Code Obfuscation

Submitted in partial fulfillment of the requirements
for the award of the degree of

Master of Computer Application (MCA)

Guru Gobind Singh Indraprastha University, Delhi

Guide(s):
Mrs. Rajashree

Submitted by:
Richa Sachdeva
Roll No.:
02811804409

Centre for Development of Advanced Computing, Noida

2011-2012

Certificate

I, Richa Sachdeva, Roll No.02811804409, certify that the Project Report/Dissertation (MCA-302) entitled "Code Obfuscation" is done by me and it is an authentic work carried out by me at C-DAC, Noida (Name of the organisation or of the Institute). The matter embodied in this project work has not been submitted earlier for the award of any degree or diploma to the best of my knowledge and belief.

Signature of the Student

Date:

Certified that the Project Report/Dissertation (MCA-302) entitled "Code Obfuscation" done by Ms. Richa Sachdeva, Roll No. 02811804409, is completed under my guidance.

Signature of the Guide

Date:

Mrs. Rajshree

Head SOIT

Address:

CDAC, Noida

Acknowledgement

I take this opportunity with much pleasure to thank all the faculty members who have helped me through the course and produce this synopsis.

I would like to thank Ms. Kriti Saroha (HOD, MCA) for her constant guidance and support, which has been of immense help in carrying out the project work.

I sincerely thank my teacher, Mrs. Rajshree for her guidance, help and motivation, without which the project could not be completed. Apart from the project domain, I have learnt a lot from her, which I am sure, will be useful in different stages of my life.

College's Library had all the books and resources required for completion of the project work.

I would also like to thank my friends who helped me in successful completion of the project.

Richa Sachdeva

Synopsis/ Executive Summary

In this project report, we discuss the use of code obfuscation as means of protecting the intellectual property of software. Obfuscation is a technique to transform the program, which aims to make the program harder to understand (which can mean it becomes unintelligible to automated program comprehension tools or that the result of program analyses become less useful to a human adversary), without affecting its behavior (output) that is we preserve its behavior while transformation.

Code Obfuscation is primarily implemented to provide security against the reverse engineering process, and its plausible side effects on the original source code.

In this project, we have implemented a Code Obfuscator, which takes in input a basic C program, and gives output as an obfuscated C code.

This project report discusses the need for obfuscators, different types of obfuscators. Then we have outlined the methodology we have followed for making this obfuscator, comprising of designing of algorithm, to its implementation.

CONTENTS

LIST OF FIGURES

METHODOLOGY FOR PROJECT WORK/DISSERTATION

Chapter-1: Introduction/Problem Definition

1.) General Description of the System under Study

The system is supposed to obfuscate the program codes written in C programming language. The System is designed in Java Programming language. Through obfuscation we make the reverse engineering of the code difficult, hence making it secure.

There are three general methods for protecting source code, namely -

a. Code authentication and verification - meant to protect against unauthorized tampering and unauthorized access to the code. This method is most efficient when authentication data is sent via the network. User has the complete code, which in theory can also be in mangled form.

b. Server side invocation - provides protection by restricting the distribution of the code. This method allows avoiding sending of the final code to the user. A fundamental requirement for this method is high bandwidth.

c. Code obfuscation - You may need to distribute the code to several entities and want to protect against reverse engineering or copying. It involves transformation of executable code to make its hard through tools like decompilers.

2.) User requirements

The user has to give as an input a source code written in C programming language. The code is supposed to be indented properly. The programs should be basic, and not relating to embedded systems or core assembly language codes.

Code Obfuscation is an emerging field and has still not been able to achieve the levels of obfuscation that can be performed and are desired. Hence, was found a need to make a new Code Obfuscator tool that could have enhanced features and functionalities to

obfuscate the code and make it more and more difficult to reverse engineer based on the levels of obfuscation desired by the source programmer.

3.) **Need of the New System**:

Since Code Obfuscation is an emerging technique, not many obfuscators have been created till date. The ones available have not been able to perform the logic obfuscation much to an extent desired, which is been desired to accomplish in this existing system.

It brings to us a lot of learning regarding how a compiler and a lexical analyzer work and also to come up with a better code obfuscation technique.

4.) **Objectives of the project**

The project aims at performing obfuscation of the code to all levels and types of obfuscations devised. Making it difficult to reverse engineer and hard to understand, the codes are analysed against the lexical analyzer designed in the system and the techniques some existing and one devised as a study during this project is supposed to be implemented.

5.) **Methodology**

We have used spiral model for development, as for the different phases of project we gathered the requirements, then designed and developed them and implemented and tested, iteratively. The software is programmed in Java programming language, hence; object oriented analysis has been carried out, as in, for a common property, abstract class has been created, while for a specific role to be performed, we have developed concrete classes.

Chapter-2: Systems Requirement Analysis

1.) The system is required to perform obfuscation of the codes written in C programming language.

The system requires only a C source code as an input. The code is expected to be well indented, especially for this system.

The activities that are required to be carried out and to be described in the project are:

a.) Scanning the whole source code, known as the lexical analysis of the source code

b.) Construction of the parse tree, corresponding to the given source code

c.) Perform obfuscation on the parse tree

d.) Perform data obfuscation

e.) Copy the final code into the output file

2.) Input: a source code written in C programming language, well indented according to the standard indentation convention.

Output: a text file containing the obfuscated source code written in C programming language.

3.) The source codes should be of basic level, they should not be the codes written for embedded systems or very low level.

