code instructions (shown in paragraph

below) have same code structure such as Abstract Syntax Tree (AST), but they have different variable names.

```
Swap1:    int x, y, t;   t = x; x = y; y = t;

Swap2:    int a, b, u;   u = a; a = b; b = u;
```

- Non trivial code clones

  Non trivial code clones perform same computation but they have different code structure (such as AST) as shown in the Figure 4.1.2.1.

The code obfuscation scheme proposes use of non trivial code clones for obfuscation. Trivial code clones may be detected using techniques such as matching Abstract Syntax Tree (AST) [25]. But non trivial code clones cannot be detected using existing code clone detection techniques as code structure (such as AST) is different for different clones.

The following Figure 4.1.2.2 shows the system after constructing non trivial code clones for sort and search logical code fragments. Note that three code clones are constructed for sort fragment namely sorting using bubble sort, selection sort and insertion sort. Similarly, three code clones are constructed for search fragment namely sequential, sorted search and binary search. For sake of understanding, different algorithms are shown for different code clones, although developer can construct different non trivial code clones using same algorithm (for example, bubble sorting) for logical code fragments (for example, sort fragment).

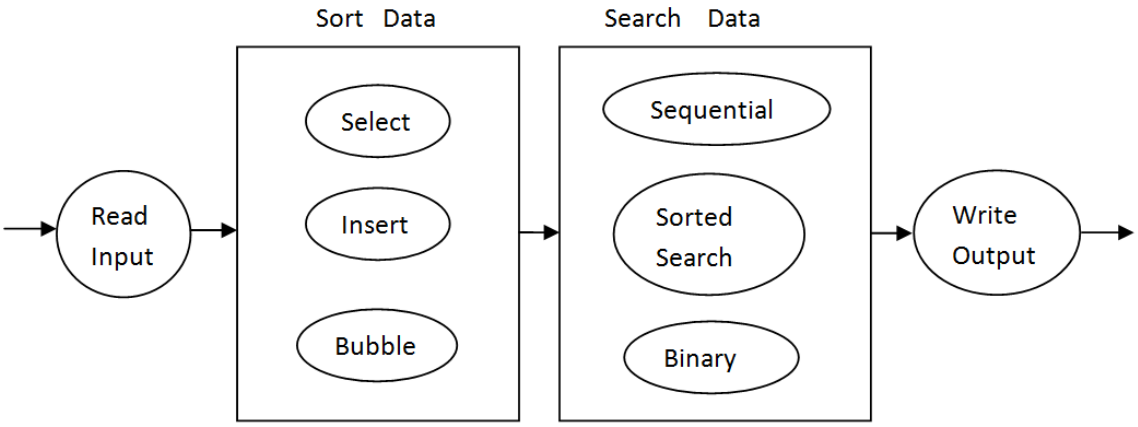


Figure 4.1.2.2: Non Trivial Code Fragments for Data Processing Application

Note that rectangular blocks are used to represent a wrapper structure for logical code fragments (for sort and search functionality) in the intermediate step during application of the obfuscation scheme. Developer need to link these non trivial code clone fragments.

4.1.3. Step-3: Linking Code Clone Fragments using Dynamic Predicate Variables

In this step, non trivial code clone fragments are linked using predicate variables. Predicate variables are set of expressions that are evaluated to boolean value (either true or false value). There are two possible choices for linking code clone fragments.

- Use of static opaque predicate variables

  Static opaque predicate variables are described by Collberg and others in paper [1]. An example of static opaque predicate variable is conditional statement "b:= (x == x+1)" which always evaluates to false value. Such predicate variables introduce fake control path in code (described by Low in paper [10]). Static predicates cannot resist dynamic analysis attacks as fake control paths are never selected for execution.

- Use of dynamic opaque predicate variables

  Palsberg and others in paper [13] describe dynamic predicate variables as a set of correlated boolean variables. These variables evaluate to same value in given run of software but they evaluate to different values at different run of software.

The obfuscation scheme uses a variant of dynamic predicate variables, a set of conditions which are evaluated at run time such that exponential number of valid combinations of values of predicate variables is possible. These variables enable selection of a particular combination of code clone fragments for a given run of obfuscated software. These dynamic predicate variables introduce valid control flow paths by keeping dynamic structures (such as linked list) shown in the Figure 4.1.3.1. These dynamic predicates increase static and dynamic analysis efforts as exponential number of code clone combinations are possible for execution.

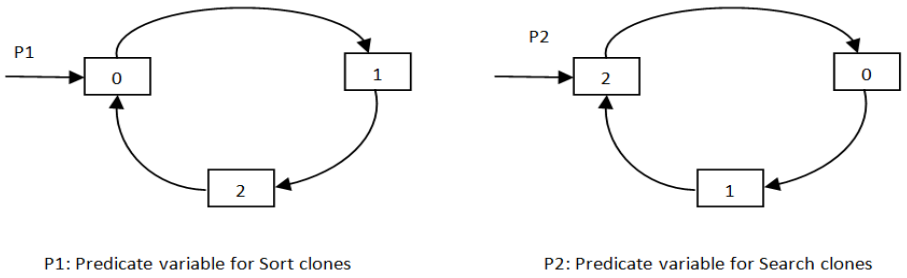


Figure 4.1.3.1: Dynamic Predicate Variables for Sort and Search Fragments

Suppose that two linked lists are maintained - one for "sort" fragment and other for "search" fragment. Variables P1 and P2 move randomly through the lists such that they point to different nodes (containing integer data values) at different times. For a given run of software, clones for respective fragments are selected by matching clone identifier (a unique integer value assigned to each code clone of fragment) with data value pointed by predicate variables.

The Figure 4.1.3.2 shows the data processing system after linking the code clone fragments using dynamic predicate variables. Any sequence of logical code clone fragments is possible for given run of software. Attacker needs to execute software multiple times to generate traces for understanding functionality of software.

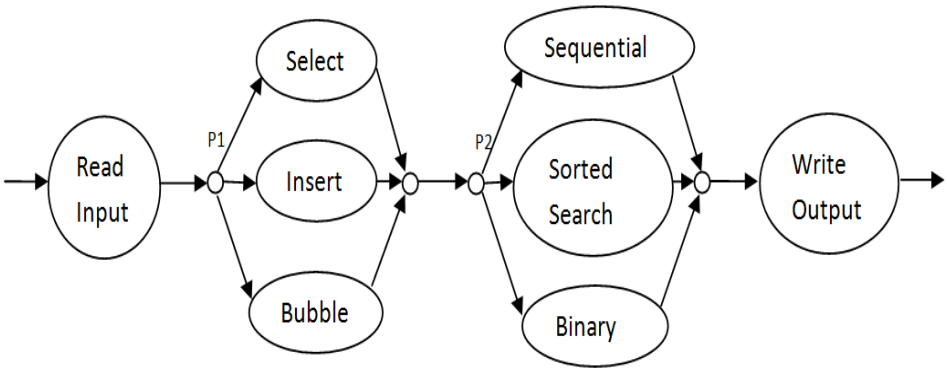


Figure 4.1.3.2 Linked Code Clone Fragments for Data Processing Application

4.1.4. Step-4: Applying Identifier Renaming Technique

In this step, identifier renaming technique is applied to logical code clone fragments. In the literature, Ceccato and others [7] experimentally show that attacker's efforts are increased if code is obfuscated using identifier renaming technique. In identifier renaming technique, meaningful names (such as "sort") are removed. It is one way transformation (as original names cannot be recovered by attacker) and it has little or no cost overhead. Although the obfuscation scheme proposes use of identifier renaming technique, the scheme does not restrict use of other state of the art code obfuscation techniques such as control flow transformations [1], control flow flattening [12] for obfuscation of logical code clone fragments.

The Figure 4.1.4.1 shows obfuscated code after applying identifier renaming code obfuscation technique. Note that in the figure, only meaningful function names (such as "insert") are replaced with random meaningless names ("h"), but in actual code, all meaningful names such as local variables, parameter can be replaced too.

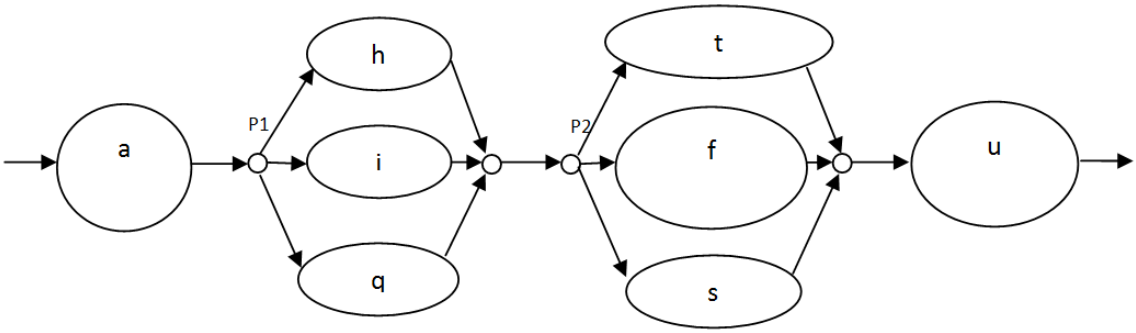


Figure 4.1.4.1: Identifier Renaming for Data Processing Application

4.2.Mathematical Analysis using Theoretical Example

In this section, the obfuscation scheme is applied to a theoretical example and mathematical analysis of the scheme is presented. It shows that total number of valid execution paths is exponential with respect to number of logical code fragments and number of code clones introduced in obfuscated code.

Consider original code of software divided into N logical code fragments by developer. Execution of software is shown as a path from start to end node in the Figure 4.2.1. Developer has knowledge of code fragmentation (that is, logical code fragments are CF_1, CF_2 which are shown in different colours) in the Figure 4.2.1. But, attacker does not know about logical code fragmentation and hence attacker's view of code is different (that is, software is a sequence of code fragments as shown in gray colour) in the Figure 4.2.2.

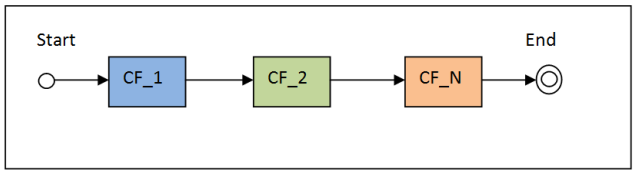


Figure 4.2.1: Developer's View of Original Code

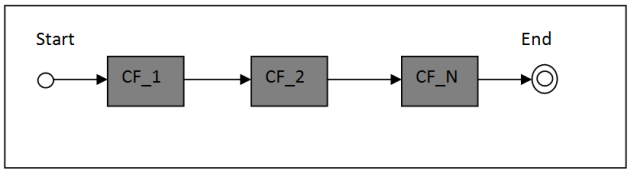


Figure 4.2.2: Attacker's View of Original Code

Now, developer constructs non trivial code clones for each logical code fragments and links these code clone fragments using dynamic predicate variables. For example, for code fragment CF_1 in original code, code clones CF_11, CF_12 and CF_13 are created in obfuscated code. Developer has knowledge of code clones fragments and hence he can identify and differentiate between code fragments which perform same and different computation as shown in the Figure 4.2.3. But, attacker does not know about code clone fragments; hence attacker cannot identify and differentiate between code fragments which perform same and different computation as shown in the Figure 4.2.4.

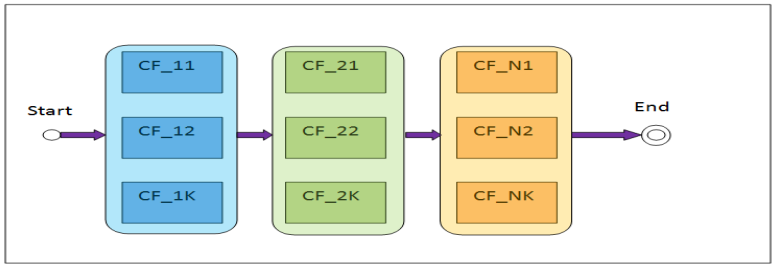


Figure 4.2.3: Developer's View of Obfuscated Code

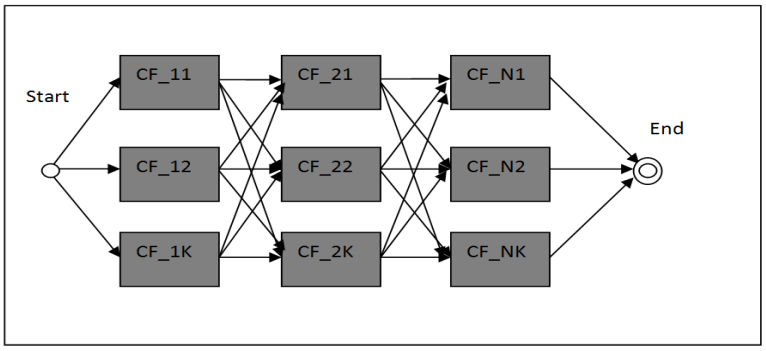


Figure 4.2.4: Attacker's View of Obfuscated Code

4.3.Mathematical Analysis

Suppose that original code of software is divided into N logical fragments and K clones are created per fragment. Suppose that execution of each code fragment is guarded by predicate variable Pij where (1 <= i <= N) and (1 <= j <= K). Let execution sequence of original code be {CF_1-…CF_i-…-CF_N}. Let execution sequence of obfuscated code be {COF_1-…COF_i-…COF_N} (where (1 <= i <= N). Now, COF_i can be any of CF_ij where j varies from 1 to K. Thus, there exists K * K * .. N times (= K^N) unique execution sequences in obfuscated code corresponding to single execution sequence in original code. All sequences are valid execution paths during any run of software.

