Project Document: Mini-C Obfuscator

Rozhin Khalilian – 40010803
Kimia Mazloomifar – 40010893
Alireza Karaminejad - 40010883

5/20/2025

—

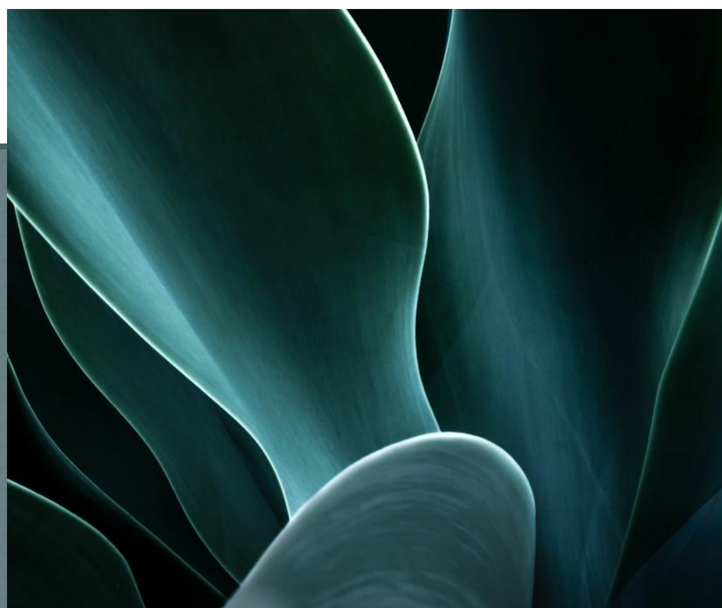
Compiler

—

Dr. Alaeiyan

INSTRUCTIONS

The goal of this project is to design and implement an Obfuscation tool that operates on Mini-C code, producing a functionally equivalent version that is more difficult for humans to comprehend



THE PROCESS

Mini-C Language Scope

For simplified control and evaluation, the Mini-C language subset includes the following features:

- Basic Data Types: `int`, `char`, `bool`
- Variables and Operators
- Control Flow: `if`, `else`, `while`, `for`, `return`
- Functions with parameters and return values
- Input/Output: Via `printf`, `scanf`
- Exclusions: Without `struct` and `pointer`

Implementation Approach

The implementation of the obfuscator follows a two-stage process:

Stage 1: Input Code Analysis

This stage involves designing a syntactic analyzer or utilizing tools like ANTLR or Flex/Bison to construct the Abstract Syntax Tree (AST).

Tool Used: The `pycparser` library is utilized for parsing Mini-C code into an AST and for generating C code from the modified AST.

Preprocessing: Before parsing, the tool removes GCC-specific and other non-standard C extensions such as `__attribute__`, `__restrict`, `__extension__`, `__volatile`, `__inline`, `__asm__`, and `__declspec` to ensure compatibility with `pycparser` and avoid parsing errors.

Stage 2: Applying Obfuscation Techniques on AST

At least three different obfuscation techniques must be implemented on the AST. This project implements five distinct obfuscation techniques.

Code related to using `pycparser`:

```
main_ast.py:

import pycparser
parser = c_parser.CParser()
ast = parser.parse(processed_c_code_cli_str)
```

Code related to removing GCC extensions:

```
main_ast.py:

processed_c_code_cli_str =
re.sub(r'__attribute__\s*\(\(([^)]*\)\)', '',
processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__restrict(?:__)?', '',
processed_c_code_cli_str)
processed_c_code_cli_str = re.sub(r'__extension__',
'', processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__volatile(?:__)?', '',
processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__inline(?:__)?', '',
processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__asm__\s*\(\s*\".*?\"\s*\)', '',
processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__asm\s*\(\s*\".*?\"\s*\)', '',
processed_c_code_cli_str)
processed_c_code_cli_str =
re.sub(r'__declspec\s*\([^)]*\)', '',
processed_c_code_cli_str)
```

This involves multiple files, each implementing a technique.

