Федеральное государственное бюджетное учреждение науки
Институт системного программирования
Российской академии наук

На правах рукописи

Бородин Алексей Евгеньевич

Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
к. ф.-м. н.
Белеванцев Андрей Андреевич

Москва – 2016

Оглавление

Введение 4

Глава 1. Поиск ошибок в исходном коде 10

1.1. Использование статического анализатора в жизненном цикле разработки программ 10

1.2. Ошибки в исходном коде 12

1.3. Корректность анализа 15

1.4. Синтаксические анализаторы 16

1.5. Обзор существующих семантических анализов 17

1.6. Анализатор Svace 27

Глава 2. Анализируемый язык 32

2.1. Промежуточное представление Svace 32

2.2. Описание языка svace0 32

2.3. Структурная операционная семантика языка 36

2.4. Функция на языке svace0 40

Глава 3. Внутрипроцедурный анализ 42

3.1. Используемый подход 42

3.2. Идентификаторы значений и ссылки 44

3.3. Передаточные функции 50

3.4. Анализ циклов 53

3.5. Корректность анализа 55

3.6. Сильные и слабые обновления 65

3.7. Слияние состояний 67

3.8. Пример анализа 68

3.9. Массивы, объединения, приведения типов 70

Глава 4. Детекторы 73

4.1. Атрибуты 73

4.2. Критерий выдачи 76

4.3. События 78
4.4. Вспомогательные атрибуты 80
4.5. Разыменование нулевого указателя 80
4.6. Обратный анализ 83
4.7. Сопоставление атрибутов ссылкам 85
4.8. Чувствительность к путям 87

Глава 5. Межпроцедурный анализ 92
5.1. Резюме функции 92
5.2. Создание резюме функции 93
5.3. Трансляция резюме в контекст вызова 97
5.4. Отложенное слияние для параметров функций 100
5.5. Трассы атрибутов 102
5.6. Параметризованные атрибуты 103
5.7. Удаление резюме 104
5.8. Анализ конструкторов и деструкторов 105

Глава 6. Реализация 108
6.1. Инструмент Svace 108
6.2. Трансляция Си и Си++-кода в LLVM 110
6.3. Вспомогательные алгоритмы 112
6.4. Создание резюме 115
6.5. Оценка времени анализа 115
6.6. Оценка выданных предупреждений 117

Заключение 122

Список сокращений и условных обозначений 124

Список литературы 126

Приложение А. Оценка скорости анализа 132

Приложение Б. Оценка выданных предупреждений 136

Введение

Актуальность темы исследования.

Дефекты и ошибки в программном обеспечении (ПО) в большой степени влияют на качество ПО. Ошибки времени выполнения снижают надёжность программы, приводя к сбоям в работе, отказам в обслуживании. Некоторые ошибки – уязвимости защиты – приводят к возможности выполнения произвольного кода злоумышленником. Дефекты исходного кода программы, такие, как, например, мёртвый или избыточный код, могут не проявляться во время её работы, но тем не менее снижают качество исходного кода.

Сложность поиска ошибок сильно выросла из-за взрывного роста размеров ПО. Количество строк кода в современных дистрибутивах ОС Tizen и Android достигает десятков миллионов, а в дистрибутиве ОС Debian GNU/Linux – миллиарда. Множество ошибок остаются не найденными в течение многих лет после выпуска ПО. Из-за этого, а также из-за доступности программ через компьютерные сети, растет ущерб от эксплуатации ошибок. Популярность библиотек с открытым исходным кодом приводит к возможности эксплуатации единственной ошибки на миллионах инсталляций самых разных программ.

В настоящее время известно множество методов поиска ошибок: экспертный аудит кода, ручное тестирование, дедуктивная верификация, динамический анализ (мониторинг выполнения программы, фаззинг и др.). Все эти методы имеют различные, присущие только им достоинства и недостатки, разные области применимости, и поэтому не исключают, а дополняют друг друга. Одним из признанных методов поиска, широко используемого в индустрии, является статический анализ исходного кода программ.

Статический анализ выдает список предупреждений о местах потенциальных ошибок в исходном коде программы. Выданное анализом предупреждение называется истинным, если в используемой анализом модели поведения программы действительно может возникнуть искомая ошибочная ситуация, и ложным в противном случае. Преимуществом статического анализа является одновременный анализ многих путей исполнения и поиск ошибок на редко выполняющихся путях, которые плохо покрываются при общесистемном тестировании

или динамическом анализе. Среди недостатков нужно отметить наличие ложных предупреждений, от которых невозможно полностью избавиться. Ложные предупреждения связаны с тем, что точное вычисление необходимых свойств программы при статическом анализе является алгоритмически неразрешимой задачей.