4.3.1. Special Cases

In this subsection, special cases for the obfuscation scheme are considered. The first case considers construction of different number of code clones per fragment. In this scenario, exponential number of execution sequences is possible as follows: Let K_1, K_2, K_3, …, K_N be number of clones created for code fragments 1, 2, …, N respectively. The total number of valid execution sequences is: $K_1*K_2*K_3…*K_N$.

The second case considers construction of obfuscated code using code duplication (in which single clone is created per fragment). In this case too an exponential number of execution sequences are possible as follows. Let there exists N code fragments and 2 clones per fragment. The total number of execution paths is 2*2*2... N times (= 2^N). Thus, total number of execution sequences is exponential in terms of logical code fragments and code clones.

4.4.Obfuscation Example

This section shows an end to end application of obfuscation scheme to data processing code described in the sections above. The first subsection shows original and obfuscated source codes for sort and search fragments. The next subsection shows cyclomatic numbers (a software complexity measure by McCabe [26]) for sort and search methods of original and obfuscated codes. In the third subsection, code coverage of obfuscated code is shown. The last subsection shows execution traces of original and obfuscated codes.

4.4.1. Original Data Processing Code

The original data processing code contains four logical code fragments which are invoked in a sequence from main method. The source code of main, sort and search methods are shown in Figure 4.4.1.1, Figure 4.4.1.2 and Figure 4.4.1.3 respectively. The source code for methods “readInput” and “writeOutput” is not shown to avoid cluttering of code. The entire source code of original and obfuscated code can be found in appendix sections [9.1] and [9.4].

DataProcessor_Orig.java

```
1
2  import java.io.BufferedReader;
3  import java.io.BufferedWriter;
4  import java.io.File;
5  import java.io.FileReader;
6  import java.io.FileWriter;
7  import java.util.ArrayList;
8
9  public class DataProcessor_Orig
10 {
11     int[] arrayData = null;
12     int[] arrayDataUpdated = null;
13     int value = 0;
14     boolean isValueFound = false;
15
16     public static void main(String[] args)
17     {
18         DataProcessor_Orig dp = new DataProcessor_Orig();
19         dp.readInputs(args);
20         dp.sortData();
21         dp.searchData();
22         dp.writeOutputs();
23     }
```

Figure 4.4.1.1: Main method of Original Data Processing Code

DataProcessor_Orig.java

```
81    public void sortData()
82    {
83        int arrLen = arrayData.length;
84        arrayDataUpdated = new int [arrLen];
85
86        for(int i=0;i<arrLen;i++)
87        {
88            arrayDataUpdated[i] = arrayData[i];
89        }
90
91        int tempVar = 0;
92        for(int i=0;i<arrLen;i++)
93        {
94            for(int j=1;j<arrLen;j++)
95            {
96                if(arrayDataUpdated[j-1]>arrayDataUpdated[j])
97                {
98                    tempVar = arrayDataUpdated[j-1];
99                    arrayDataUpdated[j-1] = arrayDataUpdated[j];
100                   arrayDataUpdated[j] = tempVar;
101               }
102           }
103       }
```

Figure 4.4.1.2: Sort method of Original Data Processing Code

```
104   }
105
106   public void searchData()
107   {
108       for(int i=0;i<arrayData.length;i++)
109       {
110           if(arrayData[i] == value)
111           {
112               isValueFound = true;
113           }
114       }
115   }
116
```

Figure 4.4.1.3: Search method of Original Data Processing Code

4.4.2. Obfuscated Data Processing Code

The Figure 4.4.2.1, Figure 4.4.2.2 and Figure 4.4.2.3 show obfuscated code for main, sort and search methods respectively. Note that non trivial code clone fragments are obfuscated using identifier renaming obfuscation technique applied to local variables. The predicate variables (m_0 and m_1) link these code clones and select a particular code clone combination for execution (this is achieved by generating all possible combinations

of code clones). Names of member variables and other methods are not changed to focus on obfuscation of methods which implement logical code fragments only.

DataProcessor_Obfuscated.java

```
9
10 public class DataProcessor_Obfuscated {
11
12     int[] arrayData = null;
13
14     int[] arrayDataUpdated = null;
15
16     int value = 0;
17
18     boolean isValueFound = false;
19
20     NDG m_0 = null, m_1 = null;
21
22     STM stm = new STM("dataprocessor", new int[] { 3, 3 });
23
24     public static void main(String[] args) {
25         DataProcessor_Obfuscated dp = new DataProcessor_Obfuscated();
26         dp.readInputs(args);
27         dp.sortData();
28         dp.searchData();
29         dp.writeOutputs();
30     }
31
```

Figure 4.4.2.1: Main method of Obfuscated Data Processing Code

```
70
71     public void sortData() {
72     if (m_0 == null) {
73     m_0 = new NDG(3, "m_0", stm);
74 } else {
75     m_0.mvn();
76 }
77     if (m_0.sn(0)) {
78     int a2110711758 = arrayData.length;
79     arrayDataUpdated = new int[a2110711758];
80     for (int c677447630 = 0; c677447630 < a2110711758; c677447630++) {
81         arrayDataUpdated[c677447630] = arrayData[c677447630];
82     }
83     for (int n1774596447 = 0; n1774596447 < a2110711758; n1774596447++) {
84         int a171330926 = arrayDataUpdated[n1774596447];
85         int a118702090 = -1;
86         for (int a979667234 = n1774596447 + 1; a979667234 < a2110711758; a979667234++) {
87             if (arrayDataUpdated[a979667234] < a171330926) {
88                 a118702090 = a979667234;
89                 a171330926 = arrayDataUpdated[a979667234];
90             }
91         }
92         if (a118702090 != -1) {
93             arrayDataUpdated[a118702090] = arrayDataUpdated[n1774596447];
94             arrayDataUpdated[n1774596447] = a171330926;
95         }
96     }
97 }
```

```
        if (a118702090 != -1) {
            arrayDataUpdated[a118702090] = arrayDataUpdated[n1774596447];
            arrayDataUpdated[n1774596447] = a171330926;
        }
    }
}
    if (m_0.sn(2)) {
    int o78573288 = arrayData.length;
    arrayDataUpdated = new int[o78573288];
    for (int u723080508 = 0; u723080508 < o78573288; u723080508++) {
        arrayDataUpdated[u723080508] = arrayData[u723080508];
    }
    for (int s1517383704 = 1; s1517383704 < o78573288; s1517383704++) {
        int l1143453413 = arrayDataUpdated[s1517383704];
        boolean f1702496723 = false;
        for (int j50892461 = s1517383704; j50892461 > 0; j50892461--) {
            if (l1143453413 < arrayDataUpdated[j50892461 - 1]) {
                arrayDataUpdated[j50892461] = arrayDataUpdated[j50892461 - 1];
            } else {
                arrayDataUpdated[j50892461] = l1143453413;
                f1702496723 = true;
              break;
            }
        }
        if (!f1702496723) {
            arrayDataUpdated[0] = l1143453413;
        }
    }
}
    if (m_0.sn(1)) {
    int m1910608582 = arrayData.length;
    arrayDataUpdated = new int[m1910608582];
    for (int g2074187290 = 0; g2074187290 < m1910608582; g2074187290++) {
        arrayDataUpdated[g2074187290] = arrayData[g2074187290];
    }
    int o1922514582 = 0;
    for (int h261779395 = 0; h261779395 < m1910608582; h261779395++) {
        for (int t2073539929 = 1; t2073539929 < m1910608582; t2073539929++) {
            if (arrayDataUpdated[t2073539929 - 1] > arrayDataUpdated[t2073539929])
                o1922514582 = arrayDataUpdated[t2073539929 - 1];
                arrayDataUpdated[t2073539929 - 1] = arrayDataUpdated[t2073539929];
                arrayDataUpdated[t2073539929] = o1922514582;
            }
        }
```

Figure 4.4.2.2: Sort method of Obfuscated Data Processing Code

```
    public void searchData() {
    if (m_1 == null) {
    m_1 = new NDG(3, "m_1", stm);
} else {
    m_1.mvn();
}
    if (m_1.sn(0)) {
    for (int e1834258326 = 0; e1834258326 < arrayData.length; e1834258326++) {
        if (arrayData[e1834258326] == value) {
            isValueFound = true;
            break;
        }
    }
}
    if (m_1.sn(2)) {
    int f1813592123 = arrayDataUpdated.length;
    int p1542792409 = 0;
    int i1669784488 = f1813592123;
    while (true) {
        if (p1542792409 > i1669784488) {
            break;
        } else if (p1542792409 == i1669784488) {
            if (value == arrayDataUpdated[p1542792409]) {
                if (value == arrayDataUpdated[p1542792409]) {
                    isValueFound = true;
                    break;
                }
            } else {
                int x325657563 = (p1542792409 + i1669784488) / 2;
                if (value == arrayDataUpdated[x325657563]) {
                    isValueFound = true;
                    break;
                } else if (value < arrayDataUpdated[x325657563]) {
                    i1669784488 = x325657563;
                } else {
                    p1542792409 = x325657563;
                }
            }
        }
}
    if (m_1.sn(1)) {
    for (int u341262682 = 0; u341262682 < arrayDataUpdated.length; u341262682++)
        if (arrayDataUpdated[u341262682] == value) {
            isValueFound = true;
            break;
        } else if (arrayDataUpdated[u341262682] > value) {
            break;
        }
    }
}
    }

```

Figure 4.4.2.3: Search method of Obfuscated Data Processing Code

4.4.3. Cyclomatic Complexity of Original and Obfuscated Code

This section augments the mathematical analysis performed on obfuscation scheme with computation of cyclomatic number (a software complexity metric described by McCabe [26]) of original and obfuscated codes. Cyclomatic complexity of a flow graph V (G) is equal to P + 1 where P is number of predicates present in code. It captures upper bound on the total number of independent paths present in code that form basis set (basis path testing by McCabe [26]). An independent path is any path which traverses at least one new set of code statements (nodes) or conditions (edges).

Cyclomatic number for methods "sortData" and "searchData" of original data processing code is 5 and 3 respectively. This is shown in Figure 4.4.3.1. Cyclomatic number for methods "sortData" and "searchData" of obfuscated data processing code is 19 and 16 respectively. This is shown in the Figure 4.4.3.2. Anckaert and others in paper [16] describe that program is more obfuscated if control flow of code (measured in terms of cyclomatic number) is highly complicated. Cyclomatic number is computed using PMD tool [29] (a static analysis tool). Thus, software complexity of the obfuscated code is higher than that of the original code. Thus, the obfuscated code increases attacker's static analysis efforts.

