Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б08-мм

Обфускация методом слияния функций

Чернобровкина Юлия Владиславовна

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. преподаватель кафедры ИАС, Смирнов К. К.

Консультант:
Старший разработчик ПО I категории ООО «Софтком», Бабанов П. А.

Санкт-Петербург
2024

Оглавление

Введение

Безопасность и надежность ПО являются критически важными темами для сферы программирования, поэтому в настоящее время существует множество средств для защиты кода. Одним из таких средств является обфускация [9] — метод, осуществляющий «запутывание кода». Под запутыванием имеется в виду преобразование кода, которое сохраняет функциональность программы, но делает затруднительным анализ кода [7]. Таким образом цель обфускации — скрыть логику работы ПО.

Обфускация может производиться на любом из трех уровней: исходный код, промежуточное представление и машинный код. Обфускация трансформирует код, при этом сохраняя исходную функциональность. Можно выделить следующие группы трансформаций:

1. трансформации потока управления
2. трансформации структур данных
3. лексическая обфускация

Обфускация не гарантирует абсолютной защиты ПО, ее главная цель — сделать процесс реверс-инжиниринга затратным по времени и средствам и потому невыгодным для злоумышленника. Поэтому часто обфускацию используют как дополнительный инструмент для защиты кода. На данный момент существует множество обфускаторов с открытым исходным кодом, однако они, как правило, сильно уязвимы перед деобфускаторами [6] и другими средствами реверс-инжиниринга. Проприетарные обфускаторы лучше противостоят атакам, но узнать принцип их работы практически невозможно.

Одной из сфер деятельности компании ООО «Софтком» является разработка обфускатора. Трансформации в обфускаторе реализованы с помощью системы LLVM проходов (LLVM-passes), позволяющих изменять промежуточное представление кода (LLVM-IR). На данный момент уже написаны некоторые трансформации [10], но отсутствует трансформация методом слияния функций.

Слияние функций — одна из базовых трансформаций потока управления программы. Тела функций помещаются в новую функцию, а управление вызовом исходных функций происходит через дополнительный аргумент.

Таким образом в рамках данной работы предлагается реализовать трансформацию методом слияния функций для упомянутого обфускатора. Тема работы была предложена компанией ООО «Софтком».

1. Постановка задачи

Целью работы является реализация метода слияния функций для обфускатора компании «Софтком». Целевыми языками для обфускатора являются C/C++.

Для её выполнения были поставлены следующие **задачи**:

1. провести обзор существующих решений и используемых технологий;
2. реализовать метод слияния функций с помощью технологии проходов LLVM;
3. провести оценку качества работы обфускатора с помощью декомпиляторов и выполнить замеры производительности.

2. Обзор

2.1. Обзор существующих обфускаторов

Цель обзора: рассмотреть существующие обфускаторы, реализацию метода слияния функций в них и сделать вывод о возможности внедрения обфускаторов в проект.

В обзор попали обфускаторы, которые удовлетворяют хотя бы одному из следующих критериев:

1. наличие в обфускаторе трансформации слиянием функций;
2. обфускатор работает с LLVM IR.

При выборе кандидатов была изучена работа [5], в которой проведено сравнение наиболее известных обфускаторов.

2.1.1. O-LLVM

O-LLVM (Obfuscator-LLVM) — обфускатор, созданный на базе LLVM фреймворка. На вход обфускатору может подаваться код в промежуточном представлении LLVM IR, сгенерированный одним из фронт-ендов LLVM, то есть компилятором высокоуровневого языка. По сути обфускатор является набором проходов (*passes*), каждый из которых осуществляет определенную обфусцирующую трансформацию. Часть проходов является открытой и доступна в репозитории проекта [1].

Открытая часть обфускатора поддерживает следующие трансформации:

1. замена инструкций;
2. создание ложных потоков управления;
3. сглаживание потока управления.

[1] репозиторий O-LLVM, https://github.com/obfuscator-llvm/obfuscator/tree/llvm-4.0 (дата обращения: 22 октября 2024 г.).

Метод слияния функций, реализованный в обфускаторе O-LLVM, описан в статье [2]. В сокращенном изложении трансформация реализована следующим образом: базовые блоки исходных функций помещаются в тело новой функции. Ее аргументами становятся аргументы прошлых функций, а также появляется дополнительный аргумент для контроля за вызовом нужных блоков. Затем тела исходных функций заменяются вызовом полученной функции с нужным параметром. Исходные функции не удаляются, чтобы сохранить API модуля.

Несмотря на наличие трансформации слиянием функций в описании обфускатора, метод отсутствует в репозитории проекта. Таким образом непосредственно применить трансформацию слиянием в обфускаторе O-LLVM невозможно.

2.1.2. Tigress

Tigress — обфускатор языка C, предоставляющий защиту от статического и динамического реверс-инжиниринга. Код обфускатора является закрытым, пользователям предлагается изучать функциональность обфускатора, применяя реализованные в нем трансформации к написанному коду. Tigress содержит набор преобразований, которые пользователь по своему усмотрению может указать в параметрах командной строки.

