Наука и Образование
МГТУ им. Н.Э. Баумана
Сетевое научное издание
ISSN 1994-0408

Ссылка на статью:
// Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. №9. С. 145–159.
DOI: **10.7463/0915.0812271**
Представлена в редакцию: 21.06.2015
Исправлена: 25.07.2015

УДК 004.4'2

Реализация межмодульного анализа для языков C и C++ в статическом анализаторе, использующем для анализа исходный код программы

Сидорин А. В.[1,2,*]
[*]alexey.v.sidorin@ya.ru

[1]МГТУ им. Н.Э. Баумана, Москва, Россия
[2]Московский исследовательский центр Samsung, Москва, Россия

Целью данного исследования является построение метода межмодульного анализа для языков C и C++. Поскольку выбранный для разработки статический анализатор (Clang Static Analyzer) использует в качестве входных данных непосредственно исходный код программы, для реализации такой возможности необходима разработка специального метода. В данной работе описывается система анализа, построенная для Clang Static Analyzer для поддержки межмодульного анализа. Также рассматривается задача слияния абстрактных синтаксических деревьев различных транслируемых модулей и связанные с ним проблемы, в частности, обработка конфликтующих определений, поддержка сложных систем сборки и сложных проектов, в том числе мультиархитектурных проектов. В статье описываются некоторые эвристики, разработанные с целью увеличения скорости слияния синтаксических деревьев. Разработанная система была проверена на исходном коде ОС Android с целью демонстрации ее способности обрабатывать проекты высокой сложности.

Ключевые слова: C++; статический анализ; символьное выполнение; межпроцедурный анализ; Clang Static Analyzer; межмодульный анализ

Введение

В связи с возрастающей сложностью программных комплексов и сложностью межмодульных связей внутри них возрастает трудоемкость поиска ошибок в таких системах. Цепочки вызовов функций, затрагивающие несколько модулей программы, трудно не только тестировать. В них также затруднительно отслеживать изменения, которые могут привести к дефекту. Поскольку интеграционное тестирование обычно проводится на поздних этапах разработки системы, стоимость исправления ошибок на данном этапе гораздо выше, особенно если речь идет о ручном тестировании. В связи с этим возникает необходимость в средствах автоматизированного поиска ошибок с возможностью глубокого анализа сложных программ.

В настоящее время статический анализ становится все более популярным средством поиска дефектов в программном коде. Это во многом объясняется ростом компьютерных

мощностей, доступных разработчикам: многие виды статического анализа являются ресурсоемкими, причем как в отношении процессорного времени, так и в отношении используемой анализатором памяти. Как правило, чем выше чувствительность метода анализа и его точность, тем выше требования, предъявляемые к вычислительным ресурсам. Самые простые и наименее ресурсоемкие виды анализа являются внутрипроцедурными, т. е. рассматривающими функцию безотносительно других функций, вызываемых или вызывающих. В отличие от внутрипроцедурного, межпроцедурный анализ подразумевает использование определений других функций, вызываемых анализируемой функцией, для более точного моделирования эффекта вложенного вызова функции. Использование межпроцедурного анализа позволяет производить поиск ошибок, к которым приводит целая цепочка вызовов функций.

В свою очередь, межпроцедурный анализ можно разделить на две категории в зависимости от области поиска определений вызываемых функций: на внутримодульный и межмодульный. Внутримодульный анализ подразумевает поиск доступных определений функций только в анализируемом модуле трансляции, тогда как в случае межмодульного анализа поиск определений может производиться и в других модулях трансляции. Очевидно, что в случае внутримодульного анализа анализатору может быть доступна лишь часть пользовательских определений функций, несмотря на потенциальную доступность исходного кода моделируемых функций. При межмодульном анализе потенциально недоступными являются лишь библиотечные функции с недоступным исходным кодом.

Межмодульный анализ позволяет находить различные классы дефектов программного кода, не обнаруживаемые при внутримодульном анализе или труднообнаружимые с его помощью. К таким дефектам относятся ошибки интеграции модулей и подсистем программы или программного комплекса, некорректное использование программных интерфейсов (API). Межмодульный анализ становится особенно полезным для языков программирования, допускающих раздельную компиляцию исходных файлов, поскольку в этом случае информация о программе, содержащаяся в одном исходном файле, становится крайне ограниченной и затрагивает лишь малую часть программного проекта. В число таких языков входят C и C++, исходные файлы которых обычно сначала компилируются в объектный код, и уже затем результирующие модули компонуются между собой, что позволяет получить всю информацию о программе лишь на поздних этапах ее построения. Таким образом, проблема межмодульного анализа должна быть решена любым анализатором, выполняющим межпроцедурный анализ программ, разработанных с использованием данного языка.

