Implemented Obfuscation Techniques

Name Surname

March 9, 2025

Introduction

The project is based on the implementation of two code obfuscation techniques, inspired by the paper *Property-driven Code Obfuscations*.
I implemented the **Parity Distorter** and the **Flattening Distorter** using Python and ANTLR4.

Motivation of Choices

ANTLR4 was preferred for:

- Simplicity in creating the parser.
- Flexibility in implementing code transformations.
- Ease of integration with Python for automation.

Grammar Highlights

The following slides present key portions of the ANTLR4 grammar used for the obfuscation techniques.

Commands Rule

```
prog: LBRACE com* RBRACE EOF;

com:
    ID ASSIGN exp SEMICOLON                 # Assignment
    | SKIP_CMD SEMICOLON                    # Skip
    | IF LPAREN bExp RPAREN LBRACE
    com* RBRACE elseTail SEMICOLON              # If
    | WHILE LPAREN bExp RPAREN LBRACE
    com* RBRACE SEMICOLON                       # WhileLoop;

elseTail:
    ELSE LBRACE com* RBRACE                 # IfElseTail
    | ELSE IF LPAREN bExp RPAREN
    LBRACE com* RBRACE elseTail             # IfElseIfTail
    |                                       # IfElseTailEmpty;
```

Expressions Rule

```
exp: aExp | bExp;
```

Arithmetic Expressions Rule

```
aExp:
    aExp '*' aExp   # Multiplication
    | aExp '/' aExp   # Division
    | aExp '+' aExp   # Addition
    | aExp '-' aExp   # Subtraction
    | LPAREN aExp RPAREN # AParenthesis
    | INT     # Integer
    | ID     # ArithmeticVariable;
```

Boolean Expressions Rule

```
bExp:
    | ID bExp_tail
    | TRUE bExp_tail
    | FALSE bExp_tail
    | aExp bExp_tail
    | LPAREN bExp RPAREN bExp_tail
    ;

bExp_tail:
    EQ exp bExp_tail
    | NEQ exp bExp_tail
    | GT exp bExp_tail
    | LT exp bExp_tail
    | AND exp bExp_tail
    |
    ;
```

Interpreter

First, the ANTLR4 grammar was used to create an interpreter for the language. The interpreter was then used to test the correctness of the grammar and the language constructs.

Parity Distorter

The **Parity Distorter** aims at obfuscating the *parity abstraction* on numerical values.

$$\mathbf{f}_{\vdash}(\mathbf{x}) \stackrel{\text{def}}{=} 2 * x \qquad \mathbf{f}_{\dashv}(\mathbf{x}) \stackrel{\text{def}}{=} x/2$$

$$\mathbf{f_c}(\mathbf{c}) \stackrel{\text{def}}{=} \begin{cases} 2 * \mathbf{f_{ex}}(\mathbf{a}) & \text{if } c = x := a \\ \mathbf{skip} & \text{if } c = \mathbf{skip} \end{cases} \qquad \mathbf{f_b}(\mathbf{b}) \stackrel{\text{def}}{=} \mathbf{f_{ex}}(\mathbf{b})$$

$$\mathbf{f_{ex}}(\mathbf{x}) \stackrel{\text{def}}{=} x/2 \qquad \mathbf{f_{ex}}(\mathbf{n}) \stackrel{\text{def}}{=} n$$

$$\mathbf{f_{ex}}(\mathbf{e}\ \text{bop}\ \mathbf{e}) \stackrel{\text{def}}{=} \mathbf{f_{ex}}(\mathbf{e})\ \text{bop}\ \mathbf{f_{ex}}(\mathbf{e}) \qquad \mathbf{f_{ex}}(\neg \mathbf{b}) \stackrel{\text{def}}{=} \neg \mathbf{f_{ex}}(\mathbf{b})$$

Parity Distorter: Class Definition

```
class ParityDistorter(langVisitor):
    def __init__(self):
        self.vars = set()
```

Parity Distorter: Main Program

```
def visitProg(self, ctx: langParser.ProgContext):
    result = "{\n"
    for com in ctx.com():
        result += self.visit(com)
    for var in self.vars:
        result += f"{var} := {self.__obf_var(var)};\n"
    return result + "}"
```

Parity Distorter: Assignments

```
def __obf_com(self, ctx):
    if isinstance(ctx, langParser.AssignmentContext):
        return f"{ctx.ID().getText()} := 2 * ({self.visit(ctx.exp())});\n"
```

Parity Distorter: Skip

```
elif isinstance(ctx, langParser.SkipContext):
    return f"skip; "
```

Parity Distorter: While Loop

```
elif isinstance(ctx, langParser.WhileLoopContext):
    result = f"while ({self.visit(ctx.bExp())}) {{\n"
    for com in ctx.com():
        result += self.visit(com)
    return result + "};\n"
```

Parity Distorter: Variables

```
def __obf_var(self, varName: str):
    self.vars.add(varName)
    return f"({varName} / 2)"
```

Flattening Distorter

The **Flattening Distorter** aims at obfuscating the *Control Flow Graph*. The flattening is made by making the *program counter (pc)* dynamic.
The idea is to introduce a new variable **pc** that will be used to determine the next block of code to execute.

