

گزارش کار فاز اول پروژه درس طراحی کامپایلر (طراحی مبهم ساز)

(اعضای گروه: امیرعلی محمدپور، ارمیا میرزایی، علی نوروزی)

بخش ۱) شرح کامل تکنیک های پیاده سازی شده

۱. تغییر نام متغیرها و توابع (Rename Obfuscation)

این متد با استفاده از کلاس (RenameVisitor) و قابلیت بازدید درخت نحوی (Parse Tree)) با ابزار (ANTLR) پیاده سازی شده است.

در این روش، تمام نامهای مرتبط با کد شامل:

- نام توابع ((Function names))
- نام پارامترهای ورودی توابع
- نام متغیرهای تعریف شده در بلوک ها
- و ارجاعات به آن ها در (assignment) ها یا (expression) ها

با استفاده از رشته هایی تصادفی جایگزین شدن. این رشته ها ترکیبی از حروف الفبا، اعداد و کاراکتر underscore (هستند و به گونه ای انتخاب می شوند که با نام های قبلی تداخلی نداشته باشند. به منظور جلوگیری از تکرار و حفظ همارزی رفتاری (mapping))، نکاشتی (Behavioral Equivalence)) بین نام اصلی و نام جدید در قالب (name_map) نگهداری می شود.

токن هایی که به این شناسه ها اشاره دارند توسط (TokenStreamRewriter) جایگزین می شوند. این تکنیک باعث از بین رفتن معنا و شفافیت کد می شود، چرا که خواننده دیگر نمی تواند با استناد به نام ها منطق برنامه را دنبال کند.

مزایا:

- حفظ عملکرد کامل برنامه
- دشوار کردن خوانایی و درک منطقی کد برای انسان با نام های بی معنا
- ساده سازی پیاده سازی و کمترین تأثیر روی ساختار کلی کد

۲. درج کد مرده (Dead Code Insertion)

با استفاده از کلاس (DeadCodeInjector) پیاده‌سازی شد. در این تکنیک، بعد از هر (statement)، یک یا چند بلوک کد مرده به صورت تصادفی ایجاد و درج می‌شود. این کدها درون ساختارهایی مانند (if (0)) یا (while (0) {...}) قرار می‌گیرند تا هیچ‌گاه اجرا نشوند.

محتوای این بلوک‌ها ممکن است:

- متغیرهای بدون استفاده را تعریف کند.
- مقادیری تصادفی به متغیرها تخصیص دهد.
- شامل حلقه یا شرطی باشد که اجرای آن غیرممکن است.

هدف این است که ظاهر کلی برنامه را به شدت شوغ کرده و حجم آن را افزایش دهیم، بدون اینکه عملکرد واقعی آن تغییر کند.

مزایا:

- افزایش حجم کد و گمراه کردن مهندس معکوس
- حفظ کامل رفتار برنامه
- تنوع پذیری بالا به دلیل تصادفی بودن ساختار و مقادیر

۳. تصادفی‌سازی عبارات معادل (Equivalent Expression Transformation)

در این متدهای با کلاس (ExpressionTransformer) پیاده‌سازی شد، از معادلهای منطقی و ریاضی برای بازنویسی برخی از عملیات‌ها استفاده می‌شود. به طور مثال:

- عبارت $a + b$ تبدیل به $(-b) - a$ می‌شود.
- عبارت $x == y$ تبدیل به $(x != y) != true$ می‌شود.
- عبارت $x != y$ تبدیل به $(x == y) != true$ می‌شود.

این تغییرات، با وجود حفظ معنای اصلی، تحلیل دستی کد را برای خواندن سخت‌تر می‌کند. همچنین در صورتی که ابزار تحلیل ایستا یا مهندسی معکوس به شکل ساده‌ای روی عبارات عمل کند، این تکنیک می‌تواند مانعی برای تحلیل صحیح باشد.