Figure 4.4.3.1: Cyclomatic Number of Original Data Processing Code

33	src/DataProcessor_Obfuscated.java	68	Avoid printStackTrace(); use a logger call instead.
34	src/DataProcessor_Obfuscated.java	72	The method 'sortData' has a Cyclomatic Complexity of 19.
35	src/DataProcessor_Obfuscated.java	72	The method sortData() has an NPath complexity of 4050
36	src/DataProcessor_Obfuscated.java	81	System.arraycopy is more efficient
37	src/DataProcessor_Obfuscated.java	86	Found 'DD'-anomaly for variable 'z255406657' (lines '86'-'89').
38	src/DataProcessor_Obfuscated.java	89	Found 'DD'-anomaly for variable 'z255406657' (lines '89'-'89').
39	src/DataProcessor_Obfuscated.java	102	System.arraycopy is more efficient
40	src/DataProcessor_Obfuscated.java	106	Found 'DU'-anomaly for variable 'e552378090' (lines '106'-'139').
41	src/DataProcessor_Obfuscated.java	107	Found 'DD'-anomaly for variable 'o1708414748' (lines '107'-'113').
42	src/DataProcessor_Obfuscated.java	125	System.arraycopy is more efficient
43	src/DataProcessor_Obfuscated.java	128	Found 'DD'-anomaly for variable 'a996899852' (lines '128'-'132').
44	src/DataProcessor_Obfuscated.java	128	Found 'DU'-anomaly for variable 'a996899852' (lines '128'-'139').
45	src/DataProcessor_Obfuscated.java	141	The method 'searchData' has a Cyclomatic Complexity of 16.
46	src/DataProcessor_Obfuscated.java	141	The method searchData() has an NPath complexity of 320

Figure 4.4.3.2: Cyclomatic Number of Obfuscated Data Processing Code

4.4.4. Code Coverage of Obfuscated Code

This section provides statement code coverage of the obfuscated code gathered by using Emma code coverage tool [30] (a dynamic analysis tool). The Figure 4.4.4.1, Figure 4.4.4.2, Figure 4.4.4.3, Figure 4.4.4.4, Figure 4.4.4.5 and Figure 4.4.4.6 show that all statements in the obfuscated code clones fragments are valid (highlighted with green colour) and thus, obfuscated code does not contain any fake control paths. The obfuscation scheme introduces valid control paths in the obfuscated code. Thus, the obfuscation scheme increases attacker's dynamic analysis efforts as number of valid control flow paths are exponential in terms of code clones and logical code fragments.

```
72      public void sortData() {
73      if (m_0 == null) {
74      m_0 = new NDG(3, "m_0", stm);
75  } else {
76      m_0.mvn();
77  }
78      if (m_0.sn(2)) {
79      int a1940733601 = arrayData.length;
80      arrayDataUpdated = new int[a1940733601];
81      for (int z1974462879 = 0; z1974462879 < a1940733601; z1974462879++) {
82          arrayDataUpdated[z1974462879] = arrayData[z1974462879];
83      }
84      for (int f1684376023 = 0; f1684376023 < a1940733601; f1684376023++) {
85          int n1386676837 = arrayDataUpdated[f1684376023];
86          int z255406657 = -1;
87          for (int w307268958 = f1684376023 + 1; w307268958 < a1940733601; w307268958++)
88              if (arrayDataUpdated[w307268958] < n1386676837) {
89                  z255406657 = w307268958;
90                  n1386676837 = arrayDataUpdated[w307268958];
91              }
92          }
93          if (z255406657 != -1) {
94              arrayDataUpdated[z255406657] = arrayDataUpdated[f1684376023];
95              arrayDataUpdated[f1684376023] = n1386676837;
96          }
97      }
98  }
```

Figure 4.4.4.1: Code Coverage - Selection Sort Code Clone

```
 99      if (m_0.sn(1)) {
100     int j791267447 = arrayData.length;
101     arrayDataUpdated = new int[j791267447];
102     for (int c1460318 = 0; c1460318 < j791267447; c1460318++) {
103         arrayDataUpdated[c1460318] = arrayData[c1460318];
104     }
105     for (int z1599154569 = 1; z1599154569 < j791267447; z1599154569++) {
106         int e552378090 = arrayDataUpdated[z1599154569];
107         boolean o1708414748 = false;
108         for (int s990386792 = z1599154569; s990386792 > 0; s990386792--) {
109             if (e552378090 < arrayDataUpdated[s990386792 - 1]) {
110                 arrayDataUpdated[s990386792] = arrayDataUpdated[s990386792 - 1];
111             } else {
112                 arrayDataUpdated[s990386792] = e552378090;
113                 o1708414748 = true;
114                 break;
115             }
116         }
117         if (!o1708414748) {
118             arrayDataUpdated[0] = e552378090;
119         }
120     }
121 }
```

Figure 4.4.4.2: Code Coverage - Insertion Sort Code Clone

```
122     if (m_0.sn(0)) {
123     int c304244384 = arrayData.length;
124     arrayDataUpdated = new int[c304244384];
125     for (int y1751033464 = 0; y1751033464 < c304244384; y1751033464++) {
126         arrayDataUpdated[y1751033464] = arrayData[y1751033464];
127     }
128     int a996899852 = 0;
129     for (int c597356788 = 0; c597356788 < c304244384; c597356788++) {
130         for (int c1942352882 = 1; c1942352882 < c304244384; c1942352882++) {
131             if (arrayDataUpdated[c1942352882 - 1] > arrayDataUpdated[c1942352882]) {
132                 a996899852 = arrayDataUpdated[c1942352882 - 1];
133                 arrayDataUpdated[c1942352882 - 1] = arrayDataUpdated[c1942352882];
134                 arrayDataUpdated[c1942352882] = a996899852;
135             }
136         }
137     }
138 }
139     }
```

Figure 4.4.4.3: Code Coverage - Bubble Sort Code Clone

```
141     public void searchData() {
142     if (m_1 == null) {
143     m_1 = new NDG(3, "m_1", stm);
144 } else {
145     m_1.mvn();
146 }
147     if (m_1.sn(2)) {
148     for (int g1213302694 = 0; g1213302694 < arrayData.length; g1213302694++) {
149         if (arrayData[g1213302694] == value) {
150             isValueFound = true;
151             break;
152         }
153     }
154 }
```

Figure 4.4.4.4: Code Coverage - Sequential Search Code Clone

```
155     if (m_1.sn(0)) {
156     int m1926592784 = arrayDataUpdated.length;
157     int y355866210 = 0;
158     int e616522352 = m1926592784;
159     while (true) {
160         if (y355866210 > e616522352) {
161             break;
162         } else if (y355866210 == e616522352) {
163             if (value == arrayDataUpdated[y355866210]) {
164                 isValueFound = true;
165                 break;
166             }
167         } else {
168             int d918923957 = (y355866210 + e616522352) / 2;
169             if (value == arrayDataUpdated[d918923957]) {
170                 isValueFound = true;
171                 break;
172             } else if (value < arrayDataUpdated[d918923957]) {
173                 e616522352 = d918923957;
174             } else {
175                 y355866210 = d918923957;
176             }
177         }
178     }
179 }
```

Figure 4.4.4.5: Code Coverage - Binary Search Code Clone

```
180     if (m_1.sn(1)) {
181     for (int x904480249 = 0; x904480249 < arrayDataUpdated.length; x904480249++) {
182         if (arrayDataUpdated[x904480249] == value) {
183             isValueFound = true;
184             break;
185         } else if (arrayDataUpdated[x904480249] > value) {
186             break;
187         }
188     }
189 }
190     }
```

Figure 4.4.4.6: Code Coverage - Sorted Search Code Clone

4.4.5. Execution Traces of Original and Obfuscated Code

This section shows execution traces of original and obfuscated codes on same set on inputs. Output of the original code is shown in the Figure 4.4.5.1. For all paths in the obfuscated code the same output is generated. Output of the obfuscated code is shown in the Figure 4.4.5.2.

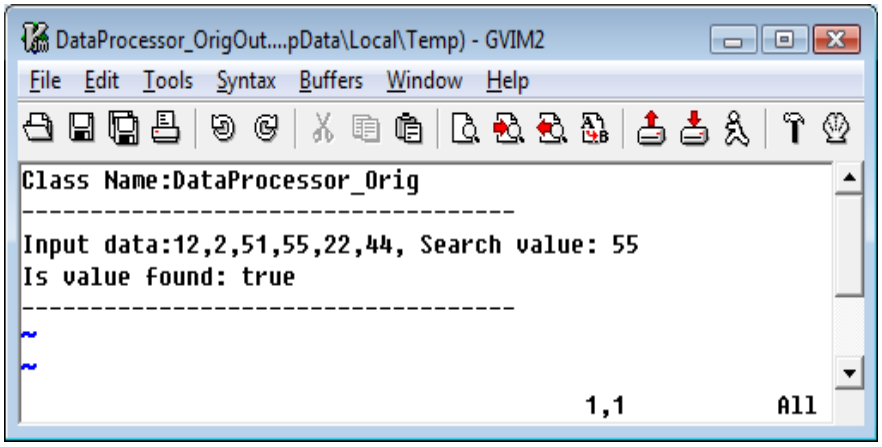


Figure 4.4.5.1: Execution trace of Original Code

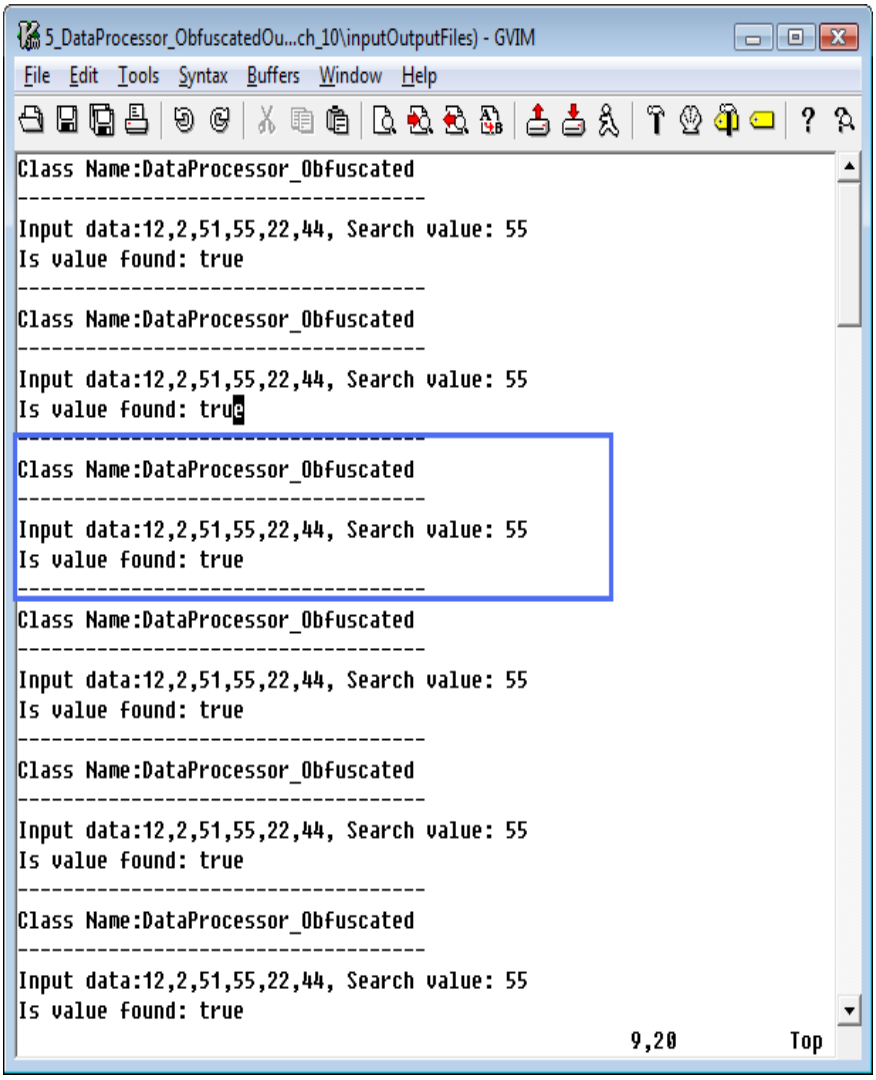


Figure 4.4.5.2: Execution trace of Obfuscated Code

The obfuscated code contains 9 (3*3) valid execution paths (as shown in the Figure 4.4.5.3) corresponding to single execution path in the original code. Each line shows a unique combination of code clone fragments which are selected for execution of software. For example, strings “m_0 cloneID-2” and “m_1 cloneID-1” together select one possible combination of sort and search code fragments respectively. Note that string “m_0” indicates predicate variable for sort method and string “m_1” indicates predicate variable for search method. Each method contains 3 clones. These clones are identified by integers in range [0-2].

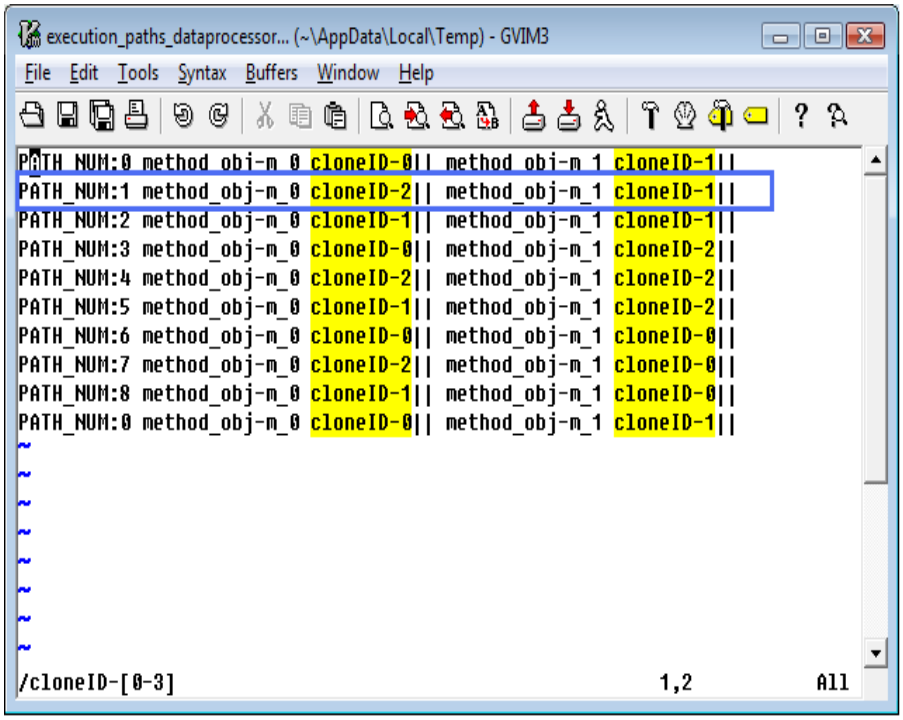


Figure 4.4.5.3: Exponential Number of Unique Paths for Data Processing Application

4.4.6. Evaluation of Code Obfuscation Scheme

In this section, we measure effectiveness of obfuscated code constructed by applying the scheme on the set of criteria (C-1, C-2 and C-3) (described in section [3.2]) to evaluate our code obfuscation scheme.

