ObfusC Project Report

محمدعرفان خبرتی. محمد مهدی امیرپور

تکنیک‌های مبهم‌سازی (Obfuscation Techniques)

این سند تکنیک‌های مبهم‌سازی پیاده‌سازی شده دراین پروژه را توضیح می‌دهد. این سیستم با اعمال این تغییرات، کد منبع را برای خواندن و درک دشوارتر می‌کند، در حالی که عملکرد آن را حفظ می‌نماید.

درج کد مرده (Dead Code Insertion)

توضیحات

این تکنیک کدهایی که از نظر نحوی معتبر هستند اما از نظر عملکردی بی‌ربط هستند را به برنامه اضافه می‌کند. کد درج شده هرگز اجرا نمی‌شود یا تأثیری بر رفتار برنامه ندارد.

پیاده‌سازی

کلاس `DeadCodeInserter` نقاط بالقوه درج در کد را شناسایی کرده و موارد زیر را اضافه می‌کند:

- بلوک‌های شرطی غیرقابل دسترسی (مانند `if(false)` یا شرایط معادل)
- اعلان متغیرهای استفاده نشده
- حلقه‌های بی‌معنی که هرگز اجرا نمی‌شوند
- فراخوانی‌های تابع ساختگی بدون اثرات جانبی

مثال

کد اصلی:

```
int sum(int a, int b) {
    return a + b;
}
```

کد مبهم‌سازی شده:

```
int sum(int a, int b) {
    int _dummy = 42;
```

```
    if(_dummy > 100) { printf("Never reached"); }

    if(1) {
        int _unused = 0;
        while(_unused < 0) { _unused++; }
    }
    return a + b;
}
```

تغییر نام متغیرها (Variable Renaming)

توضیحات

این تکنیک نام‌های معنادار متغیرها را با شناسه‌های تصادفی یا گمراه‌کننده جایگزین می‌کند، درک هدف متغیرها را دشوار می‌سازد.

پیاده‌سازی

کلاس `Renamer` :

- یک جدول نماد برای تمام متغیرها ایجاد می‌کند
- نام‌های جایگزین تصادفی تولید می‌کند
- با جایگزینی تمام نمونه‌های هر متغیر، سازگاری را تضمین می‌کند
- قواعد محدوده را حفظ می‌کند

مثال

کد اصلی:

```
int calculateArea(int width, int height) {
    int area = width * height;
    return area;
}
```

کد مبهم‌سازی شده:

```
int calculateArea(int a_x25, int b_f34) {
    int c_e09 = a_x25 * b_f34;
    return c_e09;
}
```

بازنویسی عبارات (Expression Rewriting)

توضیحات

این تکنیک عبارات ساده را به عبارات پیچیده‌تر اما معادل تبدیل می‌کند، درک منطق را دشوارتر می‌سازد.

پیاده‌سازی

کلاس `ExpressionRewriter` قوانینی را برای تغییر عبارات با حفظ معناشناسی آنها اعمال می‌کند:

- عملیات ساده را به چندین مرحله تقسیم می‌کند
- هویت‌های جبری را اضافه می‌کند (مانند `x → x * 1` ، `x → x + 0`)
- عملیات‌های اضافی که یکدیگر را خنثی می‌کنند معرفی می‌کند
- شرط‌های ساده را با شرط‌های پیچیده معادل جایگزین می‌کند

مثال

کد اصلی:

```
int calculate(int x) {
    if (x > 10) {
        return x * 2;
    }
    return x;
}
```

کد مبهم‌سازی شده:

```
int calculate(int x) {
    if ((x - 0) > (10 + 0)) {
        return (x * 1) * (1 + 1);
    }
    return x + 0;
}
```

فرآیند اعمال

فرآیند مبهم‌سازی این تکنیک‌ها را به ترتیب اعمال می‌کند:

1. کد منبع با استفاده از ANTLR4 تجزیه می‌شود تا یک درخت نحوی انتزاعی (AST) ایجاد شود
2. هر مبهم‌ساز درخت AST را پیمایش کرده و تغییرات خود را اعمال می‌کند

3. درخت AST اصلاح شده به کد منبع تبدیل می‌شود
4. مبهم‌ساز بعدی روی کد مبهم‌شده قبلی عمل می‌کند

این رویکرد لایه‌ای کد را به طور پیشرونده‌ای دشوارتر برای درک می‌کند، زیرا هر تکنیک بر تغییرات تکنیک قبلی بنا می‌شود.

