مستندات پروژه عملی درس طراحی کامپایلر: مبهم‌ساز کد Mini-C

دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

درس: طراحی کامپایلر

استاد: دکتر محمدهادی علائیان

ترم تحصیلی: ۱۴۰۲-۲

اعضای گروه:

امیرعلی حبیبی 40004353

امیرعلی صفایی 40120313

امیرمحمد علیخانی 40121023

فهرست مطالب:

---

۱. مقدمه

۱.۱. هدف پروژه

هدف اصلی این پروژه، طراحی و پیاده‌سازی یک ابزار مبهم‌ساز کد (Code Obfuscator) برای زیرمجموعه‌ای ساده از زبان C موسوم به Mini-C است. این ابزار، کد منبع Mini-C

را به عنوان ورودی دریافت کرده و نسخه‌ای مبهم‌شده از آن را تولید می‌کند که از نظر عملکردی معادل کد اصلی بوده اما درک و تحلیل آن برای انسان دشوارتر باشد.

۱.۲. مبهم‌سازی کد چیست؟

مبهم‌سازی کد فرآیندی است که در آن کد منبع یا کد ماشین به گونه‌ای تغییر داده می‌شود که فهم منطق و ساختار آن برای انسان دشوار گردد، بدون آنکه عملکرد اصلی برنامه تغییر کند. این تکنیک معمولاً با اهدافی نظیر حفاظت از مالکیت معنوی، جلوگیری از مهندسی معکوس، و دشوار ساختن تحلیل بدافزارها مورد استفاده قرار می‌گیرد.

۲. زبان Mini-C

۲.۱. مشخصات و محدودیت‌ها

زبان Mini-C که در این پروژه پشتیبانی می‌شود، شامل موارد زیر است:

* انواع داده پایه: `int`, `char`, `bool` (همراه با لیترال‌های `true` و `false`).
* متغیرها و عملگرها: عملگرهای استاندارد محاسباتی، رابطه‌ای، منطقی و انتساب.
* دستورات کنترل جریان: `if-else`, `while`, `for`, `return`.
* توابع: تعریف توابع با پارامتر و مقدار بازگشتی.
* ورودی/خروجی: توابع `printf` و `scanf` به عنوان فراخوانی تابع در نظر گرفته می‌شوند (بدون تحلیل رشته فرمت توسط مبهم‌ساز). برای کامپایل نهایی با GCC/Clang، هدرهای `stdio.h` (و `stdbool.h` در صورت استفاده از `bool`) باید لحاظ شوند.
* عملگر آدرس (`&`): برای استفاده در `scanf` پشتیبانی می‌شود.
* محدودیت‌ها: این زبان شامل `struct`، اشاره‌گر، دستورات پیش‌پردازنده (مانند `#include`) نمی‌باشد.

گرامر کامل زبان در فایل `grammar/MiniC.g4` تعریف شده است.

۳. معماری و طراحی کلی سیستم

سیستم مبهم‌ساز کد از چندین مرحله اصلی تشکیل شده است:

۳.۱. مراحل پردازش

1. تحلیل لغوی (Lexical Analysis): کد ورودی Mini-C به توکن‌های معنادار شکسته می‌شود.
2. تحلیل نحوی (Syntax Analysis): توکن‌ها بر اساس گرامر زبان بررسی شده و یک درخت تجزیه (Parse Tree) یا درخت نحو مجرد (AST) ساخته می‌شود.
3. پیمایش و تبدیل AST (Obfuscation Pass): یک یا چند پیمایشگر (Visitor) درخت نحو را طی کرده و تکنیک‌های مبهم‌سازی مختلف را روی گره‌های آن اعمال می‌کنند.
4. تولید کد (Code Generation): کد مبهم‌شده Mini-C از روی درخت تغییریافته (یا در حین پیمایش) بازسازی می‌شود.

۳.۲. استفاده از ANTLR برای تحلیل لغوی و نحوی

برای فاز تحلیل لغوی و نحوی، از ابزار ANTLR نسخه ۴ استفاده شده است.

* گرامر زبان Mini-C در فایل `MiniC.g4` تعریف گردید.
* با استفاده از ANTLR، کلاس‌های تحلیلگر لغوی (`MiniCLexer.py`)، تحلیلگر نحوی (`MiniCParser.py`) و یک کلاس پایه پیمایشگر (`MiniCVisitor.py`) به زبان پایتون تولید شدند.
* اسکریپت‌های `run_antlr.bat` (برای ویندوز) و `run_antlr.sh` (برای لینوکس/مک) برای سهولت در تولید این کلاس‌ها فراهم شده‌اند.