Список поддерживаемых трансформаций Tigress значительно превосходит прошлый обфускатор [8], из них можно выделить следующие:

1. слияние и разделение функций;
2. создание случайной функции;
3. сглаживание потока управления;
4. трансформация арифметических выражений и литералов;
5. использование контрольной суммы для контроля целостности программы.

Официальный сайт проекта приводит два возможных сценария работы метода слияния. В первом случае тела функций просто помещаются в конструкцию `if-else`. Этот вариант достаточен, когда метод слияния используется как подготовительный этап для следующих трансформаций, например `jit` или `virtualize`. Во втором случае к функциям сперва применяется трансформация сглаживания потока управления, затем полученные базовые блоки соединяются и встраиваются в один из методов управления, например `switch` или `goto`. Отметим то преимущество, что в обфускаторе предусмотрен выбор из нескольких методов управления, чего не было обнаружено в других проектах.

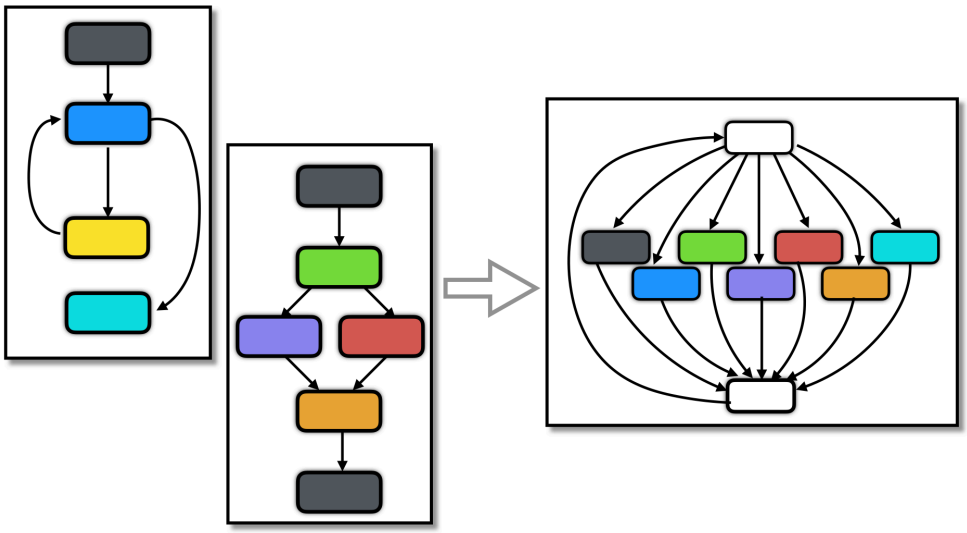

Рис. 1: трансформация слияния функций. Изображение взято из источника https://tigress.wtf/merge.html(дата обращения: 22 октября 2024 г.).

Таким образом в обфускаторе реализована трансформация слиянием функций, имеется описание реализации, но код закрыт. К тому же целевым для обфускатора является язык C, а не промежуточное представление LLVM-IR.

2.1.3. Petráček's Obfuscator

Petráček's Obfuscator — основанный на LLVM обфускатор [3]. Трансформации в обфускаторе являются наборами независимых проходов, как и в случае обфускатора O-LLVM.

Список реализованных трансформаций следующий:

1. встраивание кода;
2. выделение кода в отдельный блок;
3. чередование кода;
4. разделение блоков;
5. ложный поток управления;
6. табличная интерпретация.

Несмотря на широкий набор трансформаций, обфускатор не поддерживает метод слияния функций.

2.1.4. Šíma's Obfuscator

Šíma's Obfuscator — обфускатор, который также основан на фреймворке LLVM [4]. Обфускатор поддерживает следующие трансформации:

1. вставка мертвого кода;
2. чередование кода;
3. табличная интерпретация.

В обфускаторе метод слияния функций также отсутствует.

2.1.5. Выводы

Среди рассмотренных решений только обфускатор Tigress действительно реализовал метод слияния функций. Tigress предоставляет описание используемых трансформаций, в том числе рассматриваемого метода. Однако код проекта закрыт, более того целевой язык обфускатора C. Поэтому использование данного обфускатора в проекте невозможно.

Остальные обфускаторы работают с промежуточным представлением LLVM IR, однако они либо вовсе не реализуют трансформацию слиянием функций, либо предоставляют лишь краткое описание реализации, как обфускатор O-LLVM.

Таким образом в ходе обзора рассмотрены некоторые существующие обфускаторы. У тех решений, которые описывают трансформацию слияния функций, было изучено описание метода. Сделан вывод, что ни один из обфускаторов не может быть использован для решения задачи данного проекта.

2.2. Обзор используемых технологий

Трансформации в обфускаторе «Софтком» реализованы в виде LLVM проходов, поэтому для лучшего понимания работы в данном разделе будет дано описание системы проходов LLVM.

2.2.1. LLVM passes

Проходы (*passes*) в LLVM — это инструмент для выполнения преобразований над кодом в промежуточном представлении LLVM IR. Типичное применения проходов — оптимизация кода, также проходы могут использоваться для анализа кода или обфускации. В структуре LLVM проходы занимают место middle-end, то есть они работают сразу после компилятора высокоуровневого языка.

