

# Design and Implementation of a Code Obfuscator for the Mini-C Language

## Project Report Phase.1

Prof. Alaeiyan

*Sarina Babadi - Mobina Davoodi - Fatemeh Amirabadi*

دانشگاه صنعتی خواجه نصیرالدین طوسی

## Project Overview

Code obfuscation is a process that transforms readable source code into a version that preserves functionality but becomes more difficult to interpret or reverse-engineer. This is widely used in software protection to prevent intellectual property theft and increase software security.

This project presents the design and implementation of a code obfuscator targeting Mini-C, a simplified C-like language. The goal is to convert valid Mini-C source code into an equivalent but obfuscated version, using static code transformation techniques.

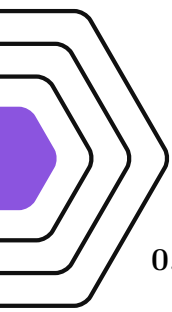
## Project Goals

- Build a working code obfuscator for Mini-C using Python and ANTLR.
- Apply at least three static obfuscation techniques.
- Maintain functional equivalence between input and output.
- Demonstrate the transformation on real Mini-C code snippets.

## Mini-C Language Scope

The Mini-C subset includes:

Data types: int, char, bool Control flow: if, else, while, for, return Function definitions and calls Input/output via printf and scanf No pointers or structs (excluded by design)



# Obfuscation Techniques

## 0.0.1 Identifier Renaming

The first technique applied is identifier renaming, a classic static obfuscation strategy. This technique systematically replaces all meaningful names in the code—such as variable names, function names, and parameter identifiers—with randomly generated, meaningless strings.

This transformation breaks any intuitive connection between the identifier and its functionality. For instance, a function originally named `calculateSum` might be renamed to `abxqke`, rendering the code semantically intact but much harder to read or reverse-engineer.

Importantly, this process preserves referential integrity throughout the code. That means any reference to the original identifier is consistently replaced across all usages, ensuring that the program continues to function correctly after obfuscation.

## 0.0.2 Dead Code Insertion

The second technique is dead code insertion. In this approach, syntactically valid but semantically irrelevant statements are added to the code. Common examples include declaring unused variables or inserting no-op operations.

These additions do not alter the actual flow or output of the program but serve to clutter the codebase. This misleads both human readers and automated analysis tools, increasing the cognitive load needed to understand the core logic.

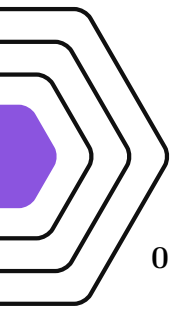
Such code can be inserted at strategic points (e.g., at the beginning of function bodies or code blocks), blending in with real logic while adding noise that does not affect runtime behavior.

## 0.0.3 Expression Rewriting (Equivalence)

The third technique involves rewriting expressions using mathematically equivalent alternatives. For example, a simple addition operation such as  $a + b$  might be transformed into  $a - (-b)$ .

These transformations retain the exact same computational outcome but introduce a layer of complexity to the expression tree. They are particularly effective in masking obvious arithmetic intent or distorting the simplicity of conditions and calculations.

This kind of obfuscation increases resistance to reverse-engineering by avoiding recognizable coding patterns while preserving the original semantics.



# Functional Equivalence Justification

## 0.0.4 Identifier Renaming

All variable and function names have been replaced with randomized, meaningless identifiers:

Original Name	Obfuscated Name
main	kepace
x	nmwskv
y	qgzyti
z	plyyym

### Explanation:

Impact:

These changes are purely lexical. Each identifier is consistently renamed throughout the program, ensuring logical and referential integrity.

## 0.0.5 Dead Code Insertion

Unused variables such as xtbbjc, ypbhxn, and mppjel have been inserted within the code:

```
1 int xtbbjc;  
2 ...  
3 int ypbhxn;  
4 ...  
5 int mppjel;
```

### Explanation:

Impact:

These variables do not affect program execution or logic. They serve only to obfuscate and clutter the code, making it harder to analyze manually.

## 0.0.6 Expression Rewriting (Semantic Equivalence)

The addition expression  $x + y$  has been rewritten as:

```
7 | plyyym = nmwskv - (-qgzyti);
```

Mathematical Principle:

$$a + b = a - (-b)$$

### Explanation:

#### Impact:

This transformation maintains the result but adds complexity to the code, thus reducing readability without altering semantics.

## Input and Output Demonstration

Input (input.mc):

```
8 int main() {
9     int x = 5;
10    int y = 10;
11    int z = x + y;
12    if (z > 10) {
13        return z;
14    } else {
15        return 0;
16    }
17 }
```

AST tree:

```
18 program
19   function_decl(main)
20     type(int)
21     params
22     block
23       var_decl
24         type(int)
25         var_list
26           var_item(x)
27             int(5)
28       var_decl
29         type(int)
30         var_list
31           var_item(y)
32             int(10)
33       var_decl
34         type(int)
35         var_list
36           var_item(z)
37             binop(+)
38               lvalue(x)
39               lvalue(y)
40       if
41         binop(>)
42         lvalue(z)
43         int(10)
```

```

44     block
45         return
46         lvalue(z)
47     block
48         return
49         int(0)

```

Obfuscated AST:

```

50 program
51     function_decl(lukxiz)
52     type(int)
53     params
54     block
55         var_decl
56             type(int)
57             var_list
58                 var_item(rmyudc)
59     var_decl
60         type(int)
61         var_list
62             var_item(kglfyf)
63             int(5)
64     var_decl
65         type(int)
66         var_list
67             var_item(jrpmru)
68             int(10)
69     var_decl
70         type(int)
71         var_list
72             var_item(nvxcdz)
73             binop(-)
74                 lvalue(kglfyf)
75             unary(-)
76                 lvalue(jrpmru)
77     if
78         binop(>)
79             lvalue(nvxcdz)
80             int(10)
81     block
82         var_decl
83             type(int)
84             var_list
85                 var_item(tsqusp)
86         return
87             lvalue(nvxcdz)
88     block
89         var_decl
90             type(int)
91             var_list
92                 var_item(oczawr)
93         return
94             int(0)
95     label(wcsxzq)

```

---

Output (output.mc):

```
96 int kepace() { //rename variable
97     int xtbbjc; //dead code
98     int nmwskv = 5;
99     int qgzyti = 10;
100    int plyyym = (nmwskv - (-qgzyti)); //expression equivalence
101    if ((plyyym > 10)) {
102        int ypbhxn; //dead code
103        return plyyym;
104    } else {
105        int mppjel; //dead code
106        return 0;
107    }
108 }
```



## Technical Challenges Faced

- Constructing a correct AST from ANTLR-based grammar
- Handling name collisions in variable renaming
- Managing recursive AST traversal and modification
- Guaranteeing semantic equivalence post-obfuscation
- Building a code generator from modified AST back to Mini-C



## Conclusion

This project successfully demonstrates a multi-stage, static-code obfuscator that can process Mini-C input and generate valid, functionally equivalent but obfuscated code. The modular design (AST, obfuscator, code generator) allows future expansion, such as:  
Support for pointer manipulation More advanced control flow rewriting Integration with runtime obfuscation