Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: **Разработка модуля расширения статических проверок для компилятора Clang**

Студент гр. 43501/3 Н.А. Салин

Санкт-Петербург – 2015

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Диссертация допущена к защите
зав. кафедрой

_____ В.Ф. Мелехин

«___» _____ 2015 г.

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: **Разработка модуля расширения статических проверок для компилятора Clang**

Направление: 230100 – Информатика и вычислительная техника

Выполнил студент гр. 43501/3 _____ Н.А. Салин

Научный руководитель,
м. т. т., Ассистент _____ М.Х. Ахин

Рецензент,
к. т. н., Старший разработчик _____ Р.Е. Цензент

Консультант по нормоконтролю,
к. т. н., Старший преподаватель _____ С.А. Нестеров

Санкт-Петербург – 2015

Эта страница специально оставлена пустой.

РЕФЕРАТ

Отчет, 43 стр., 1 прил.

CLANG, AST, СТАТИЧЕСКИЙ АНАЛИЗ, МОДУЛЬ РАСШИРЕНИЯ

Целью данной работы является разработка модуля статических проверок для компилятора Clang.

Рассмотрены основные задачи статического анализа исходного кода, а также преимущества и недостатки использования инструментов для статического анализа. Сделан обзор существующих статических анализаторов, таких как PVS-Studio, Coverity, Flint и Clang Static Analyzer. Также проведен анализ абстрактного синтаксического дерева, создаваемого с помощью Clang. Разработан модуль расширения для компилятора Clang. Данный модуль представляет собой динамическую библиотеку, которая подключается к компилятору с помощью специальных флагов. Для выявления подозрительных участков кода использовалась библиотека AST matchers. Данная библиотека позволяет составлять шаблоны абстрактного синтаксического дерева Clang. При нахождении фрагмента кода соответствующего заданному шаблону вызывается обработчик для последующего анализа. В модуль расширения были добавлены проверки, которые отсутсвуют в анализаторе Clang Static Analyzer, но присутствуют в PVS-Studio, Coverity и Flint. В ходе тестирования были выявленные ошибки первого и второго рода.

ABSTRACT

Report, 43 pages, 1 appendicies

CLANG, AST, STATIC ANALYSIS, PLUGIN

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время компьютеры уже проникли во все сферы жизни человека. С каждым годом все больше возрастает сложность программного обеспечения, что влечет за собой увеличение количества программных ошибок. Некоторые программные ошибки незначительно влияют на работу программы и могут быть не обнаружены длительное время, однако другие могут привести к некорректной работе, зависанию или завершению программы. Известны случаи, когда ошибки в программе привели к серьезным последствиям. Например ошибка в ПО аппарата для лучевой терапии Therac-25 стала причиной смерти нескольких пациентов [**?**], ошибка в системе управления полетом ракеты привела к падению и стала самой дорогой в истории[**?**]. По подсчетам экономистов программные ошибки обходятся в миллиарды долларов в год.

Очевидно, что обеспечение качества ПО является одной из важнейших задач в сфере информационных технологий. Для обеспечения качества могут применяться такие методы как:

- Тестирование
- Статический анализ кода
- Динамический анализ кода
- Инспекция кода

В данной работе будет рассмотрен подход использования статического анализа исходного кода. Для этого был разработан модуль расширения статических проверок для компилятора Clang. Разработанный модуль расширения представляет собой динамическую библиотеку, которая подключается к компилятору Clang. Модуль расширения занимает небольшой объем памяти и для проведения статического анализа пользователю не нужно устанавливать специальные программы, достаточно добавить флаги для компилятора, подключающие разработанный модуль.

Для выявления фрагментов кода, потенциально содержащих ошибки, используется библиотека AST Matchers. Данная библиотека предоставляет краткий способ определения шаблонов по которым происходит поиск фрагментов кода в абстрактном синтаксическом дереве

Clang. В результате нет необходимости писать много повторяющегося код для похожих шаблонов.

Разработчики статического анализатора PVS-Studio постояно производят поиск ошибок в открытых проектах. Из проведенных анализов видно, что часто ошибки допускают по невнимательности. Поэтому при выборе проверок для выявления ошибок, были выбраны ошибки появляющиеся во время дублирования существующего исходного кода, а так же ошибки, которые допускают программисты по невнимательности. К примеру распространенной ошибкой является ошибка вида «выражение1 оператор выражение2», где выражение1 эквивалентно выражение2. Выбранные ошибки присутствуют в PVS-Studio, но не представлены в Clang Static Analyzer.

1. СТАТИЧЕСКИЙ АНАЛИЗ

1.1. Введение

Статический анализ кода — процесс выявления ошибок и недочетов в исходном коде без реального выполнения исследуемой программы. В большинстве случаев анализ производится над какой-либо версией исходного кода, однако анализу может подвергаться объектный код. В основном данный термин применяют к анализу, который производится автоматизировано, с использованием специального программного обеспечения. В настоящее время статические анализаторы являются частью всех современных компиляторов. Найденные подозрительные участки кода компилятор выдает в виде предупреждений (warnings). Однако для того чтобы провести более детальный анализ исходного кода, необходимо больше процессорного времени и памяти. Поскольку компиляция должна происходить быстро, в компиляторах используются проверки только на простые ошибки. Для выявления более сложных ошибок в исходном коде необходимо использовать специализированные инструменты для статического анализа кода. В настоящее время статический анализ широко используется в областях:

- ПО для медицинских устройств
- ПО для ядерных станций и систем защиты реактора
- ПО для авиации

1.2. Решаемые задачи

1.2.1. Выявление ошибок в программах

Статический анализ в большинстве случаев используется для выявления ошибок в исходном коде. Статический анализатор может быть как общего назначения, так и специализированным для поиска определенных классов ошибок. При исполльзовании статического анализа можно выявить значительное количество ошибок еще на этапе написания кода, что существенно снижает стоимость их исправления.