مزایا:

- بدون نیاز به بازنویسی ساختار یا فحوى کد
- حفظ عملکرد در عین تغییر شکل بیان ریاضی و منطقی
- قابلیت گسترش برای عملیات‌های بیشتر در آینده

۴. ادغام توابع (Function Inlining)

در این تکنیک که توسط (FunctionInliner) پیاده‌سازی شد، توابع ساده‌ای که تنها شامل یک `return` هستند شناسایی شده و عبارت بازگشتی آن‌ها مستقیماً در محل فراخوانی جایگزین می‌شود. برای مثال:

قبل:

```
int sum(int a, int b) { return a + b; }
int z = sum(3, 4);
```

بعد:

```
int z = (3) + (4);
```

این روش باعث می‌شود ساختار توابع از بین برود و کد در هم آمیخته‌تر شود. علاوه بر این، ارتباط معنایی بین تابع و محل استفاده از آن حذف می‌شود و همین موضوع درک روابط بین اجزای برنامه را دشوارتر می‌کند.

مزایا:

- کاهش قابلیت ردیابی منطق توابع
- افزایش پیچیدگی تحلیل برنامه برای انسان و ابزارها
- حذف توابع کوچک و ساده از سطح بالای برنامه

۵. اضافه کردن توابع تصادفی کاملاً بیهوده (Dummy Functions)

این متد با استفاده از کلاس (DummyFunctionInjector) پیاده‌سازی شد. در این تکنیک، یک یا چند تابع با بدنه‌ای پیچیده اما بی‌اثر به فایل نهایی برنامه اضافه می‌شود. این تابع شامل حلقه‌ها، شرط‌ها، متغیرهای تصادفی، و عملیات‌های محاسباتی بی‌معنا هستند.

این تابع معمولاً در هیچ جای برنامه فراخوانی نمی‌شوند، اما وجود آن‌ها باعث می‌شود حجم و پیچیدگی ظاهری کد بالا رود.

مزایا:

- افزودن نویز و اغتشاش به ساختار کلی برنامه
- فریب دادن ابزارهای تحلیل آماری یا امنیتی
- کاهش اثربخشی مهندسی معکوس از طریق افزایش تابع غیرضروری

۶. درهم‌سازی ساختار کنترلی (Control Flow Flattening)

با استفاده از کلاس (ControlFlowFlattener) پیاده‌سازی شد. در این روش، توالی اجرای دستورات تابع (و در صورت نیاز تابع دیگر) به صورت `case` هایی درون یک `switch` قرار می‌گیرد که خودش داخل `main` یک `while` بی‌نهایت است.

هر `case` معادل یک (statement) یا (declaration) است. متغیر `selector` مشخص می‌کند کدام بخش از کد اجرا شود و در پایان هر بخش، مقدار آن تغییر می‌کند تا بخش بعدی اجرا شود.

مثال ساده:

```
int selector = 1;
while(selector > 0) {
    switch(selector) {
        case 1: x = 5; selector = 2; break;
        case 2: y = x + 2; selector = 3; break;
        case 3: selector = 0; break;
    }
}
```

مزایا:

- حذف جواب کنترلی واضح
 - ایجاد ساختاری مبهم برای مسیر اجرای برنامه
 - مقاومسازی در برابر تحلیل جواب داده و کنترل در ابزارهای مهندسی معکوس
-

تمام متدها با استفاده از (ANTLR) و (TokenStreamRewriter) انجام شدند تا تغییرات عمیق و دقیق در کد ایجاد کنند بدون ایجاد (behavior) یا تغییر در رفتار یا (syntax error) برنامه.

بخش ۲) توجیح تصمیمات طراحی (به انتخاب، ۸ تصمیم طراحی توضیح داده شده)

تصمیم ۱: استفاده از TokenStreamRewriter به جای بازنویسی دستی کد

توضیح:

برای بازنویسی کد به جای بازسازی کامل AST یا تولید مجدد کد از صفر، از TokenStreamRewriter استفاده شد تا تغییرات موضعی و دقیق در توکن‌ها انجام شود.

دلایل طراحی:

- بازنویسی در سطح توکن بسیار دقیق و بدون شکستن ساختار کد است.
- تغییرات کوچک مانند جایگزینی متغیر، عبارت یا عبارت بازگشتی را آسان می‌کند.
- نیازی به تولید مجدد کامل کد از AST ندارد.