Целью данного исследования является построение метода анализа программ с целью обнаружения потенциальных дефектов, подходящего для проведения межпроцедурного межмодульного анализа крупных программных комплексов, разработанных с использованием языков C и C++, в частности, ОС Android и ОС Tizen. В качестве основы для реализации был выбран статический анализатор Clang Static Analyzer (CSA) [1]. Этот анализатор кода является опциональным модулем компилятора Clang [2] , который, в свою очередь, явля-

ется частью проекта LLVM [3]. Clang Static Analyzer является статическим анализатором с поддержкой анализа исходного кода на языках C, C++, Objective C и Objective C++. CSA поддерживает различные виды анализа, наиболее мощным из которых является символьное выполнение с возможностью межпроцедурного анализа.

Различные инструменты анализа кода реализуют межмодульный анализ по-разному. В случае динамического анализа анализируются не определения функций, а сгенерированный код: объектный код или промежуточное представление. В этом случае код вызываемых функций непосредственно становится доступным анализатору или в момент загрузки анализируемого модуля, или в процессе выполнения программы при загрузке подгружаемых модулей.

Большинство статических анализаторов, имеющих возможность межмодульного анализа, используют в качестве входных данных анализа промежуточное представление. Так, статический анализатор Svace [4] использует для анализа байт-код LLVM, Coverity SAVE [5] — компилятор Edison Design Group, и строят глобальный граф вызовов анализируемого проекта, производя разбор байт-кода, предварительно сгенерированного из исходных файлов проекта. Clang Static Analyzer, в отличие от многих других инструментов анализа исходного кода методом символьного выполнения, использует в качестве входных данных не промежуточное представление или объектный код, а непосредственно исходные файлы. Одной из основных причин этого является удобная и удачно спроектированная реализация абстрактного синтаксического дерева, в котором представлена вся информация о программе без каких-либо предварительных оптимизаций или потерь информации. Это обстоятельство требует применения иных подходов к межмодульному анализу, нежели в Coverity SAVE или Svace. В связи с этим в данной работе для проведения межмодульного анализа с использованием фреймворка Clang Static Analyzer предлагается выполнять слияние синтаксических деревьев различных файлов, связанных вызовами функций.

Единицей анализа в Clang Static Analyzer является транслируемый модуль, представляющий собой препроцессированный файл исходного кода. Однако для выполнения межмодульного анализа информации, содержащейся в одном транслируемом модуле, недостаточно. Необходимо знать расположение определений функций, необходимых для анализа других функций. Кроме того, необходимо знать не только имя и путь к файлу, где располагается определение функции. Для корректного построения импортируемого синтаксического дерева файла с исходным текстом необходимо знать, например, аргументы команды сборки файла, расположение включаемых файлов, использовавшихся для построения, и некоторую другую информацию.

Именно Clang Static Analyzer является целевым анализатором для реализации межмодульного анализа в данной работе, поскольку ранее автором был реализован ряд других проектов с использованием данного анализаторного фреймворка, и реализация межмодульного анализа является их завершающей частью. Кроме того, использование межмодульного анализа резко увеличивает количество требующих анализа путей программы и представляет

интерес при сравнении производительности и качества межпроцедурного анализа методом встраивания и разработанного автором метода резюме. Таким образом, в данной работе рассматривается решение проблемы межмодульного анализа для случая использования анализатором непосредственно исходного кода программы.

1. Реализация межмодульного анализа

Для решения проблемы определения местоположения анализируемых функций, в данной работе реализован трехфазный анализ с сохранением промежуточных результатов в файлах в директории проекта. Таким образом, анализ разделяется на три фазы: фаза сборки, фаза предобработки данных и непосредственно сам анализ исходных кодов. На рис. 1 представлена схема взаимодействия инструментов, используемых на различных фазах анализа, в виде диаграммы IDEF0. На этой схеме модули, реализованные в данной работе полностью, обозначены белым цветом, а серым обозначены модули, существовавшие ранее и доработанные для использования при межмодульном анализе: clang является непосредственно статическим анализатором, использованным для реализации межмодульного анализа, а Perl-скрипт ccc-analyzer входит в состав вспомогательного пакета scan-build и служит для формирования командной строки запуска статического анализатора clang и его запуска.