برنامه به کاربران اجازه می‌دهد تکنیک‌های فردی را انتخاب کنند یا همه آنها را برای حداکثر مبهم‌سازی اعمال کنند.

چالش‌های پروژه Obfusc2

تعامل متقابل بین تکنیک‌های مبهم‌سازی

چالش تداخل بین تغییر نام متغیرها و درج کد مرده

یکی از اصلی‌ترین چالش‌های این پروژه، تأثیر منفی تکنیک تغییر نام متغیرها بر تکنیک درج کد مرده بود. زمانی که ابتدا تکنیک تغییر نام اجرا می‌شد، متغیرهای درج شده توسط تکنیک کد مرده نیز تغییر نام پیدا می‌کردند و این باعث می‌شد ساختار کد مرده تغییر کند و در برخی موارد نامعتبر شود.

راه حل: ترتیب اجرای تکنیک‌ها را تغییر دادیم تا ابتدا کد مرده اضافه شود و سپس تغییر نام‌ها صورت گیرد. همچنین برای متغیرهای کد مرده از پیشوندهای مشخص (مانند `dummy_`) استفاده کردیم تا در مرحله تغییر نام قابل تشخیص باشند.

```
var obfuscators = List.<Obfuscator>of(
    DeadCodeInserter::insertDeadCode,  // اول اجرا می‌شود
    Renamer::renameVar,                // سپس این اجرا می‌شود
    ExpressionRewriter::rewriteExpressions
);
```

تعارض بین بازنویسی عبارات و سایر تکنیک‌ها

تکنیک بازنویسی عبارات می‌توانست ساختار عبارات را به‌گونه‌ای تغییر دهد که باعث ایجاد مشکل در کد مرده یا متغیرهای تغییر نام داده شده می‌شد.

راه حل: این تکنیک را به‌عنوان آخرین مرحله در زنجیره مبهم‌سازی قرار دادیم تا تداخلی با سایر تکنیک‌ها ایجاد نشود.

چالش‌های پارسر ANTLR

ساخت گرامر کامل برای زبان Minic

ایجاد یک گرامر ANTLR کامل که بتواند تمام ساختارهای زبان C یا حتی زیرمجموعه Minic را پشتیبانی کند، بسیار چالش‌برانگیز بود.

مشکلات اصلی:

- پشتیبانی از تمام انواع اعلان متغیرها (محلی و سراسری)
- مدیریت صحیح عبارات پیچیده و اولویت عملگرها
- پشتیبانی از ساختارهای کنترلی مختلف (if, while, for, switch)
- مدیریت فراخوانی توابع و پارامترهای آنها

راه حل: به جای پیاده‌سازی کامل گرامر C، تمرکز بر زیرمجموعه محدودتری از زبان (Minic) و افزودن تدریجی قابلیت‌ها بر اساس نیاز پروژه.

چالش‌های تکنیک درج کد مرده

تولید کد مرده معتبر

ایجاد کد مرده‌ای که از نظر نحوی و معنایی معتبر باشد اما هرگز اجرا نشود یا تأثیری در برنامه نداشته باشد، نیاز به دقت زیادی داشت.

راه حل: طراحی الگوهای مختلف کد مرده با شرایط ثابت و قابل پیش‌بینی (مانند `if(false)` یا معادل‌های آن) و اطمینان از اینکه این کدها در مکان‌های مناسب درج می‌شوند.

مدیریت محدوده (Scope)

اضافه کردن متغیرهای جدید در کد مرده می‌توانست باعث تداخل با متغیرهای موجود در محدوده اصلی برنامه شود.