1.2.2. Рекомендации по оформлению кода

Стив МакКоннелл (Steve McConnell) сказал в своём выступлении на SD West '04: «код должен удобно читаться, а не удобно писаться»[**?**]. Грамотное оформление кода крайне важное условие для людей, работающих в одной команде. Следует помнить, что стиль и соглашения о форматировании у каждой организации свои. Обычно эти соглашения определяются на ранней стадии создании проекта. В результате соблюдения принятых соглашений, код будет легче читаться и станет более единообразным и, как следствие, уменьшится стоимость поддержки и отладки существующего кода. Статический анализ для решения данной[**?**] задачи применяют как для контроля уже давно работающих сотрудников, так и для обучения новых, которые недостаточно хорошо знакомы с принятыми соглашениями.

1.2.3. Подсчет метрик

Метрики кода представляют собой набор оценок программного обеспечения, которые дают разработчикам более глубокое представление о разрабатываемом коде. Например с помощью метрик кода разработчики могут оценить сложность разработанного ПО, понять какие места необходимо переработать или более тщательно тестировать. Примеры метрик:

1. Колличественные метрики (количество строк кода, количество использованных операторов и др.)

2. Связность

1.2.4. Поиск уязвимостей

Статический анализ можно успешно применять для поиска уязвимостей в ПО (Static Application Security Testting, SAST). Однако не только разработчики программы могут использовать статический анализ. В случае, если исходный код находится в открытом доступе или был украден, злоумышленники могут использовать статический анализ для нахождения потенциальных уязвимостей. Вместо того чтобы искать слабые места программы вслепую или просматривать огромное количество исходного кода, куда проще провести статический анализ. После нахождения слабых мест, злоумышленник может начать

исследование слабых мест для атаки. Хотя в большинстве случаев хакеру не доступен исходный код, все равно имеется возможность проведения статического анализа с помощью специальных анализаторов, которые работают с двоичным кодом.

1.2.5. Другие

Перечисленные выше задачи — не единственное применение для статического анализа. К примеру при написании кроссплатформенного кода довольно не просто учесть все особенности каждой ОС или аппаратной платформы. На помощь в данном случае могут придти специализированные анализаторы, которые смогут найти большинство небезопасных участков кода. Так же статический анализ можно использовать в обучении. Чтобы проверить работу ученика на ошибки не нужно просматривать весь исходный код вручную, взамен можно автоматизировать проверку всех работ сразу. Это очень удобно, когда у преподавателя много учеников и на ручную проверку всех работ уйдет много времени.

1.3. Преимущества и недостатки

Как и любой подход, статический анализ не является универсальным и имеет как преимущества, так и недостатки. К преимуществам можно отнести:

- Полное покрытие кода. В отличии от динамического анализа, статические анализаторы проверяют даже те участки кода, которые получают управление крайне редко.

- Независимость от компилятора и среды выполнения. В виду того, что статический анализ производится над исходным кодом без реального выполнения, появляется возможность находить скрытые ошибки. Данные ошибки могут появляться в определенных ситуациях и зависеть от реализации компилятора или заданных ключей компилятора. Примером скрытых ошибок могут служить ошибки «неопределенного поведения» (англ. undefined behaviour).

- Возможность использования на ранних этапах жизненного цикла ПО. Основным преимуществом является возможность на-

хождения программных ошибок в программе на раннем этапе. Вследствие чего происходит существенное уменьшение стоимости исправление ошибки. Так же данное преимущество удобно для проектов больших встраиваемых систем, в случае если невозможно использовать средства динамического анализа до тех пор, пока ПО не будет готово к запуску на целевой системе.

- Низкие стоимостные затраты. Для того, чтобы использовать статический анализ, не нужно тратить дополнительные средства на создание тестовых программ или фиктивных модулей (stubs).

Недостатки:

- Ложные срабатывания. При поиске ошибок статический анализатор пытается предсказать поведение программы, используя ее исходный код. Это приводит к тому, что происходит множество ложных срабатываний. В большинстве случаев подозрительный участок кода будет компилироваться и работать верно. Для определения является ли подозрительный участок кода ошибкой или нет, необходима помощь человек. В результате просмотр ложных срабатываний отнимает много времени и может отвлекать от просмотра участков кода, где есть реальная ошибка.

- Скорость. Поскольку для нахождения потенциальных ошибок необходимо проводить детальный анализ каждого участка кода, анализ может занимать значительное время. Поэтому обычно статический анализ используется при «ночных сборках».

- Статические анализаторы полностью не заменяют ручной аудит кода. Такие категории дефектов, как логические ошибки, уязвимости и проблемы с производительностью, могут быть обнаружены только экспертом или инструментами динамического анализа.

1.4. Существующие статические анализаторы

Существует множество статический анализаторов кода, у каждого есть свои плюсы и минусы. Ниже будут рассмотрены некоторые из существующих анализаторов. С большим списком синтаксических анализаторов можно ознакомиться на сайте Wikipedia[**?**].

1.4.1. PVS-Studio

PVS-Studio это статический анализатор для языка C/C++ отечетсвенной разработки. В нем можно выделить 4 набора правил для диагностики ошибок:

1. Диагностика общего назначения
2. Диагностика возможных оптимизаций
3. Диагностика 64-битных ошибок (Viva64)
4. Диагностика параллельных ошибок (VivaMP)

Подробный список осуществляемых проверок можно найти на сайте PVS-Studio[**?**].

Одним из главным преимуществ является интеграция с Microsoft Visual Studio начиная с версии 2005. При этом анализатор поставляется в виде плагина и для начала работы с анализатором не нужно проводить настройку. Для удобства использования предоставляется пользовательский интерфейс для навигации по коду, анализа файлов и получения справочной информации. Также разработчиками реализована работа анализатора на многоядерных процессорах, что существенно ускоряет процесс нахождения ошибок.

Однако из преимуществ PVS-Studio вытекают и недостатки. Так как анализатору для работы необходима Visual Studio, то недостатками является:

- Анализ можно производить используя только операционную систему Windows
- Для возможности использования плагинов не подходит бесплатная версия Visual Studio (Express)

Хотя на сайте разработчиков есть описание того, как можно произвести запуск PVS-Studio на Linux, но официально нет поддержки Linux.

Для демонстрации возможностей PVS-Stduio разработчики постоянно проводят проверки проектов с открытым исходным кодом. Со списком уже проверенных проектов и найденными ошибками можно ознакомиться на официальном сайте[**?**].