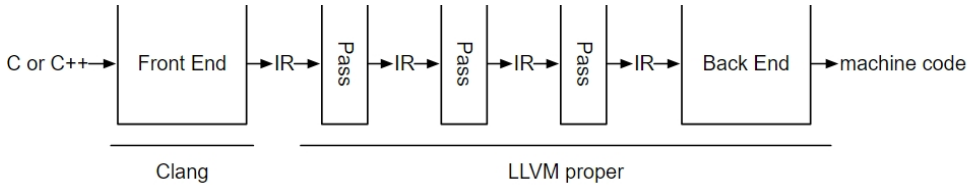

Рис. 2: структура компонент LLVM. Изображение взято из источника https://www.cs.cornell.edu/~asampson/blog/llvm.html(дата обращения: 22 октября 2024 г.).

LLVM предоставляет для разработчиков специальное API, позволяющее создавать программное обеспечение, которое можно встроить в

структуру LLVM. В частности API содержит классы для представления модуля, функций, базовых блоков и так далее.

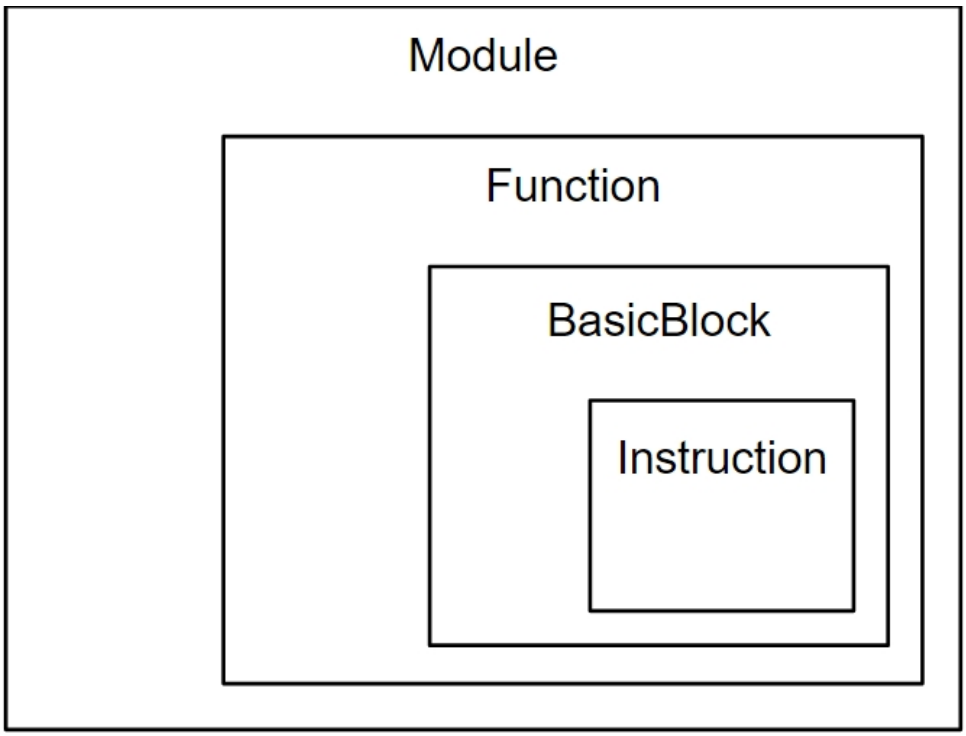

Рис. 3: классы из API LLVM. Изображение взято из источника https://www.cs.cornell.edu/~asampson/blog/llvm.html(дата обращения: 22 октября 2024 г.).

Документация LLVM содержит подробное описание API и приводит примеры его использования [1]. Таким образом, реализованную в виде прохода трансформацию можно использовать как динамическую библиотеку. Инструменты LLVM, такие, как clang или opt, могут принимать библиотеку в параметрах командной строки и автоматически применять проход к целевому коду.

В листинге 1 представлен пример вызова инструмента opt, который применит проход к входному файлу и поместит результат в выходной файл.

Листинг 1: Пример применения прохода с помощью opt.

```
$ opt -load-pass-plugin=<path_to_pass>/pass_lib.so
-S input_for_opt.ll -o out.ll
```

3. Описание реализации

Основные инструменты, используемые при реализации — LLVM API, написанный на языке C++.

3.1. Структура прохода

На вход проходу передается объект класса `llvm::Module` — класс, который содержит все объекты, используемые в модуле, такие как глобальные переменные, функции, внешние библиотеки и другое.

Для реализации метода слияния проход должен обработать все функции в модуле, не являющиеся объявлениями. Объявления функций не содержат тела, поэтому их обработка не требуется.

Работу прохода можно разделить на следующие компоненты:

1. группирование функций модуля по типу возвращаемого значения;
2. слияние функций каждой группы в новую единую функцию;
3. замена тел исходных функций вызовом полученной функции, то есть создание оберток функций.

При этом для создания новой функции, являющейся результатом слияния, необходимо:

1. создать сигнатуру для новой функции;
2. переместить тела исходных функций в новую;
3. скорректировать инструкции в блоках так, чтобы они обращались к корректным аргументам;

4. создать блок-диспетчер для управления переходами между блоками.