Для промышленного применения статического анализатора нужно выполнить следующие требования:

- масштабируемость: анализ больших программных систем (миллионы строк кода) должен длиться не более 4-6 часов (ночная сборка);
- качество: высокий уровень истинных предупреждений (50-70%) вместе с пропуском небольшого количества ошибок;
- понятность: как можно более подробное описание ошибки, т. е. места её проявления и (по возможности) возникновения, иногда – цепочки событий, приведших к ошибке;
- полнота: поддержка популярных языков программирования, поддержка известных типов ошибок разной критичности;
- расширяемость: простое добавление новых типов ошибок, в том числе специфичных для конкретной программы.

Ключевой сложностью при разработке статического анализа является поиск компромисса между уровнем истинных срабатываний и количеством пропущенных ошибок при условии масштабируемости анализа для заданного размера программ. С одной стороны, попытка выдачи предупреждений о всех потенциальных местах ошибок приводит к потоку ложных срабатываний. С другой стороны, максимальная консервативность анализа, проявляющаяся в срабатывании анализа только для ситуаций, в которых есть практически полная уверенность в ошибке, приводит к пропуску значительной доли реальных ошибок. Следовательно, необходимо применять нестрогий анализ (т. н. unsound analysis), который вынужденно будет пропускать некоторые ошибочные ситуации, регулируя точность алгоритмов анализа с помощью набора эвристик.

Используемые эвристики необходимо регулярно пересматривать при повышении требований к масштабируемости анализа, потому что с ростом размера анализируемой программы увеличивается вероятность обнаружения ранее не встреченных программных конструкций, и сбой эвристик анализа всего лишь на нескольких новых конструкциях может серьезно ухудшить качество анализа. Кроме того, нужно повышать точность используемых алгоритмов анализа потока данных и управления за счет лучшей контекстной чувствительности межпроцедурного анализа, большей точности алгоритмов анализа указателей, анализа циклов, учета влияния отдельных путей выполнения. Таким способом разрабатываются ведущие мировые промышленные анализаторы (инструменты Coverity CMC, Klocwork K10, HP Fortify, инструмент Svace Института системного программирования РАН).

Статический анализатор Svace ищет ошибки в исходном коде программ, написанных на языках Си, Си++, Java и Си#. Предыдущие версии анализатора демонстрировали промышленное качество анализа при обработке программ из сотен тысяч строк исходного кода. Однако при переходе к анализу ПО размером в миллионы строк кода уровень истинных срабатываний для многих типов ошибок упал до 15-20%. Актуальной является задача разработки алгоритмов анализа потока данных и управления программы, обеспечивающих высокое качество статического анализа при поиске ошибок в сверхбольших программах, и их реализация в инструменте Svace.

Цели и задачи диссертационной работы: разработка алгоритмов внутрипроцедурного анализа потока данных и управления, алгоритмов межпроцедурного контекстно- и потоково- чувствительного анализа, предназначенных для поиска дефектов в исходном коде программ, написанных на языках Си и Си++, и их реализация в инструменте Svace. Разработанные алгоритмы должны обеспечивать качество анализа, соответствующее современным требованиям, выдвигаемым к промышленным статическим анализаторам, и масштабируемость анализа для программ в миллионы строк исходного кода.

Для достижения поставленных целей были сформулированы и решены следующие задачи:

- Разработка внутрипроцедурных алгоритмов ядра анализа, позволяющих

на основе вычисленной ими информации о потоке данных и управления программы выполнять поиск широкого класса дефектов, и доказательство их корректности.

- Разработка межпроцедурных алгоритмов ядра анализа на основе предложенных внутрипроцедурных алгоритмов, обеспечивающих контекстную и потоковую чувствительность анализа.

- Разработка детекторов для часто встречающихся критических ошибок и дефектов исходного кода, использующих предложенные алгоритмы ядра анализа, в том числе критерия выдачи предупреждений на основе вычисленной детектором информации.

- Реализация разработанных алгоритмов в инструменте статического анализа Svace.

Научная новизна. В работе получены следующие основные результаты, обладающие научной новизной:

- Разработан алгоритм внутрипроцедурного анализа функции, интегрирующий анализ указателей и нумерацию значений и обеспечивающий возможность поиска широкого класса дефектов. Доказана корректность разработанного алгоритма.