Constraints/Limitations of our Project

- Obfuscators cannot obfuscate a program completely.
- The proper indented source code is required for obfuscator to work.
- External variables, unions and structures are not obfuscated.
- Haven't obfuscated all of pre-defined functions, will be obfuscating some of them, like functions in string library.
- It is assumed that the variable names have spaces on either sides so that they can be well identified and recognized
- It is assumed that the global variables defined are also declared there itself using the assignment operator, helps parsing in a more flexible way since it becomes easier to decipher between the variable and the function definition.

Chapter-3: Systems Design

Following is the algorithm devised.

Basic Outline of the algorithm:

1.) Scan the whole program.

2.) Parse tree construction

3.) Apply obfuscations in the parse tree.

4.) Copy the parse tree to the final target file, in the proper way.

5.) Apply Layout obfuscation.

Detailed description of the algorithm:

1.) Scan the whole program

The first step is to read the whole program that is coded in C programming language. This involves breaking the program into tokens. The program in this phase is broken into lines as is written in the source code. Each line would then be considered and the parsing over them would be conducted.

In a way, Lexical Analysis is the first phase, where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning.

The lexical analyzer takes a source program as input, and produces a stream of tokens as output. Such specific instances are called lexemes. A lexeme is the actual character sequence forming a token; the token is the general class that a lexeme belongs to. Some tokens have exactly, for others there are many lexemes.

2.) **Parse tree construction**

The same source code can be constructed and laid out in many different ways. The code being the same, but because of different styles of writing, we need to construct a standard form, with the help of which the code can be understood. The understanding of the code is necessary so as to understand where are the variable defined, what is its scope, what is its visibility, which function is declared where and where is its definition placed, for example, the functions defined before the main() function don't need to be declared in the main's function definition, while the ones placed after main(), will have to be declared in the main() function. Further if a predefined function is used, it is "known" through the header files included, and if a new predefined is to introduced in the program, for which the user has not included the header file, we will have to again add. Hence, for these reasons we need to construct a parse tree.

In lexical analysis of the program, we have two types of trees constructed, the syntax tree and the parse tree. The syntax tree is just the representation of the code according to the syntax rules, while a parse tree is different from the syntax tree, in terms that the nodes of the parse tree have a meaning, and is self descriptive. Looking at the nodes, what it signifies can be understood.

By understanding the code, it is meant that the syntax, semantics and pragmatics are understood.

a.) **Syntax** refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language. It defined the formal relations between the constituents of a language, thereby providing a structural description of the various expressions that make up legal strings in the language. It deals solely with the form and structure of symbols in a language without any consideration given to their meaning.

b.) **Semantics** reveals the meaning of syntactically valid strings in a language. For natural languages, this means correlating sentences and phrases with the objects, thought and feelings based on our experiences.

c.) **Pragmatics**, alludes to those aspects of language that involve the users of the language, namely psychological and sociological phenomena such as utility, scope of application, and effects on the users. For programming languages, pragmatics includes issues such as ease of implementation, efficiency in application, and programming methodology.

The codes written in C programming language have the following major sections:

a.) The Pre-processor section

b.) The global declaration section

c.) The function definition section

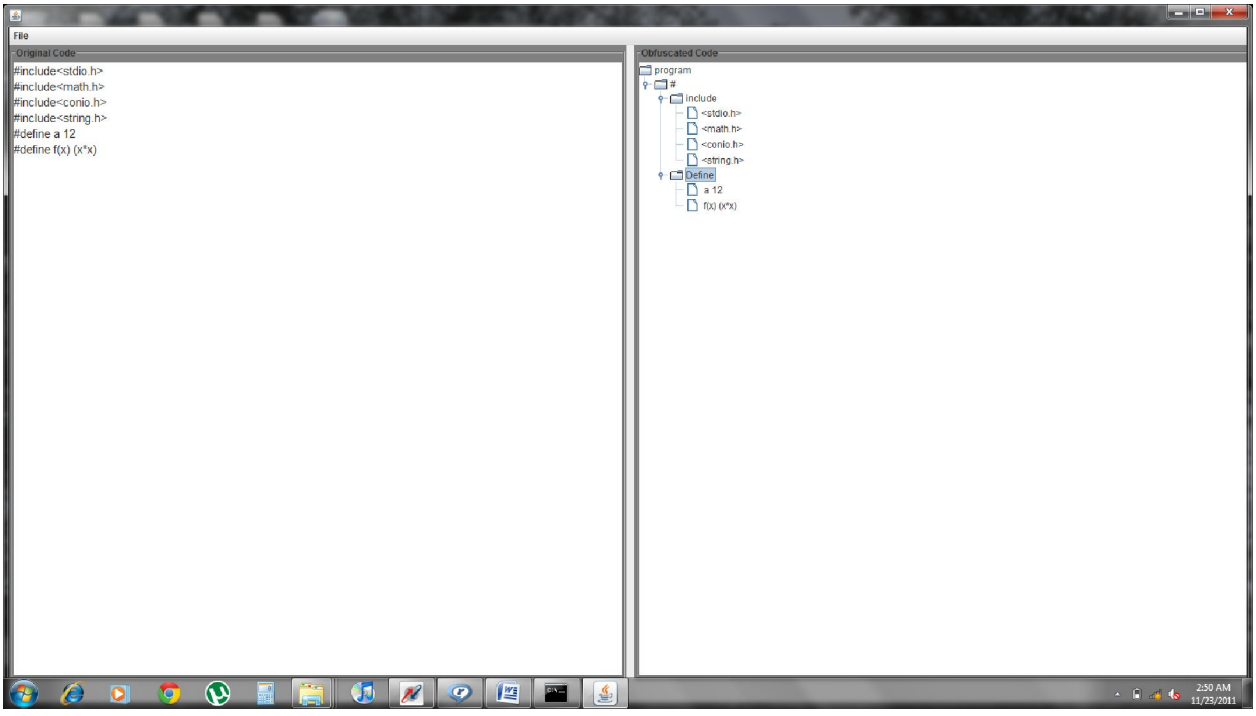
d.) The main() function definition

e.) The function defined after the main() function

The Pre-processor section:

9 pre-processor directives are taken into consideration, namely; include, define, error, line, pram, if, ifdef, endif. The parsing is started and as expected, we encounter the pre-processor directives. The "#" in the beginning of the lines clearly signifies the presence of a pre-processor directive.