مزایا:

- پیاده‌سازی ساده‌تر و سریع‌تر تکنیک‌های Obfuscation.
- کاهش ریسک syntax error.
- حفظ ساختار اصلی برنامه.

تصمیم ۲: محدودسازی Control Flow Flattening فقط به تابع main

توضیح:

به جای اعمال Flattening به تمام توابع، فقط تابع main هدف قرار گرفت.

دلایل طراحی:

- تابع main اغلب شامل مسیر اصلی اجرای برنامه است.
- اعمال این تکنیک روی همهٔ توابع ممکن است ساختار را بیش از حد پیچیده و خراب کند.
- اجرای مطمئن‌تری دارد چون معمولاً در main متغیرهای گلوبال و IO کمتری هستند.

مزایا:

- جلوگیری از اضافه کاری بی‌مورد.
- حفظ کارایی برنامه.
- حفظ ساختار تابعی در سایر بخش‌ها برای توسعه‌پذیری بعدی.

تصمیم ۳: طراحی تکنیک inlining فقط برای توابع بسیار ساده با یک return

توضیح:

فقط توابعی که یک return دارند شناسایی شده و بدنده‌شان به جای فراخوانی درون خطی می‌شود.

دلایل طراحی:

- توابع پیچیده‌تر نیاز به جایگزینی متغیر، مدیریت scope، و گاه بازنویسی چند خط دارند.
- ساده‌سازی پیاده‌سازی با کمترین رسیک.

مزایا:

- دقیق‌تر در inlining بدون ایجاد خطای.
- پیاده‌سازی سریع و قابل اعتماد.
- مناسب برای فاز اول پروژه و گسترش پذیر برای مراحل بعدی.

تصمیم ۴: شناسایی و استفاده از متغیرهای تعریف شده برای درج کد مرد

توضیح:

در تکنیک Dead Code Insertion ، ابتدا تمام متغیرهای معرفی شده ذخیره می شوند و سپس در بلوک های مرده به کار می روند.

دلایل طراحی:

- استفاده از متغیر واقعی باعث می شود کد مرد واقعی تر به نظر برسد.
- کد جعلی کمتر قابل تشخیص و حذف توسط ابزارهای اتوماتیک باشد.

مزایا:

- افزایش مخفی سازی و واقع نمایی کد.
- شباهت بیشتر به کد کاربردی از نظر ابزارهای تحلیل.

تصمیم ۵: تولید تابع های جعلی با ساختارهای ترکیبی متنوع

توضیح:

در تکنیک Dummy Function ، توابعی با ساختار پیچیده شامل حلقه، شرط، محاسبات و متغیرهای درون بخشی تولید می شوند.

دلایل طراحی:

- ساده بودن تابع جعلی می تواند به راحتی توسط optimizer حذف شود.
- ترکیب ساختارهای واقعی کد باعث گمراه سازی بیشتر می شود.

مزایا:

- افزایش اثربخشی تکنیک در برابر حذف خودکار.
- افزایش بار تحلیل برای مهندسی معکوس.
- بالا رفتن entropy ظاهری پروژه.

تصمیم ۶: ذخیره‌سازی مپ نام‌ها برای حفظ انسجام در Rename

توضیح:

در تکنیک تغییر نام (Rename Obfuscation)، از یک دیکشنری به نام `name_map` استفاده شد تا هر بار که یک نام دیده می‌شود، همان نام جدید به جای آن جایگزین شود.

دلایل طراحی:

- اگر یک نام چندین بار در کد استفاده شده باشد (مثلًا متغیری که در چند خط به کار رفته)، باید در همه جا به یک نام جدید پرسان تغییر کند.
- اگر نکاشتی وجود نداشته باشد، انسجام برنامه از بین می‌رود و ممکن است کد دچار خطای semantic شود.

مزایا:

- جلوگیری از خطای عدم انطباق نام‌ها.
- اجرای مطمئن و بدون تغییر در منطق برنامه.
- افزایش انسجام و دقت در فرآیند بازنویسی توکن‌ها.