Рассмотрим подробнее некоторые детали реализации прохода.

3.2. Создание сигнатуры новой функции

При создании сигнатуры новой функции возможно применение двух подходов. Первый подход подразумевает простое объединение параметров каждой функции, при чем каждой исходной функции будет соответствовать некоторая последовательность параметров новой функции.

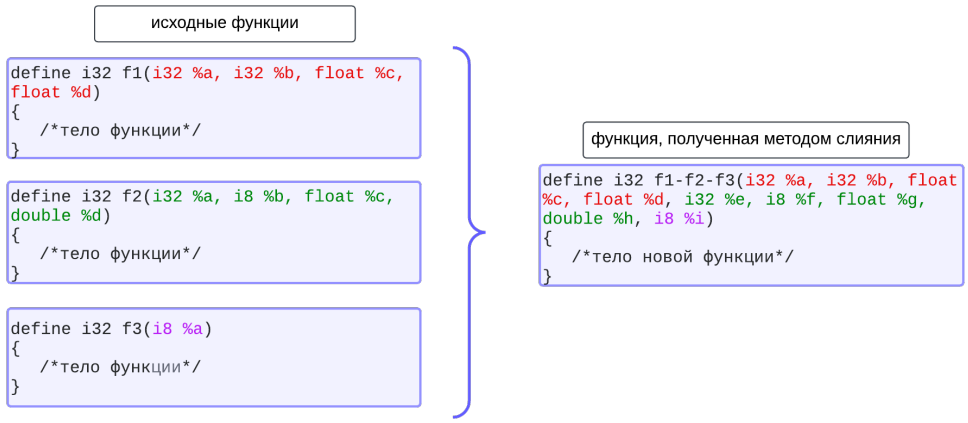

Рис. 4: Пример, демонстрирующий применение первого подхода создания сигнатуры функции. Функции представлены в промежуточном представлении LLVM IR, цветом показано соответствие параметров.

Второй подход предполагает, что параметрам одного типа в разных исходных функциях соответствует один параметр новой функции. Данный подход несколько труднее в реализации, так как предполагает создание таблицы соответствия параметров исходных функций параметрам новой функции. Тем не менее подход способен усложнить реверс-инжиниринг программы, так как соответствие параметров становится менее очевидным в сравнении с первым способом. По этой причине при написании прохода было решено реализовать второй подход.

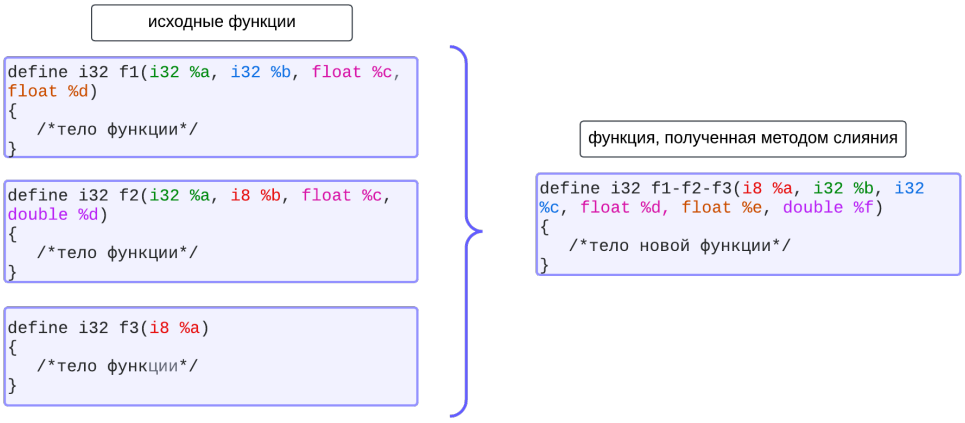

Рис. 5: Пример, демонстрирующий применение второго подхода создания сигнатуры функции. Функции представлены в промежуточном представлении LLVM IR, цветом показано соответствие параметров.

В обоих способах последним шагом необходимо добавить дополнительный параметр для идентификации конкретной функции.

3.3. Создание таблицы соответствия параметров

Таблица соответствия параметров является ключевым объектом для прохода, так как она необходима для корректной работы сразу нескольких методов: для создания сигнатуры новой функции, исправления инструкций и создания оберток исходных функций.

Создание таблицы реализовано в отдельном методе прохода, его шаги таковы:

1. создается словарь, в котором будет храниться максимальное количество параметров каждого используемого функциями типа. Необходимо помнить, что в рамках одной функции разным параметрам одного типа должны соответствовать разные параметры в новой функции, вот почему нужно знать максимальное количество параметров одного типа;

2. в итерации по параметрам каждой функции заполняется словарь;

3. на основе словаря строится таблица соответствия параметров старых функций параметрам новой функции.