As the pre-processor directive statement is identified, we further look for the above-mentioned pre-processor directives, and make the corresponding nodes.



As in the above-mentioned list of pre-processor directives taken into consideration, the corresponding nodes are created and are shown in the diagram.

The global declaration section

Moving ahead, as we find that the beginning of the lines does not have "#" next we can expect a global variable definition or a user defined data type declaration or a function definition.

Hence if we encounter something of the form:

<return type> <variable name>;

Or

<return type> <variable name>=<value>;

Or

Struct <structure name>

{

Body

}

Or

Union <union name>

{

Body

}

Or

<return type> <function name> (parameters)

{

Body of the function

}

Or

<return type> main()

{

Body of main

}

Considering all the above condition, the corresponding patterns of occurrences are taken into account and used to identify and understand what exactly the next node in the tree is supposed to be and accordingly we add the nodes.

The global variables will have their own self describing node, where in we can make changes in their names while performing obfuscation.

The format of the node is like:

<return type> <variable name> [=<value>];

The structures and unions and other user defined data types

Only structures and unions are considered in the software, rest are not touched upon. No obfuscations would be performed on the structures and the unions.

The node for struct or for the union is identified with the help of the corresponding keywords. This node will only be used for the copying into the final code.

The user defined function definitions

It's with the help of the function call operator, or the assignment operator or the statement termination operator that we are able to decipher and "understand" amongst the options given above.

The different sub-tree for each function is constructed and the obfuscations are performed on them individually. The construction of sub-trees for each function facilitates better understanding, operation and copying.

Here each function node will have child nodes each representing a statement, characterized by the termination of the semicolon. The internal occurring braces are well taken into account, and the parentheses matching algorithm is maintained using a count, which also helps in identifying the termination of the function.

The main() function definition node

Similar to the user defined function node, we follow the same working while parsing the main() function.

Quite similarly is the body of the function broken into statement characterized by the semicolon and the parentheses are well matched.

The functions defined under the main() block

Again the same treatment is undertaken for such functions. These function need to be declared before in the main() function. And hence any dynamically generated functions we add to the program, we add them before the main() function, taking the advantage of the fact that, they don't need to be declared separately in the main().

Applying obfuscations in the parse tree

1.) First we consider the global variables, change their names with their hexadecimal equivalent ones. Hence in the whole program, i.e. the parse tree wherever the variable is in use, we change it with its hexadecimal equivalent. Even if the variable is declared again inside a block, we replace that one also including its definition and use, which finally does not make any difference, just adds difficulty in understanding and remembering the variable, as they are not obvious by nature.

2.) Then we take into consideration those functions (predefined), for which we have given our own definitions (hence pre defined in a way defined by us). They are replaced by function calls, these functions are dynamically generated and put in the code, which either make a call to the predefined function called or call our version of the function.

For example, we have a call to strlen("string");
It can be replaced by cdacfun("string");
While we dynamically generate a function definition for cdacfun(), add it as a function node before the main() node.
While the cdacfun() function's definition will be a little different from what is expected of a function finding the length of string, it will have some unnecessary variable, some illogical computations, some always true resulting if loops, for example,
If(1<2) will always make the true block executed while the else would never be executed.

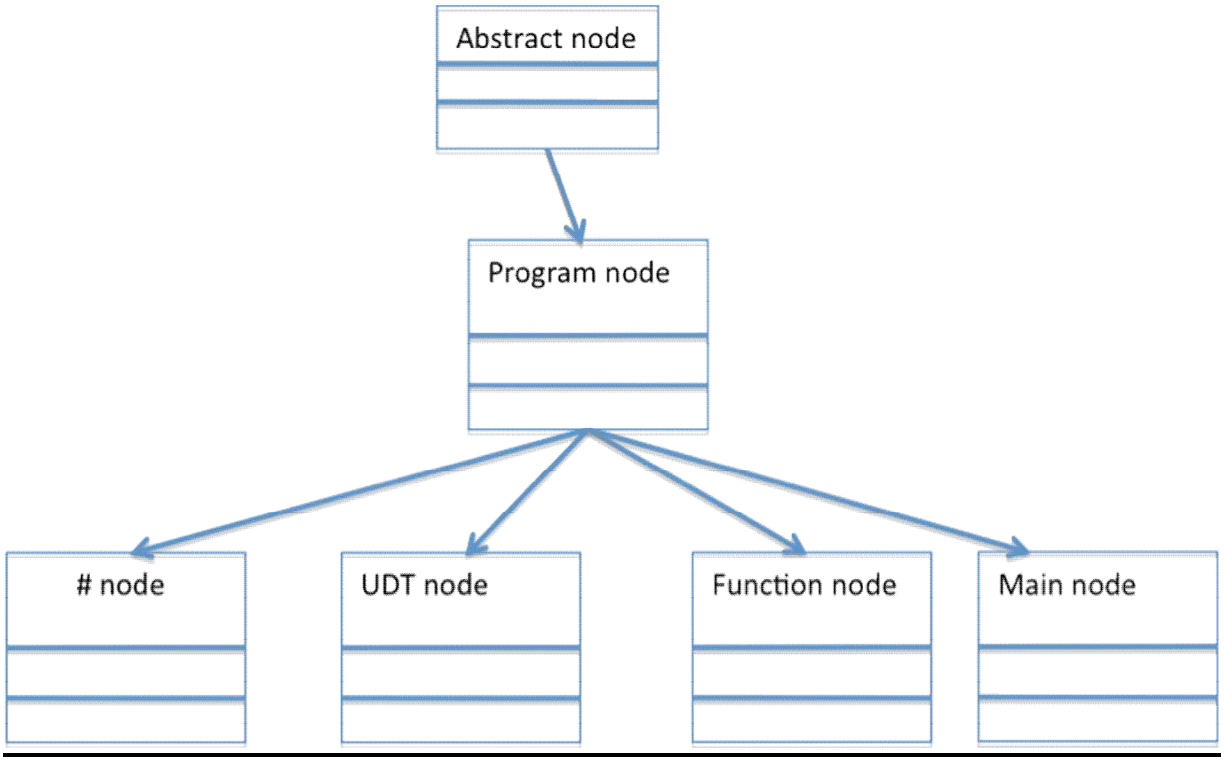
3.) Then we will make changes in the layout of the code. The indentation will have a pattern, or probably end up having no particular pattern. The purpose is just make it a little more unreadable