تصمیم ۷: پردازش مجذای expression در درخت نحوی برای بازنویسی دقیق

توضیح:

در تکنیک‌هایی مثل Expression Transformation و Rename، به جای بازنویسی کلی تابع یا بلاک، فقط گره‌های مربوط به هدف قرار گرفتند. `additiveExpr` یا `expression` به

دلایل طراحی:

- تغییر در گره‌های کوچک‌تر باعث می‌شود ساختار کلی برنامه دست‌فخورده باقی بماند.
- هدف‌گیری موضعی، ریسک syntax error را کاهش می‌دهد و عملکرد پروژه را قابل اطمینان‌تر می‌کند.

مزایا:

- افزایش دقت و کنترل در تغییرات نحوی.
- کاهش احتمال تخریب ساختار یا منطق برنامه.

- امکان بازنویسی‌های پیشرفته‌تر در آینده با توسعه‌ی گرامر.

تصمیم ۸: طراحی modular به‌گونه‌ای که بتوان تکنیک‌ها را ترکیب کرد

توضیح:

تمام تکنیک‌های Obfuscation به صورت کلاس‌های مستقل پیاده‌سازی شدند که به صورت جداگانه روی درخت نحوی اعمال می‌شوند. این ساختار به‌گونه‌ای است که بتوان چند تکنیک را پشت سر هم روی یک فایل اعمال کرد (pipeline processing).

دلایل طراحی:

- هر تکنیک اهداف خاص خود را دارد و ممکن است کاربر بخواهد ترکیبی از آن‌ها را اجرا کند.
- استقلال کلاس‌ها باعث می‌شود ترتیب اجرا مهم نباشد و هر تکنیک فقط تغییرات خود را اعمال کند.

مزایا:

- افزایش انتظاف‌پذیری در استفاده و تست.
- امکان اجرای selective یا progressive تکنیک‌ها.
- تسهیل توسعه ابزار GUI یا خط فرمان برای کنترل تکنیک‌های اعمال شده.

بخش (۳) قبل و بعد چند مثال واقعی

مورد ۱:

```
≡ input.mc
1 int main() {
2     int a = 3;
3     int b = 5;
4     int g = a + b;
5     printf("%d\n", a + b);
6     return 0;
7 }
8 |
```

OBFUSCATOR

Grammar

MiniC Obfuscator

Input File: D:/KNTU_STUDY/4032/Compiler/phase1/project/Obfuscator/run_mc.py

Output File: output.mc

MiniC Obfuscator Options:

- Rename Variables & Functions
- Inject Dead Code
- Control Flow Flattening
- Insert Dummy Functions
- Inline Functions
- Equivalent Expression Replacement

Run Obfuscator

ASTbuilder.py

nodes.py

input.mc

output.mc

```
1 int fEfWBf() {  
2     int Ok93aJir = 3;  
3     int Mbw = 5;  
4     int Wp2Jbce6B = Ok93aJir + Mbw;  
5     printf("%d\n", Ok93aJir + Mbw);  
6     while (0) {  
7         Wp2Jbce6B = 27;  
8     }  
9     bool CmVnn07C = false;  
10    bool NtmeKmx = true;  
11    return 0;  
12    while (0) {  
13        Ok93aJir = 54;  
14        Wp2Jbce6B = 46;  
15        Mbw = 58;  
16    }  
17    char waP = 'o';  
18}
```

مورد ۲:

The screenshot shows the MiniC Obfuscator application window. On the left, the 'input.mc' file is displayed with the following code:

```
1 int main() {
2     int a = 3;
3     int b = 5;
4     int g = a + b;
5     int z = b - a;
6     printf("%d\n", a + b);
7     return 0;
8 }
```

On the right, the 'Minic Obfuscator' window shows the following configuration:

- Input File: input.mc
- Output File: output.mc
- Checkboxes:
 - Rename Variables & Functions
 - Inject Dead Code
 - Control Flow Flattening
 - Insert Dummy Functions
 - Inline Functions
 - Equivalent Expression Replacement
- Run Obfuscator button

Output:

The screenshot shows the 'output.mc' file with the following obfuscated code:

```
1 int main() {
2     int selector_5663 = 1;
3     while (selector_5663 > 0) {
4         switch(selector_5663) {
5             case 1:
6                 int a = 3;
7                 selector_5663 = 2;
8                 break;
9             case 2:
10                int b = 5;
11                selector_5663 = 3;
12                break;
13             case 3:
14                 int g = a - (-b);
15                 selector_5663 = 4;
16                 break;
17             case 4:
18                 int z = b + (-a);
19                 selector_5663 = 5;
20                 break;
21             case 5:
22                 printf("%d\n", a - (-b));
23                 selector_5663 = 6;
24                 break;
25             case 6:
26                 return 0;
27                 selector_5663 = 7;
28                 break;
29             case 7:
30                 selector_5663 = 0;
31                 break;
32         }
33     }
34 }
```

On the right side of the screen, there is another block of obfuscated code:

```
35     int qdemmg() {
36         int x = 75;
37         int y = 12;
38         if (!(x%2 != 0)) {
39             int y = 1;
40             for (int i = 0; i < 3; i++) {
41                 y*=i - (-1);
42             }
43             x += y;
44         } else {
45             int temp = 0;
46             while (temp < 5) {
47                 x += temp;
48                 temp++;
49             }
50         }
51         int z = x - (-y);
52         return z*4;
53     }
54 }
```

مورد ۳:

The screenshot shows the MiniC Obfuscator application window. On the left, the 'Input File' is set to 'D:/KNTU_STUDY/4032/Compiler/phase1project/Obfuscator' and the 'Output File' is 'C:/Users/Lenovo/Desktop/output.mc'. A list of obfuscation options is visible, with 'Rename Variables & Functions', 'Insert Dummy Functions', and 'Inline Functions' checked. Below the options is a 'Run Obfuscator' button. To the right, the 'input.mc' file contains the following C-like pseudocode:

```
input.mc
1 int yek(int x) {
2     return x * 10;
3 }
4
5 int main() {
6     int a = 3;
7     int b = 5;
8     int g = a + b;
9     int z = yek(g);
10    printf("%d\n", a + b);
11    return 0;
12 }
13
```

Output:

```
C:\> Users > Lenovo > Desktop > output.mc
1 int IJ8wc9it7D(int ApW) {
2
3
4     return ApW * 10;
5 }
6
7 }
8
9 int MwlN7() {
10
11     int ApW = 84;
12
13
14     int chrw2bj = 30;
15
16
17     if (ApW % 2 == 0) {
18
19         int chrw2bj = 1;
20
21         for (int i = 0; i < 3; i++) {
22
23             chrw2bj *= i + 1;
24
25         }
26
27         ApW += chrw2bj;
28
29     } else {
30
31
32     }
33
34     ApW += chrw2bj;
35
36 } else {
37 }
```

```
37
38
39     int WaVXsR = 0;
40
41     while (WaVXsR < 5) {
42
43         ApW += WaVXsR;
44
45         WaVXsR++;
46
47     }
48
49
50
51     }
52
53
54
55
56
57     int QLU8zL = ApW + chrw2bj;
58
59
60     return QLU8zL*4;
61
62
63     }
64
65     int xGdyKndNyp() {
66
67         int v9Lr = 3;
68
69         int oFR6VFj = 5;
70
71         int tP8xYg = v9Lr + oFR6VFj;
72
73         int QLU8zL = (tP8xYg) * 10;
```

```

73     int QLU8zL = (tP8xYg) * 10;
74
75     printf("%d\n", v9Lr + oFR6VFj);
76
77     return 0;
78 }
79
80 int L0xFrWvLcy() {
81
82     int ApW = 9;
83
84     int chrw2bj = 75;
85
86     if (ApW % 2 == 0) {
87
88         int chrw2bj = 1;
89
90         for (int i = 0; i < 3; i++) {
91
92             chrw2bj *= i + 1;
93
94         }
95
96         ApW += chrw2bj;
97
98     } else {
99
100        int WaVXsR = 0;
101
102        while (WaVXsR < 5) {
103
104            ApW += WaVXsR;
105
106            WaVXsR++;
107
108        }
109    }
110
111
112     int QLU8zL = ApW + chrw2bj;
113
114
115     return QLU8zL*4;
116
117 }
```

:۴ مورد

The screenshot shows a terminal window on the left containing the source code for 'input.mc' and a graphical user interface for the 'MiniC Obfuscator' on the right.