- Разработан алгоритм межпроцедурного контекстно- и потоково- чувствительного анализа функций, основанный на алгоритме создания резюме функции, которое обобщает результаты внутрипроцедурного анализа функции, и применения созданного резюме при обработке вызова функций.

- Разработанные алгоритмы реализованы в статическом анализаторе Svace для программ на языках Си и Си++. Экспериментальные результаты анализа больших программных систем подтвердили масштабируемость реализованных алгоритмов при высоком качестве анализа (более 60% истинных срабатываний критических детекторов разыменования нулевых указателей, утечки ресурсов, переполнения буфера, использования неинициали-

зированных переменных, и в разы возросшее количество предупреждений по сравнению с предыдущей версией анализатора).

Теоретическая и практическая значимость. Методы и алгоритмы, изложенные в диссертации, были реализованы в статическом анализаторе Svace для программ, написанных на языках Си и Си++. Предложенные алгоритмы поиска дефектов были реализованы для поиска ошибок разыменования нулевых указателей, переполнения буфера, утечек памяти и ресурсов, использования неинициализированных данных, использования памяти после удаления и других. Разработанный анализатор продемонстрировал масштабируемость и качество анализа на уровне лучших мировых аналогов и внедрён для промышленного использования в коммерческой компании Samsung.

Положения, выносимые на защиту:

- алгоритм внутрипроцедурного анализа функции, интегрирующий анализ указателей и нумерацию значений и обеспечивающий возможность поиска широкого класса дефектов;
- алгоритм создания резюме функции, обобщающий результаты внутрипроцедурного анализа функции и обеспечивающий потоковую чувствительность межпроцедурного анализа;
- алгоритм анализа инструкций вызова функций, обеспечивающий масштабируемый контекстно-чувствительный межпроцедурный анализ;
- инструмент статического анализа, реализующий разработанные алгоритмы для программ на языках Си и Си++.

Апробация результатов. Основные результаты диссертации докладывались на следующих конференциях:

1. IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST) (Cleveland, Ohio, USA, 2014);
2. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия, 2014);

3. Открытая конференция по компиляторным технологиям (Москва, Россия, 2015).

Публикации. Материалы диссертации опубликованы в 9 печатных работах, из них 6 статей в рецензируемых журналах [1–6], 3 статьи в сборниках трудов конференций [7–9]. В статье [1] личный вклад автора заключается в описании возможностей расширения анализа на примере разработанных детекторов. В работе [2] личный вклад состоит в описании ядра анализатора, межпроцедурного анализа и взаимодействия ядра и детекторов. Совместная работа [3] посвящена описанию инструмента Svace, личный вклад автора состоит в описании подсистемы анализа исходного кода на языках Си и Си++. Статья [4] является переводом [3]. В статье [6] личный вклад автора заключается в описании ядра анализатора Svace и детекторов разыменования нулевых указателей.

Личный вклад автора. Все представленные в диссертации результаты получены лично автором.

Структура и объем диссертации. Диссертация состоит из введения, 6 глав, заключения, библиографии и 2 приложений. Общий объем диссертации 137 страниц, из них 131 страница текста. Библиография включает 72 наименования на 6 страницах.

Глава 1

Поиск ошибок в исходном коде

Статический анализатор осуществляет поиск ошибок в программе (иногда говорят – поиск дефектов[1]) без её фактического запуска. После анализа выдаётся список предупреждений о местах потенциальных ошибок в исходном коде программы. Предупреждение называют истинным, если оно соответствует ошибке в исходном коде, т. е. описывает ситуацию в исходном коде, которую необходимо исправить. Если предупреждение не описывает ошибку, то его называют ложным.

1.1. Использование статического анализатора в жизненном цикле разработки программ

Анализ многих путей сразу, поиск ошибок на редко выполняемых путях, диагностика места ошибки и раннее выявление ошибок сделали популярным использование статических анализаторов для поиска ошибок. В настоящее время жизненный цикл разработки программ в крупных компаниях обязательно включает в себя применение инструментов статического анализа. При таком использовании анализаторы запускаются во время ночной сборки проекта, результаты сохраняются на сервере, и разработчики на следующий день могут просмотреть выданные анализом предупреждения.

Идеальный анализатор после анализа программы за приемлемое время найдёт все дефекты в программе и не выдаст ни одного ложного срабатывания. К сожалению, создание такого анализатора невозможно [10], что следует из теоремы Райса, согласно которой для любого нетривиального свойства функции программы определение того, вычисляет ли произвольный алгоритм функцию с таким свойством, является алгоритмически неразрешимой проблемой. Свойство считается нетривиальным, если существуют как функции, обладаю-

[1] Под дефектами в статическом анализе иногда понимают только недостатки исходного кода программы, а под ошибками – ситуации, проявляющиеся во время выполнения. В работе оба термина будут использоваться как синонимы.