1.4.2. Coverity

Coverity является набором программ, которые используются для выявления и исправления дефектов безопасности и качества в программах критического назначения. Одной из программ выявления дефектов является статический анализатор Coverity Code Advisor умеющий находить дефекты и уязвимости в исходном коде, написанном на языке C,C++, Java и C#.

Отличительной особенностью Coverity является минимальное количество ложных срабатываний, большинство выдаваемых предупреждений действительно соответствует дефектным участкам кода, которые в последствии могут привести к серьезным ошибкам.

Coverity Code Advisor может выявлять следующие дефекты:

- Ошибки использования API
- Переполнение буффера
- Неправильная обработка ошибок
- Переполнения целлых типов данных
- Проблемы с производительностью
- Неинициализированные переменные
- SQL инъекции
- Разименование нулевого указателя
- Другие[**?**]

Недостатком данного анализатора является высокая цена, однако есть бесплатный онлайн сервис Coverity Scan доступный для зарегистрированных open source проектов. Данный сервис начал сотрудничать со Стенфордским университетом с 2006 года и в течении первого года было обнаружено и устранено более 6000 программных ошибок в 50 C/C++ open source проектах[**?**].

Статический анализатор Coverity использовали в ЦЕРНе для анализа программ, использующихся в большом адронном коллайдере[**?**]. В результате было найдено и устранено более 40000 программных ошибок, которые могли повлиять на точность физических исследований частиц. Так же статический анализ широко использовался во

время разработки программного обеспечения для полета марсохода Curiosity[**?**].

1.4.3. Flint

Flint — это статический анализатор от компании Facebook с открытым исходным кодом[**?**] предназначенный для анализа программ написанных на C++. Изначально Flint был написан на языке C++, но затем был переписан на язык D, в следствии чего ускорилось время анализа исходного кода.

Так как Flint разрабатывался для внутрених нужд компании, в качестве доступных проверок используются только часто встречающиеся ошибки:

1. Запрет на идентификаторы из черного списока
2. Инициализация переменной самой собой
3. Все исключения должны передаваться по ссылке
4. Ошибки при определении конструктора
5. Передача небольших типов по значению
6. Другие[**?**]

Хотя Flint не сможет найти большинство ошибок как PVS-Studio или Coverity, большим достоинством является открытость кода. Если необходимо добавить собственную проверку, это легко сделать.

1.4.4. Clang Static Analyzer

Clang Static Analyzer является инструментом для статического анализа, который находит ошибки программ написанных на языке C,C++ и Objective-C. В данный момент имеется возможность использовать статический анализатор как отдельную программу или в интегрированной среде разработки XCode. Отдельная программа запускается из командной строки и разработана для запуска вместе с компиляцией исходного кода. Весь исходный код анализатора открыт и является частью проекта Clang. Анализатор реализован как C++ библиотека, которая может быть использована другими инструментами и приложениями.

В настоящее время ведется активная разработка и производятся постоянные улучшения как для повышения точности, так и для увеличения возможностей алгоритмов анализа.

Доступные проверки разделены на 6 категорий:

1. Базовые проверки. Производятся проверки общего назначения, такие как деление на 0, разыменование NULL указателя, использование неинициализированных переменных и так далее

2. C++ проверки. Сюда входят проверки, специфичные для языка C++

3. Проверки для нахождения неиспользуемого кода.

4. OS X проверки. Данные проверки необходимы для нахождения ошибок, специфичных для языка Objective-C и ошибок неправильного использования Apple SDK (OS X и iOS)

5. Проверки безопасности. Используются для выявления небезопасного использования API. Производятся на основе CERT Secure Coding Standards

6. Unix проверки. Необходимы для проверки корректного использования Unix и POSIX API

С полным списком доступных проверок можно ознакомиться на сайте анализатора[**?**].

1.4.5. Выводы

Подводя итог вышесказанного, статический анализатор является удобным инструментом для поиска ошибок. Проанализировав рассмотренные анализаторы, было принято решение создать статический анализатор, который легко интегрировать в существующую среду сборки. Так же должна иметься возможность добавления новых проверок к уже реализованным.

2. КОМПИЛЯТОР CLANG И МОДУЛИ РАСШИРЕНИЯ

2.1. Введение

Clang[**?**] является компилятором для языков программирования C, C++, Objective-C, Objective-C++ и OpenCL. Для оптимизации исходного кода и кодогенерации используется фреймворк LLVM[**?**]. Хотя Clang разрабатывается как фронтенд для LLVM, теоретически возможно использовать и другие бэкенды. К примеру в качестве бэкенда можно использовать GCC. Однако комбинация Clang и LLVM предоставляет набор инструментов, позволяющий полностью заменить GCC.

Разработка Clang началась ввиду того, что был необходим компилятор, который предоставлял бы детальные диагностики ошибок, удобную интеграцию с интегрированной средой разработки и имел бы лицензию, позволяющую использовать компилятор в коммерческих целях. В результате в 2007 году был представлен компилятор Clang под лицензий University of Illinois/NCSA Open Source License. Для интегрированной среды разработки XCode, Clang становится основным компилятором начиная с версии 3.2. С ноября 2012 года в качестве основного компилятора для FreeBSD используется Clang. Вероятно в ближайшие годы Clang сменит GCC и будет основным компилятором для языков C и C++ во многих диструбутивах и других Unix-подобных системах.

Одной из главных целей Clang является поддержка инкрементной компиляции, которая позволяет более тесно интегрировать компилятор и среду разработки. Для GCC это непросто сделать так как он разрабатывался для использования в классическом цикле «компиляция-линковка-отладка». Поэтому Clang стремится предоставить фреймворк, позволяющий производить парсинг, индексацию, статический анализ и компиляцию языков семейства C.

Ввиду того, что изначально Clang был спроектирован для максимального сохранения информации во время процесса компиляции, появляется возможность предоставлять контекстно-ориентированные сообщения об ошибках. Такие сообщения понятны программистам и удобны для сред разработки. Так же благодаря модульности дизай-

на компилятора, можно использовать необходимые модули в среде разработки для индексирования кода, подсветки синтаксиса и рефакторинга.