При наличии словаря с максимальным количеством параметров каждого типа, создание сигнатуры сводится к созданию объекта класса `llvm::FunctionType`, который в структуре LLVM API отвечает за сигнатуру функции. В данном проходе это реализовано следующим образом:

1. методом `llvm::Function::getReturnType` получается тип возвращаемого значения функций (тип возвращаемого значения новой функции должен совпадать с типом переданных функций);

2. создается массив, который будет заполняться типами параметров;

3. словарь перебирается по ключам, и в массив добавляется нужное количество экземпляров каждого конкретного типа;

4. создается параметр типа `llvm::ConstantInt`, который будет использоваться для идентификации конкретной функции, и также добавляется в массив;

5. полученный массив вместе с возвращаемым типом передается в метод `llvm::FunctionType::get` и создается объект `llvm::FunctionType` для сигнатуры новой функции.

3.4. Блок-диспетчер

Имея дополнительный аргумент, отвечающий за идентификацию исходных функций, необходимо осуществлять переход на блоки, соответствующие конкретной функции. Для этой цели в теле новой функции создается блок-диспетчер, который становится также входным блоком. Поскольку базовые блоки исходных функций в новой функции располагаются последовательно, в блоке диспетчере достаточно передавать управление на входной блок нужной функции. Так как изменений в порядке следования блоков нет и инструкции перехода в блоках не меняются, функция отработает как исходная.

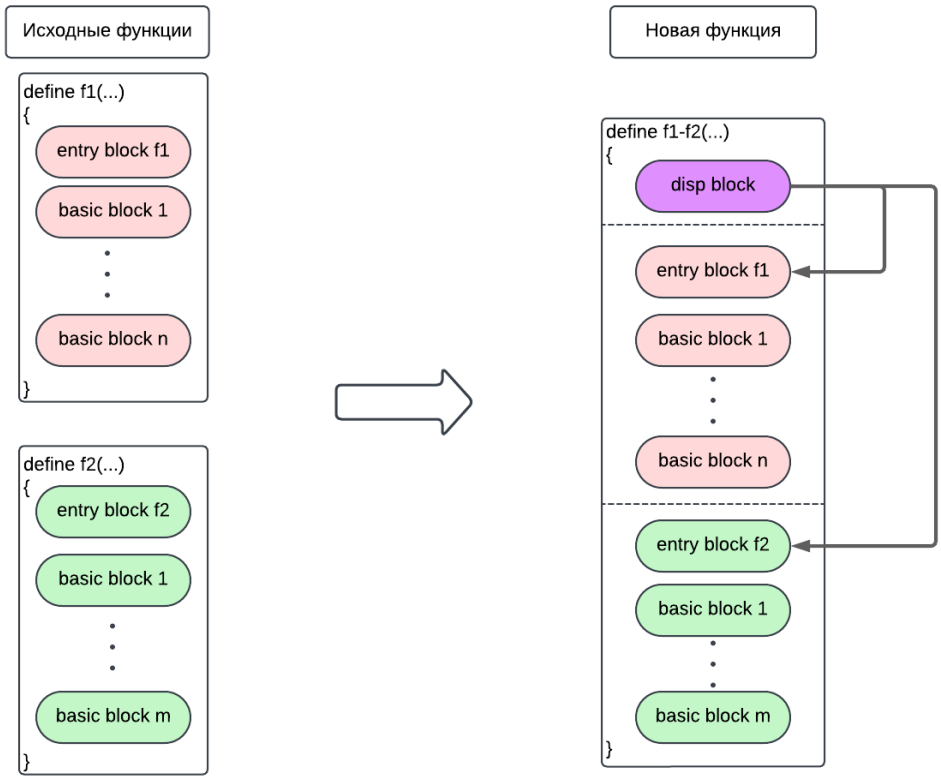

Рис. 6: Схема расположения базовых блоков и блока-диспетчера в функции, полученной методом слияния.

Основные детали, связанные с созданием блока-диспетчера, таковы:

1. в отдельном массиве сохраняются входные блоки старых функций, чтобы впоследствии блок-диспетчер мог передавать управление на них;

2. сам блок состоит из switch инструкции, case соответствуют входным блокам старых функций;

3. получить входные блоки можно методом `llvm::Function::getEntryBlock`, создать `switch` инструкцию методом `llvm::IRBuilder::CreateSwitch`.

3.5. Перемещение тел функций и исправление инструкций

В цикле по функциям в отдельный массив сохраняются базовые блоки функций. Полученные блоки открепляются от старых функций и закрепляются за новой с помощью методов `llvm::BasicBlock::removeFromParent` и `llvm::BasicBlock::insertInto` соответственно. Заметим, что в итерации по блокам функции, осуществляемой с помощью итератора на базовые блоки, нельзя удалять или откреплять блоки, так как это приведет к ошибке компиляции. Поэтому перед удалением необходимо скопировать блоки в отдельный контейнер. Следующий шаг — скорректировать инструкции, которые обращаются к аргументам, так как сигнатура новой функции отлична от сигнатур прошлых функций. На этом шаге понадобится описанная ранее таблица соответствия параметров. Исправление инструкций реализовано следующим образом:

1. для каждой инструкции создается массив использующих ее инструкций (метод `llvm::Argument::use`);

2. с помощью таблицы соответствия и позиции прошлого аргумента аргумент в инструкции меняется на нужный с помощью метода `llvm::Instruction::setOperand`.