2. Фаза сборки

На фазе сборки специальный инструмент (strace_interceptor), использующий программу strace, собирает информацию о транслируемых модулях, которые должны быть проанализированы. Программа strace является утилитой ОС Linux, задачей которой является трассировка указываемого процесса (и, опционально, его потомков) и вывод информации о системных вызовах трассируемых процессов в стандартный поток вывода. Инструмент strace_interceptor реализован с использованием языка Python. Данный инструмент был разработан в рамках данной работы с целью поддержки различных систем сборки, в том числе, использующих сборку проекта в «песочнице», без изменения сборочных конфигураций. В число поддерживаемых систем сборки входят Makefile, OpenSUSE Build System (OBS) и Git Build System (GBS), причем поддерживается, в том числе, сборка в «песочнице» другой архитектуры с использованием эмуляторов (например, Qemu). Разработанный инструмент выполняет разбор вывода утилиты strace и использует информацию о системных вызовах chroot(), chdir(), vfork() и execve() для поиска вызовов компилятора и записывает текущую директорию, корневую директорию, команду сборки и переменные окружения в файл. Корневая директория отличается от стандартной («/») в тех случаях, когда был выполнен вызов chroot(). Текущая директория задается относительно корневой. Задачей этого же инструмента является поиск директорий с включаемыми файлами, связанными с запускаемым при сборке компилятором. Затем для каждой обнаруженной команды сборки другой инструмент, использующий программный интерфейс (API) Clang (clang-func-mapping), запи-

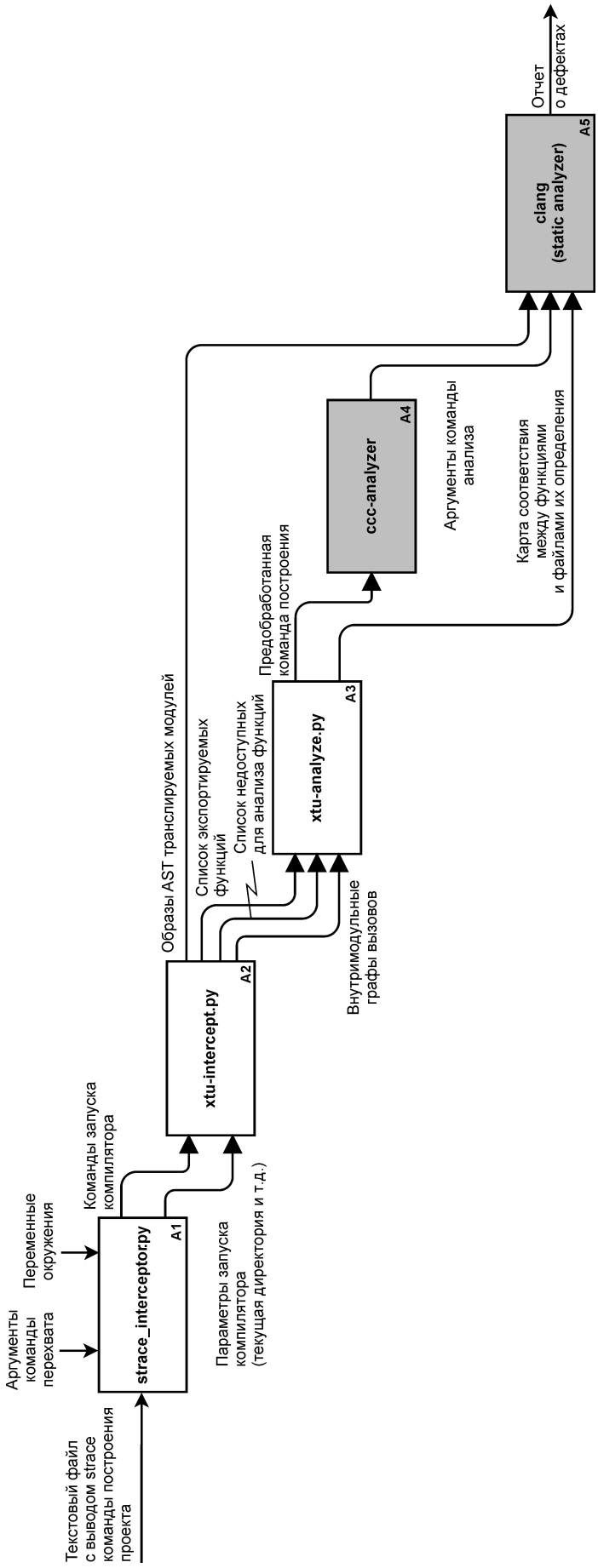

Рис. 1. IDEF0-диаграмма взаимодействия разработанных программных инструментов при межмодульном анализе