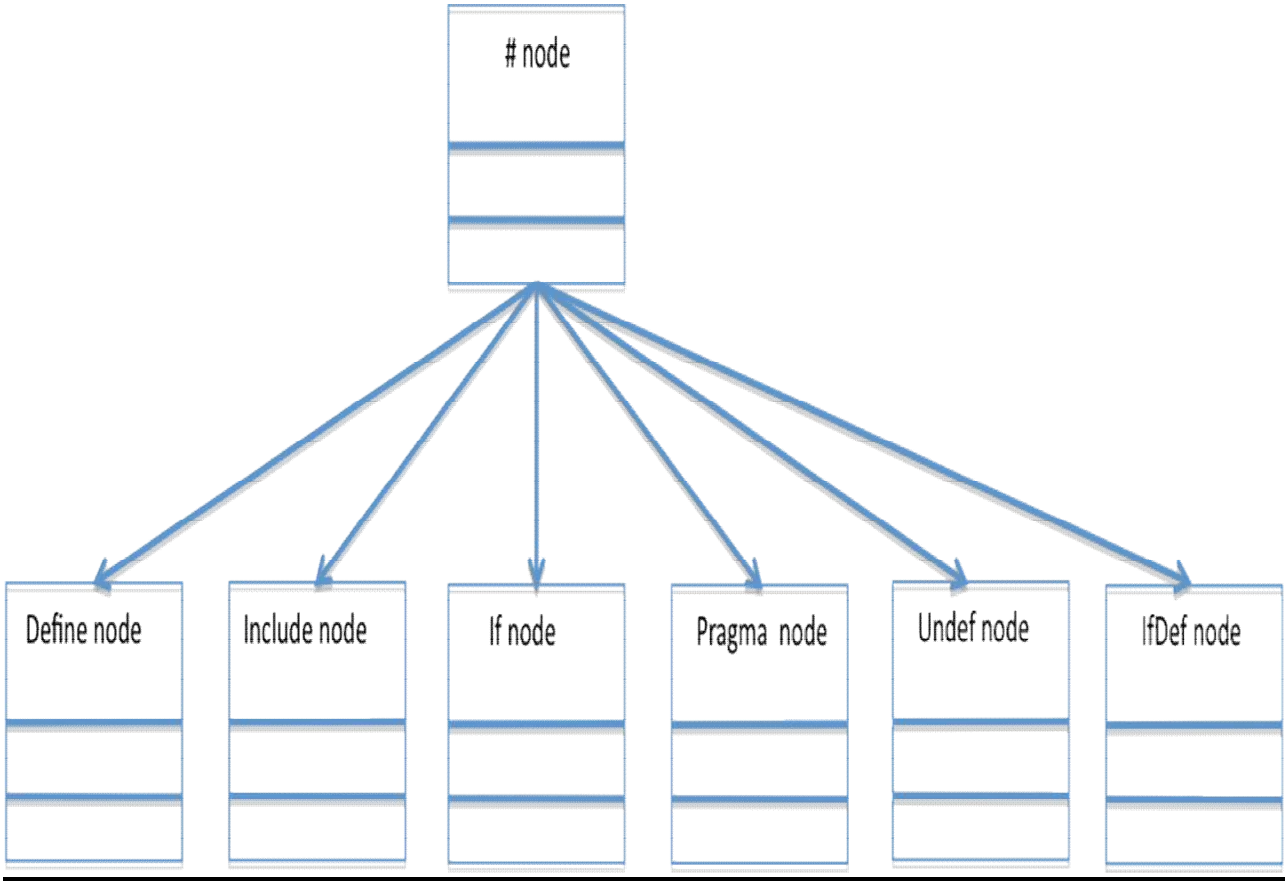
Any changes we make here are first operated upon the parse tree.

