گزارش پروژه مبهم سازی زبان Mini-c

دانشگاه : دانشگاه صنعتی خواجه نصیر الدین طوسی

درس: اصول طراحی کامپایلر

نام اعضای گروه: پرنیان بهرامی، سمیرا شفیعی

نام استاد: دکتر محمدهادی

مقدمه

هدف این پروژه طراحی و پیاده سازی یک مبهم ساز برای زبان mini-c است . ما در این پروژه از antlr استفاده کردیم برای ساخت گرامر زبان mini-c تا بتوانیم درخت نحو را ایجاد کنیم . هم چنین زبان برنامه نویسی مورد استفاده ما python است . از نرم افزار graphviz هم برای ایجاد درخت به صورت گرافیکی استفاده کردیم .

روند اجرای پروژهMiniC Obfuscator

طراحی گرامر

در ابتدا برای زبان minic یک گرامر طراحی می کنیم و در فایل minic.g4 قرار دادیم .

گرامر به صورت سلسله‌مراتبی نوشته شده و از سطح بالای برنامه شروع شده و به اجزای کوچکتر مانند عبارات، شناسه‌ها و توکن‌ها می‌رسد. قابلیت‌هایی مانند استفاده از توابع، ورودی/خروجی (scanf,printf) و عملیات رویstruct ها نیز در آن گنجانده شده است. این گرامر به گونه‌ای طراحی شده که بتواند یک زبان ساده ولی کاربردی را پوشش دهد و برای آموزش یا تحلیل نحوی مناسب باشد.

پیش از شروع فرآیند اصلی مبهم‌سازی، لازم بود یک‌سری عملیات پیش‌پردازشی روی کد ورودی انجام گیرد. این مراحل در فایل test.pyپیاده‌سازی شده و پایه اجرای ابزار ما را شکل می‌دهند.

۱ .خواندن فایل منبع

در گام اول، فایل منبع با پسوند mc.که شامل کد ورودی است، خوانده می‌شود. این فایل به‌عنوان ورودی برای مراحل بعدی پردازش می‌گردد.

۲ .تحلیل واژگانی(Lexer)

با بهره‌گیری از ابزار ANTLR و گرامری که در فایل MiniC.g۴تعریف شده است، Lexerکد ورودی را به لیستی از توکن‌ها تجزیه می کند . این توکن‌ها شامل کلمات کلیدی (مثل (return، شناسه‌ها، اعداد، عملگرها و سایر اجزای زبان MiniC هستند.

۳ .تحلیل نحوی(Parser)

پس از تولید توکن‌ها، گام بعدی ساخت درخت نحوی یا Parse Tree با استفاده از Parser است. این درخت ساختار نحوی برنامه را با توجه به قواعد زبان MiniC نشان می‌دهد و مبنایی برای پیاده‌سازی تکنیک‌های بازنویسی کد فراهم می‌کند. اگر کد ورودی بر اساس قواعد گرامری زبان نباشد درخت نحو ایجاد نمی شود

۴ .نمایش متنی درخت نحوی

ابتدا درخت نحوی به‌صورت متنی در خروجی ترمینال چاپ می‌شود.

۵ .تولید نمایش گرافیکی از درخت

نسخه‌ای گرافیکی از درخت نحو نیز تولید می‌شود. بدین منظور:

- ابتدا نمایش متنی درخت به فرمت dot.که قابل فهم برای Graphviz است، ذخیره می‌شود.
- سپس این فایل با استفاده از دستورهای Graphviz به فرمت تصویری نظیر png.یا pdf.تبدیل می‌شود.
- تمام این فرآیند از طریق توابع تعریف‌شده در test.pyقابل اجرا است و به‌صورت خودکار صورت می‌گیرد.

نتیجه نهایی

در پایان این مراحل کد اولیه در قالب درخت نحوی در اختیار داریم که آماده انجام تکنیک‌های متنوع مبهم‌سازی بر روی آن است. این مراحل پایه‌ای، تضمین می‌کنند که ابزار ما عملکرد دقیق و قابل اطمینانی در پردازش کد ورودی دارد.

تکنیک‌های اعمال‌شده

در بخش بعدی پروژه به دنبال اعمال تکنیک هایی هستیم تا خوانایی و درک منطقی کد برای افراد غیر مجاز کاهش پیدا کند . تکنیک رایج برای مبهم‌سازی کد منبع زبان MiniC پیاده‌سازی شده است.

تغییر نام شناسه‌ها ۱.

در این تکنیک، هدف جایگزینی اسامی اصلی توابع، متغیرها و سایر شناسه‌ها با نام‌های تصادفی یا غیرمعنادار است.
:مراحل به صورت زیر پیاده‌سازی شده‌اند

در فایل ObfuscatorVisitor.pyکلاسی با نام ObfuscatorVisitorتعریف شده که از کلاس پایه MiniCVisitorارث‌بری می‌کند.