Obfuscation Techniques Implemented

The project implements the following obfuscation techniques:

6.1. Renaming Identifiers

This technique replaces meaningful variable and function names with meaningless, obfuscated names (e.g., `var1`, `xyz2`).

Implementation Details: The `ASTIdentifierRenamer` class (in `ast_rename.py`) traverses the AST. It maintains a mapping of original names to new, obfuscated names.

- **Variable Renaming**: Variables are renamed with a `var` prefix followed by a counter (e.g., `var1`, `var2`).
- **Function Renaming**: Functions are renamed with an `xyz` prefix followed by a counter (e.g., `xyz1`, `xyz2`).
- **Reserved Keywords**: C reserved keywords (e.g., `if`, `for`, `int`) are not renamed.
- **Temporary Names**: The renamer also handles temporary names generated by other obfuscation passes (e.g., `deadcode_`, `dummyVariable_`, `opaque_var_`, `dummyFunction_`) to ensure consistency and further obscure the code. This prevents the detection of obfuscated elements by their default temporary names.

File: `ast_rename.py`

Key classes and functions:

`ASTIdentifierRenamer` class: Implements the AST traversal and renaming logic.

`get_or_create_renamed_name()`: Manages the mapping of original names to obfuscated names.

`_generate_new_obfuscated_name()`: Generates new names.

Code snippets:

```
class ASTIdentifierRenamer(c_ast.NodeVisitor):
    # ... (rest of the class definition)
def get_or_create_renamed_name(original_name,
category_key):
    # ...
def _generate_new_obfuscated_name(category_key):
          # ...
```

6.2. Dead Code Insertion

This technique inserts unused code sections that do not affect the program's execution.

Implementation Details: The `DeadCodeInserter` class (in `ast_dead_code.py`) inserts declarations of unused integer variables initialized with random values into compound statements (code blocks). This increases code size and complexity without changing its functionality. For example, `int deadcode_1 = 1532;` is inserted into various `Compound` AST nodes.

File: `ast_dead_code.py`

Key classes and functions:

`DeadCodeInserter` class: Inserts dead code into the AST.

`create_dead_variable()`: Creates the AST node for a dead variable declaration.

Code snippets:

```
class DeadCodeInserter(c_ast.NodeVisitor):
    # ...
    def visit_Compound(self, node):
        # ...
    def create_dead_variable(self):
          # ...
```

File: `ast_equivalent_expr.py`

Key classes and functions:

`EquivalentExpression` class: Transforms expressions in the AST.

`apply_equivalent_expression()`: Entry point to apply the transformation.

Code snippets:

```
class EquivalentExpression(c_ast.NodeVisitor):
    # ...
    def visit_BinaryOp(self, node):
        # ...
def apply_equivalent_expression(ast_root_node):
        # ...
```

6.3. Equivalent Expression Transformation

This technique modifies expressions to use semantically equivalent but syntactically more complex forms.

Implementation Details: The `EquivalentExpression` class (in `ast_equivalent_expr.py`) transforms binary operations:

- `a + b` is transformed into `a - (-b)`.
- `a - b` is transformed into `a + (-b)`. This is applied by visiting `BinaryOp` nodes in the AST and replacing them with their equivalent but more obscure forms.

6.4. Dummy Function Insertion

This technique involves injecting new, self-contained functions that are never called within the original code.

Implementation Details: The `DummyFunctionInjector` class (in `ast_dummy_function.py`) creates new function definitions. These functions typically declare local variables, perform conditional assignments, and return a calculated value, but they are designed to be unreachable and serve only to increase code volume and complexity. For example, `int dummyFunction_1() { int dummyVariable_1 = ...; ... return dummyVariable_1; }`.

File: `ast_dummy_function.py`

Key classes and functions:

`DummyFunctionInjector` class: Inserts dummy functions into the AST.

`create_dummy_function_ast()`: Creates the AST node for a dummy function.