сывает сигнатуры видимых извне (экспортируемых) определений функций данного модуля трансляции и сигнатуры используемых (импортируемых) функций, определения которых в данном модуле трансляции недоступны, в служебные файлы в директории проекта. Этот же инструмент для каждой обнаруженной функции строит локальный граф вызовов, который также сохраняется в служебный файл. На фазе сборки дополнительно создаются образы абстрактных синтаксических деревьев для всех модулей трансляции, что упрощает дальнейший импорт и позволяет избежать повторных вызовов компилятора для каждого импортируемого файла.

Необходимо учитывать целевую архитектуру сборки для каждого файла исходных кодов. Сборка одного и того же файла может выполняться одновременно для нескольких архитектур в пределах одной сессии сборки. Это особенно актуально при использовании в качестве основных тестовых комплексов мультиплатформенных систем с эмуляторами в своем составе. Например, в случае ОС Android, некоторые файлы строятся как для хост-архитектуры (на которой запускается эмулятор), так и для целевой архитектуры (эмулируемой). Это значит, что для корректного импорта определения функции необходимо различать целевые тройки (target triple) цели сборки файла. Нельзя использовать определения функций из файлов, предназначенных для разных архитектур, по следующим причинам:

1) определения функций для различных архитектур могут не совпадать непосредственно в отношении текстового сравнения компилируемого исходного кода. Это возможно, поскольку для разных архитектур могут быть использованы различные фрагменты кода в директивах условной компиляции;

2) по тем же причинам для разных архитектур может не совпадать окружение функции: определения используемых типов, зависимые определения. Кроме того, для разных архитектур могут использоваться различные включаемые файлы;

3) даже при совпадении окружения и текста функций, для различных архитектур могут не совпадать определения основных типов. Так, тип char является по умолчанию знаковым (signed char) для платформы x86 и беззнаковым (unsigned char) для платформы ARM.

Кроме того, существует еще одна проблема. Один и тот же модуль трансляции может собираться несколько раз в рамках сборки для одной архитектуры. При этом также могут быть использованы различные директивы препроцессора и включаемые файлы, что может привести к потенциальной несовместимости импортируемого и импортирующего синтаксических деревьев.

Все вышеперечисленные факторы означают, что выбирать модуль трансляции для импорта определения функции надо крайне осторожно. Для решения этой проблемы можно предложить несколько способов. В настоящей работе было использовано менее общее, но более простое в реализации решение. Для такого приближенного учета архитектур достаточно знать, объектные модули каких архитектур могут компоноваться друг с другом. Например, объектные модули, скомпонованные для архитектур «arm» и «thumb» могут быть

использованы для компоновки в один объектный файл при использовании определенных опций компилятора, тогда как «arm» и «x86» не могут. После построения такой матрицы, можно различать определения функций по тройке «путь к файлу, сигнатура функции, целевая архитектура», и использовать эту тройку для выбора нужного определения функции и его импорта. Этот подход, хотя и прост в реализации, однако, не решает проблемы, связанной с использованием различных опций компилятора для одной и той же архитектуры, поскольку выбираться будет только один файл. Другим, и более общим решением видится поддержка «дерева сборки». В этом случае на фазе сборки отслеживается полное дерево компиляции, ассемблирования и компоновки (и, возможно, сборки в бинарный образ) для объектных файлов верхнего уровня. Это позволяет точно определить, какая функция из какого исходного файла должна быть использована для моделирования межфайлового вызова. Кроме того, это частично решает проблему модулей трансляции, которые собираются несколько раз в рамках сборки для одной архитектуры, поскольку для каждой из сборок становится известен высокоуровневый модуль, для которого она производится. Вместе с тем, этот подход имеет и некоторые проблемы. Во-первых, при его использовании необходимо отслеживать не только вызовы компилятора, но также вызовы ассемблера и компоновщика. Во-вторых, некоторые файлы собираются очень часто. Так, при построении образа ОС Android встречаются файлы, которые собираются 38 раз. Это может означать необходимость множественного повторного анализа уже проанализированных модулей трансляции. Таким образом, выбор между этими вариантами нельзя назвать однозначным, поскольку у каждого из них есть свои преимущества и недостатки.