Подводя итог выше сказанного, можно выделить основные цели использования Clang:

- Особенности для конечного пользователя:
  - Быстрая компиляция и небольшое использование памяти.
  - Понятные диагностики.
  - Совместимость с GCC.
- Преимущества для приложений:
  - Модульная архитектура.
  - Поддержка разнообразных инструментов (рефакторинг, статический анализ, генерация кода и так далее).
  - Тесная интеграция с средой разработки.
  - Использование BSD-подобной лицензии.
- Внутренний дизайн и реализация:
  - Простой и легко изменяемый исходный код.
  - Унифицированный парсер для языков C, Objective-C, C++ и Objective-C++.
  - Поддержка последних версий C/C++/Objective-C.

С полным и подробным списком всех особенностей можно ознакомиться на сайте Clang[**?**].

Clang разработан как набор библиотек, которые могут быть использоваться независимо друг от друга. К примеру, если необходимо создать препроцессор, достаточно использовать библиотеки «Basic» и «Lex». Для инструментов рефакторинга или статического анализа есть библиотеки для получения абстрактного синтаксического дерева.

В данной работе будет производится анализ на основе абстрактного синтаксического дерева. В отличии от GCC и TCC, Clang предоставляет средства для получения абстрактного синтаксического дерева в несокращенном виде, что особенно удобно для разрабатываемого модуля статических проверок.

2.2. Структура абстрактного синтаксического дерева Clang

Абстрактным синтаксическим деревом (Abstract Syntax Tree) называется представление исходного кода в виде дерева, в котором вершинами являются операторы языка программирования, а листьями - операнды. Листья могут представлять только переменные и константы.

Узлы в абстрактном синтаксическом дереве Clang организованы так, что у них нет общего предка. Вместо этого, есть несколько больших иерархий классов для простых типов, таких как *Decl* и *Stmt*.

- Decl представляет объявления (declarations). От данного класса наследуются разнообразные объявления типов. К примеру класс FunctionDecl отвечает за объявление функции, а класс ParmVarDecl за объявление параметра функции.

- Stmt представляет операторы (statements). Все классы, описывающие операторы, наследуются от данного класса. К примеру для оператора «if» существует класс IfStmt, оператору «return» соответствует класс ReturnStmt.

Многие важные AST узлы наследуются от классов *Type, Decl, DeclContext* или *Stmt*, а некоторые классы наследуются от *Decl* и *DeclContext*. В Clang AST выражения (expressions) представляются классом Expr, наследуются от класса Stmt и являются операторами. Однако существует множество классов, которые не являются частью большой иерархии классов и доступны только из определенных узлов, к примеру класс CXXBaseSpecifier. Так же важно помнить, что комментарии не входят в состав абстрактного синтаксического дерева.

Все информация о абстрактном синтаксическом дереве для единицы трансляции (translation unit) собрана в классе ASTContext. Он представляет обход всей единицы трансляции или доступ к таблице идентификаторов Clang для разобранной единицы трансляции.

2.3. Способы использования Clang

Clang предоставляет необходимую инфраструктуру для создания инструментов, которым необходима синтаксическая и семантическая информация из исходного кода. Для того, чтобы использовать все

возможности Clang есть три способа создания инструментов: использование библиотеки LibClang, использование библиотеки LibTooling или создание плагина для Clang. Рассмотрим каждый из методов.

LibClang

LibClang представляет стабильный высокоуровневый интерфейс к Clang, написанный на языке C. Является хорошим выбором если необходим стабильный API, так как Clang периодически меняется. В случае если использовать LibTooling или создавать плагин, то нужно обновлять написанный код, чтобы он соответствовал изменениям в Clang. Так же LibClang позволяет использовать другие языки программирования для доступа к Clang API, а не только C++. Однако LibClang не предоставляет полного доступа к абстрактному синтаксическому дереву. Для работы с Clang предоставляется небольшой набор типов и функций, посредством которых происходит взаимодействие с ядром. Поэтому многие средства недоступны или предоставляются в урезонном виде.

LibTooling

LibTooling это C++ интерфейс, используемый для написания приложения. Типовые примеры использования LibTooling:

- Простое приложение для проверки синтаксиса
- Инструменты рефакторинга

В отличии от LibClang, предоставляется полный доступ к абстрактному синтаксическому дереву. Анализировать можно как один файл, так и заданный набор файлов, независимо от системы сборки.

Примерами использования LibTooling являются Clang Tools. Clang Tools — это коллекция специальных инструментов для разработчика, используемых для автоматизации и улучшения разработки на языках C и C++. Примеры созданных или планируемых инструментов как часть проекта Clang:

- Проекра синтаксиса (clang-check)
- Автоматическое исправление ошибок компиляции (clang-fixit)
- Автоматическое форматирование кода (clang-format)

- Инструменты для миграции кода под новый стандарт языка (clang-modernize)
- Инструменты для рефакторинга

Модуль расширения для Clang

Использование плагина для Clang позволяет производить дополнительные действия при обходе абстрактного синтаксического дерева как часть процесса компиляции. Плагины представляют собой динамические библиотеки, которые могут подключаться к компилятору во время выполнения. Благодоря динамическому подключению к компилятору, плагин легко интегрировать в среду сборки проекта. Так же плагин имеет возможность останавливать процесс сборки. Как и при использовании LibTooling, плагину предоставляется полный контроль над абстрактным синтаксическим деревом.

Однако стоит помнить, что нецелесообразно создавать плагин, если:

- Необходимо использовать плагин вне среды сборки
- Необходимо использовать плагин для определенного набора файлов в проекте, которые не обязательно вызывают пересборку проекта

Выводы

Как видно из проведенного анализа методов взаимодействия с Clang, самым удобным способом является создание модуля расширения. Для интеграции в существующую среду разработки достаточно подключить созданный модуль. Также модулю расширения предоставляется полный доступ ко всем возможностям ядра Clang.