щие этим свойством, так и функции, не обладающие этим свойством. Поэтому проблема поиска дефектов в произвольной программе является алгоритмически неразрешимой задачей, т. к. наличие ошибки в программе является нетривиальным свойством. Из-за проблемы неразрешимости статические анализаторы осуществляют поиск приближенного решения. При этом приближенное решение находится не только для путей, которые возможны при выполнении программы, но и для некоторых путей, которые не могут быть выполнены, но анализ не смог это определить. Наличие таких путей – одна из причин появления ложных предупреждений.

Можно выделить следующие технические характеристики статических анализов:

- количество выдаваемых истинных предупреждений;
- процент ложных срабатываний;
- время анализа;
- масштабируемость. Причём масштабируемость определяет не только зависимость времени анализа от размера анализируемых проектов, но также и зависимость уровня ложных срабатываний и количества выдаваемых истинных предупреждений в зависимости от размера проекта;
- количество действий, требуемых от пользователя;
- множество принимаемых программ и конструкций языка.

Реализация конкретного анализатора является компромиссом между этими характеристиками. Выбор компромисса во многом определяется задачей и лишь частично обусловлен дизайном самого анализатора.

Использование статического анализатора в жизненном цикле разработки программ имеет свои особенности. Такой анализ имеет жёсткие требования ко времени анализа и масштабируемости, необходимо укладываться во время, отведённое для ночной сборки (4-6 часов, чтобы оставить время на сборку проекта).

Требование любых действий от пользователя во время анализа является недопустимым, а запуск анализа должен осуществляться прозрачно для процесса сборки. Анализатор не должен ограничивать множество принимаемых программ. В некоторых случаях допустимо не анализировать некоторые конструкции языка либо какие-то специфичные ситуации, если это не сильно влияет на результаты.

Особое внимание уделяется проценту ложных срабатываний. Количество ложных срабатываний должно быть достаточно низким, т. к. каждое ложное срабатывание отнимает время программиста и фактически выражается в убытках для компании. Долю ложных срабатываний в 30-40% можно считать приемлемой. При таком соотношении истинных и ложных срабатываний на одно истинное срабатывание приходится не более одного ложного, и пользователь не тратит много времени на просмотр предупреждений. Если доля ложных срабатываний выше 50-60%, то у анализатора мало шансов на использование в жизненном цикле разработки больших программ.

1.2. Ошибки в исходном коде

1.2.1. Ошибки в исходном коде и ошибки времени выполнения

Следует различать ошибки времени выполнения программы и ошибки в исходном коде. Предупреждение об ошибке во время выполнения программы будет истинным, если можно так подобрать входные данные для программы, что выполнение приведёт к ошибке. Эти ошибки влияют на наблюдаемое поведение программы. Примеры таких ошибок:

- Разыменование нулевого указателя. По стандартам языков Си и Си++ [11, 12] нулевой указатель не указывает на какой-либо объект в памяти, и его разыменование не определено. На практике разыменование нулевого указателя может приводить к ошибке сегментации памяти и последующему аварийному завершению программы.

- Переполнение буфера – чтение либо запись за пределы массива. По стандартам Си и Си++ поведение программы после переполнения буфера также не определено. Подобная ошибка может по-разному проявляться

во время выполнения: от изменения поведения программы до аварийного завершения [13]. В том случае, когда переполнение зависит от входных данных, в программе будет уязвимость – злоумышленник сможет влиять на выполнение программы.

- Использование неинициализированных переменных. При такой ошибке происходит чтение значения переменной, которое не было определено. Ошибка может по-разному проявляться, поэтому такие ошибки тяжело находить с помощью тестирования и отладки.

- Утечки памяти происходят, если в программе не освобождается память, которая была выделена. Программа не обязательно упадёт, но ошибка влияет на поведение программы и может быть обнаружена. В случае нехватки памяти выполнение программы завершится аварийно. Особенно критичны такие ошибки для программ, работающих длительное время. В этом случае даже незначительная утечка памяти, которая возникает периодически, приведёт к замедлению и последующему падению программы.

Ошибка в исходном коде не обязательно проявится во время выполнения программы. Такие ошибки менее формализуемы, и во многом их классификация основана на субъективном мнении эксперта. Но можно выделить ситуации в исходном коде, которые можно формализовать. Примеры ошибок в исходном коде:

- Наличие недостижимого кода.

