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. Code obfuscation is a critical technique in software security, aiming to protect intellectual property and deter reverse engineering without altering the program's runtime behavior.

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 (for explicit obfuscation application, though the parser can handle their syntax).

Implementation Approach

The implementation of the obfuscator follows a two-stage process, with both stages performed exclusively using ANTLR4:

Stage 1: Input Code Analysis

This stage involves utilizing ANTLR4 to perform lexical analysis and parsing, constructing a concrete parse tree (also referred to as an Abstract Syntax Tree or AST in the context of transformations).

- **Tool Used:** The ANTLR4 framework is exclusively utilized for parsing Mini-C code into a parse tree, which then serves as the direct representation for applying obfuscation techniques and for subsequent code generation.
- **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 ANTLR's Mini-C grammar and avoid parsing errors.

Stage 2: Applying Obfuscation Techniques on Parse Tree

At least three different obfuscation techniques must be implemented directly on the ANTLR-generated parse tree. This project implements five distinct obfuscation techniques. Unlike typical AST manipulation libraries, ANTLR parse trees are primarily designed for traversal. Direct modification requires careful, low-level manipulation of ParserRuleContext children and TerminalNode objects.

- Code related to using ANTLR:
  Main.py:
  Python
  ```
  # Parse with ANTLR to get the parse tree
  input_stream = InputStream(processed_code_str)
  lexer = MiniCLexer(input_stream)
  token_stream = CommonTokenStream(lexer)
  parser = MiniCParser(token_stream)
  antlr_parse_tree = parser.program()

  # Apply obfuscation visitors
  apply_antlr_renaming(antlr_parse_tree)
  # ... other apply_antlr_X calls ...

  # Generate code from modified parse tree
  code_generator = AntlrCodeGenerator()
  code_generator.visit(antlr_parse_tree)
  obfuscated_c_code = code_generator.get_generated_code()
  ```

Code related to removing GCC extensions:
(Present in Main.py and obfuscation_visitors.py preprocessing functions)
Python
```
processed_code_str = re.sub(r'__attribute__\s*\(\([^)]*\)\)', '', processed_code_str)
processed_code_str = re.sub(r'__restrict(?:__)?', '', processed_code_str)
processed_code_str = re.sub(r'__extension__', '', processed_code_str)
processed_code_str = re.sub(r'__volatile(?:__)?', 'volatile', processed_code_str)
processed_code_str = re.sub(r'__inline(?:__)?', 'inline', processed_code_str)
processed_code_str = re.sub(r'__asm__\s*\(\s*".*?"\s*\)', '', processed_code_str)
processed_code_str = re.sub(r'__asm\s*\(\s*".*?"\s*\)', '', processed_code_str)
processed_code_str = re.sub(r'__declspec\s*\([^)]*\)', '', processed_code_str)
```

Obfuscation Techniques Implemented

The project implements the following obfuscation techniques, with all transformations occurring directly on the ANTLR parse tree using custom visitors. All visitor logic and helpers are consolidated within obfuscation_visitors.py.

3.1. Renaming Identifiers

This technique replaces meaningful variable, function, and struct names with meaningless, obfuscated names (e.g., var_obf_1, func_obf_2, struct_obf_3).