2.4. Выявление фрагментов кода на основе AST

Чтобы найти фрагменты кода, содержащие ошибку, будем использовать абстрактное синтаксическое дерево. Для выявления ошибок необходимо найти все совпадения узлов, соответсвующих фрагменту кода с ошибкой, в абстрактном синтаксическом дереве. Для обхода и нахождения интересующих мест в абстрактном синтаксическом дереве Clang есть два подхода:

- RecursiveASTVisitor
- ASTMatcher

Рассмотрим как с помощью каждого подхода найти интересующий фрагмент кода. В качестве примера будет рассмотрено нахождение фрагмента кода, содержащий оператор if и в качестве условия происходит сравнение указателя и 0:

```
int main()
{
    int var = 5;
    int *ptr = &var;
    if (ptr < 0) // <==
        var++;
    return 0;
}
```

Абстрактное синтаксическое дерево для интересующего фрагмента кода:

```
-IfStmt 0x7fdb75024790 <line:18:2, line:19:6>
|-<<<NULL>>>
|-BinaryOperator 0x7fdb75024720 <line:18:6, col:12> 'int' '<'
| |-ImplicitCastExpr 0x7fdb750246f0 <col:6> 'int *' <LValueToRValue>
| | `-DeclRefExpr 0x7fdb750246a8 <col:6> 'int *' lvalue Var 0x7fdb750245f0
'ptr' 'int *'
| `-ImplicitCastExpr 0x7fdb75024708 <col:12> 'int *' <NullToPointer>
|   `-IntegerLiteral 0x7fdb750246d0 <col:12> 'int' 0
|-UnaryOperator 0x7fdb75024770 <line:19:3, col:6> 'int *' postfix '++'
| `-DeclRefExpr 0x7fdb75024748 <col:3> 'int *' lvalue Var 0x7fdb750245f0
'ptr' 'int *'
`-<<<NULL>>>
```

2.4.1. RecursiveASTVisitor

Класс RecursiveASTVisitor позволяет обойти каждую вершину абстрактного синтаксического дерева. Для этого необходимо создать класс, который будет отвечать за обход абстрактного синтаксического дерева. Данный класс должен наследоваться от класса RecursiveASTVisitor. Чтобы обойти интересующие вершины типа *NodeType*, достаточно переопределить функцию «bool VisitNodeType(NodeType *)». Так для обхода всех Stmt необходимо переопределить функцию «bool VisitStmt(Stmt *)». Так же можно переопределить функцию для поиска вершин, которые являются подклассами Stmt. К примеру для нахождения оператора «if» нужно переопределить функцию «bool VisitIfStmt(IfStmt *)». В этом случае

для оператора «if» будет вызываться два обработчика: VisitStmt и VisitIfStmt.

Для Visit* функций необходимо возвращать значение «true», чтобы продолжить обход остальных вершин абстрактного синтаксического дерева. Если же возвратить значение «false», обход дерева полностью прекращается и Clang завершает работу.

Рассмотрим как стоит использовать RecursiveASTVisitor для нахождения участка кода из вышеописанного примера. Проанализировав полученное для исходного кода абстрактное синтаксическое дерево видно, что корнем для искомого фрагмента является IfStmt. Поэтому в классе, созданном для обхода AST следует переопределить функцию с именем "VisitIfStmt". Так же нетрудно заметить, что условием оператора "if"должен быть оператор сравнения (BinaryOperator), в котором в левой части находится указатель, а в правой константа, равная 0. Между левой и правой частью оператора сравнения должен быть символ «<». Основываясь на вышесказанном, можно написать код для выявления заданного шаблона.

```
class Example:
    public RecursiveASTVisitor<Example>
{
public:
bool VisitIfStmt(IfStmt *s) {
  if (const BinaryOperator *binOP =
    llvm::dyn_cast<BinaryOperator>(s->getCond()))
  {
    if (binOP->getOpcode() == BO_LT)
    {
      const Expr *LHS = binOP->getLHS();
      if (const ImplicitCastExpr *Cast =
            llvm::dyn_cast<ImplicitCastExpr>(LHS))
      {
        LHS = Cast->getSubExpr();
      }

      const Expr *RHS = binOP->getRHS();
      if (const ImplicitCastExpr *Cast =
            llvm::dyn_cast<ImplicitCastExpr>(RHS))
      {
```

```
        RHS = Cast->getSubExpr();
      }

      if (const DeclRefExpr *dRef=
            llvm::dyn_cast<DeclRefExpr>(LHS) &&
          const IntegerLiteral *intVal =
            llvm::dyn_cast<IntegerLiteral>(RHS))
      {
        if (const VarDecl *Var =
        llvm::dyn_cast<VarDecl>(dRef->getDecl()))
        {
          if (Var->getType()->isPointerType() &&
            intVal->getValue().getZExtValue() ==
               0)
          {
            s->dump();  // found
          }
        }
      }
    }
  }
  return true;
}
};
```

Как видно для нахождения простого шаблона нужно написать довольно много кода. Главной проблемой является то, что данный код трудно читать. Чтобы понять, какой шаблон ищет приведенный код, необходимо вчитываться и смотреть, что делает каждая строчка кода.

2.4.2. ASTMatcher

С недавного временив Clang появилась библиотека ASTMatcher, которая предоставляет простой, мощный и краткий способ определять шаблоны в абстрактном синтаксическом дереве. Реализованные как предметно ориентированный язык на основе макросов и шабло-

нов, matchers[1] похожи на алгебраические типы данных[2] из функциональных языков программирования.

К примеру для исследования только бинарных операторов есть matcher который делает именно это и называется "binaryOperator". Для того, чтобы исследовать только бинарный оператор, в левой части у которого используется литера 0, необходимо написать следующий matcher:

```
binaryOperator(hasLHS(integerLiteral(equals(0))))
```

Данный matcher не будет находить фрагменты кода, где используются другие формы 0, такие как '\0' или NULL, но matcher будет работать, если используется макрос, который раскрывается в 0. Matcher так же не будет срабатывать если используется перегруженный бинарный оператор, так как для этого есть специальный matcher, который называется "operatorCallExpr"и используется для обработки перегруженных операторов.

