به نام خدا

دانشکده مهندسی کامپیوتر

فاز اول پروژه درس اصول طراحی کامپایلر

طراحی و پیاده‌سازی مبهم‌ساز کد برای زبان Mini-C

محمد مهاجری

آرش برخورداریون محمدی

آرین دلیری

استاد:

دکتر علائیان

بهار ۱۴۰۴

فهرست مطالب

۱-مقدمه

مبهم‌سازی کد فرآیندی است که در آن کد منبع به گونه‌ای تغییر داده می‌شود که درک آن برای انسان دشوارتر گردد، بدون آنکه عملکرد اصلی برنامه تحت تأثیر قرار گیرد. این تکنیک کاربردهای متنوعی در زمینه امنیت نرم‌افزار، حفاظت از مالکیت معنوی و جلوگیری از مهندسی معکوس دارد. در این پروژه، یک ابزار مبهم‌ساز برای زیرمجموعه‌ای از زبان C موسوم به Mini-C طراحی و پیاده‌سازی شده است. هدف اصلی، تولید کدی است که از نظر عملکردی با کد اصلی یکسان بوده اما خوانایی آن برای انسان به مراتب کمتر باشد. این گزارش به تشریح جزئیات پیاده‌سازی، تکنیک‌های به کار رفته، تصمیمات طراحی و چالش‌های مواجه شده در طول انجام پروژه می‌پردازد.

۲-مروری بر زبان Mini-C

زبان Mini-C که در این پروژه مورد استفاده قرار گرفته است، زیرمجموعه‌ای ساده شده از زبان C است و شامل موارد زیر می‌باشد:

- انواع داده پایه: bool، char، int
- متغیرها و عملگرها: عملگرهای محاسباتی (+, -, *, /, %)، رابطه‌ای (<, <=, >, >=, ==, !=) و منطقی (&&, ||, !).
- کنترل جریان: دستورات if-else، while، for و return.
- توابع: تعریف توابع با پارامترهای ورودی و مقدار بازگشتی.
- ورودی/خروجی: از طریق توابع استاندارد (مانند printf و scanf که به عنوان توابع با نام خاص در نظر گرفته شده‌اند).

محدودیت‌ها: این زبان فاقد ساختار (struct)، اشاره‌گر (pointer)، آرایه‌های پیچیده و سایر ویژگی‌های پیشرفته زبان C است.

این محدودیت‌ها به منظور ساده‌سازی فرآیند تحلیل و پیاده‌سازی مبهم‌ساز در نظر گرفته شده‌اند.

۳-معماری کلی سیستم مبهم‌ساز

سیستم مبهم‌ساز پیاده‌سازی شده از چندین مرحله اصلی تشکیل شده است که در ادامه تشریح می‌شوند:

۳-۱- تحلیل لغوی و نحوی (Parsing)

- در این مرحله، کد ورودی Mini-C (فایل با پسوند .mc) خوانده شده و ابتدا به توکن‌های لغوی شکسته می‌شود (Lexing).
- سپس، این توکن‌ها توسط یک تحلیل‌گر نحوی (Parser) بررسی شده تا ساختار گرامری کد تأیید شود.

برای این منظور، از ابزار ANTLR (ANother Tool for Language Recognition) استفاده شده است. گرامر زبان Mini-C در قالب فایل MiniC.g4 تعریف و توسط ANTLR، کد پایتون مربوط به تحلیل‌گر لغوی و نحوی تولید شده است.

۳-۲- ساخت درخت نحو انتزاعی (Abstract Syntax Tree- AST)

- خروجی تحلیل‌گر نحوی ANTLR یک درخت تجزیه (Parse Tree) است.

یک کلاس Visitor سفارشی (ASTBuilderVisitor) این درخت تجزیه را پیمایش کرده و یک درخت نحو انتزاعی (AST) سفارشی تولید می‌کند. ساختار گره‌های این AST سفارشی در فایل ast_nodes.py تعریف شده‌اند. این AST سفارشی برای اعمال تغییرات و مبهم‌سازی مناسب‌تر است.

۳-۳- اعمال پاس‌های مبهم‌سازی (Obfuscation Passes)

- پس از ساخت AST سفارشی، یک یا چند پاس مبهم‌سازی بر روی آن اعمال می‌شود. هر پاس، یک تکنیک خاص مبهم‌سازی را پیاده‌سازی می‌کند.

این پاس‌ها AST را پیمایش کرده و گره‌های آن را به منظور کاهش خوانایی کد تغییر می‌دهند، در حالی که معادل بودن عملکردی برنامه حفظ می‌شود. این بخش در obfuscator_passes.py پیاده‌سازی شده است.

۳-۴-تولید کد خروجی (Code Generation / Unparsing)

- در نهایت، پس از اعمال تمامی پاس‌های مبهم‌سازی، AST تغییریافته توسط یک مولد کد (CodeGenerator) پیمایش می‌شود.