راه حل: استفاده از نام‌های خاص با پیشوند مشخص برای متغیرهای کد مرده و اطمینان از تعریف آنها در بلوک‌های جداگانه برای جلوگیری از تداخل محدوده.

چالش‌های تکنیک تغییر نام متغیرها

حفظ ساختار محدوده (Scope Preservation)

تغییر نام متغیرها به‌گونه‌ای که ساختار محدوده آنها حفظ شود، چالش بزرگی بود. اگر دو متغیر هم‌نام در محدوده‌های مختلف وجود داشتند، باید به شکل متفاوتی تغییر نام می‌یافتند.

راه حل: پیاده‌سازی یک سیستم مدیریت محدوده دقیق که محدوده هر متغیر را بر اساس موقعیت آن در AST شناسایی و تفکیک می‌کند.

جلوگیری از تغییر نام توابع استاندارد

تغییر نام شناسه‌های توابع استاندارد مانند `printf` می‌توانست باعث غیرقابل استفاده شدن کد شود.

راه حل: ایجاد یک لیست سفید از توابع استاندارد که نباید تغییر نام پیدا کنند و بررسی هر شناسه در مقابل این لیست قبل از تغییر نام.

چالش‌های عملکردی

حفظ معنای اصلی برنامه

پس از اعمال تمام تکنیک‌های مبهم‌سازی، اطمینان از اینکه برنامه هنوز همان عملکرد اصلی را دارد، چالش مهمی بود.

راه حل: ایجاد تست‌های جامع برای مقایسه خروجی برنامه قبل و بعد از مبهم‌سازی و اطمینان از اینکه هر تکنیک مبهم‌سازی عملکرد برنامه را تغییر نمی‌دهد.

بهینه‌سازی زمان پردازش

اعمال متوالی چندین تکنیک مبهم‌سازی می‌توانست زمان پردازش را به‌شدت افزایش دهد، به‌ویژه برای فایل‌های بزرگ.

راه حل: بهینه‌سازی الگوریتم‌های مبهم‌سازی و استفاده از روش‌های کارآمد برای تجزیه و تحلیل AST.

نتیجه‌گیری

تلفیق تکنیک‌های مختلف مبهم‌سازی برای ایجاد یک ابزار یکپارچه، نیازمند توجه دقیق به تعاملات متقابل این تکنیک‌ها و تأثیرات آنها بر یکدیگر است. ترتیب اعمال تکنیک‌ها، مدیریت محدوده متغیرها و حفظ کارکرد اصلی برنامه، سه چالش اصلی در این پروژه بودند که با طراحی دقیق و آزمایش مستمر برطرف شدند.

نمونه‌های مبهم‌سازی کد

این مستند شامل نمونه‌هایی از کدهای اصلی و نسخه‌های مبهم‌سازی شده آن‌ها با استفاده از ابزار Obfusc2 است. در هر مثال، تکنیک‌های مختلف مبهم‌سازی نشان داده شده‌اند.

مثال ۱: تابع جمع ساده

کد اصلی

```c
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;
}
```

کد مبهم‌سازی شده

```c
#include <stdio.h>
int _dummy1;
_dummy1 = 0;
int _dummy2;
_dummy2 = 0;
int _dummy3;
_dummy3 = 0;
int _dummy4;
_dummy4 = 0;
int _dummy5;
_dummy5 = 0;
int _dummy6;
_dummy6 = 0;
int _dummy7;
_dummy7 = 0;
int _dummy8;
_dummy8 = 0;
int _dummy9;
_dummy9 = 0;
int _dummy10;
_dummy10 = 0;
int sum(int a, int b) {
    while(_dummy6 < 94) { _dummy6 = _dummy6 + 1; }

    if(_dummy2 > 50) { _dummy2 = 49; } else { _dummy2 = 44; }

    int result = a + b;
    return result;
}
int main() {
    while(_dummy8 < 48) { _dummy8 = _dummy8 + 1; }

    int _dummy1;
    _dummy1 = 3;
```

```
    int x = 3;
    int y = 4;
    int total = sum(x, y);
    printf("%d\n", total);
    return 0;
}
```