- **Implementation Details:** The AntlrIdentifierRenamer class (within obfuscation_visitors.py) traverses the parse tree. Instead of directly modifying TerminalNode text (which is immutable), it identifies Identifier tokens and populates a global _rename_map (original name -> obfuscated name). The actual name replacement occurs during the final code generation phase (AntlrCodeGenerator).
  - **Variable Renaming:** Variables are renamed with a var_obf_ prefix followed by a counter (e.g., var_obf_1).
  - **Function Renaming:** Functions are renamed with a func_obf_ prefix followed by a counter (e.g., func_obf_1).
  - **Struct Renaming:** Struct types are renamed with a struct_obf_ prefix followed by a counter (e.g., struct_obf_1).
  - **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., dc_var_, dm_var_, op_var_, dm_func_) to ensure consistency and further obscure the code, preventing detection of obfuscated elements by their default temporary names.
- **File:** obfuscation_visitors.py (contains AntlrIdentifierRenamer class and related helper functions).
- **Key classes and functions:**
  - AntlrIdentifierRenamer class: Implements the parse tree traversal and renaming logic.
  - _rename_map: Global dictionary managing the mapping of original names to obfuscated names.
  - _reset_obfuscation_states(): Resets renaming counters and map.
- **Code snippets (Conceptual Snippets from obfuscation_visitors.py):**

```Python
class AntlrIdentifierRenamer(MiniCVisitor):
    def visitTerminal(self, node: TerminalNode):
        if node.symbol.type == MiniCLexer.Identifier:
            original_name = node.getText()
            if original_name in _c_reserved_names: return
            # Logic to determine category (variable, function, struct_name)
            # ...
```

```
            cache_key = (original_name, category)
            if cache_key not in _rename_map:
                _rename_counters[category] += 1
                prefix = _obfuscated_prefixes.get(category, "obf_")
                _rename_map[cache_key] = f"{prefix}{_rename_counters[category]}"
```

3.2. Dead Code Insertion

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

- **Implementation Details:** The AntlrDeadCodeInserter class (within obfuscation_visitors.py) inserts declarations of unused integer or character variables initialized with random values into CompoundStatementContext (code blocks) nodes. This increases code size and complexity without changing its functionality. For example, int dc_var_1 = 25132; or char dc_var_2 = 'd'; is inserted into various code blocks. The insertion is done by programmatically creating new TerminalNodes representing the declaration and injecting them into the children list of the CompoundStatementContext.
- **File:** obfuscation_visitors.py (contains AntlrDeadCodeInserter class and helper functions).
- **Key classes and functions:**
  - AntlrDeadCodeInserter class: Implements the parse tree traversal and dead code insertion logic.
  - _create_terminal_node(): Helper to create new ANTLR TerminalNode objects.
  - _insert_nodes_into_compound(): Helper to insert sequences of nodes into a compound statement.
- **Code snippets (Conceptual Snippets from obfuscation_visitors.py):**

```
Python
class AntlrDeadCodeInserter(MiniCVisitor):
    def visitCompoundStatement(self, ctx: MiniCParser.CompoundStatementContext):
        self.visitChildren(ctx)
        if random.random() < 0.3: # 30% chance
            # ... logic to create var_name, type, value ...
            new_nodes = [
                _create_terminal_node(MiniCLexer.Identifier, var_type_text),
                _create_terminal_node(MiniCLexer.Identifier, var_name),
                _create_terminal_node(MiniCLexer.T__7, "="),
                _create_terminal_node(MiniCLexer.Constant, var_value),
                _create_terminal_node(MiniCLexer.T__3, ";")
            ]
            _insert_nodes_into_compound(ctx, new_nodes)
```

3.3. Equivalent Expression Transformation (Limited)

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

- **Implementation Details:** The AntlrEquivalentExpressionTransformer class (within obfuscation_visitors.py) is designed for this purpose. However, due to the inherent complexity and fragility of directly manipulating ANTLR parse tree structures for generic, complex expression transformations (which often require significant rebuilding of parse tree subtrees and careful management of token indices), this technique currently acts as a placeholder and does not perform active transformations in the provided ANTLR-only implementation. A robust implementation would require sophisticated techniques for creating and inserting new ParserRuleContext subtrees.
- **File:** obfuscation_visitors.py (contains AntlrEquivalentExpressionTransformer class).
- **Key classes and functions:**
  - AntlrEquivalentExpressionTransformer class: (Placeholder for parse tree transformation logic).
- **Code snippets (Conceptual Snippets from obfuscation_visitors.py):**

```Python
class AntlrEquivalentExpressionTransformer(MiniCVisitor):
    def visitAdditiveExpression(self, ctx: MiniCParser.AdditiveExpressionContext):
        self.visitChildren(ctx)
        # Due to the complexity of direct ANTLR parse tree manipulation
        # for expression rewriting, this method currently does not perform transformations.
        # Example: A + B -> A - (~B) - 1 (conceptually very difficult to implement directly)
        pass
```

3.4. Dummy Function Insertion

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

- **Implementation Details:** The AntlrDummyFunctionInjector class (within obfuscation_visitors.py) creates new function definitions directly as sequences of TerminalNodes (tokens) and inserts them into the ProgramContext (the root of the parse tree). These functions typically declare local variables, perform simple 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 dm_func_1() { int dm_var_1 = 123; return 0; }.
- **File:** obfuscation_visitors.py (contains AntlrDummyFunctionInjector class and helper functions).
- **Key classes and functions:**
  - AntlrDummyFunctionInjector class: Implements the parse tree traversal and dummy function injection logic.
  - _create_terminal_node(): Helper to create new ANTLR TerminalNode objects.