Code snippets:

```
class DummyFunctionInjector():
    def __init__(self):
        # ...
    def create_dummy_function_ast(self):
        # ...
    def dummy_function_injection(self, ast_root_node,
num_dummy_functions=1):
         # ...
def apply_dummy_function_insertion(ast_root_node):
    # ...
```

6.5. Opaque Predicate Insertion

This technique inserts `if-else` statements with conditions that are always true or always false, making control flow analysis more difficult.

Implementation Details: The `OpaquePredicateInserter` class (in `ast_opaque_predicate.py`) generates `if-else` blocks where the condition is based on a variable initialized with a known value, leading to a deterministically true or false outcome (e.g., `if (opaque_var_1 == 42)` where `opaque_var_1` is initialized to `42`). Both the `if` and `else` branches typically declare dummy variables, thus adding dead code paths. These are randomly inserted into `Compound` statements.

File: `ast_opaque_predicate.py`

Key classes and functions:

`OpaquePredicateInserter` class: Inserts opaque predicates.

`create_opaque_if_statement()`: Creates the `if-else` statement with the opaque predicate.

Code snippets:

```
class OpaquePredicateInserter(c_ast.NodeVisitor):
    # ...
    def visit_Compound(self, node):
```

Input and Output

Input: The obfuscator takes a text file with a `.mc` extension (e.g., `input.mc`) containing a valid Mini-C program.

Output: It produces a text file named `output.mc` (or a user-specified name) that is the obfuscated version of the input program.

Functional Equivalence: The behavior of the input and output programs must be functionally identical in terms of input and output.

User Interface (UI)

The project provides both a Graphical User Interface (GUI) and a Command-Line Interface (CLI).

8.1. Graphical User Interface (GUI)

The `main_ast.py` script implements a Tkinter-based GUI (`ObfuscatorGUI` class).

File Operations: Allows users to load input Mini-C files and select output file paths.

Obfuscation Selection: Provides checkboxes to enable or disable individual obfuscation techniques (Rename Identifiers, Inject Dead Code, Equivalent Expressions, Insert Dummy Functions, Insert Opaque Predicates).

Code Display: Features two text areas for real-time display of the input and obfuscated code.

Execution: A "Run Obfuscator" button initiates the obfuscation process based on selected options.

File: `main_ast.py`

Class: `ObfuscatorGUI`

Code snippets:

```
class ObfuscatorGUI:
    def __init__(self, root_window):
        # ... (GUI initialization code)
```

```
    def load_file(self):
        # ...
    def run_obfuscator(self):
        # ...
```

8.2. Command-Line Interface (CLI)

The `main_ast.py` script also supports a CLI mode.

Usage: Users can provide an input file path and an optional output file path as command-line arguments.

Default Behavior: In CLI mode, all implemented obfuscation techniques are applied by default.

File: `main_ast.py`

Function: `run_obfuscator_cli()`

Code snippets:

```
def run_obfuscator_cli(input_file_arg,
output_file_arg=None):
    # ... (CLI logic)
if __name__ == "__main__":
    if len(sys.argv) > 1:
        # ... (CLI argument parsing)
```