تکنیک‌های استفاده شده

- **درج کد مرده:** اضافه کردن متغیرهای `dummy1_` تا `dummy10_` و دستورات شرطی و حلقه‌ای که هیچ تأثیری بر کد اصلی ندارند
- در این مثال، نام متغیرهای اصلی تغییر نکرده‌اند (تکنیک تغییر نام اعمال نشده است)

مثال ۲: برنامه پیچیده‌تر با چندین تابع

کد اصلی

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

double calculateAverage(int count) {
    double sum = 0.0;
    int i = 0;

    printf("Counting from zero to the number: \n");
    while (i < count) {
        printf("%d\n",i);
        sum = sum + i;
        i = i + 1;
    }

    if (count > 0) {
        return sum / count;
    } else {
        return 0.0;
    }
}

int isPrime(int num) {
    if (num <= 1) {
        return 0;
    }
```

```
    int i = 2;
    while (i * i <= num) {
        if (num % i == 0) {
            return 0;
        }
        i = i + 1;
    }

    return 1;
}

int main() {
    int i = 5;
    printf("Number is %d\n",i);
    int number = i;
    int fact = factorial(number);
    printf("Its factorial is %d\n",fact);

    int isPrimeNum = isPrime(number);
    printf("Is it prime? ");
    if (isPrimeNum) {
        printf("yes\r\n");
    } else {
        printf("no\r\n");
    }

    double avg = calculateAverage(i);
    printf("Its average: ");
    printf("%f ",avg);

    return 0;
}
```

کد مبهم‌سازی شده

```
#include <stdio.h>
int _vmxvovfse = 0;
int _vahwgthxh = 0;
int _vcxuuuiqa = 0;
int _vqrinmbuv = 0;
int _viosrynym = 0;
int _vyudrmnok = 0;
int _vojsehoqg = 0;
int _vcjtsnpke = 0;
int _vlogkvvpr = 0;
int _vkdvmsabc = 0;
int _vpqzalgoj(int n) {
    while(_vcxuuuiqa < 36) { _vcxuuuiqa = (_vcxuuuiqa - ((1 * -1))); }
```

```c
    while(_vlogkvvpr < 10) { _vlogkvvpr = (_vlogkvvpr - ((1 * -1))); }

    if (n <= 1) {
        return 1;
    } else {
        return n * _vpqzalgoj(n - 1);
    }
}

double _vchvmwlnf(int count) {
    while(_vcjtsnpke < 63) { _vcjtsnpke = ((_vcjtsnpke * 3) - (-1 * 3)) / 3; }

    while(_vojsehoqg < 31) { _vojsehoqg = (_vojsehoqg - (-1)); }

    double _vabrxgedt = 0.0;
    int _vxhnredqc = 0;

    printf("Counting from zero to the number: \n");
    while (_vxhnredqc < count) {
    while(_vcjtsnpke < 78) { _vcjtsnpke = ((_vcjtsnpke * 2) - (-1 * 2)) / 2; }

    _vlogkvvpr = 50;

    if(_vcxuuuiqa > 23) { _vcxuuuiqa = 74; } else { _vcxuuuiqa = 6; }

    while(_vlogkvvpr < 32) { _vlogkvvpr = (_vlogkvvpr - ((1 * -1))); }

    _vahwgthxh = 76;

        printf("%d\n",_vxhnredqc);
        _vabrxgedt = ((_vabrxgedt * 2) - (-_vxhnredqc * 2)) / 2;
        _vxhnredqc = ((_vxhnredqc * 3) - (-1 * 3)) / 3;
    }

    if (count > 0) {
        return _vabrxgedt / count;
    } else {
        return 0.0;
    }
}

int _vdstmhyoq(int num) {
    _viosrynym = 76;

    _vyudrmnok = 56;

    if (num <= 1) {
        return 0;
    }

    int _vxhnredqc = 2;
    while (_vxhnredqc * _vxhnredqc <= num) {
```

```c
    while(_vyudrmnok < 76) { _vyudrmnok = ((_vyudrmnok * 2) - (-1 * 2)) / 2; }