این مولد، کد Mini-C جدید و مبهم‌شده را بر اساس ساختار AST نهایی تولید کرده و در فایل خروجی (مثلاً output.mc) ذخیره می‌کند.

۴- تکنیک‌های مبهم‌سازی پیاده‌سازی شده

در این پروژه، تکنیک‌های مبهم‌سازی زیر پیاده‌سازی و اعمال شده‌اند:

۴-۱-تغییر نام شناساگرها(Identifier Renaming)

در این تکنیک، تمامی نام‌های شناساگرهای تعریف شده توسط کاربر شامل نام متغیرهای محلی، نام پارامترهای توابع و نام توابع) به جز توابع خاص مانند (main, printf, scanf با نام‌های بی‌معنی و تولید شده به صورت خودکار (مثلاًobf_1, var_temp_a, f_002) جایگزین می‌شوند. این فرآیند با حفظ کامل قلمرو (scope) شناساگرها انجام می‌شود تا از تداخل نام و تغییر در عملکرد برنامه جلوگیری شود.

این پاس در دو مرحله بر روی AST عمل می‌کند:

- فاز تعریفAST (Definition Phase): پیمایش شده و تمامی تعاریف شناساگرها (توابع، پارامترها، متغیرهای محلی) شناسایی می‌شوند. برای هر شناساگر اصلی، یک نام مبهم جدید تولید شده و در جداول نماد (Symbol Tables) مناسب (یک جدول برای نام‌های سراسری مانند توابع و جداول جداگANE برای قلمروهای محلی هر تابع) نگاشت می‌شود.
- فاز اعمال AST (Application Phase): مجدداً پیمایش شده و هرگونه استفاده از شناساگرها و همچنین تعریف آن‌ها، با نام مبهم معادل از جداول نماد جایگزین می‌شود. دقت در مدیریت قلمروها برای جلوگیری از تغییر نام اشتباه شناساگرها حیاتی است.

۴-۲- درج کد مرده(Dead Code Insertion)

کد مرده، بخشی از کد است که از نظر نحوی صحیح بوده اما اجرای آن هیچ تأثیر معناداری بر خروجی نهایی برنامه ندارد یا نتایج آن هرگز مورد استفاده قرار نمی‌گیرد. نمونه‌هایی از کد مرده شامل تعریف متغیرهایی است که هرگز استفاده نمی‌شوند، یا تخصیص مقادیری به متغیرها که بلافاصله بازنویسی می‌شوند، یا عبارات محاسباتی که نتیجه آن‌ها ذخیره یا استفاده نمی‌شود.

این پاس AST را پیمایش می‌کند. در نقاط مناسبی از کد، معمولاً درون بلاک‌های دستور (BlockNode) و پس از دستورات موجود (به جز دستورreturn)، به صورت تصادفی (با یک احتمال مشخص) گره‌های AST جدیدی که نماینده کد مرده هستند، درج می‌شوند. کد مرده تولیدی معمولاً شامل تعریف یک متغیر محلی جدید با یک نام مبهم و مقداردهی اولیه آن با یک عدد تصادفی است (مثلاً: int unused_val_723 = 9812;).

۴-۳- پیچیده‌سازی عبارات(Expression Complication)

در این تکنیک، عبارات محاسباتی و منطقی ساده به فرم‌های پیچیده‌تر اما از نظر عملکردی معادل تبدیل می‌شوند. هدف، دشوار کردن تحلیل سریع و درک معنای اصلی عبارت برای خواننده است.

- مثال محاسباتی: x = y + 1; ممکن است به x = y - (-1); یا x = (y * 2 + 2) / 2; تبدیل شود.
- مثال منطقی: if (a > b) ممکن است به if (!(a <= b)) تبدیل شود.

این پاس گره‌های مربوط به عبارات در AST (مانند BinaryOpNode یا UnaryOpNode) را پیمایش می‌کند. با شناسایی الگوهای خاص، گره عبارت اصلی با یک زیردرخت AST جدید که نماینده عبارت پیچیده‌تر و معادل است، جایگزین می‌شود. برای این کار ممکن است از قوانین جبر بولی یا اصول محاسباتی استفاده شود.

۵-توجیه تصمیمات طراحی

۵-۱-انتخاب ANTLR

ابزار ANTLR به دلیل قدرت بالا در تعریف گرامرهای پیچیده، تولید خودکار تحلیل‌گرهای کارآمد به زبان‌های مختلف (از جمله پایتون) و ارائه مکانیزم‌های Visitor/Listener برای پیمایش درخت تجزیه، به عنوان ابزار اصلی برای فاز تحلیل انتخاب شد. این انتخاب مطابق با پیشنهاد صورت پروژه نیز بود.