Input File: input.mc

```

1 int first(int x) {
2     return x * 100;
3 }
4
5 int main() {
6     int a = 3;
7     int b = 5;
8     int g = a + b;
9     int z = b - a;
10    int p = first(b);
11    printf("%d\n", a + b);
12    return 0;
13 }
```

Output File: output.mc

Obfuscation Settings (checked):

- Rename Variables & Functions
- Inject Dead Code
- Control Flow Flattening
- Insert Dummy Functions
- Inline Functions
- Equivalent Expression Replacement

Buttons:

- Run Obfuscator
- Save As

Output:

```

output.mc
1 int tLB(int OUTBcy) {
2     return OUTBcy * 100;
3     while (0) {
4         int KKA = 94;
5     }
6     bool k1CUlo5R = true;
7     bool kPb = false;
8 }
9 int jIDVLf5() {
10    int OUTBcy = 93;
11    int Pj7a_ = 2;
12    if (!(OUTBcy%2 != 0)) {
13        int Pj7a_ = 1;
14        for (int i = 0; i < 3; i++) {
15            Pj7a_*=i - (-1);
16        }
17        OUTBcy += Pj7a_;
18    } else {
19        int smKUyxx = 0;
20        while (smKUyxx < 5) {
21            OUTBcy += smKUyxx;
22            smKUyxx++;
23        }
24    }
25    int DERoX3PR = OUTBcy - (-Pj7a_);
26    return DERoX3PR*4;
27 }
28 int main() {
29     int selector_2050 = 1;
30     while (selector_2050 > 0) {
31         switch(selector_2050) {
32             case 1:
33                 int Wjz3IEN = 3;
34                 selector_2050 = 2;
35                 break;
36             case 2:
37                 int bpf = 5;
38                 selector_2050 = 3;
39                 break;
36 case 2:
37     int bpf = 5;
38     selector_2050 = 3;
39     break;
36 case 3:
37     int jKxJq = Wjz3IEN - (-bpf);
38     selector_2050 = 4;
39     break;
36 case 4:
37     int DERoX3PR = bpf + (-Wjz3IEN);
38     selector_2050 = 5;
39     break;
36 case 5:
37     int Ghp = first(bpf);
38     selector_2050 = 6;
39     break;
36 case 6:
37     printf("%d\n", Wjz3IEN - (-bpf));
38     selector_2050 = 7;
39     break;
36 case 7:
37     if (0) {
38         jKxJq = 46;
39         Ghp = 53;
40     }
41     selector_2050 = 8;
42     break;
43 case 8:
44     char dm_2M = 'i';
45     selector_2050 = 9;
46     break;
47 case 9:
48     return 0;
49     selector_2050 = 10;
50     break;
51 case 10:
52     if (0) {
53         bpf = 60;
54     }
55     selector_2050 = 11;
56     break;
57 case 11:
58     int mmDzuGh = 56;
59     selector_2050 = 12;
60     break;
61 case 12:
62     selector_2050 = 0;
63     break;
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 int qDhnnrT() {
88     int OUTBcy = 86;
89     int Pj7a_ = 9;
90     if (!(OUTBcy%2 != 0)) {
91         int Pj7a_ = 1;
92         for (int i = 0; i < 3; i++) {
93             Pj7a_*=i - (-1);
94         }
95         OUTBcy += Pj7a_;
96     } else {
97         int smKUyxx = 0;
98         while (smKUyxx < 5) {
99             OUTBcy += smKUyxx;
100            smKUyxx++;
101        }
102    }
103    int DERoX3PR = OUTBcy - (-Pj7a_);
104    return DERoX3PR*4;
105 }

```

بخش ۴) چالش ها و نکات مهم

۱: حفظ رفتار اصلی برنامه (Semantic Preservation)

توضیح:

مهم‌ترین و پایه‌ای ترین چالش پروژه، اعمال تغییرات روی کد بدون ایجاد اختلال در عملکرد نهایی آن است. مثلاً اگر در تغییر نام یا جایگزینی عبارت‌ها کوچک‌ترین اشتباہی رخ دهد، ممکن است متغیر نادرستی جایگزین شود و منطق برنامه تغییر کند یا حتی کد به درستی کامپایل نشود.