    _vcxuuuiqa = 54;

        if (num % _vxhnredqc == 0) {
            return 0;
        }
        _vxhnredqc = ((_vxhnredqc * 3) - (-1 * 3)) / 3;
    }

    return 1;
}

int main() {
    _vcxuuuiqa = 7;

    int _vxhnredqc = 5;
    printf("Number is %d\n",_vxhnredqc);
    int _vvdtfjpqz = _vxhnredqc;
    int _vyimrddet = _vpqzalgoj(_vvdtfjpqz);
    printf("Its factorial is %d\n",_vyimrddet);

    int _vfncmffqy = _vdstmhyoq(_vvdtfjpqz);
    printf("Is it prime? ");
    if (_vfncmffqy) {
    _vkdvmsabc = 25;

    _vahwgthxh = 50;

    _vahwgthxh = 57;

    _vqrinmbuv = 52;

    _vlogkvvpr = 88;

    if(_vyudrmnok > 97) { _vyudrmnok = 45; } else { _vyudrmnok = 15; }

    if(_viosrynym > 38) { _viosrynym = 92; } else { _viosrynym = 45; }

    while(_vqrinmbuv < 51) { _vqrinmbuv = ((_vqrinmbuv * 2) - (-1 * 2)) / 2; }

        printf("yes\r\n");
    } else {
        printf("no\r\n");
    }

    double _vvabzwaku = _vchvmwlnf(_vxhnredqc);
    printf("Its average: ");
    printf("%f ",_vvabzwaku);

    return 0;
}
```

تکنیک‌های استفاده شده

1. **درج کد مرده:**
   - متغیرهای سراسری بی‌استفاده (`_vahwgthxh , vmxvovfse, _` و غیره)
   - حلقه‌های `while` و دستورات شرطی `if` که تأثیری بر عملکرد اصلی ندارند
   - عملیات‌های ریاضی پیچیده که معادل عملیات‌های ساده هستند
2. **تغییر نام متغیرها:**
   - نام تابع‌ها به شناسه‌های نامفهوم تغییر یافته‌اند:
     - `factorial → _vpqzalgoj`
     - `calculateAverage → _vchvmwlnf`
     - `isPrime → _vdstmhyoq`
   - متغیرهای محلی با نام‌های تصادفی و گمراه‌کننده جایگزین شده‌اند:
     - `sum → _vabrxgedt`
     - `i → _vxhnredqc`
     - و غیره
3. **بازنویسی عبارات:**
   - عبارات ساده به عبارات پیچیده تبدیل شده‌اند، مانند:
     - `i = i + 1 → _vxhnredqc = ((_vxhnredqc * 3) - (-1 * 3)) / 3`
     - `sum = sum + i → _vabrxgedt = ((_vabrxgedt * 2) - (-_vxhnredqc * 2)) / 2`
     - `vojsehoqg = _vojsehoqg + 1 → _vojsehoqg = (_vojsehoqg - _(-1))`

تأثیر مبهم‌سازی

مقایسه این نمونه‌ها نشان می‌دهد که کد مبهم‌سازی شده، با حفظ عملکرد اصلی، بسیار پیچیده‌تر و دشوارتر برای فهم است:

1. **خوانایی:** نام‌های متغیرها دیگر معنادار نیستند و درک هدف هر متغیر را دشوار می‌کنند
2. **پیچیدگی:** عبارات ساده به عبارات پیچیده تبدیل شده‌اند که درک آنها نیاز به تجزیه و تحلیل بیشتری دارد
3. **حجم کد:** کد مبهم‌سازی شده بسیار طولانی‌تر است و بخش‌های زیادی از کد هیچ عملکردی ندارند
4. **جریان کنترل:** حلقه‌ها و دستورات شرطی اضافی، درک جریان کنترل برنامه را دشوار می‌کنند

با این حال، برنامه مبهم‌سازی شده دقیقاً همان خروجی برنامه اصلی را تولید می‌کند.