Есть три простых категории для matchers:

1. *Node Matchers.* Используются для поиска в AST по определенному типу. Любой шаблон определяющий выражение должен начинаться с Node Matcher, который затем можно уточнить с помощью Narrowing Matcher или Traversal Matcher. Все Traversal Matcher используют Node Matcher в качестве аргументов. Так же только Node Matcher имеют метод "bind"для связывания заданного узла и имени, для того чтобы в дальнейшем можно было получить найденный узел по имени.

2. *Narrowing Matchers.* Используются для поиска по атрибутам вершины абстрактного синтаксического дерева. Есть специальные логические Narrowing Matchers которые позволяют пользователям более мощные выражения.

3. *Traversal Matchers.* Используются для нахождения узла по связям между узлами абстрактного синтаксического дерева. Есть

[1] Matchers —шаблоны абстрактного синтаксического дерева Clang, созданные с использованием библиотеки ASTMatcher

[2] Алгебраический тип данных —тип данных получаемый из других типов с помощью алгебраических операций

специальные Traversal Matchers которые работают для всех узлов и позволяют пользователям писать более универсальные выражения.

Для полного списка всех доступных AST Matchers смотрите документацию.

Если Matcher описывает сущность в абстрактном синтаксическом дереве Clang и может быть связан с именем, то на него можно получить ссылку когда шаблон найден. Для этого нужно вызвать метод "bind"на нужном шаблоне. К примеру следующий шаблон находит все переменные типа int и связывает с этой переменной имя "intVar":

```
variable(hasType(isInteger())).bind("intvar")
```

Для обработки фрагментов кода, удовлетворяющих заданному шаблону необходимо создать класс обработчик. Данный класс должен наследоваться от класса MatchCallback. Класс MatchCallback является вложенным классом для класса MatchFinder. У класса MatchCallback есть три функции, которые можно переопределить:

1. **void run(const MatchResult &Result)**. Является обязательной для переопределения. Вызов данной функции происходит каждый раз, когда был найден участок кода, соответствующий заданному шаблону

2. **void onStartOfTranslationUnit()**. В отличии от функции "void run(const MatchResult &Result)"данная функция не является обязательной для переопределения. Вызывается каждый раз в начале разбора для каждой единицы трансляции

3. **void onEndOfTranslationUnit()**. Функция "void onEndOfTranslationUnit()"аналогична ранее рассмотренной функции "void onStartOfTranslationUnit()"с единственным отличием, что вызывается каждый раз в конце разбора для каждой единицы трансляции.

Как видно из заголовка функции "run в качестве параметра данной функции передается ссылка на результат класс MatchResult. Данный класс необходим для получения всей информации при совпадении абстрактного синтаксического дерева с заданным шаблоном.

За работу по нахождению участков кода, соответствующих заданному шаблону отвечает класс MatchFinder. Для регистрации обработчика и шаблона, который данный обработчик будет анализировать необходимо использовать функцию *addMatcher*. Первым параметром данной функции является шаблон который необходимо выявить в абстрактном синтаксическом дереве. Вторым параметром необходимо передать указатель на экземпляр класса обработчика.

Рассмотрим насколько просто нахождение участка кода из вышеописанного примера нахождения сравнения указателя и 0 с использованием ASTMatcher.

Как уже было рассмотрено ранее, корнем для искомого фрагмента кода должен быть IfStmt обозначающий оператор "if". Затем условием оператора "if"должен быть оператор сравнения (BinaryOperator), в левой части которого находится указатель, а в правой литера 0. В результате AST matcher будет выглядеть следующим образом:

```
ifStmt( hasCondition( binaryOperator(
    hasOperatorName("<"),
    hasLHS(ignoringParenImpCasts(declRefExpr(
    to(varDecl(hasType(pointsTo(AnyType)))))),
    hasRHS(ignoringParenImpCasts(
        integerLiteral(equals(0))))
    ))).bind("if")
```

Обработчик, вызываемый при нахождении данного совпадения выглядит так:

```
class Example:
public MatchFinder::MatchCallback
{
public:
void run(const MatchFinder::MatchResult &Result)
{
const IfStmt *s =
    Result.Nodes.getNodeAs<clang::IfStmt>("if");
    if (s)
      s->dump();
  }
};
```

Как видно необходимо писать намного меньше кода, чем при использовании RecursiveASTVisitor, вследствии чего труднее допустить ошибки. Так же важным преимуществом использования AST Matcher является наглядность. Увидев такой шаблон, можно сразу понять какому исходному коду он соответствует.

3. РАЗРАБОТКА МОДУЛЯ ДЛЯ CLANG

3.1. Список доступных проверок

3.1.1. Одинковые условия

Часто во время написания кода программист может допустить опечатку и не заметить следующую логическую ошибку: в исходном коде имеется один из операторов сравнения, логический или побитовый, в левой и правой части которого находятся одинаковые выражения.

Рассмотрим пример:

```
if (pos.x==0 && pos.x==0)
```

В данном случае слева и справа от оператора && расположены одинаковые выражения pos.x==0, что скорее всего свидетельствует о наличие ошибки, допущенной из-за невнимательности. Корректный код, не вызывающий ошибок должен выглядеть так:

```
if (pos.x==0 && pos.y==0)
```

Однако невсегда при использовании двух идентичных выражений выдается данное предупреждение. При использовании сравнения двух идентичных выражений типа float, double или long double можно определить, является ли значение NaN:

```
bool isnan(double X) { return X != X; }
```

Для приведенного выше прмера кода не будет выдано предупреждение.

3.1.2. Одинаковые ветки if-else

Данное предупреждение выдается на фрагмент кода, в котором полностью совпадают истинная и ложная ветка оператора "if". Такую опечатку нетрудно допустить при копировании кода.

Пример:

```
if (isCorrect())
    flags |= CORRECT;
```

```
else
    flags |= CORRECT;
```

Скорее всего, это неверный код и его стоит пересмотреть.

3.1.3. Одинаковые уловия в if-else-if

Данная ошибка часто появляется в результате множественного копирования исходного кода после чего некоторые условия не были исправлены и остались прежними.