3. Фаза предобработки данных

Фаза предобработки данных необходима для обработки служебной информации, собранной на фазе сборки. На этой фазе строится соответствие между сигнатурами импортируемых функций Как только соответствие становится известным, мы можем построить глобальный граф вызовов с использованием локальных графов вызовов, которые были сгенерированы на предыдущей фазе. Каждый узел графа вызовов, таким образом, представляет собой тройку «файл определения, сигнатура функции, архитектура». После этого выполняется топологическая сортировка построенного глобального графа вызовов. Сначала анализируются функции верхнего уровня, затем функции, участвующие в рекурсивных цепочках вызовов, а затем — функции нижнего уровня. При этом сортируются не сами функции, а файлы, их содержащие, поскольку анализ отдельных функций из файла означает многократную загрузку одних и тех же файлов. И, наконец, после фазы предобработки данных запускается анализ модулей трансляции в топологическом порядке глобального графа вызовов.

Топологическая сортировка улучшает производительность, поскольку анализ производится по одному транслируемому модулю. Так как импорт определений функций и создание их резюме производится при первом моделировании вызова, выгоднее производить анализ,

начиная с верхних уровней иерархии к нижним, что позволяет избежать повторных анализов одной и той же функции. В данной разработке используется список сигнатур проанализированных функций, чтобы исключить их повторный анализ вне контекста вызовов, т. е. если анализ функции был произведен в контексте вызова, повторный анализ производиться не будет. Это объясняется тем, что анализ функции при сборке резюме не отличается от отдельного анализа функции, поэтому нет причин производить анализ функции вне контекста, если она уже была проанализирована для сбора резюме.

4. Фаза анализа. Слияние синтаксических деревьев

В данной разработке был реализован межмодульный анализ с использованием реализации класса ASTImporter, который является частью интерфейса сериализации синтаксических деревьев компилятора Clang и отвечает за слияние синтаксических деревьев различных транслируемых модулей. Импорт фрагментов синтаксического дерева (т. е. данный класс) уже был частично реализован в Clang. Реализованная функциональность была расширена, так как значительная часть необходимых функций не была реализована ранее. В результате появилась возможность полноценного импорта фрагментов синтаксических деревьев функций в основной контекст синтаксического дерева. Когда анализатор обнаруживает функцию с недоступным определением, производится поиск сигнатуры этой функции в сгенерированном отображении. Если в результате поиска сигнатура функции была найдена, загружается синтаксическое дерево файла, содержащего определение этой функции. Затем эта функция импортируется в основной контекст синтаксического дерева вместе с необходимыми определениями и объявлениями.

Задача импорта фрагментов синтаксического дерева обычно выполняется с помощью поиска определения в импортированном контексте объявления (DeclContext), и поиска их аналогов в основном (целевом) контексте AST. Если аналогичное определение (или объявление) не найдено, оно создается в целевом синтаксическом дереве с использованием специального интерфейса. Новый фрагмент является рекурсивной копией исходного, но в процессе импорта зависимостей также производится поиск в целевом контексте, и не все части нового фрагмента синтаксического дерева обязательно являются созданными заново.

Поскольку ASTImporter уже был частично реализован на момент разработки межмодульного анализа, этот раздел посвящен различным проблемам при импорте и их возможным методам решения.

Первой проводимой операцией при импорте объявления из исходного контекста является поиск похожего объявления в целевом синтаксическом дереве. Этот поиск часто включает в себя рекурсивный обход вложенных объявлений для определения, являются ли два объявления структурно эквивалентными. В данной работе, однако, испытан ряд простых и легковесных эвристик, ускоряющих поиск за счет частичного отказа от рекурсивного об-

хода. Рекурсивная проверка структурной эквивалентности выполняется только в случае, если эти эвристики не смогли однозначно показать различие или эквивалентность.