Flattening Distorter: Class Definition

```
class FlatteningDistorter(langVisitor):
    def __init__(self):
        self.vars = set()
        self.pc = 0
```

Flattening Distorter: Main Program

```
def visitProg(self, ctx: langParser.ProgContext):
    self.pc = 1
    result = "{\n\tpc := 1;\n\twhile (pc != 0) {"
    for com in ctx.com():
        result += self.visit(com)
    result += f"""
    if(pc = {self.pc}) {{
        pc := 0;
    }};\n\n"""  # End of program
    return result + "\t};\n}"
```

Flattening Distorter: Assignments

```
def __obf_com(self, ctx, exit_pc=None):
    if isinstance(ctx, langParser.AssignmentContext):
        result = f"""
    if (pc = {self.pc}) {{
        {ctx.ID().getText()} := {self.visit(ctx.exp())};
        pc := {self.pc + 1 if exit_pc is None else exit_pc};
    }};\n"""
        self.pc += 1
        return result
```

Flattening Distorter: Skip

```
elif isinstance(ctx, langParser.SkipContext):
    return "skip; "
```

Flattening Distorter: While Loop

```
elif isinstance(ctx, langParser.WhileLoopContext):
    starting_pc = self.pc
    self.pc = starting_pc + 1
    body = ""
    for com in ctx.com():
        body += self.visit(com)
    header = f"""
if(pc = {starting_pc}) {{
    if ({ctx.bExp().getText()}) {{
        pc := {starting_pc + 1};
    }} else {{
        pc := {self.pc + 1 if exit_pc is None else exit_pc};
    }};
}};
            """
    result = header + body
    result += f"""
if (pc = {self.pc}) {{
    pc := {starting_pc};
}};\n"""
    self.pc = self.pc + 1
    return result
```

Flattening Distorter: If Statement

```
def visitIf(self, ctx: langParser.IfContext):
    branches = [{
        "bExp": self.visit(ctx.bExp()),
        "ctx": ctx,
        "coms": ctx.com()
    }]
    branches.extend(self.visit(ctx.elseTail()))

    start_pc = self.pc                          # start: current pc
    exit_pc = self.pc + len(branches)           # exit node pc

    self.pc = exit_pc + 1                       # body program counter

    body = ""
    branches_pc = []
    for i, b in enumerate(branches):
        coms = b["coms"]
        branches_pc.append(self.pc)
        for j, c in enumerate(coms):
            pc = exit_pc if j == (len(coms) - 1) else None
            body += self.__obf_com(c, exit_pc=pc)

    last_pc = self.pc
```

Flattening Distorter: If Statement

```
self.pc = start_pc                          # header program counter
header = ""
for i in range(len(branches_pc)):
    if not isinstance(branches[i]["ctx"], langParser.IfElseTailContext):
        header += f"""
            if (pc = {self.pc}) {{
                if({branches[i]["bExp"]}) {{
                    pc := {branches_pc[i]};
                }} else {{
                    pc := {self.pc + 1};
                }};
            }};
        """
    else:
        header += f"""
            if (pc = {self.pc}) {{
                pc := {branches_pc[i]};
            }};
        """
    self.pc += 1

self.pc = last_pc
```

Flattening Distorter: If Statement

```
exit_node = f"""
    if(pc = {exit_pc}) {{
        pc := {self.pc};
    }};
"""
return header + exit_node + body
```

Flattening Distorter: If Statement

```
def visitIfElseTail(self, ctx: langParser.IfElseTailContext):
    return [{
        "bExp": None,
        "ctx": ctx,
        "coms": ctx.com()
    }]


def visitIfElseIfTail(self, ctx: langParser.IfElseIfTailContext):
    branches = [{
        "bExp": self.visit(ctx.bExp()),
        "ctx": ctx,
        "coms": ctx.com()
    }]
    branches.extend(self.visit(ctx.elseTail()))
    return branches


def visitIfElseTailEmpty(self, ctx: langParser.IfElseTailEmptyContext):
    return []
```