در این کلاس، تمام محل‌هایی که شناسه‌ها در کد ظاهر شده‌اند (تعریف و استفاده)، شناسایی می‌شود.

برای هر شناسه، یک نام جایگزین تولید و در سراسر کد اعمال می‌شود.

یک دیکشنری نگاشت (Mapping Dictionary) بین نام اصلی و نام جدید ایجاد می‌شود تا انسجام در کل کد حفظ گردد.

این نگاشت در سایر مراحل (مثلاً در بازنویسی عبارات) نیز مورد استفاده قرار می‌گیرد.

```
int fn_1(int var_1,
int var_2) {

int var_3 = var_1 +
var_2;

    return var_3;}
```

```
int sum(int a, int b) {

   int result = a + b;

   return result;}
```

۲. بازنویسی عبارات

این تکنیک با هدف سخت‌تر کردن درک منطق برنامه بدون تغییر در خروجی آن اجرا می‌شود. در فایل obfuscate_to_file.py، تابعی با نام replace_expression_pattern وظیفه بازنویسی عبارات را برعهده دارد.

ورودی‌های تابع:

- tokens: لیستی از توکن‌های کد
- i: ایندکس فعلی بررسی
- name_map : نگاشت شناسه‌ها برای جایگزینی نام‌ها

مراحل اجرا:

۱. ابتدا بررسی می‌شود که آیا حداقل سه توکن بعد از موقعیت فعلی وجود دارد یا خیر. در صورت نبود، تابع با مقدار None و گام پیشروی ۱ خاتمه می‌یابد.

۲. اگر الگوی سه‌تایی Operand۱ Operator Operand۲ شناسایی شود (با شرایط زیر):

- Operands از نوع Identifier یا IntegerConstant باشند.
- Operator از میان عملگرهای +, -, *, /, ==, &&, || باشد.

۳. در صورت وجود شناسه‌ها در name_map، با نام‌های جدید جایگزین می‌شوند.

۴. بسته به نوع عملگر، بازنویسی انجام می‌گیرد

```
int check(int x, int y) {
    if (x == y || x > 0) {
        return 1;
    }
    return 0;
}
```

```
int check(int x, int y) {
    if (!(x != y) || !(x <= -1)) {
        return 1;
    }
    return 0;
}
```

3 . درج کد مرده

کد مرده، بخشی از کد است که اجرا نمی‌شود یا تأثیری در خروجی ندارد اما با افزودن آن، خوانایی و تحلیل کد برای مهاجم دشوارتر می‌شود.

نحوه پیاده‌سازی:

- ابتدا یک تابع با نام get_random_dead_code() برای تولید نمونه‌های کد مرده نوشته شده است.
- این کدها در مکان‌هایی از برنامه قرار داده می‌شوند، از جمله:
  - ابتدای بلاک‌ها پس از {
  - پس از هر دستور، با احتمال ۳۰٪ یک کد مرده درج می‌شود.

```
int main() {
    int a = 5;
    int b = 10;
    int c = a + b;
    return c;
}}
```

```c
int main() {
    if (0) { printf("Debugging...\n"); }
    int dead1 = 12345;


    int a = 5;
    if (0) { int unused = 999; }
    int b = 10;


    int c = a + b;
```

نحوه اجرا

:برای اجرای ابزار می‌توان از خط فرمان استفاده کرد .ساختار کلی اجرای پروژه به صورت زیر است

python obfuscator.py --input test.c --output out.c --techniques rename,deadcode,expr

بخش امتیازی

پیاده سازی cli

برای اینکه بتوانیم با زدن دستور در خط فرمان، پروژه را اجرا کنیم :

ابتدا باید با استفاده از `import argparse`در فایل اصلی (main) تعریف‌های لازم را انجام دهیم تا خط فرمان را پردازش کند.

آرگومان‌های ورودی در خط فرمان به شکل زیر تعریف می‌شوند:

input-- بعد از این گزینه، نام فایل ورودی قرار می‌گیرد.

`output-- - بعد از این گزینه، نام فایل خروجی مشخص می‌شود.

`techniques-- - لیست تکنیک‌های مبهم‌سازی که می‌خواهیم اعمال شوند.

-تکنیک‌ها که به‌صورت رشته هستند، باید به لیست تبدیل شوند تا بتوان روی آن‌ها پردازش انجام داد.

-در نهایت تابع `obfuscate_and_save`را بازنویسی می‌کنیم تا:

- پارامترهای ورودی و خروجی را به صورت آرگومان دریافت کند.
- تکنیک‌های انتخاب‌شده را بررسی کرده و منطق اجرا را بر اساس آن‌ها کنترل کند.