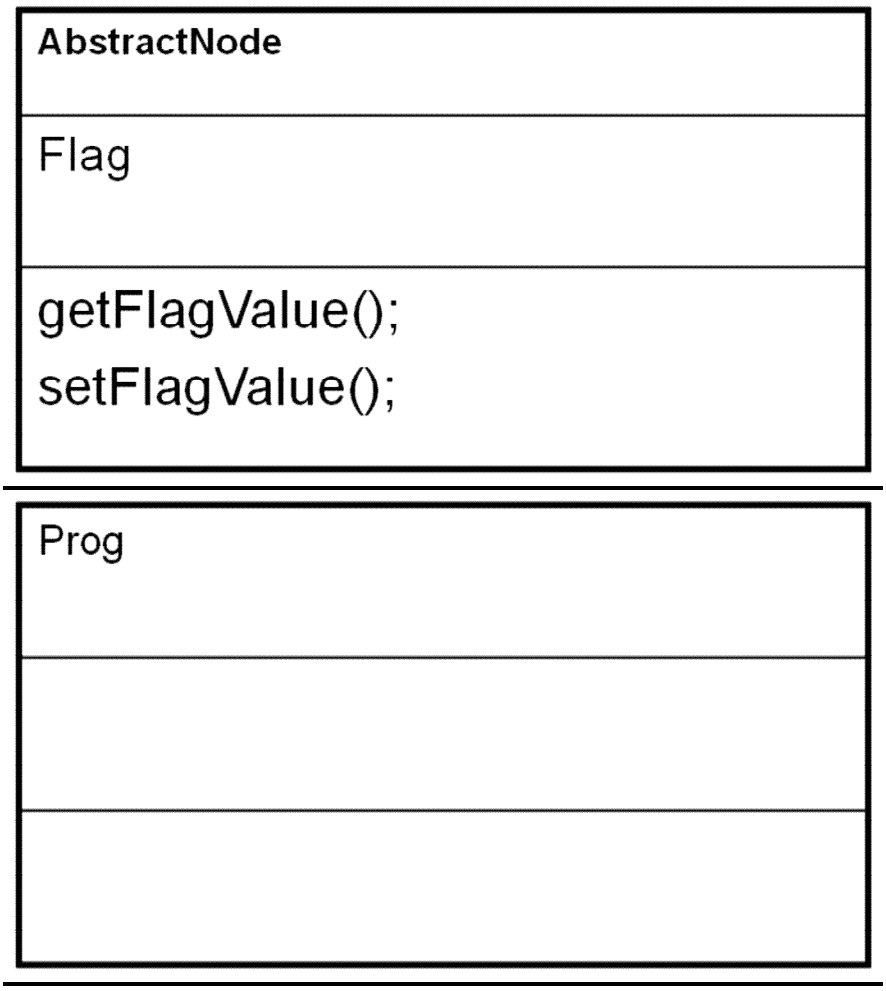
Copying the final code

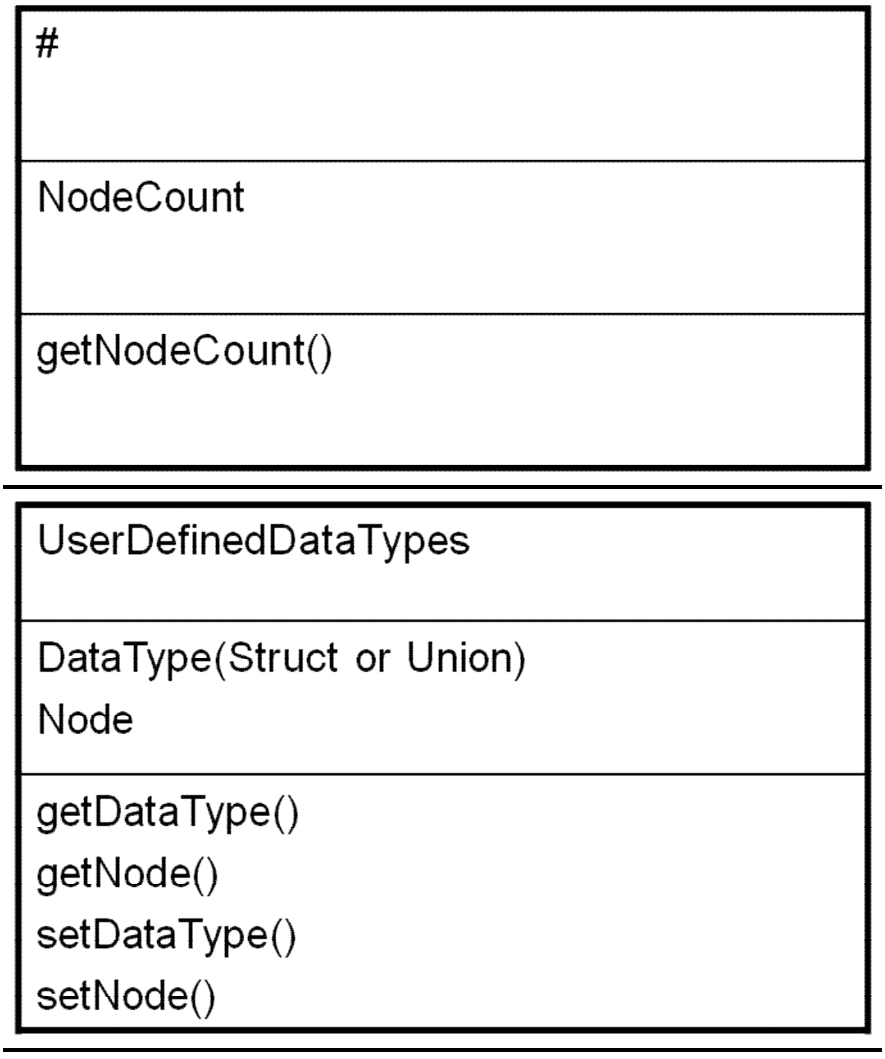
Since the nodes of the constructed parse tree are self-describing, we use this description to copy the parse tree back into the format of a source code.