- Лишняя проверка, результат которой либо всегда является истиной, либо всегда ложью.

- Неконсистентность работы с указателями, когда в одних частях функции значение указателя проверяется на ноль, а в других нет.

- Дефект на недостижимом пути.

- Объявление либо определение переменной, которая никогда не будет использоваться.

1.2.2. Использование ошибок в исходном коде

Как правило, разработчики не пишут специально неэффективный и бессмысленный код. Сами по себе такие дефекты не приводят к серьёзным последствиям. Современные оптимизирующие компиляторы удаляют недостижимый код, лишние проверки, присваивания неиспользуемых переменных. Но появление описанных выше ошибок часто является результатом опечатки, неправильного исправления кода либо непонимания программистом логики программы. Поэтому поиск таких мест позволяет находить подозрительные места в программе. В работе [14] было показано, что существует корреляция между критическими ошибками и лишними вычислениями в программе или недостижимым кодом. Было показано, что часто лишние вычисления в программе и недостижимый код являются результатом опечаток и других ошибок программиста и приводят к более серьёзным ошибкам в программах. Т. е. поиск простых дефектов в исходном коде позволяет найти сложные дефекты времени выполнения программы.

Поиск ошибок в исходном коде облегчает задачу, т. к. не обязательно проверять достижимость некоторого пути для выдачи предупреждения. Многие инструменты используют разные критерии "подозрительного" кода, т. е. производят поиск ошибок в исходном коде.

1.2.3. Библиотечные функции

При анализе библиотечных функций поиск ошибок в исходном коде особенно полезен, т. к. неизвестен контекст использования функции. Не все функции могут быть вызваны в произвольном контексте. Например, функция стандартной библиотеки Си `memcpy`, осуществляющая копирование из одной области памяти в другую, имеет три параметра: два указателя на области памяти и количество копируемых байт. Если количество копируемых байт не нулевое, то два указателя не могут иметь нулевые значения. Если вызвать функцию `memcpy` с параметрами (0, 0, 10), то произойдёт ошибка разыменования нулевого указателя. Но это будет ошибка вызывающего кода, а не кода `memcpy`. Поэтому нельзя выдавать предупреждение каждый раз, когда происходит разыменова-

ние указателя без проверки на нулевое значение, если нет информации о том, в каком контексте функция может быть вызвана.

Многие функции в программах пишут как библиотечные. Т. е. предполагается, что функция может использоваться не только в анализируемой программе, но и в других потенциальных контекстах вызова. Поэтому, если анализатор найдёт дефекты в гипотетических контекстах вызова, то это будет не ложное срабатывание, а желательное поведение анализатора. Т. е. функцию необходимо проанализировать в том числе и саму по себе.

1.3. Корректность анализа

Многие подходы статического поиска ошибок исторически развились из области компиляции программ и оперируют абстракциями, взятыми оттуда: поток токенов, абстрактное синтаксического дерево, граф вызовов, граф потока управления, поток данных. При оптимизации программы компилятор должен обеспечить корректность трансформации: оптимизированная программа должна иметь ту же семантику, что и исходная.

В отличие от оптимизирующих компиляторов, для задачи поиска дефектов корректность анализа не является обязательной. Отказ от корректности (консервативности) некоторых конструкций может существенно улучшить как скорость анализа, так и точность. Если к статическому анализатору не предъявляется требований корректности, то улучшение временных характеристик анализа, масштабируемости и точности может осуществляться за счёт отказа от корректности.

При анализе кода свойства переменных зависят от решения задачи алиасов, которая является неразрешимой в общем случае [15]. Точности быстрых анализов алиасов [16] может быть недостаточно для обеспечения высокого уровня истинных срабатываний. Более точные анализы часто имеют значительную сложность. Например, потоково-нечувствительный анализ Андерсона является одним из наиболее точных потоково-нечувствительных алгоритмов, но при этом алгоритм имеет кубическую сложность анализа в худшем случае[17].

Из-за недостаточной точности и масштабируемости анализов алиасов во

многих статических анализаторах вообще не проводят анализ алиасов либо проводят его частично. При этом используется предположение, что все переменные либо часть переменных программы не имеют алиасов. В статье [18] описывается, как после удаления анализа алиасов, необходимого для гарантии корректности, но не обязательного в большинстве случаев, из алгоритма поиска дефектов скорость анализа существенно улучшилась. В одном случае время анализа изменилось с нескольких дней до нескольких минут.