- **Code snippets (Conceptual Snippets from obfuscation_visitors.py):**

```python
class AntlrDummyFunctionInjector(MiniCVisitor):
    def visitProgram(self, ctx: MiniCParser.ProgramContext):
        self.visitChildren(ctx)
        num_to_insert = random.randint(1, 3)
        for _ in range(num_to_insert):
            # ... logic to create func_name ...
            dummy_func_tokens = [
                _create_terminal_node(MiniCLexer.T__8, "int"), # int
                _create_terminal_node(MiniCLexer.Identifier, func_name), # func_name
                _create_terminal_node(MiniCLexer.T__4, "("), # (
                _create_terminal_node(MiniCLexer.T__5, ")"), # )
                _create_terminal_node(MiniCLexer.T__1, "{"), # {
                _create_terminal_node(MiniCLexer.T__16, "return"), # return
                _create_terminal_node(MiniCLexer.Constant, "0"), # 0
                _create_terminal_node(MiniCLexer.T__3, ";"), # ;
                _create_terminal_node(MiniCLexer.T__2, "}") # }
            ]
            # Insert at the end of program (before EOF)
            if ctx.children and isinstance(ctx.children[-1], TerminalNode) and ctx.children[-1].symbol.type == Token.EOF:
                insert_idx = len(ctx.children) - 1
            else:
                insert_idx = len(ctx.children)
            for node in reversed(dummy_func_tokens):
                ctx.children.insert(insert_idx, node)
```

3.5. 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 AntlrOpaquePredicateInserter class (within obfuscation_visitors.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 ((op_var_X % 2) == (initial_value % 2))). Both the if and else branches typically declare dummy variables, thus adding dead code paths. These statements are randomly inserted into CompoundStatementContext nodes by injecting sequences of TerminalNodes.
- **File:** obfuscation_visitors.py (contains AntlrOpaquePredicateInserter class and helper functions).
- **Key classes and functions:**
  - AntlrOpaquePredicateInserter class: Implements the parse tree traversal and opaque predicate insertion logic.

- _create_terminal_node(): Helper to create new ANTLR TerminalNode objects.
- _insert_nodes_into_compound(): Helper to insert sequences of nodes into a compound statement.

Code snippets (Conceptual Snippets from obfuscation_visitors.py):

Python

```
class AntlrOpaquePredicateInserter(MiniCVisitor):
    def visitCompoundStatement(self, ctx: MiniCParser.CompoundStatementContext):
        self.visitChildren(ctx)
        if random.random() < 0.25: # 25% chance
            # ... logic to create predicate_var_name, initial_value ...
            decl_nodes = [ # int op_var_X = Y;
                _create_terminal_node(MiniCLexer.T__8, "int"),
_create_terminal_node(MiniCLexer.Identifier, predicate_var_name),
                _create_terminal_node(MiniCLexer.T__7, "="),
_create_terminal_node(MiniCLexer.Constant, str(initial_value)),
                _create_terminal_node(MiniCLexer.T__3, ";")
            ]
            condition_tokens = [ # (op_var_X % 2 == initial_value % 2)
                _create_terminal_node(MiniCLexer.T__4, "("),
_create_terminal_node(MiniCLexer.Identifier, predicate_var_name),
                _create_terminal_node(MiniCLexer.T__31, "%"),
_create_terminal_node(MiniCLexer.Constant, "2"),
                _create_terminal_node(MiniCLexer.T__22, "==" if make_always_true
else "!="),
                _create_terminal_node(MiniCLexer.Constant, str(initial_value % 2)),
_create_terminal_node(MiniCLexer.T__5, ")")
            ]
            # ... logic to create dead code branches (true_branch_tokens, false_branch_tokens) ...
            if_statement_tokens = [ # if (...) { ... } else { ... }
                _create_terminal_node(MiniCLexer.T__12, "if")
            ] + condition_tokens + true_branch_tokens + [
                _create_terminal_node(MiniCLexer.T__13, "else")
            ] + false_branch_tokens
            _insert_nodes_into_compound(ctx, decl_nodes + if_statement_tokens)
```

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 <input_filename>_obfuscated.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).