Functional Equivalence Justification

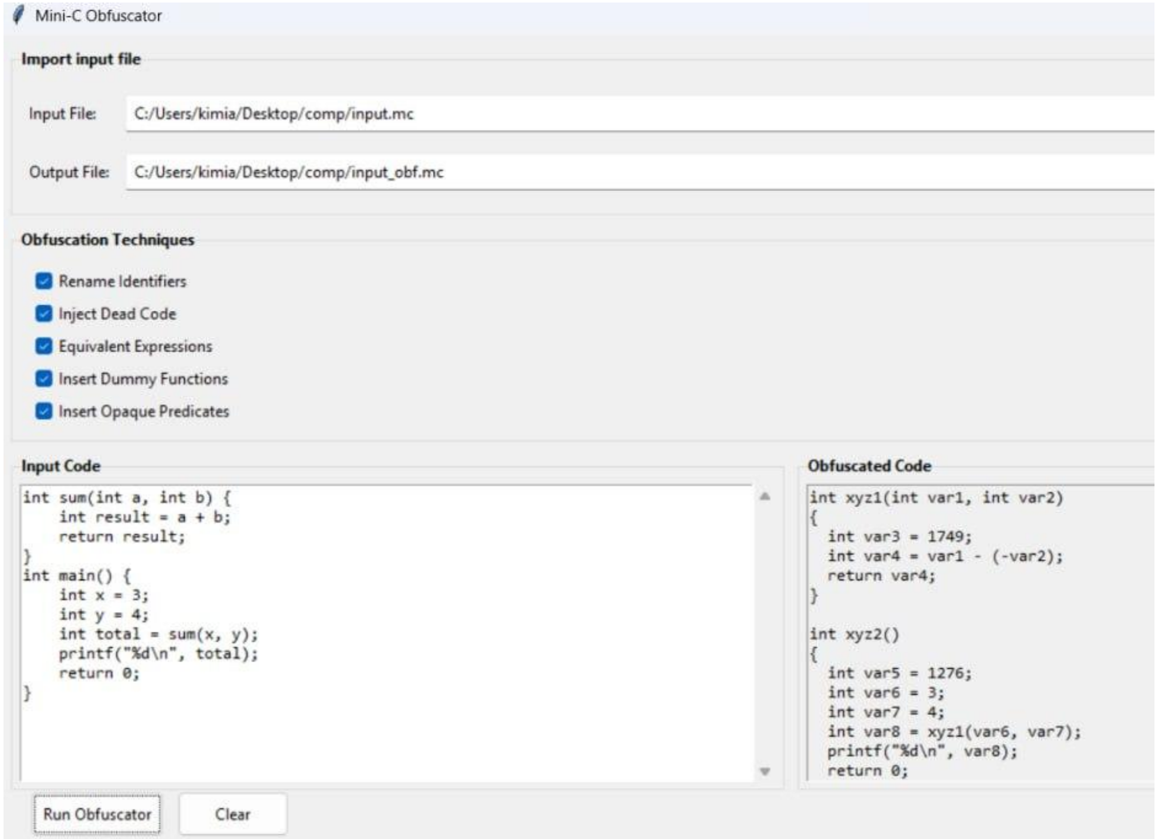
The obfuscator ensures functional equivalence by operating exclusively on the Abstract Syntax Tree (AST) representation of the code. Each obfuscation technique is designed to modify the code's structure and appearance without altering its semantic meaning or execution flow.

AST Manipulation: By working on the AST, the tool ensures that the fundamental logical operations and control flow of the program remain intact, only their representation is changed.

Technique-Specific Guarantees:

- **Renaming**: Only identifier names are changed; their references and scope remain consistent.
- **Dead Code**: Inserted code is explicitly unreachable and consists of variable declarations that do not affect the program's state or output.
- **Equivalent Expressions**: Mathematical identities are preserved (e.g., `a+b` is equivalent to `a-(-b)`).
- **Dummy Functions**: These functions are never called, ensuring they have no runtime effect on the original program's execution.
- **Opaque Predicates**: The conditions are always true or always false, meaning only one branch is ever executed, and the dummy declarations within these branches have no side effects on the program's output.

Results :



Mini-C Obfuscator
Import input file
Input File: C:/Users/kimia/Desktop/comp/input.mc
Output File: C:/Users/kimia/Desktop/comp/input_obf.mc
Obfuscation Techniques
Rename Identifiers
Inject Dead Code
Equivalent Expressions
Insert Dummy Functions
Insert Opaque Predicates
Input Code
int sum(int a, int b) {
int result = a + b;
return result;
}
int main() {
int x = 3;
int y = 4;
int total = sum(x, y);
printf("%d\n", total);
return 0;
}
Obfuscated Code
int xyz1(int var1, int var2)
{
int var3 = 1749;
int var4 = var1 - (-var2);
return var4;
}
int xyz2()
{
int var5 = 1276;
int var6 = 3;
int var7 = 4;
int var8 = xyz1(var6, var7);
printf("%d\n", var8);
return 0;
Run Obfuscator
Clear