1.4. Синтаксические анализаторы

В результате работы синтаксического анализа компилятором строится абстрактное синтаксического дерево (АСД), которое позже передаётся на следующие фазы компиляции программы. Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил. Время работы таких анализаторов линейно зависит от размера АСД.

На этой фазе недоступна информация об алиасах программы, графе потока управления, графе вызовов, и не производится межпроцедурный анализ. Поэтому сложные дефекты не могут быть обнаружены только с помощью синтаксического анализа. Но, т. к. критичные ошибки могут быть найдены простым анализом, а также из-за того, что выданные синтаксическим анализатором предупреждения имеют высокий процент истинных срабатываний из-за относительно простой реализации, такой анализ на основе АСД позволяет находить множество дефектов в реальных программах. Кроме этого, многие виды дефектов могут быть обнаружены только на стадии синтаксического анализа, т. к. необходимая информация не передаётся на следующие стадии.

Рис. 1.1 содержит предупреждение NO_EFFECT.SELF_ASSIGN, выданное синтаксическим анализатором, реализованным в инструменте Svace. Детектор проверяет аргументы для узла присваивания в АСД. Если аргументы соответствуют одному и тому же символу, то выдаётся предупреждение.

Детектор NO_EFFECT.SELF_ASSIGN способен найти ошибки вида: “x = x” или “p->q = p->q”. Но для выдачи предупреждения для фрагмента кода

```
913| if (rawptr >= end) {
 pubk->u.fortezza.DSSKey.len = pubk->u.fortezza.KEAKey.len;
 pubk->u.fortezza.DSSKey.data=
                pubk->u.fortezza.KEAKey.data;
921|    pubk->u.fortezza.DSSprivilege.data =
                pubk->u.fortezza.DSSprivilege.data;
goto done; }
```

Рис. 1.1. Пример срабатывания NO_EFFECT.SELF_ASSIGN для nss-3.12.9+ckbi-1.82. Строка 921 содержит присваивание для одной и той же переменной. Скорее всего, ошибка является результатом опечатки.

```
x = y;
y = x;//здесь переменная x уже имеет значение y.
```

недостаточно только синтаксической информации. Таким образом, синтаксический анализ позволяет находить ошибки в исходном коде программ, описываемые шаблонами на абстрактном синтаксическом дереве. Не все дефекты могут быть обнаружены таким образом. Для поиска более сложных дефектов необходим как минимум анализ алиасов в программе, побочных эффектов вызываемых функций, межпроцедурный анализ, учёт потока управления внутри функции.

1.5. Обзор существующих семантических анализов

В данном разделе приводится обзор статических анализаторов семантического уровня для поиска дефектов.

1.5.1. Анализ на основе резюме

За последние 15 лет анализ на основе резюме стал популярным для статического поиска ошибок в исходном тексте программ. При таком анализе для каждой функции строится *резюме* – краткое описание поведения функции. Резюме используется для анализа инструкции вызова функции и позволяет избежать повторного анализа тела функции. Резюме создаётся после анализа функции.

Анализ заключается в выполнении обхода графа вызовов снизу-вверх таким образом, что вызываемые функции обходятся до вызывающих. Обработка

циклов в графе вызовов зависит от анализатора, используются следующие подходы:

- преобразование графа в ациклический с помощью разрыва ребёр и дальнейший анализ на основе топологической сортировки;
- выполнение анализа потока данных с анализом каждой функции до сходимости уравнений потока данных.

Каждая функция анализируется независимо от других, не используется какая-либо информация о контексте вызова функции. Информация распространяется только от вызываемой функции к вызывающей.

Все описываемые далее анализаторы построены на основе использования резюме. Основные отличия конкретных анализаторов заключаются в способе проведения внутрипроцедурного анализа, алгоритме создания и применения резюме и в способе обхода графа вызовов (работа с рекурсивными функциями). При создании резюме необходимо выбрать правильный баланс между точностью и количеством сохраняемой информации. Чем больше информации будет сохранено, тем более точный анализ можно построить. Но одновременно размер структур данных для хранения резюме во многом определяет скорость анализа. При анализе программ требуется хранить резюме для многих функций, поэтому слишком большой размер резюме сделает невозможным их одновременное хранение в оперативной памяти. Сохранение же на диск замедлит анализ и также зависит от размера резюме.