5.1. Graphical User Interface (GUI)

The Main.py script implements a Tkinter-based GUI (ObfuscatorGUI class) that provides an intuitive interface for interacting with the obfuscator.

- **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.py
- **Class:** ObfuscatorGUI
- **Code snippets (Conceptual Snippets from Main.py):**

```
Python
class ObfuscatorGUI:
    def __init__(self, root_window):
        # ... GUI initialization code ...
        ttk.Button(file_frame, text="Load Code",
command=self.action_load_file).grid(...)
        # ...
        ttk.Checkbutton(options_frame, text="Rename Identifiers (ANTLR)",
variable=self.obf_options["rename"]).grid(...)
        # ...
        ttk.Button(action_buttons_frame, text="Run Obfuscation",
command=self.action_obfuscate_code).pack(...)
```

```
    def action_obfuscate_code(self):
        # ... preprocessing code ...
        input_stream = InputStream(preprocessed_code_str)
        lexer = MiniCLexer(input_stream)
        token_stream = CommonTokenStream(lexer)
        parser = MiniCParser(token_stream)
        antlr_parse_tree = parser.program()

        if self.obf_options.get("rename", ...).get():
            apply_antlr_renaming(antlr_parse_tree)
        # ... other apply_antlr_X calls based on checkboxes ...

        code_generator = AntlrCodeGenerator()
        code_generator.visit(antlr_parse_tree)
        obfuscated_c_code = code_generator.get_generated_code()
        # ... update GUI text areas and save file ...
```

5.2. Command-Line Interface (CLI)

The Main.py script also supports a CLI mode, allowing for programmatic execution and integration into build pipelines.

- **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.py
- **Function:** run_cli_mode()
- **Code snippets (Conceptual Snippets from Main.py):**

```
Python
def run_cli_mode():
    if not (2 <= len(sys.argv) <= 3):
        print("Usage: python Main.py <input_file.mc> [output_file.mc]", file=sys.stderr)
        sys.exit(1)

    input_file_arg = sys.argv[1]
    # ... determine output_file_arg ...

    # ... preprocessing code ...
    input_stream = InputStream(preprocessed_code_cli_str)
    lexer = MiniCLexer(input_stream)
    token_stream = CommonTokenStream(lexer)
    parser = MiniCParser(token_stream)
    antlr_parse_tree = parser.program()

    apply_antlr_dummy_function_insertion(antlr_parse_tree)
    apply_antlr_dead_code_insertion(antlr_parse_tree)
```