Во-первых, если два объявления имеют различные разновидности, они, очевидно, не являются структурно эквивалентными. У этого правила, однако, есть одно исключение: класс C++ (CXXRecordDecl) может быть импортирован как структура языка C (RecordDecl) и наоборот в случае, если это POD-структура и целевой и исходный контексты имеют различные языковые настройки. Но это исключение может быть проверено отдельно.

Во-вторых, если два объявления имеют различные имена, их можно определенно считать различными без дальнейшего просмотра.

В-третьих, в большинстве случаев объявления с совпадающими местоположениями в исходных файлах являются эквивалентными. В случае Clang данная эвристика не подходит для частичных специализаций шаблонов, поскольку они наследуют исходное местоположение специализируемых шаблонов. Основная проблема этой эвристики заключается в обработке конфликтующих объявлений.

Если эвристика не сработала, происходит возврат к рекурсивному обходу, что является одной из основных проблем импорта. Для импорта объявления необходимо сначала импортировать его контекст объявления. Этот контекст, в свою очередь, может иметь большое количество вложенных объявлений и их зависимостей. В результате происходит массовый рекурсивный импорт зависимостей как самого объявления, так и его контекста. Иногда встречаются циклические зависимости, образуемые опережающими объявлениями.

При импорте структуры или класса для создания его раскладки в памяти необходимо соблюдать порядок объявления полей в структуре, для чего поля структуры должны импортироваться в порядке объявления. Однако, если поле структуры имеет некоторый сложный тип, импорт этого типа может при рекурсивном импорте вызвать импорт другого поля структуры, например, при импорте метода, использующего это поле. В этом случае определение поля-зависимости импортируется вне очереди импорта определений полей. Подобное поведение является нежелательным, поскольку нарушает раскладку структуры, что, в свою очередь, ведет к различным ошибкам и невыполнению условия структурной эквивалентности. Для решения этой проблемы определения полей определения структуры переупорядочиваются после того, как определение структуры было полностью импортировано, в соответствии с их порядком в импортируемой структуре.

Во время тестирования разработанной системы было обнаружено, что код, успешно прошедший компиляцию и компоновку, может содержать несовместимые друг с другом определения. Проблема при наличии конфликтующих определений заключается в выборе стратегии поведения анализатора. Первой стратегией может стать выдача предупреждения об обнаружении конфликтующего определения с последующим завершением работы анализатора или пропуском импорта данного определения. Несмотря на логичность такого подхода, данная стратегия имеет недостаток: разработанный программный код, возможно, все равно имеет смысл проанализировать, поскольку его работоспособность, как правило,

проверяется при тестировании. Вторая стратегия заключается в разрешении конфликтов между определениями. Ее недостаток заключается в том, что у анализатора может не быть данных о программе для корректного разрешения конфликта.

У проблемы конфликтующих определений два основных источника. Во-первых, некоторые пакеты и программы поставляются со своими версиями библиотек, отличными от общесистемных. Различные версии могут иметь различающиеся объявления типов, функций и переменных. Эта проблема непосредственно связана с проблемой множественных компиляций одного файла. Для решения этой проблемы необходимо корректно выбирать импортируемую вызываемую функцию.

Во-вторых, источником конфликтующих определений может являться непосредственно анализируемый код. Так, например, иногда в исходных кодах обнаруживались определения-«пустышки» для поддержки старых компиляторов. Такие определения вызывают конфликт, поскольку невозможно автоматически определить, какое из определений структуры данных является корректным.

Ниже приведен пример подобных конфликтующих определений, найденный в библиотеке zlib [6]. В заголовочном файле zlib.h находится определение следующего вида:

```
1740  /* hack for buggy compilers */
1741  #if !defined(ZUTIL_H) && !defined(NO_DUMMY_DECL)
1742      struct internal_state {int dummy;};
1743  #endif
```

тогда как в другом заголовочном файле (deflate.h) находится следующее определение:

```
97  typedef struct internal_state {
98      z_streamp strm;      /* pointer back to this zlib stream */
99      int   status;        /* as the name implies */
100     Bytef *pending_buf;  /* output still pending */
        ...
273 } FAR deflate_state;
```

Очевидно, что в данном примере первое определение используется для поддержки некоторых специфических компиляторов, однако узнать это при слиянии определений достаточно затруднительно. В приведенном случае проблема возникает, когда транслируемый модуль, включающий определение-«пустышку», импортирует модуль, использующий настоящее определение и содержащий функции, обращающиеся к полям настоящего определения.

