گزارش کار پروژه کامپایلر - فاز دوم

طراحی و پیاده‌سازی ابزار رفع ابهام (De-obfuscator) برای زبانMini-C

درس :طراحی کامپایلر

دانشگاه :صنعتی خواجه نصیرالدین طوسی

استاد :دکتر محمدهادی علائیان

---

۱ .مقدمه و اهداف پروژه

در فاز اول این پروژه، ابزاری برای مبهم‌سازی کد Mini-C طراحی شد که با استفاده از تکنیک‌هایی نظیر تغییر نام، تسطیح جریان کنترل و تبدیل عبارات، خوانایی کد را به شدت کاهش می‌داد.

هدف اصلی در فاز دوم، طراحی و پیاده‌سازی فرآیند معکوس بود: ساخت یک ابزار **De-obfuscator** که بتواند یک کد مبهم‌شده Mini-C را به عنوان ورودی دریافت کرده و نسخه‌ای خوانا، تمیز و از نظر عملکردی معادل با کد اصلی تولید کند. این ابزار در فرآیندهای مهندسی معکوس، تحلیل بدافزار و درک کدهای محافظت‌شده کاربرد دارد.

---

۲ .معماری سیستم

معماری این پروژه نیز همانند فاز اول، بر اساس یک خط لوله (Pipeline) چندمرحله‌ای بنا شده است. بخش بزرگی از زیرساخت پروژه، از جمله تحلیلگر و ساختارAST ، از فاز اول به ارث برده شده و منطق اصلی در مرحله پردازش AST تغییر کرده است.

1. **تحلیلگر لغوی و نحوی :(Parser)** کد مبهم‌شده ورودی ابتدا توسط تحلیلگر ANTLR خوانده می‌شود. گرامر زبان (`ObfuMiniC.g4`) در این فاز به‌روزرسانی شد تا ساختارهای جدیدی که در کد مبهم وجود داشتند (مانند دستورات `goto`و برچسب‌ها `label`)را شناسایی کند.
2. **سازنده درخت نحو انتزاعی :(AST Builder)** خروجی پارسر به یک درخت AST تبدیل می‌شود. کلاس `ASTBuilder`نیز گسترش یافت تا گره‌های جدید مربوط به `goto`و `label`را در درخت ایجاد کند.
3. **موتور رفع ابهام :(De-obfuscation Engine)** این بخش، هسته اصلی پروژه فاز دوم است. مجموعه‌ای از الگوریتم‌های هوشمند به صورت پیمایشگر (Visitor) بر روی AST اعمال می‌شوند تا پیچیدگی‌های آن را کاهش داده و ساختار اصلی کد را بازیابی کنند.
4. **تولیدکننده کد :(Code Generator)** در نهایت، پیمایشگر `CodeGenerator`درخت ASTساده‌سازی‌شده را پیمایش کرده و کد نهایی تمیز و خوانا را به زبان Mini-C تولید می‌کند.

---

۳ .تکنیک‌های رفع ابهام پیاده‌سازی‌شده

برای رسیدن به هدف پروژه، سه تکنیک اصلی رفع ابهام پیاده‌سازی شد که در ادامه به تفصیل شرح داده می‌شوند.

۳.۱ .ساده‌سازی عبارات(Expression Simplification)

- **فایل**`deobfuscator/techniques/expression_simplifier.py` **:**
- **شرح :**این تکنیک به عنوان اولین گام، وظیفه ساده‌سازی عبارات ریاضی و منطقی پیچیده را بر عهده دارد. پیمایشگر مربوطه با عبور از رویAST ، الگوهای ریاضی که توسط مبهم‌ساز ایجاد شده بودند را شناسایی کرده و آن‌ها را با معادل ساده‌ترشان جایگزین می‌کند. به عنوان مثال:
  - `a - (-b)` به `a + b`تبدیل می‌شود.
  - `a + 0` به `a`تبدیل می‌شود .این فرآیند به طور مستقیم خوانایی محاسبات در کد را افزایش می‌دهد.