Рассмотрим пример:

```
if (x==1)
    Func1();
else if (x==2)
    Func2();
else if (x==1)
    Func3();
```

В приведенном выше коде функция "Func3()"никогда не будет вызвана так как условия "x==1"уже проверялось. Корректный код должен быть:

```
if (x==1)
    Func1();
else if (x==2)
    Func2();
else if (x==3)
    Func3();
```

3.1.4. Ошибки использования функции memset

Для обнуления области памяти часто используется функция memset. Но из-за невнимательности программист может перепутать местами второй и третий аргумент, в результате чего не происходит обнуления:

```
memset(buf,sizeof(buf),0)
```

Помимо приведенной выше ошибки, часто неопытные специалисты не знают, что третьим аргументом функции memset принимается значение в байтах.

Рассмотрим пример:

```
int arr[10];
memset(arr, 0, 10);
```

В данном примере только первые 10 байт массива arr будут заполнены 0. Анализатор выдает предупреждения, только в том случае, когда третий аргумент не является кратным размеру типа первого аргумента. Если первым аргументом передается указатель, имеющий тип не char, short, int или long, а третьим константа, то выдается предупреждение, что не стоит использовать константу.

3.1.5. Неверный размер для выделения памяти под строку

Часто многие начинающие и даже опытные программист забывают что в C/C++ строки должны заканчиваться терминальным символом ’
0’. В результате появляется ошибка недостаточного выделения памяти под строку.

Пример:

```
char * str = (char)malloc(strlen(name));
strcpy(str, name);
```

В результате выполнения приведенного кода будет выход за предел выделенной памяти. Поэтому необходимо использовать следующий код:

```
char * str = (char)malloc(strlen(name)+1);
strcpy(str, name);
```

Ошибка так же находится в случае, если для выделения памяти используется оператор **new**

3.1.6. Опечатка при использовании strlen

Как уже описывалось ранее, для хранения строк в C/C++ необходимо не забыть про наличие терминального символа ’

0'. Однако иногда из-за невнимательности можно поставить неправильно закрывающуюся скобку:

```
char * str = (char)malloc(strlen(name+1));
strcpy(str, name);
```

В результате выделяется на два байта меньше памяти, чем необходимо.

Предупреждение о потенциальной ошибке выдастся только в случае, если в качестве аргумента strlen является выражение сложения, правой частью которого является литера **1**.

3.1.7. Потенциальная ошибка при использовании strcmp в условии

Часто по незнанию начинающие специалисты не знают как работают функции "strcmp"и "strncmp". Данные функции используются для сравнения двух строк и возвращают 0, если строки совпадают. Рассмотрим пример:

```
if (strcmp(str1, str2))
```

Условие в операторе if выполняется только в том случае, когда строки не совпадают. Однако человек, редко использующий данные функции может подумать, что условие истинно, когда строки совпадают. Хотя приведенный код не содержит ошибку, лучше переписать его следующим образом:

```
if (strcmp(str1, str2) != 0)
```

3.1.8. Одинаковые аргументы функций strcmp,strncmp,memmove,memcpy,strstr

Часто в программе происходит вызов функции, в которую передается два одинаковых аргумента и для многих функций такая ситуация является нормальной. Однако для таких функций как strcmp,strncmp,memmove,memcpy,strstr такая ситуация является подозрительной.

3.1.9. Выражение внутри sizeof

Выражения написанные внутри оператора sizeof не выполняются, поэтому такой фрагмент кода является подозрительным.

Рассмотрим пример:

```
int i=0;
size_t normalSize = sizeof(i++);
size_t doubleSize = sizeof(i *2);
```

В результате выполнения переменная i не увеличится и останется равной 0, а переменная doubleSize будет равна переменной normalSize, хотя скорее всего в данном случае программист надеялся получить значение в два раза больше.

3.1.10. Перемножение операторов sizeof

Исходный код, в котором происходит перемножение двух операторов sizeof(), практически всегда свидетельствует о наличии ошибки.

Рассмотрим пример:

```
TCHAR buf[256];
DWORD count = sizeof(buf)*sizeof(TCHAR);
```

В данном коде тип TCHAR определяется во время компиляции и в зависимости от настроек может иметь размер один и более байт. В результате при размере THCAR больше одного байта, количество элементов буфера будет неверным.

Для корректного определения количество элементов необходимо использовать следующий код:

```
TCHAR buf[256];
DWORD count = sizeof(buf)/sizeof(TCHAR);
```

3.1.11. Выделение памяти для одного простого типа с инициализацией

Если в коде происходит динамическое создание одного единственного целочисленного объекта с инициализацией, то скорее всего произошла опечатка и программист хотел использовать квадратные скобки вместо круглых.

Пример:

```
int *arr = new int(N);
```

Предупреждение появляется только при попытке создания объекта с типом char, short, int или long.

3.1.12. Сравнение указателя и 0

Анализатор выдает предупреждение при нахождении сравнения указателя и литеры 0 с использованием оператора $<$.

Пример кода:

```
class ResourceManager
{
public:
    SomeObj* Find(const char * filename);
    void LoadResource(const char * filename);
    ...
} resources;
...
if (resources.Find("foo") < 0)
    resources.LoadResource("foo");
```

Приведенный выше код мог легко появится в процессе рефакторинга если раньше был следующий код:

```
class ResourceManager
{
public:
    int Find(const char * filename);
    SomeObj* Get(int index);
    void LoadResource(const char * filename);
    ...
} resources;
...
if (resources.Find("foo") < 0)
    resources.LoadResource("foo");
```

Метод "Find"изначально возвращала индекс, по которому можно было получить ресурс, если же ресурс не был найден, возвращалось -1.

Но после рефакторинга метод "Find"стал возвращать указатель на ресурс или 0, в результате метод LoadResource никогда не будет вызван.

3.2. Динамическое подключение проверок

Не всегда удобно использовать все доступные проверки. В случае если какая-то проверка на ошибки выдает слишком много ложных предупреждений, ее лучше отключить и сосредоточиться на более важных ошибках.