پشتیبانی از structوpointer

گرامر این زبان را که در فایل minic.g4 قرار دارد به نحوی تعریف شده که از این دو مورد هم پشتیبانی میکند .

برای تست فایل وردی testsp.mc را تعریف کردیم در داخل ان کدی قرار دادیم که این دو مورد را استفاده میکند خروجی را با ران کردن برنامه می تونیم ببینیم و خطایی نداریم پس ساختار درست است .

در گرامر داخل typeSpecifier این تایپ structرا تعریف کردیم و هم چنین قاعده structDefinition داریم .

گرامر قاعده برای اشاره گر ها دارد و در جاهایی مثل declaration , parameter استفاده شده است .

دسترسی به اعضای struct , pointer هم پشتیبانی می شود از طریق postfixexpression

p.name برای struct معمولی و p->name برای اشاره گر به struct

توجیه معادل بودن عملکردی

با وجود تغییرات اعمال‌شده توسط تکنیک‌های مبهم‌سازی، کد حاصل همچنان **از نظر عملکردی معادل کد اولیه است** .به بیان دیگر، برای هر ورودی یکسان، خروجی برنامه‌ی اصلی و نسخه‌ی مبهم‌شده کاملاً برابر است. این موضوع از چند جنبه قابل توجیه است:

۱.در تکنیک تغییر نام شناسه‌ها:(Rename Identifiers)

تنها نام متغیرها یا توابع عوض شده‌اند، ولی نگاشت آن‌ها حفظ شده و هیچ تغییری در منطق یا جریان داده‌ها ایجاد نشده است.

۲.در تکنیک بازنویسی عبارات:(Expression Replacement)

جایگزینی‌هایی مانند a + b → a - (-b) یا a == b → ! (a != b) همگی بر اساس قواعد جبری و منطقی معتبر هستند و **ارزش نهایی عبارت را تغییر نمی‌دهند.**

3.در تکنیک درج کد مرده:(Dead Code Insertion)

کدهای اضافه‌شده مانند if(0){...} یا متغیرهای استفاده‌نشده هیچ تاثیری بر مسیر اجرای برنامه ندارند و تنها خوانایی را کاهش داده‌اند.

چالش های فنی

۱. حساسیت گرامر antlr به خطا های نحوی

Antlr برای ساخت درخت نحو از parser استفاده میکند در صورتی که ورودی طبق گرامر باشد اگر طبق قوانین mini-c اشتباه باشد parser درخت تولید نمی کند .

۲. مدیریت فاصله بین توکن ها

زمانی که بعد از مبهم سازی دوباره کد را بازنویسی می کنیم باید حواسمان باشد بین یک سری موارد حتما فاصله وجود داشته باشد مثلا بین int , x . یعنی باید هوشمندانه تشخیص بدهیم کجاها فاصله باشد و کجاها نه

۳. پس از مبهم سازی به شیوه پیچیده کردن عملیات . نتیجه عملیات ها نباید تغییر کند .

یعنی باید طوری عملیات را پیچیده کنیم که با ورودی یکسان با حالت ساده خروجی یکسانی بدهد .

۴. باید مکان مناسب قرارگیری کد ها ی مرده را پیدا کنیم .

اگر مکان قرارگیری کد مرده به درستی انتخاب نشود کد قابل کامپایل نیست

۵. در هنگام رسم گرافیکی درخت با نرم افزار اگر بخواهیم باهمون فرمت string دچار مشکل می شویم .

باید به حالت dot. اول تبدیل کنیم بعد ان را با نرم افزار رسم کنیم

۶. باید در نوشتن کد obfuscatorVistor.py دقت کنیم یعنی فقط متغییر های تعریف شده را تغییر بدهیم

اگر به درستی اسم های جدید نگاشت نشوند اجرای کد مختل می شود

۷. در طراحی گرامر باید به نحوی باشد که ترتیب عملگر ها به درستی انجام شود .

تا عبارت های عملیاتی به درستی تجزیه شوند

۸. تفکیک نقش توکن های چند منظوره :

مثلا * هم می تواند اشاره گر باشد و هم ضرب پس باید در تجزیه این دو مفهوم تفکیک شوند .antlr این را با توجه به موقعیت توکن در درخت نحوی تفسیر می کند .

۹. حذف فضای خالی و کامنت ها

نباید فضای خالی یا کامنت ها باعث اختلال در تجزیه شوند .

۱۰. تعریف توکن های اصلی و جلوگیری از تداخل

نباید Identifierبا Keywordاشتباه شود، یا true/falseبا متغیرها قاطی شوند. برای جلوگیری از این اتفاق در گرامر BooleanConstantرا قبل از Identifierتعریف کردید تا ابهام ایجاد نشود .