Еще одним примером является компиляция с различными опциями препроцессора (такими как определения символов препроцессора) различных исходных файлов, использующих один и тот же включаемый файл, что приводит к появлению различающихся определений

одной и той же структуры данных в результате условной компиляции. Например, из-за опций препроцессора могут быть объявлены дополнительные поля структуры данных. Эти определения являются различными для компоновщика, поскольку они не эквивалентны побайтово. С другой стороны, выбор между различными конфликтующими определениями подобного рода не является тривиальным на уровне компилятора (а это тот уровень, на котором работает Clang Static Analyzer и анализ на уровне исходных кодов), поскольку эти определения должны быть помещены в одну область видимости. Данная проблема, возможно, может быть решена с помощью внутреннего переименования конфликтующих определений. Пример подобной структуры приведен ниже.

```
1  struct Sample
2      int field_1;
3  ...
4 #ifdef DEBUG_MODE
5      int access_counter;
6 #endif
7  ...
8      int field2;
9  };
```

В приведенном случае проблема возникает при условиях, аналогичном предыдущему случаю: если при слиянии импортирующий модуль имеет определение структуры без поля, а импортируемый модуль содержит определение структуры с полем и код программы, это поле использующий, т. е. при слиянии транслируемого модуля, в котором символ препроцессора не определен, и модуля, в котором он определен. Проблемы возникает и в обратном случае, поскольку оба определения структуры имеют различные смещения полей, находящихся после опционального поля.

Еще одной причиной несоответствия определений является тот факт, что некоторые элементы определений структур, согласно стандарту языка C++, создаются «по требованию», т. е. лишь в том случае, если они реально используются в коде [7, с. 269, 276]. К таким необязательным определениям относятся конструкторы по умолчанию, конструктор копирования и деструктор класса, которые создаются лишь в том случае, если они, во-первых, используются в коде, и, во-вторых, не имеют перегруженных определений. В отношении анализа программы данная проблема становится особенно важной в случаях, когда поля класса сами имеют нетривиальные конструкторы и деструкторы, поскольку в этом случае генерируемые компилятором специальные методы должны вызывать конструкторы и деструкторы полей класса. В результате допустима ситуация, при которой импортируемое определение класса имеет созданный компилятором специальный метод, а аналогичное определение в импортирующем модуле трансляции его не имеет. Решением этой проблемы является либо импорт специальных методов в случае обнаружения подобного несоответствия, либо самостоятель-

ное создание недостающих специальных методов в синтаксическом дереве. В данной работе необходимые специальные методы создаются с использованием методов класса Sema — реализации семантического анализатора в составе компилятора Clang.

Заключение

Целью данной работы была разработка метода межмодульного анализа, применимого для анализатора, использующего в качестве входных данных для анализа непосредственно исходный код программ.

В работе рассмотрен ряд проблем, решение которых потребовалось при разработке межмодульного анализа для исходных кодов: межмодульный анализ для мультиархитектурных программных комплексов, конфликтующие определения, импорт сложных структур данных между синтаксическими деревьями. В данной работе предложены способы решения данных проблем.

Проанализированы особенности межмодульного анализа при использовании исходных кодов в качестве входных данных. С учетом рассмотренных особенностей разработана архитектура системы, позволяющая проводить межмодульный анализ при помощи статического анализатора, использующей исходный код в качестве входных данных. Данная система реализована с использованием языков Python (для реализации инфраструктурных частей) и C++ (непосредственно код анализатора) и опробована на ОС Linux с использованием Clang Static Analyzer в качестве статического анализатора.

Разработанная система была успешно испытана с использованием исходного кода пакетов ОС Android в качестве входных данных с целью демонстрации возможности ее работы с крупными и сложными программными комплексами. Анализ прошел успешно, имелись срабатывания, трасса которых затрагивала несколько файлов, часть этих срабатываний была квалифицирована как корректные, т. е. указывающие на потенциальные дефекты. Помимо получения новых срабатываний, в дальнейшем планируется провести дополнительное исследование с целью изучения влияния межмодульного анализа на качество статического анализа и проверки возможности устранять ложные срабатывания, связанные с недостатком информации о вызываемой функции.

Результирующая система не зависит от используемого метода межпроцедурного анализа и допускает произвольную смену его алгоритмов. Это позволит провести сравнение различных методов межпроцедурного анализа в отношении их качества и быстродействия. Испытания проводились как с использованием межпроцедурного анализа методом встраивания, который используется в Clang Static Analyzer, так и с использованием межпроцедурного анализа разработанным и реализованным ранее методом резюме.