3.6. Создание оберток функций

Чтобы сохранить API модуля, после создания новой функции необходимо заменить тела исходных функций на вызов новой, то есть создать обертки.

После открепления блоков тела функций становятся пустыми. Создается входной блок с инструкцией вызова новой функции. Все аргументы инициализируются специальным значением `llvm::ConstantAggregateZero`, и только аргументы, отвечающие данной функции, инициализируются аргументами старой функции. Сопо-

ставление аргументов исходных и новой функций осуществляется с помощью рассмотренной ранее таблицы соответствия.

3.7. Обработка исключений

Поскольку язык C++ способен генерировать исключения, необходимо предусмотреть обработку исключений. Для этого в LLVM API с функцией, генерирующей исключения, связывается так называемая *personality function*. Она отвечает за обработку возможных исключений.

Соответственно, в проходе необходимо предусмотреть обработку исключений в новой функции. Для этого в реализации:

1. проверяется наличие *personality function* для всех функций, переданных на слияние. Проверка осуществляется методом `llvm::Function::hasPersonalityFn`;

2. при наличии таких функций, с новой функцией методом `llvm::Function::setPersonalityFn` связывается обработчик `__gxx_personality_v0` — это стандартная функция для обработки ошибок в языке C++ при использовании GCC и Clang.

Таким образом применение прохода сохраняет возможность обработки исключений, которые могут быть сгенерированы в исходных функциях.

4. Эксперимент

В данном разделе представлено экспериментальное исследование разработанного обфусцирующего прохода. Задачи эксперимента:

1. Исследовать качество обфускации с использованием декомпиляторов.

2. Замерить снижение производительности программ, подвергнутых обфускации.

4.1. Тестовый стенд

При проведении эксперимента было использовано следующее программное обеспечение: LLVM 18, clang version 18.1.8, Ghidra version 11.2.1, IDA Free 9.0.

В качестве тестовых программ для оценки качества обфускации были использованы написанные для прохода тесты на языке C++. Для замеров производительности использовались тесты Linpack Benchmark на языке C [2].

Характеристики компьютера, на котором производились замеры производительности: Ubuntu 22.04.5 LTS, 11th Gen Intel® Core™ i3-1125G4 CPU, четыре ядра, 2.00GHz, 3.8Gi RAM.

4.1.1. Подготовка тестовых программ

Опишем процесс, с помощью которого получались обфусцированные версии кода при проведении эксперимента.

Сперва с помощью clang получаем обфусцированный код в LLVM IR, флаг `-fpass-plugin` указывает clang путь до прохода, который нужно применить. Далее из LLVM IR получаем бинарный файл.

[2]Исходный код теста Linpack Benchmark доступен по ссылке https://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html(дата обращения: 15 ноября 2024 г.).

Листинг 2: Сборка обфусцированного кода с помощью clang

```
$ clang++ -S -emit-llvm -fpass-plugin=<path-to-pass>
program.cpp -o merged.ll
$ clang++ merged.ll -o merged
```

4.2. Оценка качества обфускации

Для оценки качества обфускации, осуществляемой проходом, было решено использовать инструменты Ghidra и IDA, описанные в работе [10]. В частности, были использованы реализованные в них декомпиляторы.

Схема проведения эксперимента следующая:

1. Получить бинарный файл для обфусцированной и исходной версии программы.
2. Загрузить оба варианта кода в декомпилятор.
3. Проанализировать результат декомпиляции.

4.2.1. Критерии оценки

Обфускацию можно считать качественной, если декомпилятор не способен восстановить код до обфускации. При этом считается нормальным, что декомпилятор сможет восстановить структуру кода, созданную проходом.

4.2.2. Результаты

Рассмотрим подробно процесс на одной из тестовых программ, в которой реализованы три функции с одним возвращаемым типом.

Листинг 1: Тестовая программа с тремя функциями, возвращающими значение типа int

```c
int Fibonacci(int n)
{
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    return Fibonacci(n - 2) + Fibonacci(n - 1);
}

int Factorial(int n)
{
    if(n == 0 || n == 1)
        return 1;
    return n * Factorial(n - 1);
}

int Pow(int x, int y)
{
    if(y == 0)
        return 1;
    return x * Pow(x, y - 1);
}
```

Проверим, что декомпилятор Ghidra успешно справился с восстановлением функций.

Decompile: Fibonacci - (recursion)

```c
/* Fibonacci(int) */

int Fibonacci(int param_1)

{
  int iVar1;
  undefined4 local_c;

  if (param_1 == 0) {
    local_c = 0;
  }
  else if (param_1 == 1) {
    local_c = 1;
  }
  else {
    local_c = Fibonacci(param_1 + -2);
    iVar1 = Fibonacci(param_1 + -1);
    local_c = local_c + iVar1;
  }
  return local_c;
}
```

Decompile: Factorial - (recursion)

```c
/* Factorial(int) */

int Factorial(int param_1)

{
  undefined4 local_c;

  if ((param_1 == 0) || (param_1 == 1)) {
    local_c = 1;
  }
  else {
    local_c = Factorial(param_1 + -1);
    local_c = param_1 * local_c;
  }
  return local_c;
}
```