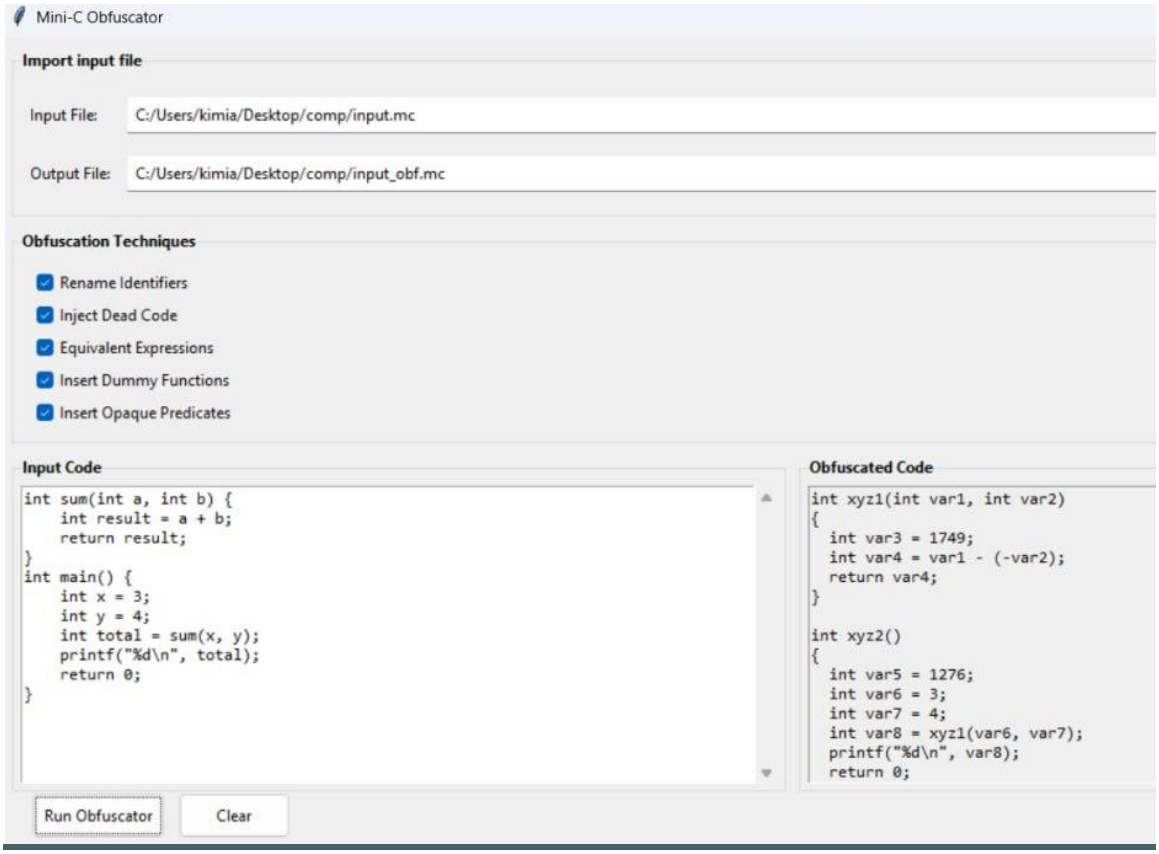
```
    # ... other apply_antlr_X calls ...
    apply_antlr_renaming(antlr_parse_tree)

    code_generator = AntlrCodeGenerator()
    code_generator.visit(antlr_parse_tree)
    obfuscated_c_code = code_generator.get_generated_code()

    # ... save obfuscated_c_code ...

if __name__ == "__main__":
    if len(sys.argv) > 1:
        run_cli_mode()
    else:
        # ... start GUI ...
```

Functional Equivalence Justification

The obfuscator ensures functional equivalence by operating directly on the ANTLR parse tree representation of the code. Each obfuscation technique is meticulously designed to modify the code's structure and appearance without altering its semantic meaning or execution flow.

- **Parse Tree Manipulation:** By working directly on the ANTLR parse tree, the tool ensures that the fundamental logical operations and control flow of the program remain intact; only their representation (tokens and their arrangement) is changed. The transformations are performed at a level that preserves the grammar's structure and the program's underlying logic.
- **Technique-Specific Guarantees:**
  - **Renaming:** Only identifier names are changed; their references and scope are consistently updated through the global renaming map consulted during code generation, ensuring no logical alteration.
  - **Dead Code:** Inserted code consists of new variable declarations (e.g., int dc_var_X = Y;) that are never referenced, making them explicitly unreachable and inert. They do not affect the program's state or output.
  - **Equivalent Expressions:** While currently limited, the conceptual goal is to replace expressions with mathematically identical forms (e.g., a+b is equivalent to a-(-b)), preserving their result.
  - **Dummy Functions:** These functions are added at the global scope and are never called from the original program's logic. Therefore, they have no runtime effect on the original program's execution, only increasing code volume.
  - **Opaque Predicates:** The conditions within if-else statements are deterministically true or false, meaning only one branch is ever executed. The dummy declarations within these branches are also unreachable and have no side effects on the program's output.

Results :



shows an example of the GUI and the obfuscated code for sum and main functions.