۵-۲-AST سفارشی

به جای کار مستقیم با درخت تجزیه ANTLR (که می‌تواند بسیار جزئی و وابسته به گرامر باشد)، یک AST سفارشی طراحی شد. این AST ساختار منطقی‌تری از برنامه را نمایش می‌دهد و دستکاری آن برای پاس‌های مبهم‌سازی ساده‌تر و مستقل‌تر از جزئیات گرامر است. گره‌های ast_nodes.py این ساختار را تعریف می‌کنند.

۵-۳-الگوی Visitor برای پیمایش AST

برای اعمال پاس‌های مبهم‌سازی و همچنین تولید کد، از الگوی طراحی Visitor استفاده شده است. این الگو امکان افزودن عملیات جدید بر روی ساختار AST را بدون تغییر در کلاس‌های گره‌های AST فراهم می‌کند و کد را ماژولارتر می‌سازد. (اگرچه در پیاده‌سازی فعلی، ممکن است متدهای visit_NODE مستقیماً در کلاس‌های پاس‌ها تعریف شده باشند که همچنان رویکرد پیمایشی مشابهی را دنبال می‌کند).

۵-۴-مدیریت قلمرو برای تغییر نام شناساگرها

برای تغییر نام صحیح شناساگرها، نگاشت نام‌های قدیمی به جدید به ازای هر قلمرو (scope) تابع به صورت جداگANE انجام شد تا از تداخل نام بین قلمروهای مختلف یا با شناساگرهای سراسری جلوگیری شود. نام توابع در یک قلمرو سراسری مدیریت می‌شوند.

۵-۵-تصادفی بودن در درج کد مرده

استفاده از یک عامل احتمالی برای درج کد مرده باعث می‌شود که خروجی‌های مبهم‌شده برای یک ورودی یکسان، در اجراهای مختلف مبهم‌ساز، متفاوت باشند (اگر seed تصادفی کنترل نشود)، که می‌تواند تحلیل الگوهای مبهم‌سازی را دشوارتر کند.

۶- مثال‌های عملی (قبل و بعد از مبهم‌سازی)

در این بخش، یک یا دو مثال از کد Mini-C ورودی و خروجی مبهم‌شده معادل آن ارائه می‌شود تا تأثیر تکنیک‌های اعمال شده به صورت ملموس نمایش داده شود.

```
// input_example1.mc
int factorial(int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}

int main() {
    int num = 5;
    int fact_val;
    fact_val = factorial(num);
    printf("Factorial of %d is %d\n", num, fact_val);
    return 0;
}
```

شکل ۱- ورودی کد Mini-C

```mc
// input_example1.mc
int factorial(int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}

int main() {
    int num = 5;
    int fact_val;
    fact_val = factorial(num);
    printf("Factorial of %d is %d\n", num, fact_val);
    return 0;
}
```

*شکل ۲- خروجی مبهم شده کد **Mini-C** داده شده*

۷-چالش‌ها و نکات مهم پروژه

- تعریف گرامر بدون ابهام برایMini-C
- تبدیل دقیق Parse Tree به AST سفارشی
- حفظ عملکرد صحیح پس از تغییرات درAST
- مدیریت صحیح scope در تغییر نام شناساگرها
- اطمینان از تولید کد معتبر و بدون خطا

۸-نتیجه‌گیری و پیشنهادات برای کارهای آینده

در این پروژه، یک سیستم کامل برای مبهم‌سازی کدهای Mini-C طراحی و پیاده‌سازی شد. این سیستم با حفظ عملکرد کد ورودی، آن را به نسخه‌ای مبهم‌تر تبدیل می‌کند که برای خواننده انسانی درک‌پذیر نیست. در گام‌های آینده، پیشنهاد می‌شود:

- اضافه کردن تکنیک‌های پیشرفته‌تر (مانند Control Flow Flattening)
- پشتیبانی از آرایه‌ها، اشاره‌گرها و ساختارهای پیچیده‌تر
- پیاده‌سازی CLI و GUI پیشرفته‌تر
- توسعه متریک‌هایی برای سنجش میزان مبهم‌سازی

۹-مراجع

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[2] ANTLR Project. (n.d.). *ANTLR (Another Tool for Language Recognition)*. Retrieved from https://www.antlr.org/

[3] GeeksforGeeks. (2022). *Code Obfuscation Techniques*. Retrieved from https://www.geeksforgeeks.org/code-obfuscation-techniques/

[4] GitHub Repository - ANTLR Example Projects: https://github.com/antlr/antlr4

[5] GitHub Repository - Mini-C Obfuscator Sample Project: https://github.com/Mohammad-OFF/CompilerProject-Obfuscator-MohammadMohajeri

[6] McPeak, S. (2001). *Elsa: An Elkhound-based C/C++ Parser*. University of California, Berkeley.

[7] Collberg, C., & Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley.

[8] Tutorial: Building a C Compiler with ANTLR. (n.d.). Retrieved from https://tomassetti.me/building-c-language-compiler-with-antlr/

[9] Python Docs. (n.d.). *ast — Abstract Syntax Trees*. Retrieved from https://docs.python.org/3/library/ast.html