راه حل:

- استفاده از `TokenStreamRewriter` برای حفظ دقت در جایگزینی توکن‌ها.
- استفاده از `name_map` برای جلوگیری از استفاده نام‌های ناسازگار.
- هدف‌گیری فقط گره‌هایی که بازنویسی‌شان کم خطرتر است، مثل `equalityExpr` یا `additiveExpr`.

۲: برخورد با ساختارهای پیچیده گرامی و AST

توضیح:

ساختار زبان Mini-C شامل ترکیب‌های نحوی متنوع مانند `statement`, `declaration`, `compoundStmt` و ... است. پیدا کردن دقیق گره‌های مربوطه در AST و تغییر دقیق آن‌ها بدون تأثیر بر گره‌های اطراف `iterationStmt` کار پیچیده‌ای است.

راه حل:

- تعریف گرامر Mini-C به صورت کامل با تدقیق دقیق سطوح مختلف AST.
- استفاده از `Flattening` در `children[1:-1]` برای حذف {} و پردازش دقیق محتوای بدنه توابع.
- استفاده از `Visitor Pattern` برای هدف‌گیری نوع خاصی از گره‌ها در درخت.

۳: جلوگیری از برخورد نام‌ها (Name Collisions) در Inlining و Rename

توضیح:

در Rename ممکن است نام تصادفی تولید شده با نامی موجود در برنامه (مثلاً کتابخانه‌ای یا تعریف‌نشده‌ای دیگر) تداخل داشته باشد. در Inlining هم ممکن است پارامترها با متغیرهای محلی در محل فراخوانی تداخل کنند و رفتار برنامه را تغییر دهند.

راه حل:

- بررسی اینکه نام جدید تولید شده با عدد شروع نشود و در لیست `used_names` نباشد.
- استفاده از پرانتز هنگام جایگزینی پارامترها در Inlining برای کنترل precedence در عبارات پیچیده.

۴: حفظ ترتیب و درستی اجرای دستورات در Control Flow Flattening

توضیح:

در Flattening، ترتیب اجرای دستورات اصلی باید با همان منطق قبلی حفظ شود، در حالی که ساختار آن به کلی تغییر کرده و به ساختار while-switch-case تبدیل شده است.

مشکلات:

- از بین رفتن مقداردهی اولیه متغیرها اگر در case اشتباہی قرار گیرند.
- عدم رسیدن به case پایانی و ورود به حلقه بی پایان.

راه حل:

- اختصاص متغیر selector با مقدار اولیه دقیق (۱) و گام به گام افزایش آن در انتهای هر case.
- افزودن selector جهت خروج از حلقه.

۵: موازن سطح پیچیدگی و حفظ کامپایل پذیری

توضیح:

افزایش بیش از حد پیچیدگی در عبارات، کدهای مرده، یا Flattening می‌تواند باعث شود که تولید شده دیگر به راحتی قابل کامپایل نباشد یا بیش از حد کند اجرا شود.

راه حل:

- محدودسازی inlining فقط به توابع بسیار ساده با یک return.
- عدم دست‌کاری ساختارهای کنترلی مانند if/for/while در سطح بالا.
- کنترل حجم dead code با استفاده از random محدود در ۱ تا ۳ خط.

۶: ترتیب اعمال تکنیک‌های مختلف و سازگاری آن‌ها

توضیح:

اعمال چند تکنیک مختلف روی یک کد ممکن است باعث تداخل بین آن‌ها شود. مثلاً اگر Rename بعد از Inlining اعمال شود، ممکن است متغیرهایی که از قابع وارد شده‌اند rename نشوند.

راه حل:

- طراحی مازوچی و pipeline-friendly برای تکنیک‌ها.