۴. پیاده‌سازی تکنیک‌های مبهم‌سازی

تکنیک‌های مبهم‌سازی در کلاس `ObfuscatorVisitor` (که از `CodeReconstructionVisitor` یا کلاس پایه `MiniCVisitor` ارث‌بری می‌کند) پیاده‌سازی شده‌اند. این کلاس درخت تجزیه را پیمایش کرده و کد مبهم‌شده را تولید می‌کند.

۴.۱. تغییر نام شناساگرها (Identifier Renaming)

* توضیح: نام متغیرهای محلی، پارامترهای توابع، و نام توابع (به جز توابع خاص مانند `main`, `printf`, `scanf`) با نام‌های کوتاه، بی‌معنی و تولیدی (مانند `v_0`, `p_1`, ``f_0``) جایگزین می‌شوند.
* پیاده‌سازی: از یک کلاس `Scope` برای مدیریت جدول نمادها و نگاشت نام‌های اصلی به نام‌های جدید در حوزه‌های مختلف استفاده می‌شود. هنگام تعریف یک شناساگر، نام جدیدی برای آن در حوزه جاری تولید و ذخیره می‌شود. هنگام استفاده از شناساگر، نام مبهم‌شده آن از جدول نمادها بازیابی می‌گردد.

۴.۲. درج کد مرده (Dead Code Insertion)

* توضیح: دستورات بی‌اثری که هیچ تاثیری بر خروجی نهایی برنامه ندارند (مانند تعریف متغیرهای استفاده‌نشده و مقداردهی آن‌ها با مقادیر تصادفی) به صورت تصادفی در بلوک‌های کد درج می‌شوند.
* پیاده‌سازی: در متد `visitBlock`، قبل از پردازش دستورات اصلی، تعدادی دستور کد مرده تولید و به ابتدای لیست دستورات بلوک اضافه می‌شوند. این کدها شامل فراخوانی توابع بازگشتی بی‌معنی نیز می‌توانند باشند.

۴.۳. تبدیل عبارات معادل (Equivalent Expression Transformation)

* توضیح: عبارات محاسباتی ساده با عبارات پیچیده‌تر اما از نظر عملکردی معادل جایگزین می‌شوند. به عنوان مثال، عبارت `a + b` ممکن است به `a-(-b)` تبدیل شود.
* پیاده‌سازی: متد `visitAdditiveExpression` برای شناسایی عملگرهای `+` و `-` بازنویسی شده و عبارت جدید را بر اساس الگوی تبدیل تولید می‌کند.

۴.۴. مسطح‌سازی جریان کنترل (Control Flow Flattening) - برای تابع `main`

* توضیح: این تکنیک با هدف پیچیده کردن جریان اجرای برنامه، دنباله‌ای از دستورات را به یک حلقه بزرگ (مانند `while`) و یک دستور `switch` توزیع‌کننده تبدیل می‌کند. یک متغیر حالت تعیین می‌کند که کدام بخش از کد اصلی (که اکنون یک `case` در `switch` است) در مرحله بعد اجرا شود.
* پیاده‌سازی: این تکنیک به صورت ساده‌شده و هدفمند برای بدنه تابع `main` پیاده‌سازی شده است. متغیرهای محلی ابتدا تعریف می‌شوند، سپس دستورات اجرایی اصلی در یک ساختار `while/switch` قرار می‌گیرند.

۴.۵. افزودن توابع بازگشتی بی‌معنی (Meaningless Recursive Functions)

* توضیح: یک یا چند تابع بازگشتی ساده که هیچ تاثیر جانبی بر متغیرهای اصلی برنامه یا خروجی ندارند، تولید شده و به کد اضافه می‌شوند. فراخوانی این توابع (با عمق بازگشت کم) می‌تواند به صورت تصادفی در بخش‌های کد مرده درج شود.
* پیاده‌سازی: متد `create_meaningless_recursive_function_` کد C این توابع را تولید و در لیستی ذخیره می‌کند. متد `visitProgram` این تعاریف را به ابتدای کد نهایی اضافه می‌کند. متد `generate_dead_code_statement` با احتمال مشخصی به جای کد مرده استاندارد، فراخوانی به یکی از این توابع را درج می‌کند.

۵. بازسازی و تولید کد مبهم‌شده

پس از اعمال تکنیک‌های مبهم‌سازی توسط `ObfuscatorVisitor`، هر متد `visit` رشته‌ای حاوی کد مبهم‌شده برای گره مربوطه را برمی‌گرداند. فراخوانی متد `visit` برای گره ریشه درخت (قاعده `program`) منجر به تولید رشته کامل کد مبهم‌شده Mini-C می‌شود. مدیریت تورفتگی (Indentation) نیز در متدهای پیمایشگر برای حفظ حداقل خوانایی انجام می‌شود.