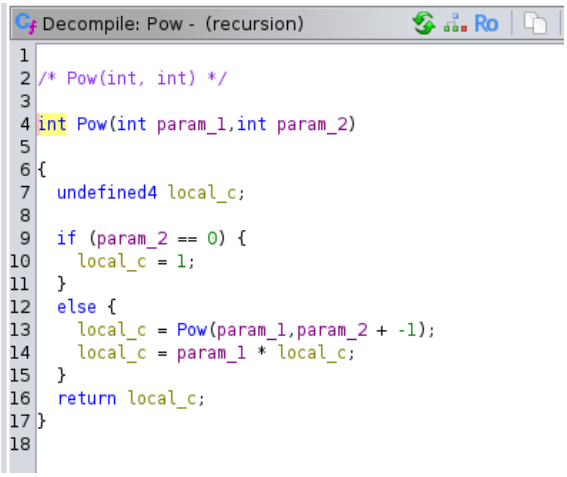

Рис. 7: Восстановленные декомпилятором Ghidra функции Fibonacci, Factorial, Pow.

Видно, что декомпилятор Ghidra правильно распознал тела всех трех функций и их сигнатуры.

Перейдем к обфускации программы. Продемонстрируем, как изменились определения функций после применения прохода.

```
; Function Attrs: mustprogress noinline optnone uwtable
define dso_local noundef i32 @_Z9Fibonaccii(i32 noundef %0) #4 {
  %2 = call i32 @_Z9Fibonaccii-_Z9Factoriali-_Z3Powii(i32 %0, i32 zeroinitializer, i32 0)
  ret i32 %2
}

; Function Attrs: mustprogress noinline optnone uwtable
define dso_local noundef i32 @_Z9Factoriali(i32 noundef %0) #4 {
  %2 = call i32 @_Z9Fibonaccii-_Z9Factoriali-_Z3Powii(i32 %0, i32 zeroinitializer, i32 1)
  ret i32 %2
}

; Function Attrs: mustprogress noinline optnone uwtable
define dso_local noundef i32 @_Z3Powii(i32 noundef %0, i32 noundef %1) #4 {
  %3 = call i32 @_Z9Fibonaccii-_Z9Factoriali-_Z3Powii(i32 %0, i32 %1, i32 2)
  ret i32 %3
}
```

Рис. 8: Объявления функции после применения прохода. Код представлен в LLVM IR.

Видно, что все тела заменены вызовом новой функции, полученной в ходе работы прохода. Последний аргумент в вызове новой функции отвечает за идентификацию.

Теперь рассмотрим результат работы декомпилятора Ghidra на обфусцированной программе.

Исходные функции Ghidra распознала следующим образом:

Decompile: Fibonacci - (recursion-merged)

```
1
2 /* Fibonacci(int) */
3
4 void Fibonacci(int param_1)
5
6 {
7   Fibonacci(int)-Factorial(int)-Pow(param_1,0);
8   return;
9 }
10
```

Decompile: Factorial - (recursion-merged)

```
1
2 /* Factorial(int) */
3
4 void Factorial(int param_1)
5
6 {
7   Fibonacci(int)-Factorial(int)-Pow(param_1,0);
8   return;
9 }
10
```

Decompile: Pow - (recursion-merged)

```
1
2 /* Pow(int, int) */
3
4 void Pow(int param_1,int param_2)
5
6 {
7   Fibonacci(int)-Factorial(int)-Pow(param_1,param_2);
8   return;
9 }
10
```

Рис. 9: Восстановленные декомпилятором Ghidra обфусцированные функции Fibonacci, Factorial, Pow.

Заметно, что декомпилятор неверно определил параметры вызова новой функции внутри Pow. Также вызов новой функции происходит с неверным дополнительным аргументом для функции Factorial.

Таким образом, декомпилятор Ghidra не смог корректно восстановить исходные функции.

Проведем те же действия для декомпилятора IDA.

```
1  __int64 __fastcall Fibonacci(int a1)
2  {
3    int v2; // [rsp+4h] [rbp-Ch]
4
5    if ( a1 )
6    {
7      if ( a1 == 1 )
8      {
9        return 1;
10     }
11     else
12     {
13       v2 = Fibonacci(a1 - 2);
14       return (unsigned int)(Fibonacci(a1 - 1) + v2);
15     }
16   }
17   else
18   {
19     return 0;
20   }
21 }
```

```
1  __int64 __fastcall Factorial(unsigned int a1)
2  {
3    if ( a1 > 1 )
4      return (unsigned int)(Factorial(a1 - 1) * a1);
5    else
6      return 1;
7  }
```

```
1  __int64 __fastcall Pow(int a1, int a2)
2  {
3    if ( a2 )
4      return (unsigned int)(Pow(a1, a2 - 1) * a1);
5    else
6      return 1;
7  }
```

Рис. 10: Результат декомпиляции функций инструментом IDA.

Можно заметить, что декомпилятор IDA также успешно справляется с восстановлением необфусцированных функций.