4.4.6.1. C-1: Potency, Resilience and Cost

The obfuscation scheme increases software complexity metrics (such as cyclomatic number described by McCabe [26]) of obfuscated code. Software complexity of obfuscated code is higher than that of original code. Thus, the obfuscation scheme

satisfies criterion C-1. As the scheme applies identifier renaming technique, potency of obfuscated code is medium (described by Collberg and others in [1]). Also, Ceccato and others [7] experimentally show that attacker's efforts are increased if code is obfuscated using identifier renaming technique. Resilience of obfuscated code to automated attacks is strong. The scheme resists automated attacks such as code clone detection attacks launched on obfuscated code. This is because of construction of non trivial code clones. These code clones have different code structure but they perform same computation. Thus, such code clones cannot be detected using existing clone detection techniques such as matching Abstract Syntax Tree (AST) described by Baxter and others [25]. This is shown with an example in the experimentation section [5.4]. Cost of obfuscation can be controlled by developer. More number of code clones introduced for obfuscation higher software complexity of obfuscated code, but higher development efforts to construct obfuscated code. Cost measure can be controlled depending upon software protection requirement. In the experimentation section [5], we show that performances of obfuscated and original code are comparable and there is negligible performance loss due to code obfuscation. This is because of use of identifier renaming technique (which has no cost overhead as described in paper [1]) for obfuscation of logical code clone fragments and linking code clone fragments using dynamic predicate variables manufactured using computationally inexpensive constructs such as linked list. This construction of predicate variables is similar to the construction described by Collberg and others in paper [9] to manufacture cheap static opaque predicates using graphs.

4.4.6.2. C-2: Resistance to Static and Dynamic Analysis Attacks

Obfuscated code constructed by applying the obfuscation scheme resists static and dynamic analysis attacks. As shown in the section [4.4.3], cyclomatic number of the obfuscated code is higher than that of the original code due to insertion of non trivial code clones and dynamic predicate variables. Control flow graph (CFG) of obfuscated code is more complicated than the original code. Thus, static analysis efforts on obfuscated code are increased. Anckaert and others [16] describe that more complicated control flow, higher code obfuscation and thus higher static analysis efforts.

The obfuscation scheme introduces exponential number of valid control flow paths in obfuscated code. If there are N logical code fragments and K code clones per fragment,

K^N number of paths are introduced in obfuscated code corresponding to single path present in original code. This is shown in the sections [4.3] and [4.4.5]. Attacker needs to execute software multiple times to obtain K^N execution traces to understand software functionality such as license checking mechanism. Thus, obfuscated code resists dynamic analysis attacks. Thus, the obfuscation scheme satisfies criterion C-2.

4.4.6.3. C-3: Increase in Program State Space

State space of obfuscated program is larger than that of original program. State space is increased by inserting non trivial code clones created for logical code fragments present in original code. As non trivial code clones have different code structure (such as Abstract Syntax Tree [AST]), it is not possible to detect such code clones using existing clone detection techniques such as matching AST (described by Baxter and others [25]). This is shown with a few examples described in the experimentation section [5.4]. Thus, it is not possible detect and remove non trivial code clones present in obfuscated code to minimize its state space. Thus, the obfuscation scheme satisfies criterion C-3.

4.4.7. Advantages and Disadvantages of Obfuscation Scheme

4.4.7.1. Advantages of Obfuscation Scheme

In this section advantages of the obfuscation scheme are presented.

- **Simple to Apply**

  The obfuscation scheme is simple to apply as it does not require any complicated operations, such as time sensitive codes [21] (which require accurate timing information of guard codes to be obtained by profiling original code) or need for distributed environment [20] (in which developer needs to store and retrieve code fragments on trusted network entities during execution of software). Our obfuscation scheme does not need profiling information and it is applicable to protection of software running on standalone machine. Thus, the obfuscation scheme is simple to apply.

- **Scheme Satisfies Code Obfuscation Evaluation Criteria**

The obfuscation scheme satisfies existing sets of criteria described in the literature to measure effectiveness of code obfuscation. The obfuscation scheme increases software complexity metrics (such as cyclomatic number [26]) and resists static and dynamic analysis attacks possible on obfuscated code.

- **Less Performance Overhead**

  Performance of obfuscated code is comparable to that of original code. Execution time overhead of obfuscated code is negligible. Also, memory consumed by obfuscated code is comparable to memory consumed by original code. This is shown with an example in the experimentation section [5]. Thus, the obfuscation scheme does not reduce performance or increase memory footprint of obfuscated code.

- **Source Code Level Obfuscation**

  The obfuscation scheme performs obfuscation at source code level of software. Developer has complete knowledge of obfuscated code. The scheme performs obfuscating transformations (for example, introduction of non trivial code clones) which cannot be undone by compiler during its optimization phase (such as elimination of common sub expressions). Madou and others in paper [18] evaluate effectiveness of transformations (such as insertion of opaque predicates) performed at source code level. Our obfuscation scheme introduces similar transformations (for example, use of dynamic predicate variables) at source code level. Thus, the transformations applied by the obfuscation scheme are equally effective.

4.4.7.2. Disadvantages of Obfuscation Scheme

In this section disadvantages of the obfuscation scheme are presented.

- **High Development Cost**

  As developer needs to divide original code of software into logical code fragments and developer has to construct non trivial code clones for logical code fragments, software development efforts are increased.

- **Difficulty of Automation**

  Developer needs to create logical code fragments and corresponding non trivial code clones to obfuscate code. Due to these constructions, it is difficult to automate the obfuscation scheme to generate obfuscated code. Thus, the obfuscation scheme requires manual efforts.

5. Experimentation and Results

In this section, we present results of experimentation on original and obfuscated code (for a few applications such as sequential data processing code which takes an integer dataset as input). Obfuscated code is constructed by applying the obfuscation scheme. Karnick and others [15] describe evaluation of cost measure of code obfuscation using the following three parameters:

- Storage (Lines of Code)
- Run time (Execution time)
- Memory

We carry out experimentation by executing original and obfuscated code and comparing parameters such as execution time and memory usage. We also compare sizes of both original and obfuscated code.

We find that execution time and memory footprint of obfuscated code is comparable and performance of original and obfuscated application is almost equal. Thus, overhead introduced due to code obfuscation (for example, insertion of dynamic predicate variables) is negligible. The detailed results of experimentation can be found in the subsections below.

Machine Configuration for Experimentation:

The experimentation is carried out on desktop machine having following configuration:

Processor: Intel Core Duo, 2.10 GHz

RAM Memory: 2 GB

OS: Windows Vista 32 bit

Execution time and memory footprint data is captured using Jensor tool [31] (Java profiler tool). A few screenshots for data processing application are shown in Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4.