۶. مقایسه زمان اجرا

این برنامه قابلیتی برای کامپایل و اجرای کد اصلی و کد مبهم‌شده و سپس مقایسه زمان اجرای آن‌ها را فراهم می‌کند.

* یک تابع کمکی `compile_and_run` در `main.py` پیاده‌سازی شده است که کد C را (پس از افزودن هدرهای لازم مانند `stdio.h` و `stdbool.h` به صورت شرطی) با استفاده از کامپایلر مشخص شده (GCC یا Clang) کامپایل کرده و سپس اجرا می‌کند.
* زمان اجرای خالص برنامه‌ها (بدون احتساب زمان کامپایل) با استفاده از `time.perf_counter()` اندازه‌گیری می‌شود.
* خروجی استاندارد برنامه‌ها نیز نمایش داده می‌شود تا از صحت عملکرد اطمینان حاصل شود.
* در نهایت، درصد سربار (Overhead) یا بهبود زمان اجرای کد مبهم‌شده نسبت به کد اصلی گزارش می‌شود. باید توجه داشت که به دلیل بهینه‌سازی‌های قدرتمند کامپایلرهای مدرن C، کد مبهم‌شده گاهی ممکن است سریع‌تر از کد اصلی اجرا شود، به خصوص اگر تکنیک‌های مبهم‌سازی ساده باشند و کامپایلر بتواند کدهای اضافی را حذف یا ساختار را بهینه کند.

۷. راهنمای استفاده از برنامه

۷.۱. پیش‌نیازها

* پایتون نسخه ۳.۷ یا بالاتر.
* کیت توسعه جاوا (JDK) نسخه ۱۱ یا بالاتر (برای اجرای ابزار ANTLR).
* فایل JAR ابزار ANTLR نسخه ۴ (مثلاً `antlr-4.13.x-complete.jar`).

۷.۲. نحوه نصب و راه‌اندازی

1. مخزن پروژه را کلون کنید.
2. یک محیط مجازی پایتون ایجاد و فعال کنید (توصیه می‌شود).
3. بسته‌های پایتون مورد نیاز را نصب کنید: `pip install antlr4-python3-runtime`
4. فایل JAR ابزار ANTLR را در ریشه پروژه قرار دهید یا مسیر آن را در اسکریپت‌های `run_antlr` به‌روز کنید.

۷.۳. نحوه اجرای مبهم‌ساز و گزینه‌های خط فرمان

* تولید کلاس‌های پارسر (در صورت تغییر گرامر):
  * ویندوز: `.\run_antlr.bat`
  * لینوکس/مک: `chmod +x run_antlr.sh && ./run_antlr.sh`
* اجرای مبهم‌ساز:

```bash
python src/main.py <input_file.mc> [options]
```

گزینه‌ها (Options):

* `input_file`: مسیر فایل ورودی Mini-C.
* `-o, --output_file`: مسیر فایل خروجی مبهم‌شده. (پیش‌فرض: `output/<input_file_base>_obfuscated.mc`)

* `--rename`: فعال‌سازی تغییر نام شناساگرها.
* `--dead-code`: فعال‌سازی درج کد مرده.
* `--flatten`: فعال‌سازی مسطح‌سازی جریان کنترل (برای تابع `main`).
* `--expr-transform`: فعال‌سازی تبدیل عبارات معادل.
* `--meaningless-funcs`: فعال‌سازی افزودن توابع بازگشتی بی‌معنی.
* `--all`: فعال‌سازی تمامی تکنیک‌های مبهم‌سازی تعریف‌شده.
* `--interactive`: انتخاب تعاملی تکنیک‌ها از طریق خط فرمان. (اگر هیچ فلگ تکنیکی مشخص نشود، این حالت به صورت پیش‌فرض فعال می‌شود).
* `--compare-runtime`: کامپایل و مقایسه زمان اجرای کد اصلی و مبهم‌شده.
* `--compiler {gcc,clang}`: انتخاب کامپایلر C برای مقایسه زمان اجرا (پیش‌فرض: `gcc`).
* `--verbose-runtime`: نمایش خروجی کامل stdout/stderr برنامه‌ها در حین مقایسه زمان اجرا.
* `-h, --help`: نمایش پیام راهنما.

۱۰۹. نتیجه‌گیری

این پروژه با موفقیت یک ابزار مبهم‌ساز کد برای زبان Mini-C با قابلیت‌های متعدد پیاده‌سازی کرد. این تجربه درک عمیق‌تری از فرآیندهای تحلیل کد، دستکاری درخت نحو مجرد و چالش‌های مربوط به تبدیل برنامه‌ها را فراهم آورد. افزودن قابلیت مقایسه زمان اجرا نیز دیدگاهی عملی نسبت به تاثیر این تغییرات بر عملکرد ارائه داد.