Анализ на основе резюме имеет ряд особенностей, обеспечивших его популярность. Анализ имеет естественную контекстную чувствительность, т. к. каждое резюме применяется для конкретного контекста вызова. Такой анализ имеет высокий потенциал для распараллеливания программы, что актуально, т. к. в настоящее время даже бюджетные машины имеют несколько процессорных ядер. Распараллеливание основано на том, что многие функции могут анализироваться независимо в соответствии с графом вызовов. Причём распараллеливание осуществляется на уровне целой функции, то есть нет необходимости использовать специальные многопоточные алгоритмы для анализа

отдельной функции, а также не требуется передавать информацию между такими функциями. Кроме этого, каждая функция анализируется ограниченное количество раз (один раз для ациклических графов вызовов), благодаря чему анализ хорошо масштабируется.

1.5.2. Консервативный анализ на основе графа чтения-записи

Консервативный анализ находит приближенное решение задачи поиска дефектов, включающее в себя все ошибки определённого типа, которые могут произойти при выполнении программы. Т. е. такой анализатор не пропускает реальные дефекты, но может выдавать ложные предупреждения.

Анализатор, описанный в [19], основан на использовании *графа чтения-записи*[2] [20, 21]. Граф чтения-записи является абстракцией для анализа указателей и позволяет изменять баланс между точностью и стоимостью анализа. В графе различаются чтение и запись в память. На основе графа можно создавать компактные резюме для функции.

В анализаторе все циклы заменяются на функции с хвостовой рекурсией, в результате чего все функции становятся ациклическими. В отличие от большинства анализаторов, описываемых в этом разделе, здесь каждая функция может анализироваться более одного раза. В анализаторе реализовано множество детекторов, включая поиск утечек памяти и ресурсов, разыменования нулевых указателей, переполнения массива.

Отличительной чертой анализатора является возможность изменять точность всего анализа. Базовым анализом является анализ, осведомлённый о потоке управления[3]. Такой анализ производится для линейной последовательности инструкций, полученной топологическим упорядочиванием инструкций графа потока управления для ациклических функций. Анализ будет различать, когда одна инструкция доминирует над другой. Но анализ условных выражений производится не точно – вначале идут инструкции одной ветки условного выражения, после чего – инструкции другой ветки. Точность анализа можно увеличивать в различных направлениях. Например, добавлять чувствительность

[2] Assign Fetch Graph

[3] flow-aware

к условиям или чувствительность к потоку.

В статье [19] приводится результаты оценки инструмента для 41 проекта. Размер большинства проектов не превышает 25 тысяч строк кода. Самый большой проект, balsa, имеет размер 110 тысяч строк кода. Время анализа для большинства проектов измерялось секундами, и для balsa составило 43 секунды. К сожалению, время анализа не включает в себя время на чтение исходных файлов, их разбор и построение промежуточного представления. Осуществлялся поиск разных видов ошибок, в том числе разыменований нулевых указателей, переполнения массивов, утечки памяти и ресурсов. Каждое предупреждение выдавалось, только если оно было дополнительно подтверждено с помощью символьного выполнения.

Всего инструмент выдал 221 предупреждение, из них 38 оценили как истинные. Т. е. уровень истинных срабатываний 17%. Можно сделать вывод, что использование консервативного анализа не является приемлемым из-за низкого процента истинных срабатываний. Кроме этого, нет гарантии, что даже такой уровень истинных срабатываний удастся сохранить при переходе к анализу проектов больших размеров.

1.5.3. Верификаторы кода

Существуют инструменты, гарантирующие поиск в программе всех дефектов определённого типа. Если в программе есть дефект, то инструмент обязан выдать предупреждение о дефекте. Если инструмент не выдаёт предупреждение, то гарантируется, что в проанализированном проекте нет дефектов определённого типа. К таким инструментам относятся Astree [22, 23], Polyspace Verifier [24].

В построении подобных инструментов достигнут существенный прогресс. Инструменты выдают немного ложных срабатываний, а для некоторых проектов вообще не выдаётся ложных срабатываний. Но в настоящий момент использование таких анализаторов ограничено проектами средних размеров. Также существенно ограничиваются конструкции, которые допустимо использовать в программе. В частности, для анализа с помощью Astree в программе не должно быть рекурсивных вызовов процедур, динамического выделения памяти, обрат-

ных goto (безусловных переходов на метки, определённые ранее), параллельного выполнения, сложных структур данных [25].

Верификаторы кода используют в программном обеспечении, где стоимость ошибки чрезмерно высока. Для таких программ важна гарантия отсутствия ошибок. Но для большинства программ этого не требуется. Если в проекте есть дефект, то это не является поводом откладывать выпуск продукта.

1.5.4. Использование символьного выполнения

Обходить пути внутри функции можно с помощью *символьного выполнения* [26]. При символьном выполнении вместо реальных входных данных для программы используются символьные. Значеними переменных являются формулы над входными символами.