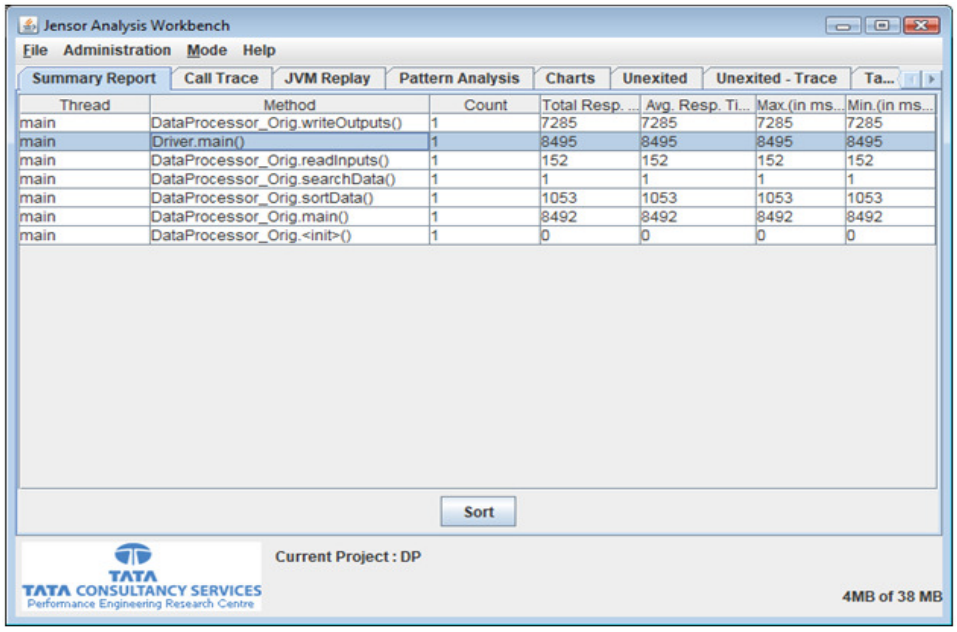


Figure 5.1: Execution Time of Original Data Processing Code

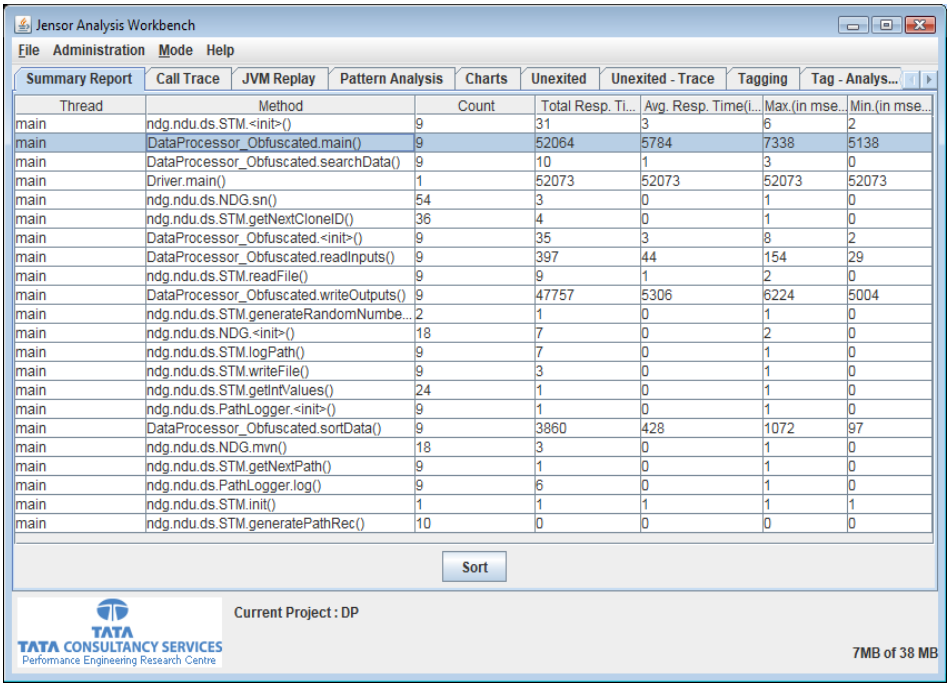


Figure 5.2: Execution Time of Obfuscated Data Processing Code

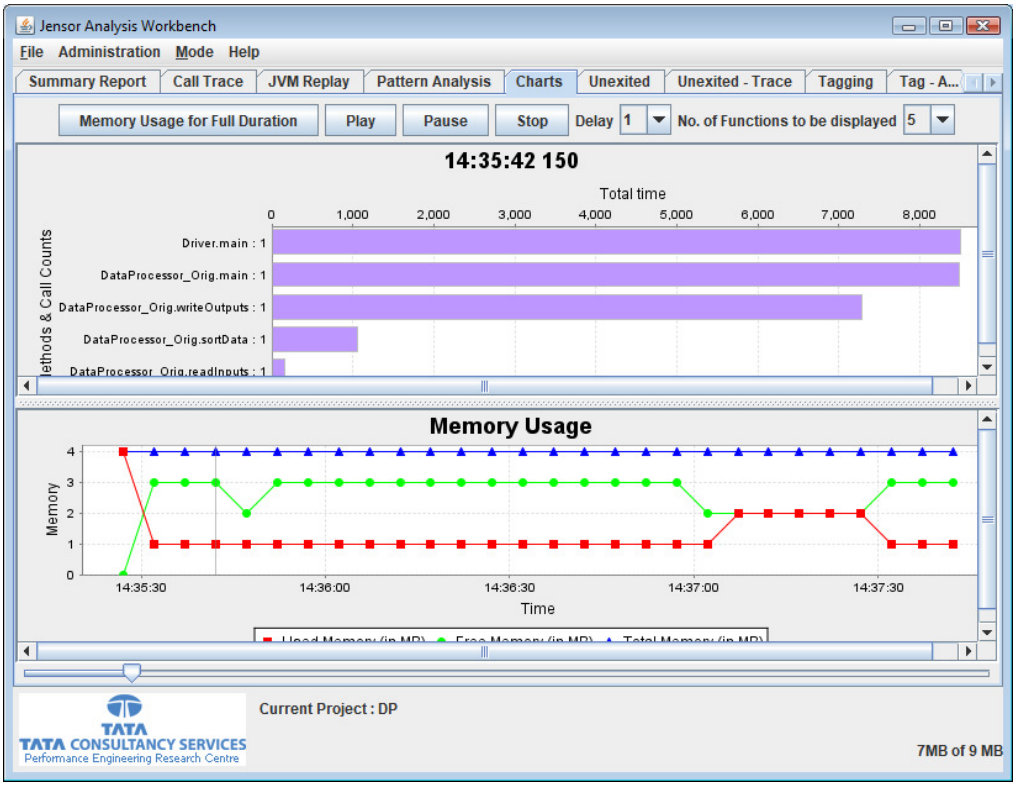


Figure 5.3: Memory Usage of Original Data Processing Code

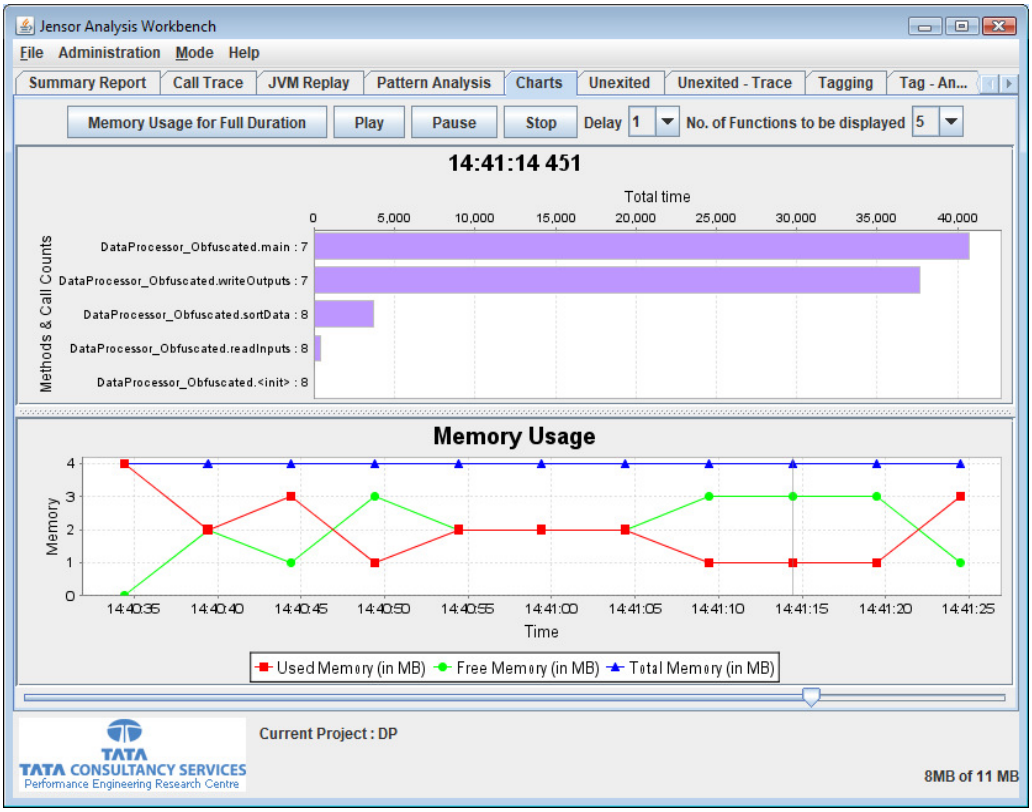


Figure 5.4: Memory Usage of Obfuscated Data Processing Code

5.1.Storage (Code Size)

As the obfuscation scheme introduces non trivial code clones, hence size of code (in terms of lines of code (LOC)) of obfuscated code is always greater than that of original code. More the code clones introduced for obfuscation, larger size of obfuscated code. But we believe that this measure is less accurate as it is possible to change code size by obscuring code (for example, addition or removal of empty lines or code comments) during obfuscation. The Table 5.1-1 shows results of comparison of code sizes of original and obfuscated programs.

Table 5.1-1: Sizes of Original and Obfuscated Codes

Code Size (LOC)	Method Name	Original Code	Obfuscated Code
Data Processing Code (Non trivial clones)	sortData	23	67
	searchData	9	50
Data Processing Code (trivial clones)	sortData	23	57
	searchData	9	27
Data Processing Code (25 trivial clones)	sortData	23	432
	searchData	9	181

Note that trivial codes are constructed by renaming identifiers. This is to check scalability of obfuscation scheme. Generation of trivial code clone for obfuscation can be automated; hence it is useful for experimenting scalability of our approach.

5.2.Execution Time

We observed little execution time overhead for obfuscated data processing application. The Table 5.2-1 presents execution time taken by original code and obfuscated code with non trivial code clones (such as select, bubble and insert clones for sorting fragment). The "Configuration" column in the table shows unique configurations used for experimentation. The "data size" column indicates number of integer elements present in data file taken as input (for example, 10K) by the applications. The "Original Code (Time)" shows time taken by application with original code to process data. The "Obfuscated Code (Max time)" column captures the maximum time taken by a path (among all valid paths) present in application with obfuscated code. Note that the "Obfuscated Code (Total time)" column is computed by adding up time taken by all the paths present in the obfuscated code. It shows cumulative time taken by obfuscated code.

The "Obfuscated Code (Avg time)" column is computed by dividing total time by number of paths present in obfuscated code.

Table 5.2-1: Execution Time of Data Processing Application with Non Trivial Code Clones

Configuration	Data Size	Original Code (Time)	Obfuscated Code (Max Time)	Obfuscated Code (Total time)	Obfuscated Code (Avg Time)
N = 2, K = 3 Total Paths = 9	10K	8 Sec	7 Sec	52 Sec	6 Sec
N = 2, K = 3 Total Paths = 9	25K	1 Min 50 Sec	1 min 35 sec	13 Min	1 Min 25 Sec
N = 2, K = 3 Total Paths = 9	50K	9 Min	9 Min	71 Min	8 Min

We find that execution time for original and obfuscated code is almost same when code fragments (N = 2) and code clones (K = 3) are small in number. Thus, no overhead is observed due to obfuscation when number of code fragments and code clones are small.

The Table 5.2-2 presents time taken by original code and obfuscated code with trivial code clones (that is, using same algorithm such as bubble sorting for all code clones). By varying the obfuscation scheme parameters (K and N) and executing the codes multiple times (for example, in a loop), scalability of approach is checked. This scenario precisely captures the overhead introduced due to dynamic predicate variables used for linking obfuscated code clone fragments. We found that the code which generates dynamic predicate variables is computationally inexpensive (as there is little cost overhead observed during execution of obfuscated code). This result validates the approach described by Collberg and others in paper [9] in which dynamic structures (such as graphs) are used in manufacturing cheap static opaque predicates.

Table 5.2-2: Execution Time of Data Processing Application with Trivial Code Clones

Configuration	Data Size	Original Code (Time)	Obfuscated (Max Time)	Obfuscated Code (Total Time)	Obfuscated Code (Avg Time)
Scaling K N = 2, K = 25 Total Paths = 625	10K	6 Sec	6 Sec	64 Min	6 Sec
Scaling N: N = 10, K = 2	10K	10 Sec	11 Sec	2 hour 20 Min	8 Sec

Total Paths = 1024					
Scaling N and K: N = 5, K = 4 Total Paths = 1024	10K	8.6 Sec	8.8 Sec	1 hour 46 Min	6.2
Execution in Loop: N = 2, K = 2 Loop Counter = 500	10K	8 Min	8 Min	32 Min	8 Min

The Table 5.2-3 shows results of execution time of original and obfuscated code for sorting application.

Table 5.2-3: Execution Time of Sorting Program with Trivial Code Clones

Configuration	Data size	Original Code (Time)	Obfuscated Code (Max time)	Obfuscated Code (Total time)	Obfuscated Code (Avg time)
N = 2, K = 2 Total Paths=4 Loop Count = 50K * 50K	50K	20 Sec	62 Sec	4 Min	62 Sec
N = 2, K = 2 Total Paths=4 Loop Count = 100K * 100K	100K	83 Sec	4 Min 15 Sec	17 Min	4 Min 15 Sec
N = 2, K = 2 Total Paths=4 Loop Count = 200K * 200K	200K	5.6 Min	8.7 Min	35 Min	8.7 Min

For sorting application, substantial performance loss was observed. The obfuscated sorting application runs approximately 3 times slower than that of original application. This is because of creation of logical code fragment for swap functionality. Swap fragment is performance critical code for sorting functionality. Predicate variables inserted at swap code fragment increased execution time of obfuscated code. To reduce performance loss of obfuscated code, creation of logical code fragmentation for performance critical code should be avoided.

5.3.Memory

The Table 5.3-1, Table 5.3-2 and Table 5.3-3 show results of memory usage of original and obfuscated codes for data processing and sorting applications respectively. The difference in memory footprint of original and obfuscated application is negligible. Hence, obfuscated code does not increase memory requirement of application.

Table 5.3-1: Memory Usage of Data Processing Application with Non Trivial Code Clones

Configuration	Data size	Original Code (Memory Usage) (MB)	Obfuscated Code (Memory Usage) (MB)
N = 2, K = 3 Total Paths = 9	10K	2	4
N = 2, K = 3 Total Paths = 9	25K	4	4
N = 2, K = 3 Total Paths = 9	50K	4	5

Table 5.3-2: Memory Usage of Data Processing Application with Trivial Code Clones

Obfuscation Configuration	Data Size	Original Code (Memory Usage) (MB)	Obfuscated Code (Memory Usage) (MB)
Scaling K: N = 2, K = 25 Paths = 625	10K	4	4
Scaling N: N = 10, K = 2 Paths = 1024	10K	4	4
Scaling N and K: N = 5, K = 4 Paths = 1024	10K	4	4
Execution in Loop: N = 2, K = 2 Loop Counter = 500	10K	3	4

Table 5.3-3: Memory Usage of Sorting Program with Trivial Code Clones

Configuration	Data size	Original Code (Memory Usage) (MB)	Obfuscated Code (Memory Usage) (MB)
N = 2, K = 2 Total Paths=4	50K	2	3

Loop Count = 50K * 50K			
N = 2, K = 2 Total Paths=4 Loop Count = 100K * 100K	100K	4	4
N = 2, K = 2 Total Paths=4 Loop Count = 200 * 200	200K	6	6

5.4. Code Clone Detection

This section shows results of experimentation performed for detecting code clones (for sort and search fragments) introduced in obfuscated code. The Table 5.4-1 and Table 5.4-2 show results of detecting trivial and non trivial code clones respectively using JCCD tool [34] (Java Code Clone Detection Tool). Results of detection of code clones are shown in column named "is detected" in these tables. Note that trivial code clones (for example Bubble-1 and Bubble-2 shown in columns "Clone-1" and "Clone-2" respectively) are created using identifier renaming technique. These trivial code clones can be detected using existing clone detection technique (described by Baxter and others in [25]) but non trivial code clones (for example, Bubble and Insert code clones) cannot be detected using the technique.

Table 5.4-1: Detection of Trivial Code Clones

Functionality	Clone-1	Clone-2	Is detected
Sort fragment	Bubble-1	Bubble-2	Yes
	Insert-1	Insert-2	Yes
	Select-1	Select-2	Yes
Search fragment	Sequential-1	Sequential-2	Yes
	Sorted-1	Sorted-2	Yes
	Binary-1	Binary-2	Yes

Table 5.4-2: Detection of Non Trivial Code Clones

Functionality	Clone-1	Clone-2	Is detected
Sort fragment	Bubble	Insert	No
	Bubble	Select	No
	Insert	Select	No
Search fragment	Sequential	Sorted	No
	Sequential	Binary	No

	Sorted	Binary	No

Sample output for detection of trivial code clones is shown in the Figure 5.4.1 below. Note that “Similarity Group 6” indicates that code structures (such as Abstract, Syntax Tree) are identical.