Разработанная и реализованная система предполагается к внедрению в Samsung Electronics в качестве инструмента для поиска дефектов в программном обеспечении различного назначения, в том числе, в приложениях для мобильных и телевизионных систем, а также ПО медицинских систем.

Список литературы

1. Clang Static Analyzer: website. Режим доступа: http://clang-analyzer.llvm.org (дата обращения 12.06.2015).
2. clang: a C language family frontend for LLVM // The LLVM Project: website. Режим доступа: http://clang.llvm.org (дата обращения 12.06.2015).
3. The LLVM compiler infrastructure // The LLVM Project: website. Режим доступа: http://llvm.org (дата обращения 12.06.2015).
4. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26, № 1. С. 231–250.
5. Almossawi A., Lim K., Sinha T. Analysis tool evaluation: Coverity Prevent. Final Report. Pittsburgh, PA: Carnegie Mellon University, 2006. 19 p.
6. A Massively Spiffy Yet Delicately Unobtrusive Compression Library: webcite. Режим доступа: http://www.zlib.net (дата обращения 25.07.2015).
7. Smith R. Working Draft, Standard for Programming Language C++. ISO/IEC N4296, 2014. 1368 p. Режим доступа: https://isocpp.org/files/papers/n4296.pdf (дата обращения 01.09.2015).

Science&Education
of the Bauman MSTU
Electronic journal
ISSN 1994-0408

Science and Education of the Bauman MSTU, 2015, no. 9, pp. 145–159.
DOI: **10.7463/0915.0812271**
Received: 21.06.2015
Revised: 25.07.2015

Implementation of inter-unit analysis for C and C++ languages in a source-based static code analyzer

Sidorin A. V.[1,2,*]

[*]alexey.v.sidorin@ya.ru

[1]Bauman Moscow State Technical University, Russia
[2]Moscow Samsung Research Center, Russia

Keywords: C++, static code analysis, symbolic execution, interprocedural analysis, Clang Static Analyzer, inter-unit analysis

The proliferation of automated testing capabilities arises a need for thorough testing of large software systems, including system inter-component interfaces. The objective of this research is to build a method for inter-procedural inter-unit analysis, which allows us to analyse large and complex software systems including multi-architecture projects (like Android OS) as well as to support complex assembly systems of projects. Since the selected Clang Static Analyzer uses source code directly as input data, we need to develop a special technique to enable inter-unit analysis for such analyzer. This problem is of special nature because of C and C++ language features that assume and encourage the separate compilation of project files. We describe the build and analysis system that was implemented around Clang Static Analyzer to enable inter-unit analysis and consider problems related to support of complex projects. We also consider the task of merging abstract source trees of translation units and its related problems such as handling conflicting definitions, complex build systems and complex projects support, including support for multi-architecture projects, with examples. We consider both issues related to language design and human-related mistakes (that may be intentional). We describe some heuristics that were used for this work to make the merging process faster. The developed system was tested using Android OS as the input to show it is applicable even for such complicated projects. This system does not depend on the inter-procedural analysis method and allows the arbitrary change of its algorithm.

References

1. Clang Static Analyzer: website. Available at: http://clang-analyzer.llvm.org, accessed 12.06.2015.

2. clang: a C language family frontend for LLVM. The LLVM Project: website. Available at: http://clang.llvm.org, accessed 12.06.2015.
3. The LLVM compiler infrastructure. The LLVM Project: website. Available at: http://llvm.org, accessed 12.06.2015.
4. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. Static analyzer Svace for finding of defects in program source code. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the RAS* (Proceedings of ISP RAS), 2014, vol. 26, no. 1, pp. 231–250. (in Russian).
5. Almossawi A., Lim K., Sinha T. Analysis tool evaluation: Coverity Prevent. Final Report. Pittsburgh, PA, Carnegie Mellon University, 2006. 19 p.
6. A Massively Spiffy Yet Delicately Unobtrusive Compression Library: webcite. Available at: http://www.zlib.net, accessed 25.07.2015.
7. Smith R. *Working Draft, Standard for Programming Language C++*. ISO/IEC N4296, 2014. 1368 p. Available at: https://isocpp.org/files/papers/n4296.pdf, accessed 01.09.2015.