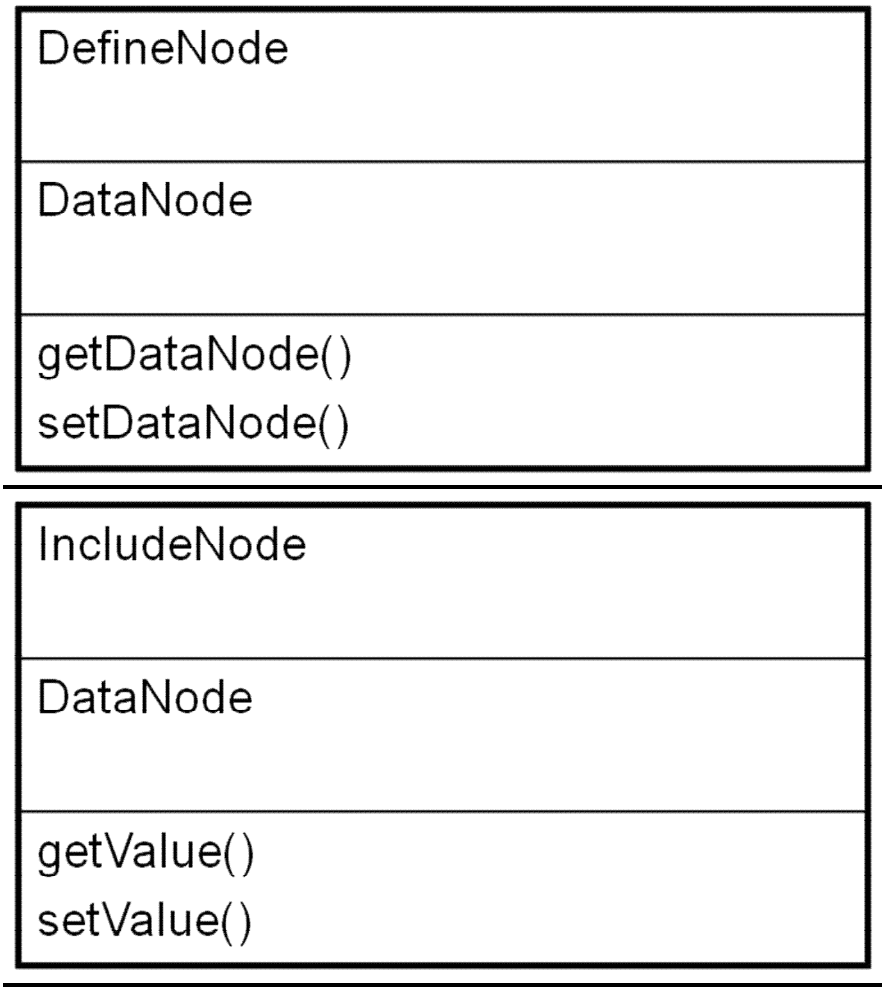
The Class Diagrams

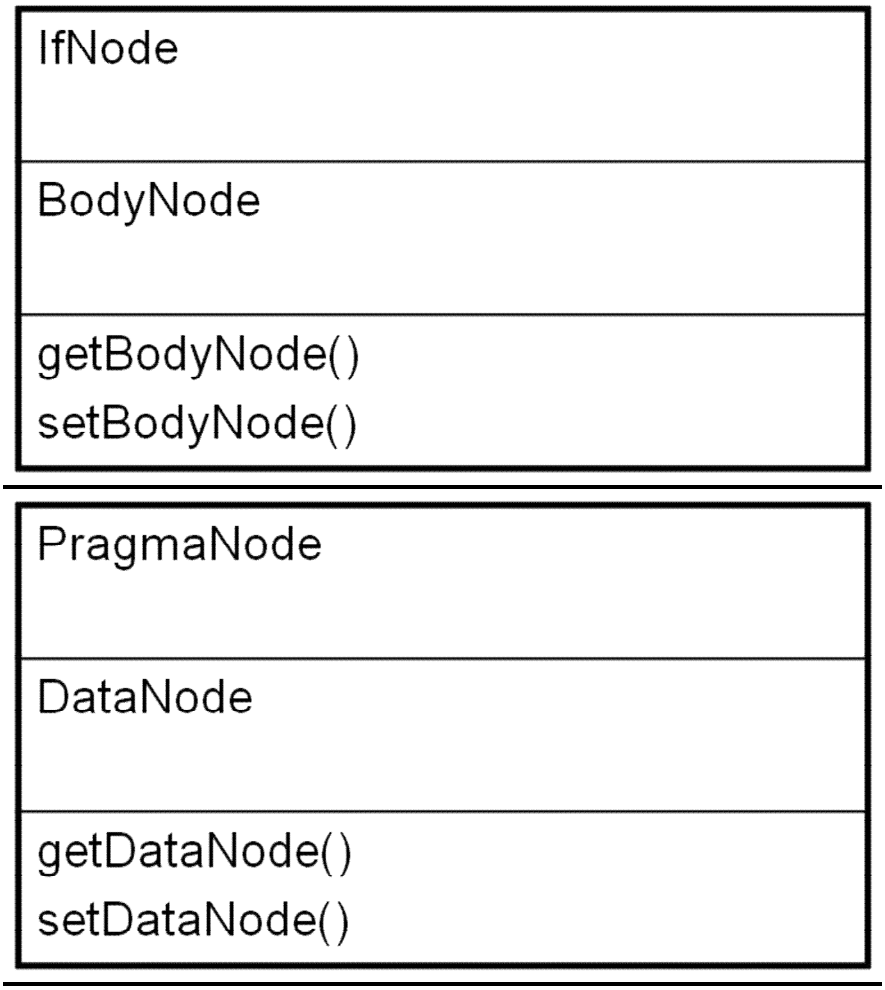


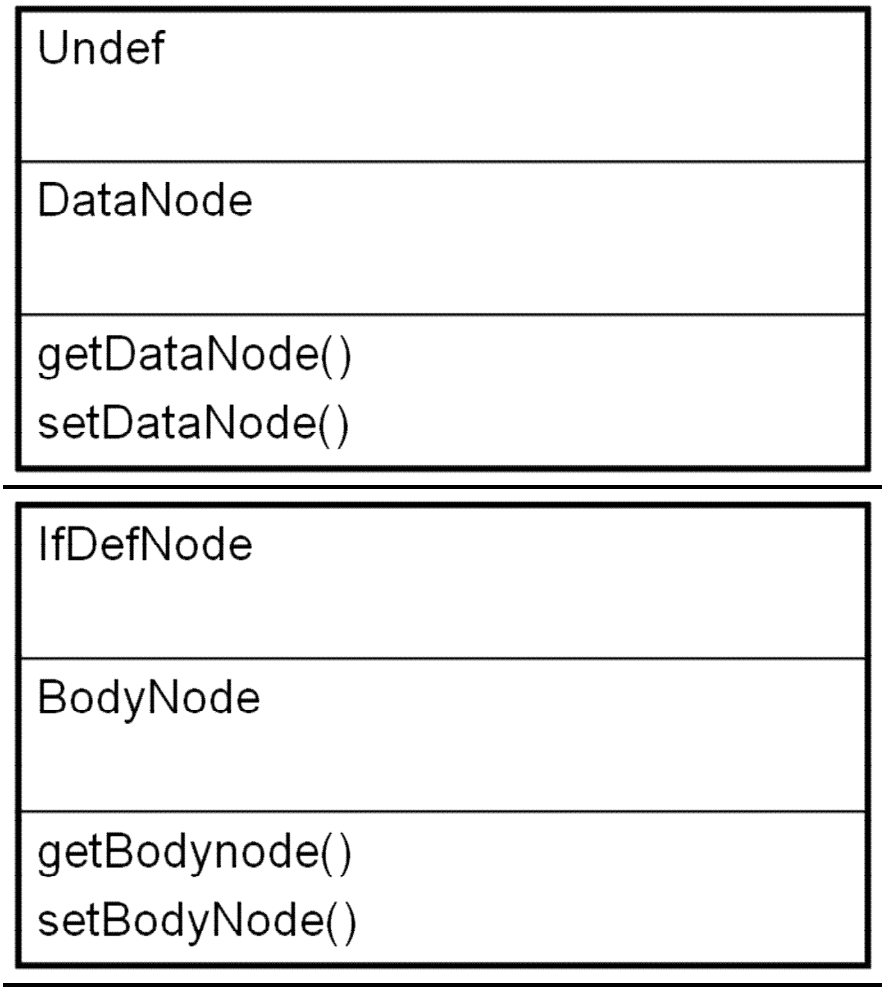

node
Define node
Include node
If node
Pragma node
Undef node
IfDef node

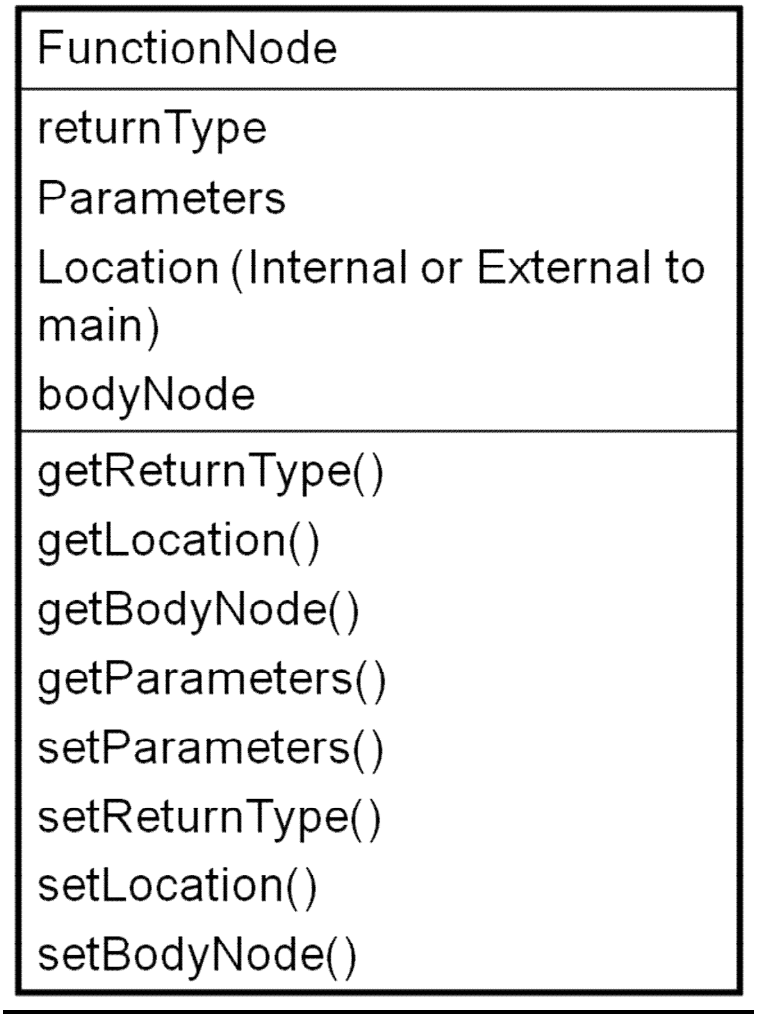

AbstractNode
Flag
getFlagValue();
setFlagValue();
Prog

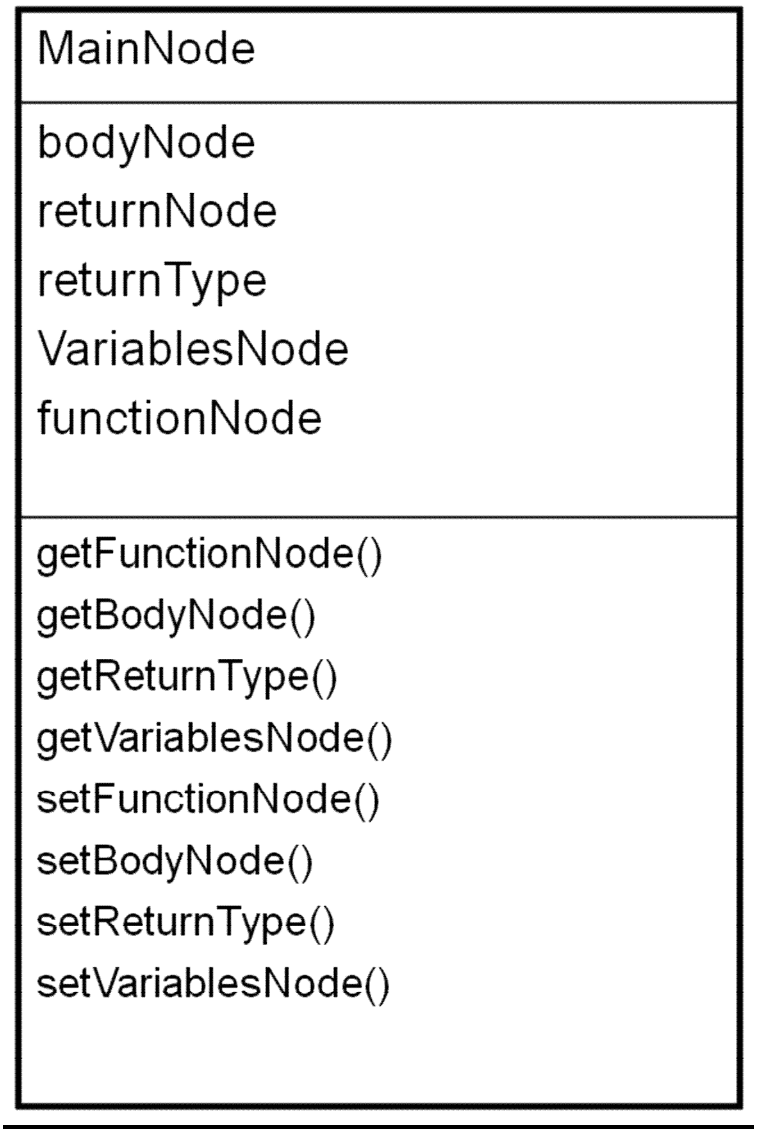

#
NodeCount
getNodeCount()
UserDefinedDataTypes
DataType(Struct or Union)
Node
getDataType()
getNode()
setDataType()
setNode()


DefineNode
DataNode
getDataNode()
setDataNode()
IncludeNode
DataNode
getValue()
setValue()


IfNode
BodyNode
getBodyNode()
setBodyNode()
PragmaNode
DataNode
getDataNode()
setDataNode()


Undef
DataNode
getDataNode()
setDataNode()
IfDefNode
BodyNode
getBodynode()
setBodyNode()


FunctionNode
returnType
Parameters
Location (Internal or External to main)
bodyNode
getReturnType()
getLocation()
getBodyNode()
getParameters()
setParameters()
setReturnType()
setLocation()
setBodyNode()


MainNode
bodyNode
returnNode
returnType
VariablesNode
functionNode
getFunctionNode()
getBodyNode()
getReturnType()
getVariablesNode()