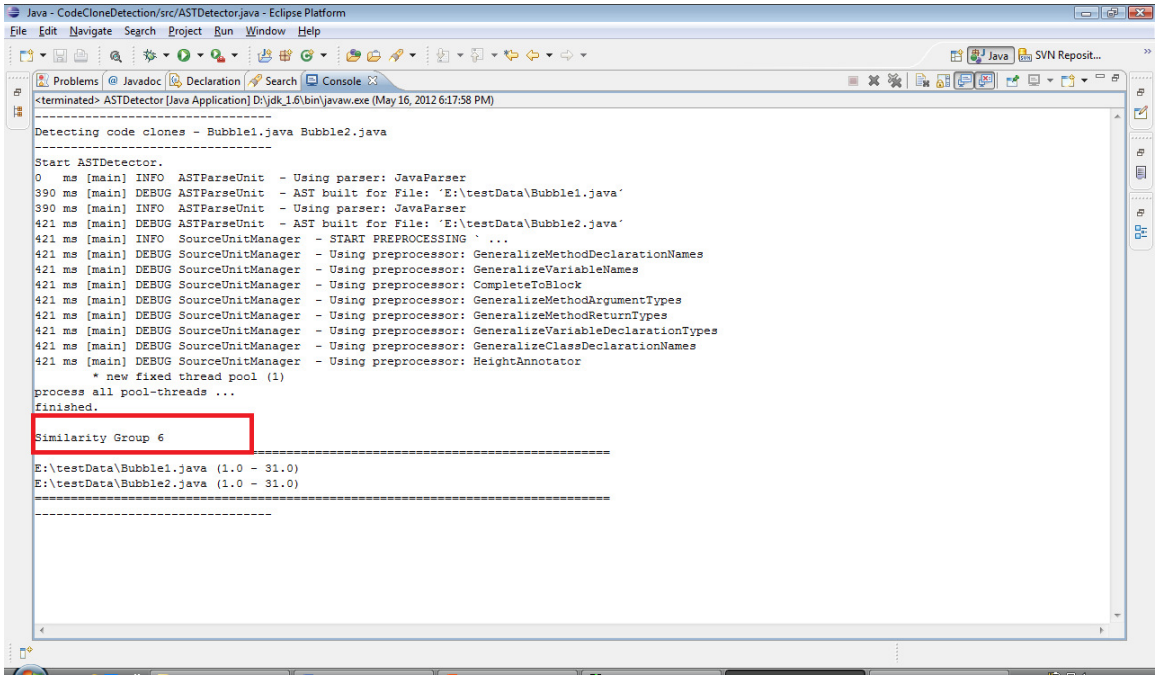


Figure 5.4.1: Sample Output of Trivial Code Clones Detection

Sample output for detection of non trivial code clones is shown in the Figure 5.4.2 below:

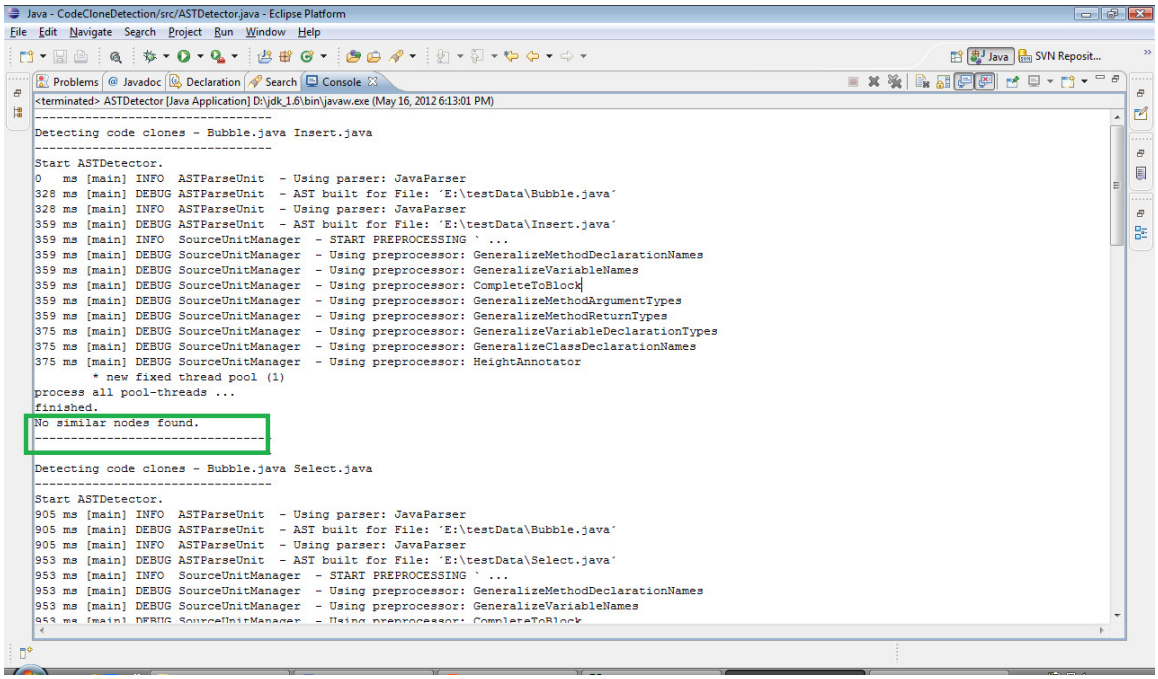


Figure 5.4.2: Sample Output of Non Trivial Code Clones Detection

6. Architecture and Design of Code Obfuscation Tool

This section describes architecture and design of code obfuscation tool (shown in the Figure 6.1.1). The tool implements the obfuscation scheme. The tool also enables addition of new obfuscation techniques by providing interface functionality.

6.1. Architectural Block Diagram

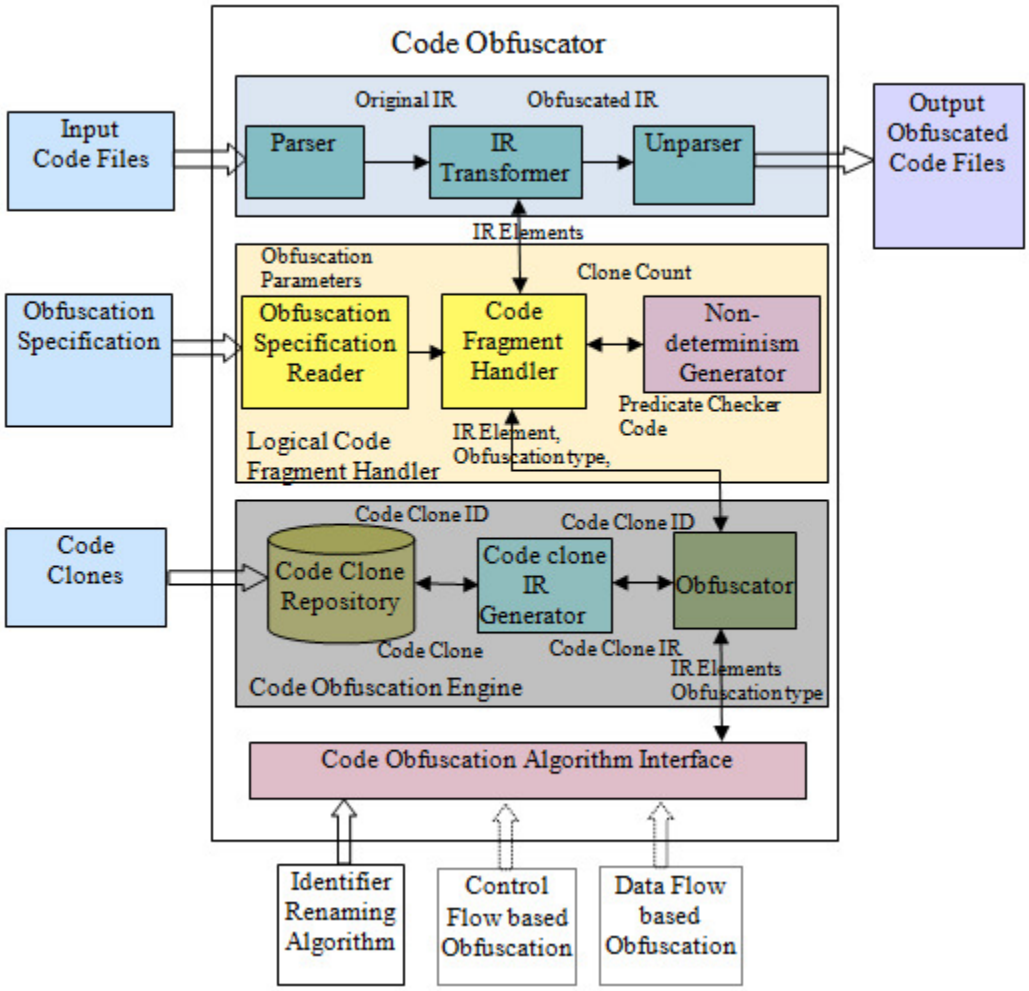


Figure 6.1.1: Architecture of Code Obfuscation Tool

6.2. Architectural Components Description

The tool consists of the following key components:

- Language Processing component
- Logical Code Fragment Handler component
- Code Obfuscation Engine
- Code Obfuscation Algorithm Interface

The tool supports identifier renaming obfuscation technique to obfuscate code clone fragments by default. The tool also enables addition of custom code obfuscation technique (such as control flow based obfuscation technique).

Inputs and outputs of the Code Obfuscator tool are as follows:

Inputs:

Original Code Files

Obfuscation Specification File

Code Clone Files

Outputs:

Obfuscated Code Files

The following subsections describe architectural blocks of the Code Obfuscator tool.

6.2.1. Language Processing Component

- Parser: Parser parses input source code files to construct Intermediate Representation (IR). The constructed IR is given to IR Transformer component and Logical Code Fragment Handler component for further processing.
- IR Transformer: IR transformer component replaces IR of original source code with IR of obfuscated source code. This component traverses IR and using code fragment handler component, it checks whether any IR element (for example, language construct such as "method" in Java) requires obfuscation or not. If yes, it passes the IR element to code fragment handler component to obtain IR of obfuscated code.

- Unparser: Unparser component generates obfuscated source code files as output. This component takes IR of obfuscated source code as input from IR Transformer component.

6.2.2. Logical Code Fragment Handler

- Obfuscation Specification Reader: Obfuscation specification reader component takes obfuscation specification file as input. Obfuscation specification file contains obfuscation parameters (for example, method names present in original source code files). For each logical code fragment (for example, method), it contains parameters such as code clones to be used for obfuscation and type of obfuscation (by default identifier renaming is used). Obfuscation specification reader component reads obfuscation specification file and constructs intermediate data structure containing obfuscation parameters. This data structure is passed to Code Fragment Handler component for further processing.

- Code Fragment Handler: This component takes obfuscation specification data structure as input from Obfuscation Specification Reader component. It also takes IR element as input from IR Transformer component. This component checks whether IR element requires obfuscation or not. If yes, it passes original IR element and obfuscation parameters (code clones identifiers, code obfuscation type) to code obfuscation engine. The code obfuscation engine component returns obfuscated code clones to Code Fragment Handler component. Code Fragment Handler passes code clone counts to non determinism generator. Non determinism generator returns code for dynamic predicates required for linking obfuscated code clone fragments.

- Non determinism Generator: The Non determinism Generator component takes clone counts for all logical fragments as input. It returns code to check dynamic predicates used for selecting a particular code clone combination during execution of obfuscated code. The component also generates of all possible code clone

combination using random number generation. A code clone combination is selected randomly for a given run of obfuscated software.

6.2.3. Code Obfuscation Engine

- Code Clone Repository: Code Obfuscation Engine consists of code clone repository which stores code clones for functionalities such as swap, sort operations. Each code clone can be identified uniquely in code repository using key (clone identifier) such as Java file path and method name. Developers add code clones manually to the code repository. These code clones are fetched and processed by Code Clone IR Generator component.
- Code Clone IR Generator: The code clone IR generator component takes code clone key (or clone identifier) as input from Code Logical Code Fragment Handler component. It obtains code clone file from Code Clone Repository using the key. This component generates IR for code clone using parser component. This code clone IR is passed to Obfuscator component for further processing.
- Obfuscator: Obfuscator takes original IR element, code clone identifier and type of obfuscation as inputs from Code Fragment Handler component. It passes code clone key to Code Clone IR generator component and it obtains IR of code clone. By default, code clones are obfuscated by applying identifier renaming technique. But this component can pass IR elements via code obfuscation algorithm interface for custom code obfuscation.

6.2.4. Code Obfuscation Algorithm Interface

This component provides interfacing functionality between internal obfuscator and external code obfuscation techniques. The internal code obfuscator supports identifier renaming technique for generation of obfuscated code clones by default. But developer can add other code obfuscation algorithms such as control flow based obfuscation and data flow based obfuscation depending upon obfuscation requirement.

- Identifier Renaming Algorithm: The Obfuscator component supports identifier renaming algorithm for obfuscation. This algorithm takes original IR elements and / or code clone IR as input. It replaces original meaningful names (such as local variable names) with random meaningless names to achieve code obfuscation.