۳.۲ .سادەسازی جریان کنترل(Control-Flow Simplification)

- **فایل :**

`deobfuscator/techniques/control_flow_simplifier.py`

- **شرح :**این الگوریتم، **پیچیدەترین و مهمترین بخش پروژه** بود. هدف آن، باز کردن ساختار درهمتنیده `while-switch-goto`است که توسط تکنیک تسطیح جریان کنترل در فاز اول ایجاد شده بود. الگوریتم پیادەسازیشده در چند مرحله عمل میکند:

1. **شناسایی الگو :**ابتدا در بدنه هر تابع به دنبال ساختار `switch`میگردد که بر روی یک "متغیر حالت (state variable) "کار میکند.
2. **استخراج بلوکهای کد :**تمام بلوکهای کد را که با یک برچسب (label) مشخص شدهاند، استخراج کرده و در یک دیکشنری ذخیره میکند.
3. **مرتبسازی هوشمند :**از حالت اولیه (معمولاً `case 0`)شروع کرده و با دنبال کردن مقادیر جدید متغیر حالت در انتهای هر بلوک، ترتیب صحیح اجرای بلوکها را بازسازی میکند.
4. **بازسازی بدنه تابع :**در نهایت، بدنه تابع را با دستورات مرتبشده جایگزین کرده و تمام عناصر اضافی مبهمسازی (متغیر حالت، `switch, goto`ها و `label`ها) را حذف میکند.

۳.۳ .حذف کدهای مرده(Dead Code Removal)

- **شرح :**این تکنیک به صورت ضمنی در الگوریتم سادەسازی جریان کنترل پیادەسازی شد. با بازسازی منطقی جریان اجرای برنامه، بلوکهای کدی که هرگز قابل دسترسی نبودند (مانند بدنه شرط `if (0)`)به طور طبیعی از جریان اجرای جدید حذف شده و در خروجی نهایی ظاهر نمیشوند. همچنین، با حذف متغیر حالت، تعریف آن نیز از کد نهایی پاک میشود.

---

۴ .چالشهای پروژه

- **تحلیل گرامر مبهم :**اولین چالش، اصلاح گرامر ANTLR برای پذیرش ساختارهای غیر استانداردی مانند `goto` و `label`بود که نیازمند دقت بالا در تعریف قوانین جدید نحوی بود.
- **دیباگ کردن الگوریتم بازسازی :**بزرگترین چالش، پیاده‌سازی و دیباگ کردن الگوریتم `ControlFlowSimplifier`بود. کوچکترین ناهماهنگی بین ساختار AST تولید شده و منطق پیمایشگر، منجر به خروجی‌های ناقص یا خالی می‌شد که فرآیند عیب‌یابی آن نیازمند تحلیل دقیق و قدم به قدم بود.

مدیریت ساختار AST: اطمینان از اینکه تمام پیمایشگرها(AST Builder ، Simplifiers، Code Generator)درک یکسانی از ساختار نودهای AST دارند، یکی از نکات کلیدی بود که در طول پروژه بارها مورد بازبینی قرار گرفت.

۵ .نمونه خروجی و مقایسه

برای نمایش قدرت و صحت عملکرد ابزار، یک مقایسه "قبل و بعد" از کد ورودی و خروجی نهایی ارائه می‌شود.

کد ورودی

```
int ial100(int gts580, int srv427) {
    int fdy779;
    if (0)
    {
        printf("Unreachable\\n");
    }
    fdy779 = (gts580 + srv427);
    return fdy779;
}
```

```c
int ial100(int gts580, int srv427) {
    int _f0_state = 0;
    int fdy779;
_f0_dispatcher:
    switch (_f0_state) {
        case 0: goto _f0_case_0;
        case 1: goto _f0_case_1;
        case 2: goto _f0_case_2;
    }
    {
    _f0_case_0:
        if (0) { printf("Unreachable\\n"); }
        _f0_state = 1;
        goto _f0_dispatcher;
    }
    {
    _f0_case_1:
        fdy779 = (gts580 - (-srv427));
        _f0_state = 2;
        goto _f0_dispatcher;
    }
    {
    _f0_case_2:
        return fdy779;
    }
}
```