setFunctionNode()
setBodyNode()
setReturnType()
setVariablesNode()

Chapter 5: System Implementation

The software run on java run time environment. The software accepts the source code as a text file, displays its parse tree and generates a text file containing the C source code file.

Summary and conclusion

Code Obfuscation being an upcoming field attracts, and provides us with a new way of exercising security and making it more difficult to reverse engineer the source code and also difficult to maintain and modify to extents desired.

The project aimed at designing software that obfuscates the source codes written in C programming language, while the software is written in Java programming language.

The process that involves the construction of the parse forms the most critical step of the project, is also displayed in the software. This basically provides us with a standard form of the given source code, and techniques can be well implemented on the parse tree, traversing through it, which again involves coming up with newer standards.

The system constructed in this project performs the lexical analysis of the input source code and constructs its parse tree, with the constraints mentioned. The obfuscation is achieved at the level of variable replacement in the main() function, which is known as Layout Obfuscation. The proposed project could not be completed due to extensive study carried out in the designing of the lexical analysis module and the parse tree construction module, and also due to time constraint. It is believed that this system if extended can prove to be a useful Code Obfuscator.

References

1.) http://en.wikipedia.org/wiki/Obfuscatedcode

2.) http://www.cs.auckland.ac.nz/~cthombor/Student/hlai/hongying.pdf

3.) http://thc.org/root/phun/unmaintain.html

4.) http://www.cs.ox.ac.uk/files/2936/RR-10-02.pdf

5.) http://www.cs.arizona.edu/~colllberg/Research/Publications/CollbergThomborsonLow97a/

6.) http://palisade.plynt.com/issues/2004Dec/code-obfuscation

7.) www.obfuscators.org

8.) Memphis Compilers Manual

9.) Alfred V Aho – Principles of Compiler Design

10.) C Reference Manual

11.) The C Programming Language by Dennis Ritchie