6.3.Example of Obfuscated Code using the Tool

The complete example on code obfuscation scheme applied to sequential data processing code can be found in appendix [9]. The example contains the following files:

a) Input code file - Data Processor

   The contents of original code file can be found in section [9.1]. It contains four methods namely readInputs, sortData, searchData, writeOutputs. These methods are invoked from main method in a sequence.

b) Input code clones - Sort and Search functionality

   The contents of sort and search code clone files can be found in section [9.2]. The code file for Sort fragment contains 3 code clones for sorting functionality namely bubble sort, insert sort and select sort. The code file for Search fragment contains 3 code clones for searching functionality namely sequential search, sorted search and binary search. These code clone files reside in code repository.

c) Obfuscation specification XML File

   The obfuscation specification file describes obfuscation parameters for data processing application. The Figure 6.3.1 shows an example of obfuscation specification. It shows that developer specifies obfuscation of methods “sortData” and “searchData” using 3 code clones written in files “Sort.Java” and “Search.Java” respectively.

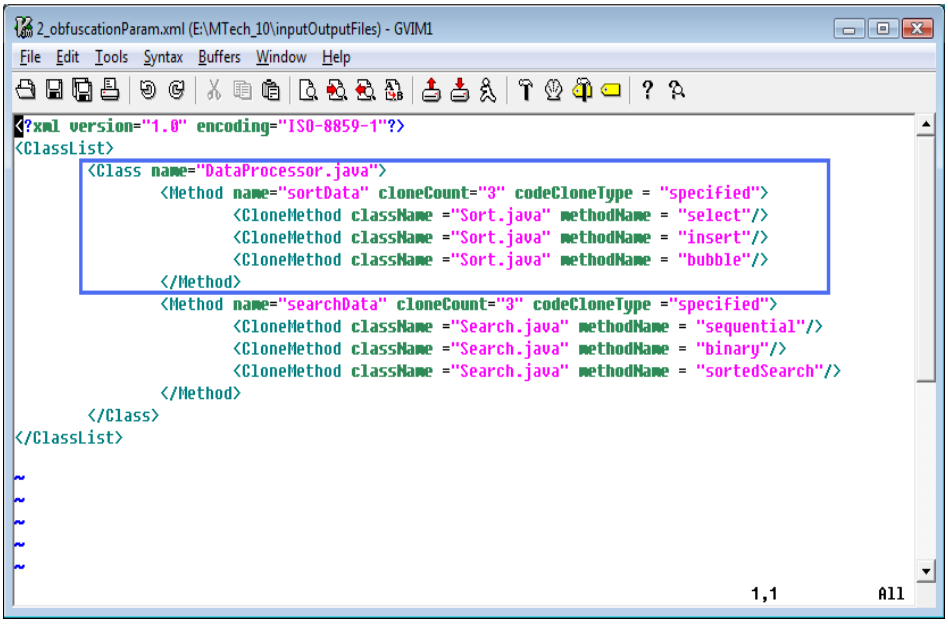


Figure 6.3.1: Sample Obfuscation Specification File

The contents of obfuscation specification file can also be found in section [9.3].

d) Output code file : Obfuscated Data Processor

The contents of obfuscated code file can be found in section [9.4]. The file contains obfuscated code for sort and search methods in which local variables are renamed to random meaningless names. It also contains code for checking dynamic predicate variables ("m_0" and "m_1") used to link the obfuscated code clone fragments.

6.4.Limitations

- Code Obfuscation Techniques: Currently only identifier renaming technique is supported for code obfuscation. It supports obfuscation of only local variables names present in method body.
- Code Clone Variable Naming Convention: The code obfuscator tool assumes that all non trivial code clones operate on member variables having same names (for example, member variable "arrayData") present in code files.

The code obfuscator tool can be extended to support other code obfuscation techniques such as control flow transformation described in paper [1].

7. Conclusion and Future Work

7.1. Conclusion

Code obfuscation is a useful technique to protect software against reverse engineering attacks. Most of the existing obfuscation techniques and schemes satisfy a few criteria defined to measure effectiveness of code obfuscation (such as resistance to static analysis attacks). In this thesis, we present a novel obfuscation scheme to protect secret algorithm present in software. We show that obfuscated code constructed by applying our scheme satisfies most of the criteria (such as increase in software complexity metrics, resistance to static-dynamic analysis attacks and increase in program state space) described in the literature to measure effectiveness of code obfuscation. The scheme achieves code obfuscation by expanding state space of original code by inserting obfuscated non trivial code clones created for logical code fragments. These code clone fragments are linked using dynamic predicate variables. We also present mathematical analysis of the scheme which shows that number of valid execution paths is exponential with respect to number of code fragments and code clones. We show that performance of obfuscated code is comparable to that of original code by performing experimentation on a few programs (such as sequential data processing code). Although the scheme requires high development efforts to construct obfuscated code, it can be useful to obfuscate code which implements secret functionality such as license checking mechanism.

7.2. Future Work

In future, we would like focus on the following work. We would like to extend the scheme for obfuscation of logical code fragments which are connected by complicated control flow statements such as conditional statements or loop constructs. We also would like to enhance the code obfuscator tool by adding state-of-the-art obfuscation techniques such as control flow transformations [1] and control flow flattening [11] for obfuscation of logical code clone fragments. We would also like to apply our scheme to large size code and to carry out experimentation on obfuscated code.

8. References

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformation", Technical report 148, Department of computer science, the University of Auckland, Auckland, New Zealand, 1997.

[2] Christian S. Collberg , Clark Thomborson, "Watermarking, tamper-proofing, and obfuscation: tools for software protection", IEEE Transactions on Software Engineering, v.28 n.8, p.735-746, August 2002.

[3] C. Collberg and J. Nagra. "Surreptitious Software: Obfuscation, Watermarking, and Tamper proofing for Software Protection", Addison Wesley Professional, 2009.

[4] P.C. van Oorschot, "Revisiting Software Protection", Proc. 6th Int'l Conf. Information Security (ISC 03),LNCS 2851, Springer-Verlag, 2003, pp. 1-13;

[5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (im)possibility of obfuscating programs." In J. Kilian, editor, Advances in Cryptology: CRYPTO 2001, 2001. LNCS 2139.

[6] B. Lynn, M. Prabhakaran, and A. Sahai. "Positive results and techniques for obfuscation", In Eurocrypt, 2004. Springer Verlag.

[7] M. Ceccato, M. DiPenta, J. Nagra, P. Falcarin, F.Ricca, M. Torchiano, and P. Tonella. "The effectiveness of source code obfuscation: an experimental assessment", In IEEE International Conference on Program Comprehension (ICPC 2009). IEEE CS Press, 2009

[8] J. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode", JOURNAL OF SYSTEMS AND SOFTWARE, vol. 71, no. 1, pp. 1-10, Apr.2004

[9] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs", In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL98, San Diego, January 1998.

[10] Low, D. (1998). "Java Control Flow Obfuscation", Master's thesis. University of Auckland, Auckland, New Zealand.

[11] Wang, C., Hill, J., Knight, J.C., and Davidson, J.W.: "Protection of software-based survivability mechanisms", In Proceedings of the 2001 conference on

Dependable Systems and Networks. IEEE Computer Society. Pages 193-202. 2001.

[12] Chow, S., Gu, Y., Johnson, H., and Zakharov, V.A.: "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", In the proceedings of 4th International Conference on Information Security, LNCS Volume 2200. Pages 144-155. Springer-Verlag. Malaga, Spain. 2001.

[13] Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y.: "Experience with software watermarking", In Proceedings of 16th IEEE Annual Computer Security Applications Conference (ACSAC'00). IEEE Press. p308. New Orleans,LA, USA. 2000.

[14] A. Balakrishnan and C. Schulze,"Code Obfuscation: Literature Survey", Technical report, Computer Science Department, University of Wisconsin, Madison, USA, 2005.

[15] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, Ravi Ramachandran, "A Qualitative analysis of Java Obfuscation", Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX,USA, November 13-15,2006.

[16] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. "Program obfuscation: a quantitative approach", In QoP '07: Proc. of the 2007 ACM Workshop on Quality of protection, pages 15-20, New York, NY, USA,2007. ACM.

[17] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and De Bosschere Koen. "Hybrid static-dynamic attacks against software protection mechanisms", In Proceedings of the 5th ACM Workshop on Digital Rights Management. ACM, 2005.

[18] Madou M, Anckaert B, De Bus, De Bosschere K, Cappaert J. Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation", Proc. of the International Conference on Software Engineering Research and Practice (SERP06), June. 2006.

[19] Sebastian Schrittwieser and Stefan Katzenbeisser, "Code Obfuscation against Static and Dynamic Reverse Engineering", Vienna University of Technology, Austria, Darmstadt University of Technology, Germany

[20] Falcarin, P.; Di Carlo, S.; Cabutto, A.; Garazzino, N.; Barberis, D.; "Exploiting code mobility for dynamic binary obfuscation", Internet Security (WorldCIS), 2011 World Congress on Publication Year: 2011 , Page(s): 114 - 120

[21] Yuichiro Kanzaki, Akito Monden, "A SOFTWARE PROTECTION METHOD BASED ON TIME-SENSITIVE CODE AND SELF-MODIFICATION MECHANISM", Proceedings of the IASTED International Conference, November 8 - 10, 2010 Marina Del Rey, USA Software Engineering and Applications (SEA 2010)

[22] Christian Collberg, "The Case for Dynamic Digital Asset Protection Techniques", Department of Computer Science, University of Arizona, June 1, 2011

[23] Larry D-Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, and Patrick LeBlanc. "Self-protecting mobile agents obfuscation report Final report", Technical Report 03-015, Network Associates Laboratories, June 2003.

[24] Kelly Heffner and Christian S. Collberg. "The obfuscation executive", In Information Security, 7th International Conference, pages 428-440. Springer-Verlag, 2004. Lecture Notes in Computer Science, #3225.

[25] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In Proceedings of ICSM. IEEE, 1998.

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

[27] Business Software Alliance (May 2011), Eighth Annual BSA and IDC Global Software Piracy Study.

[28] Neil MacDonald; Amrit Williams, et al. Gartner, Inc. "Hype Cycle for Cyberthreats, 2006", September 2006

[29] PMD (Java source code analyzer)
http://pmd.sourceforge.net/

[30] EMMA (code coverage tool)
http://emma.sourceforge.net/

[31] Jensor - (Java Profiler)

http://jensor.sourceforge.net/

[32] DashO - PreEmptive Solutions

http://www.preemptive.com/products/dasho

[33] ProGuard: java obfuscator

http://proguard.sourceforge.net/

[34] JCCD -Java Code Clone Detection API

http://jccd.sourceforge.net/

9. Appendix
9.1. Original Code of Data Processing Application

```
----------------------------
File: DataProcessor_Orig.Java
----------------------------

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.ArrayList;

public class DataProcessor_Orig
{
   int[] arrayData = null;
   int[] arrayDataUpdated = null;
   int value = 0;
   boolean isValueFound = false;

   public static void main(String[] args)
   {
      DataProcessor_Orig dp = new DataProcessor_Orig();
      dp.readInputs(args);
      dp.sortData();
      dp.searchData();
      dp.writeOutputs();
   }

   public void readInputs(String[] args)
   {
      String dataFilePath = null;
      String searchDataValue = null;
      int len = args.length;

      System.out.println("Data processor exuection started");
      if (len != 2)
      {
         System.out.println("invalid inputs");
         return;
      }

      dataFilePath = args[0];
      searchDataValue = args[1];
      value = Integer.parseInt(searchDataValue);

      ArrayList inputData = new ArrayList();
      File inputFile = new File(dataFilePath);
      try
      {
         FileReader fr = new FileReader(inputFile);
         BufferedReader br = new BufferedReader(fr);
         String lineData = null;

         while(true)
         {
            lineData = br.readLine();
            if(lineData == null)
            {
               break;
            }
            else
            {
               String[] values = lineData.split(",");
               for(int i=0;i<values.length;i++)
               {
                  inputData.add(Integer.parseInt(values[i]));
```

```
            }
         }
      }

      int arraySize = inputData.size();
      arrayData = new int[arraySize];

      for(int i=0;i<arraySize;i++)
      {
         arrayData[i] = (Integer)inputData.get(i);
      }
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
}

public void sortData()
{
   int arrLen = arrayData.length;
   arrayDataUpdated = new int [arrLen];

   for(int i=0;i<arrLen;i++)
   {
      arrayDataUpdated[i] = arrayData[i];
   }

   int tempVar = 0;
   for(int i=0;i<arrLen;i++)
   {
      for(int j=1;j<arrLen;j++)
      {
         if(arrayDataUpdated[j-1]>arrayDataUpdated[j])
         {
            tempVar = arrayDataUpdated[j-1];
            arrayDataUpdated[j-1] = arrayDataUpdated[j];
            arrayDataUpdated[j] = tempVar;
         }
      }
   }
}

public void searchData()
{
   for(int i=0;i<arrayData.length;i++)
   {
      if(arrayData[i] == value)
      {
         isValueFound = true;
      }
   }
}

public void writeOutputs()
{
   // write in log file for demo purpose
   try
   {
      String clsName = this.getClass().getName();
      String fileName = clsName + "Out.out";
      File file = new File (fileName);
      BufferedWriter bw = new BufferedWriter(new FileWriter(file, true));

      bw.write("Class Name:"); bw.write(clsName); bw.newLine();
      String str = "-----------------------------------";
      bw.write(str); bw.newLine();
      str = "Input data:";
      bw.write(str);
      str = "";
      for(int i =0;i<arrayData.length;i++)
```

```
            {
               str = str + arrayData[i] +",";
            }
            bw.write(str); bw.write(" ");
            str = "Search value: "+ value;
            bw.write(str); bw.newLine();

            str = "Is value found: "+isValueFound;
            bw.write(str); bw.newLine();
            str = "------------------------------------";
            bw.write(str); bw.newLine();
            bw.close();
            System.out.println("Data processor exuection complete");
            System.out.println("------------------------------------");
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }
```