Рассмотрим результат декомпиляции функций, подвергнутых обфускации.

```
1  __int64 __fastcall Fibonacci(__int64 a1)
2  {
3    return Fibonacci(a1, 0LL);
4  }
```

```
1  __int64 __fastcall Factorial(__int64 a1)
2  {
3    return Fibonacci(a1, 0LL);
4  }
```

```
1  __int64 __fastcall Pow(__int64 a1, __int64 a2)
2  {
3    return Fibonacci(a1, a2);
4  }
```

Рис. 11: Результат декомпиляции функций инструментом IDA.

В итоге декомпилятор IDA совершенно не справился с восстановлением функций, присутствуют ошибки в сигнатурах функций, более того, внутри тел происходят вызовы неверной функции.

4.3. Замеры производительности

Для замеров производительности использовался тест Linpack Benchmark на языке C. Количество решаемых линейных уравнений составляло одну тысячу.

Исходный код теста подвергался обфускации методом слияния функций, после чего производились замеры времени выполнения исходной программы и обфусцированной версии. Выполнялось сорок замеров.

Результаты замеров представлены в Таблице 1.

Версия программы	**Среднее время выполнения (сек)**	**Стандартная ошибка среднего (сек)**
Исходная программа	0.682	0.004
Обфусцированная программа	0.853	0.004

Таблица 1: Сравнение среднего времени выполнения исходной и обфусцированной версий программы.

Таким образом, замедление программы составило 25%.

4.4. Выводы

По результатам проведенных экспериментов можно сделать вывод, что реализованный метод слияния функций обеспечивает приемлемый уровень обфускации, так как рассмотренные декомпиляторы оказались неспособными безошибочно восстановить исходные функции. Тем не менее замеры производительности показали немалое замедление обфусцированной программы, что необходимо учитывать при использовании прохода.

Заключение

В рамках данной учебной практики были выполнены следующие **задачи**:

1. проведен обзор существующих решений и используемых технологий;
2. реализован метод слияния функций с помощью технологии проходов LLVM;
3. проведена оценка качества работы обфускатора с помощью декомпиляторов и выполнены замеры производительности.

Обфускатор компании «Софтком» является проектом с закрытым исходным кодом.

Список литературы

[1] LLVM. — The LLVM Compiler Infrastructure. — URL: https://bcain-llvm.readthedocs.io/projects/llvm/en/latest/WritingAnLLVMPass/ (дата обращения: 22 октября 2024 г.).

[2] Obfuscator-{LLVM} – Software Protection for the Masses / Pascal Junod, Julien Rinaldini, Johan Wehrli, Julie Michielin // Proceedings of the {IEEE/ACM} 1st International Workshop on Software Protection, {SPRO'15}, Firenze, Italy, May 19th, 2015 / Ed. by Brecht Wyseur. — IEEE, 2015. — P. 3–9. — URL: https://ieeexplore.ieee.org/document/7174804 (дата обращения: 22 октября 2024 г.).

[3] Petráček Martin. LLVM obfuskátor. — 2018. — URL: https://dspace.cvut.cz/handle/10467/76160 (дата обращения: 22 октября 2024 г.).

[4] Šíma Roman. Obfuskátor nad platformou LLVM. — 2019. — URL: https://dspace.cvut.cz/handle/10467/81282 (дата обращения: 22 октября 2024 г.).

[5] Turčan Lukáš. LLVM Obfuscator Based on Virtual Machines with Custom Opcodes and String Encryption. — 2019. — URL: https://dspace.cvut.cz/bitstream/handle/10467/82300/F8-DP-2019-Turcan-Lukas-thesis.pdf?sequence=-1&isAllowed=y (дата обращения: 22 октября 2024 г.).

[6] Udupa Sharath K., Debray Saumya K., Madou Matias. Deobfuscation. Reverse Engineering Obfuscated Code. — URL: https://www2.cs.arizona.edu/~debray/Publications/unflatten.pdf (дата обращения: 9 октября 2024 г.).

[7] The current state of art in program obfuscations: definitions of obfuscation security / N.P. Varnovsky, Vladimir Zakharov, N.N. Kuzurin, Shokurov Alexander // Proceedings of the Institute for System

Programming of RAS. — 2014. — Vol. 26, no. 3. — P. 167–198. — URL: https://www.researchgate.net/publication/273519884_The_current_state_of_art_in_program_obfuscationsdefinitions_of_obfuscation_security (дата обращения: 9 октября 2024 г.).

[8] The tigress c obfuscator. — URL: https://tigress.wtf/index.html (дата обращения: 22 октября 2024 г.).

[9] А.Г. Коробейников, И.М Кутузов, П.Ю. Колесников. Анализ методов обфускации // Кибернетика и программирование. — 2012. — no. 1. — P. 31–37. — URL: https://nbpublish.com/library_read_article.php?id=13858 (дата обращения: 9 октября 2024 г.).

[10] К.А. Дугушова. Защита программного кода путём встраивания мёртвого кода и данных. — URL: https://se.math.spbu.ru/thesis/texts/Dugushova_Ksenija_Andreevna_Autumn_practice_2nd_year_2023_text.pdf (дата обращения: 9 октября 2024 г.).