Для включения нужных проверок необходимо использовать следующие флаги:

- **-eqBin** Одинаковые условия, см. 3.1.1
- **-eqStmt** Одинаковые ветки if-else, см. 3.1.2
- **-eqCond** Одинаковые условия в if-else-if, см. 3.1.3
- **-memset** Неверное использование memset, см. 3.1.4
- **-allocStr** Неверный размер для выделения памяти под строку, см. 3.1.5
- **-strlen** Опечатка при использовании strlen, см. 3.1.6
- **-strcmp** Потенциальная ошибка strcmp, см. 3.1.7
- **-eqArgs** Одинаковые аргументы функций, см. 3.1.8
- **-sizeof** Выражение внутри sizeof, см. 3.1.9
- **-sizeofMul** Перемножение операторов sizeof, см. 3.1.10
- **-new** Выделение памяти для одного простого типа, см. 3.1.11
- **-ptrCmp** Сравнение указателя и 0, см. 3.1.12
- **-all** Включение всех вышеперечисленных проверок

Для передачи каждого флага в плагин необходимо использовать следующий формат:

```
-plugin-arg-<имя плагина> <флаг>
```

Где <имя плагина> соответствует имени плагина, которое задается в исходном коде при регистрации плагина, а <флаг> соответствует флагу, который получет плагин. Так для того, чтобы включить проверки одинаковых условий (3.1.1) и сравнения указателя с 0 (3.1.12), нужно передать Clang такие ключи:

```
-plugin-arg-CppSpotter -eqBin \
-plugin-arg-CppSpotter -ptrCmp
```

3.3. Добавление новых проверок

Добавление новой проверки происходит в три этапа:

1. Создание класса обработчика. Данный класс необходим для обработки подозрительных мест в исходном коде и в случае нахождения ошибки отвечает за отображение предупреждения. Созданный класс должен наследоваться от класса BasePrinter и переопределять функцию

   ```
   void addToFinder(MatchFinder * finder)
   ```

   Эта функция отвечает за регистрацию созданного обработчика.

2. Создание AST Matcher для выявления узлов абстрактного синтаксического дерева, потенциально содержащих ошибки. Созданный AST Matcher используется при регистрации обработчика в функции **addToFinder**.

3. Последним этапом является добавление нового флага для созданной проверки в функции **ParseArgs**. Если пользователь передал флаг на включение новой проверки, необходимо добавить данную проверку в список используемых проверок с помощью функции **addPrinter**.

4. ТЕСТИРОВАНИЕ РАЗРАБОТАННОГО МОДУЛЯ

4.1. Тестирование

В результате тестирования созданного модуля были выявленны некоторые ошибки первого и второго рода.

Ошибки первого рода

Так как анализатор выдает предупреждения, если найдены участки кода, в которых слева и справа от оператора находятся одинаковые выражения (см. 3.1.1), то следующий код вызывает подозрение у анализатора:

```
*++scan == *++match && *++scan == *++match
```

Однако данный код является корректным, так как переменные в левой части изменяются, после чего переменные в правой части имеют измененные значения. Так же данное ложное срабатывание будет происходить при вызове функции, которая меняет свое состояние или глобальные переменные:

```
bool foo()
{
    static bool initialized = false;
    if (initialized)
    {
        initialized = init();
        return initialized;
    }
    else
    {
        return isAvailable();
    }
}
if (foo() && foo())
```

В приведенном выше примере вызов функции *foo()* является необходимым и не соответствует ошибке, но анализатор не учитывает побочные эффекты (side effects) при вызове функции и выдает предупреждения.

Часто встречается наличие пустых веток *if* и *else*, которые появляются при использовании макросов. Такая конструкция считается корректной и безопасной:

```
if (exp) {}
else {}
```

Но анализатор выдаст предупреждение, о совпадении веток *if* и *else*. Помимо ложного срабатывания при использовании пустых веток, часто анализатор выдает предупреждение для кода:

```
    if (err1)
        return -1;
    else if (err2)
        return -1;
    else
        return 0;
```

Приведенный код соответсвует следующему:

```
    if (err1 || err2)
        return -1;
    else
        return 0;
```

Но для удобства читаемости исходного кода была использована конструкция *if-else-if*. И так как анализатор находит две одинаковые ветки (*return -1;*), выдается предупреждение.

При использовании функции *strlen* иногда необходимо подсчитать длину строки без первого символа, для этого можно использовать конструкцию *strlen(text+1)*, но с точки зрения анализатора, данный участок кода содержит потенциальную ошибку и будет выдано предупреждение. Похожее ложное срабатывание происходит, когда необходимо выделить памяти для строки без учета символа ’
0’.

Ошибки второго рода

Реализованный модуль для статических проверок построен на основе сопоставления участков кода заранее заданным шаблонам, которые соответствуют фрагментам кода, потенциально содержащим ошибки. Вследствие чего, эффективность нахождения ошибок реализованного модуля зависит от количества запрограммированных шаблонов. Однако реализованные проверки могут находить не все ошибки. Рассмотрим пример:

```
if (( (A || B) && C) || ((A && C) || (A && B)))
```

Как видно, левая часть **((A || B) && C)** эквивалентна правой части **((A && C) || (A && B))** оператора ||, но анализатор не выдаст никаких предупреждений так как абстрактное синтаксическое дерево для левой и правой части различны. Поэтому при использовании кода, которые содержит ошибку, но не соответсвует шаблону, предупреждение не будет выданно.

ЗАКЛЮЧЕНИЕ

В результате проделанной работы был разработан модуль статических проверок для компилятора Clang. Были реализованы простейшие проверки для нахождения ошибок в исходном коде. Ввиду легкости добавления новых шаблонов для проверок на ошибки в исходном коде, данный модуль может являться основой для дальнейшей разработки и улучшения статического анализатора. Произведенный экспериментальный анализ тестовых исходных кодов, содержащих дефекты, показал эффективность разработанного модуля.

Полученные результаты соответствуют ожидаемым. Преимуществом использования разработанного модуля является его автоматезированность и легкость интеграции со средой разработки. Однако в ходе тестирования были выявлены следующие недостатки:

- Значительное увеличение времени компиляции
- Большое количество ложных срабатываний

В дальнейшем для улучшения разработанного модуля планируется увеличение количества доступных проверок и уменьшение ложных срабатываний.

ПРИЛОЖЕНИЕ А
ЛИСТИНГИ