9.2.Non Trivial Code Clones for Sort and Search Functionality

```
----------------
File: Sort.Java
----------------
public class Sort
{
   int[] arrayData = null;
   int[] arrayDataUpdated = null;

    public void select()
   {
      int arrLen = arrayData.length;
      arrayDataUpdated = new int [arrLen];

      for(int i=0;i<arrLen;i++)
      {
         arrayDataUpdated[i] = arrayData[i];
      }

      for(int i=0;i<arrLen;i++)
      {
         int minElt = arrayDataUpdated[i];
         int minEltIndex = -1;
         for(int j=i+1;j<arrLen;j++)
         {
            if(arrayDataUpdated[j] < minElt)
            {
               minEltIndex = j;
               minElt = arrayDataUpdated[j];
            }
         }

         if(minEltIndex != -1)
         {
            arrayDataUpdated[minEltIndex] = arrayDataUpdated[i];
            arrayDataUpdated[i] = minElt;
         }
      }
   }

   public void insert()   // take an element and insert it in right position
   {
      int arrLen = arrayData.length;
      arrayDataUpdated = new int [arrLen];

      for(int i=0;i<arrLen;i++)
      {
         arrayDataUpdated[i] = arrayData[i];
      }

      for(int i=1;i< arrLen;i++)
      {
         int element = arrayDataUpdated[i];
         boolean isInserted = false;
         for(int j=i;j>0;j--)
         {
            if (element < arrayDataUpdated[j-1])
            {
               arrayDataUpdated[j] = arrayDataUpdated[j-1];   // move elements down
            }
            else
            {
               arrayDataUpdated[j] = element;
               isInserted = true;
               break;
            }
         }
         if(!isInserted)
         {
```

```
            arrayDataUpdated[0] = element;
         }
      }

   }

   public void bubble()   // bubble, insertion, selection
   {
      int arrLen = arrayData.length;
      arrayDataUpdated = new int [arrLen];

      for(int i=0;i<arrLen;i++)
      {
         arrayDataUpdated[i] = arrayData[i];
      }

      int tempVar = 0;
      for(int i=0;i<arrLen;i++)
      {
         for(int j=1;j<arrLen;j++)
         {
            if(arrayDataUpdated[j-1]>arrayDataUpdated[j])
            {
               tempVar = arrayDataUpdated[j-1];
               arrayDataUpdated[j-1] = arrayDataUpdated[j];
               arrayDataUpdated[j] = tempVar;
            }
         }
      }
   }
}
```

```
-----------------
File: Search.Java
-----------------

public class Search
{
   int[] arrayData = null;
   int[] arrayDataUpdated = null;
   int value = 0;
   boolean isValueFound = false;

   public void sortedSearch()
   {
      for(int i=0;i<arrayDataUpdated.length;i++)
      {
         if(arrayDataUpdated[i] == value)
         {
            isValueFound = true;
            break;
         }
         else if (arrayDataUpdated[i] > value)
         {
            break;      // element does not exist
         }
      }
   }

   public void binary()
   {
      int arrLen = arrayDataUpdated.length;
      int firstIndex = 0;
      int secondIndex = arrLen;

      while(true)
      {
         if (firstIndex > secondIndex)
         {
            break;
         }
         else if (firstIndex == secondIndex)
         {
            if (value == arrayDataUpdated[firstIndex])
            {
               isValueFound = true;
               break;
            }
         }
         else
         {
            int mid = (firstIndex + secondIndex)/2;
            if (value == arrayDataUpdated[mid])
            {
               isValueFound = true;
               break;
            }
            else if (value < arrayDataUpdated[mid])
            {
               secondIndex = mid;
            }
            else
            {
               firstIndex = mid;
            }
         }
      }
   }

   public void sequential()
   {
      for(int i=0;i<arrayData.length;i++)
      {
```

```
            if(arrayData[i] == value)
            {
               isValueFound = true;
               break;
            }
         }
      }

}
```

9.3.Obfuscation Specification for Data Processing Application

```
-------------------------
File: obfuscationParam.xml
-------------------------
<?xml version="1.0" encoding="ISO-8859-1"?>
<ClassList>
   <Class name="DataProcessor.java">
      <Method name="sortData" cloneCount="3" codeCloneType = "specified">
         <CloneMethod className ="Sort.java" methodName = "select"/>
         <CloneMethod className ="Sort.java" methodName = "insert"/>
         <CloneMethod className ="Sort.java" methodName = "bubble"/>
      </Method>
      <Method name="searchData" cloneCount="3" codeCloneType ="specified">
         <CloneMethod className ="Search.java" methodName = "sequential"/>
         <CloneMethod className ="Search.java" methodName = "binary"/>
         <CloneMethod className ="Search.java" methodName = "sortedSearch"/>
      </Method>
   </Class>
</ClassList>
```

9.4.Obfuscated Code of Data Processing Application

```
-----------------------------------
File: DataProcessor_Obfuscated.Java
-----------------------------------
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.ArrayList;
import ndg.ndu.ds.NDG;
import ndg.ndu.ds.STM;

public class DataProcessor_Obfuscated {

    int[] arrayData = null;

    int[] arrayDataUpdated = null;

    int value = 0;

    boolean isValueFound = false;

    NDG m_0 = null, m_1 = null;

    STM stm = new STM("dataprocessor", new int[] { 3, 3 });

    public static void main(String[] args) {
        DataProcessor_Obfuscated dp = new DataProcessor_Obfuscated();
        dp.readInputs(args);
        dp.sortData();
        dp.searchData();
        dp.writeOutputs();
    }

    public void readInputs(String[] args) {
        String dataFilePath = null;
        String searchDataValue = null;
        int len = args.length;
        System.out.println("Data processor exuection started");
        if (len != 2) {
            System.out.println("invalid inputs");
            return;
        }
        dataFilePath = args[0];
        searchDataValue = args[1];
        value = Integer.parseInt(searchDataValue);
        ArrayList inputData = new ArrayList();
        File inputFile = new File(dataFilePath);
        try {
            FileReader fr = new FileReader(inputFile);
            BufferedReader br = new BufferedReader(fr);
            String lineData = null;
            while (true) {
                lineData = br.readLine();
                if (lineData == null) {
                    break;
                } else {
                    String[] values = lineData.split(",");
                    for (int i = 0; i < values.length; i++) {
                        inputData.add(Integer.parseInt(values[i]));
                    }
                }
            }
            int arraySize = inputData.size();
            arrayData = new int[arraySize];
            for (int i = 0; i < arraySize; i++) {
                arrayData[i] = (Integer) inputData.get(i);
            }
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void sortData() {
    if (m_0 == null) {
    m_0 = new NDG(3, "m_0", stm);
} else {
    m_0.mvn();
}
    if (m_0.sn(0)) {
    int a2110711758 = arrayData.length;
    arrayDataUpdated = new int[a2110711758];
    for (int c677447630 = 0; c677447630 < a2110711758; c677447630++) {
        arrayDataUpdated[c677447630] = arrayData[c677447630];
    }
    for (int n1774596447 = 0; n1774596447 < a2110711758; n1774596447++) {
        int a171330926 = arrayDataUpdated[n1774596447];
        int a118702090 = -1;
        for (int a979667234 = n1774596447 + 1; a979667234 < a2110711758; a979667234++) {
            if (arrayDataUpdated[a979667234] < a171330926) {
                a118702090 = a979667234;
                a171330926 = arrayDataUpdated[a979667234];
            }
        }
        if (a118702090 != -1) {
            arrayDataUpdated[a118702090] = arrayDataUpdated[n1774596447];
            arrayDataUpdated[n1774596447] = a171330926;
        }
    }
}
    if (m_0.sn(2)) {
    int o78573288 = arrayData.length;
    arrayDataUpdated = new int[o78573288];
    for (int u723080508 = 0; u723080508 < o78573288; u723080508++) {
        arrayDataUpdated[u723080508] = arrayData[u723080508];
    }
    for (int s1517383704 = 1; s1517383704 < o78573288; s1517383704++) {
        int l1143453413 = arrayDataUpdated[s1517383704];
        boolean f1702496723 = false;
        for (int j50892461 = s1517383704; j50892461 > 0; j50892461--) {
            if (l1143453413 < arrayDataUpdated[j50892461 - 1]) {
                arrayDataUpdated[j50892461] = arrayDataUpdated[j50892461 - 1];
            } else {
                arrayDataUpdated[j50892461] = l1143453413;
                f1702496723 = true;
                break;
            }
        }
        if (!f1702496723) {
            arrayDataUpdated[0] = l1143453413;
        }
    }
}
    if (m_0.sn(1)) {
    int m1910608582 = arrayData.length;
    arrayDataUpdated = new int[m1910608582];
    for (int g2074187290 = 0; g2074187290 < m1910608582; g2074187290++) {
        arrayDataUpdated[g2074187290] = arrayData[g2074187290];
    }
    int o1922514582 = 0;
    for (int h261779395 = 0; h261779395 < m1910608582; h261779395++) {
        for (int t2073539929 = 1; t2073539929 < m1910608582; t2073539929++) {
            if (arrayDataUpdated[t2073539929 - 1] > arrayDataUpdated[t2073539929]) {
                o1922514582 = arrayDataUpdated[t2073539929 - 1];
                arrayDataUpdated[t2073539929 - 1] = arrayDataUpdated[t2073539929];
                arrayDataUpdated[t2073539929] = o1922514582;
            }
        }
    }
```

```
}
    }

    public void searchData() {
    if (m_1 == null) {
    m_1 = new NDG(3, "m_1", stm);
} else {
    m_1.mvn();
}
    if (m_1.sn(0)) {
    for (int e1834258326 = 0; e1834258326 < arrayData.length; e1834258326++) {
        if (arrayData[e1834258326] == value) {
            isValueFound = true;
            break;
        }
    }
}
    if (m_1.sn(2)) {
    int f1813592123 = arrayDataUpdated.length;
    int p1542792409 = 0;
    int i1669784488 = f1813592123;
    while (true) {
        if (p1542792409 > i1669784488) {
            break;
        } else if (p1542792409 == i1669784488) {
            if (value == arrayDataUpdated[p1542792409]) {
                isValueFound = true;
                break;
            }
        } else {
            int x325657563 = (p1542792409 + i1669784488) / 2;
            if (value == arrayDataUpdated[x325657563]) {
                isValueFound = true;
                break;
            } else if (value < arrayDataUpdated[x325657563]) {
                i1669784488 = x325657563;
            } else {
                p1542792409 = x325657563;
            }
        }
    }
}
    if (m_1.sn(1)) {
    for (int u341262682 = 0; u341262682 < arrayDataUpdated.length; u341262682++) {
        if (arrayDataUpdated[u341262682] == value) {
            isValueFound = true;
            break;
        } else if (arrayDataUpdated[u341262682] > value) {
            break;
        }
    }
}
    }

    public void writeOutputs() {
        try {
            String clsName = this.getClass().getName();
            String fileName = clsName + "Out.out";
            File file = new File(fileName);
            BufferedWriter bw = new BufferedWriter(new FileWriter(file, true));
            bw.write("Class Name:");
            bw.write(clsName);
            bw.newLine();
            String str = "-----------------------------------";
            bw.write(str);
            bw.newLine();
            str = "Input data:";
            bw.write(str);
            str = "";
            for (int i = 0; i < arrayData.length; i++) {
                str = str + arrayData[i] + ",";
```

```
            }
            bw.write(str);
            bw.write(" ");
            str = "Search value: " + value;
            bw.write(str);
            bw.newLine();
            str = "Is value found: " + isValueFound;
            bw.write(str);
            bw.newLine();
            str = "------------------------------------";
            bw.write(str);
            bw.newLine();
            bw.close();
            System.out.println("Data processor exuection complete");
            System.out.println("------------------------------------");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```