Наиболее ранней работой, которую удалось найти, является описание инструмента PREfix [27]. В инструменте для анализа функции используется символьное выполнение, каждый путь рассматривается отдельно. По умолчанию анализируется только 50 путей, но количество путей может быть изменено с помощью опции. Как замечают авторы, для большинства функций достаточно рассмотрения небольшого количества путей.

Резюме функции формируется на основании всех рассмотренных путей внутри функции. Резюме является списком возможных *исходов* программы, включающих значения и свойства переменных, условия ошибок. Для каждого исхода описываются условия контекста вызова, при которых он будет использоваться. Для создания резюме запоминаются все состояния в конце каждого пути внутри функции, после чего производится операция их объединения. Сохраняются только информация об объектах, видимых извне функции (параметры, глобальные переменные, возвращаемое значение). Объединение состояний не требуется для семантики анализа, целью объединения является минимизация хранимых данных. Исходы, которые отличаются некоторыми противоположными условиями ($x > 0$ и $x \leq 0$), объединяются, сами условия удаляются. Такое резюме не обязательно полностью описывает возможное поведение функции. При объединении используется подход, что объединение не должно привести к ложным срабатываниям, но может приводить к пропуску ошибок.

Использование символьного выполнения позволяет довольно точно проанализировать отдельный путь, но плохо масштабируется на функции с большим количеством путей. Для примера, если в функции нет циклов и есть N условных ветвлений, то необходимо рассмотреть 2^N путей. В работе не описано, как происходит анализ циклов.

В инструменте PREfix реализовано значительное количество детекторов: использование неинициализированной памяти, разыменование нулевых указателей, утечки памяти и ресурсов, деление на ноль, возвращение из функции указателя на освобождённую память и другие.

Приводятся данные о влиянии количества путей на время анализа и количество выданных предупреждений. Время анализа растёт практически линейно с ростом количества рассматриваемых путей, что довольно неожиданно, т. к. большинство функций имеет небольшое количество путей. Количество выдаваемых предупреждений почти перестаёт изменяться после рассмотрения больше 100 путей внутри функции. Авторы отмечают, что распределение предупреждений по путям очень неравномерное: на 1% всех путей приходится около трети предупреждений. Поэтому результат очень сильно зависит от выбора путей, которые будут рассматриваться.

Таким образом, подход, используемый в инструменте PREfix, является хорошей отправной точкой для создания масштабируемого и точного статического анализатора. Естественным развитием подхода является борьба с взрывным ростом количества путей внутри функции и минимизация размеров резюме (в PREfix резюме поделено на исходы, которые могут в значительной мере дублировать друг друга).

1.5.5. Инструмент Saturn

Значительное количество работ [28–33] связано с инструментом Saturn. Целью анализатора является найти как можно больше дефектов с низким уровнем ложных срабатываний, который достигается в том числе и за счёт пропуска реальных дефектов. Анализ является довольно точным и гибким. Моделирование функций, не содержащих циклы, осуществляется с точностью до бита (учитывается каждый отдельный бит переменных). Анализ является межпроцедурным

контекстно-чувствительным с чувствительностью к путям, что достигается за счёт использования SAT-решателя. Свойства программы выражаются в виде формул, и с помощью SAT-решателя определяется их выполнимость. При создании резюме формулы могут округляться с целью упростить анализ. Инструмент позволяет реализовывать различные детекторы для поиска ошибок, но все они должны использовать SAT-решатель.

Анализ основан на *симуляции выполнения* процедуры. При таком подходе для анализа отдельного базового блока используется символьное выполнение. При анализе графа потока управления процедуры в точках слияния потока управления производится объединение результатов анализа отдельных базовых блоков, подобно анализу потока данных (с округлением части информации). Благодаря этому не возникает проблемы взрывного роста возможных путей внутри функции. Симуляцию выполнения можно считать развитием символьного выполнения. Так же, как и в символьном выполнении, анализ циклов не гарантирует покрытие всех возможных путей во время реального выполнения. Т. е. анализ является неконсервативным.

В анализаторе ядро анализа выполняет только базовые вещи: чтение внутреннего представления, анализ указателей. Анализ функции во многом определяется конкретным детектором, в том числе способ обхода циклов основан на эвристиках и зависит от детектора. Время анализа также сильно меняется в зависимости от используемых детекторов. В табл. 1.1 приведено сравнение времени анализа различных детекторов инструмента Saturn и текущей версии анализатора Svace [3] для проектов dnprogs-2.62, openssh-6.6p1 и wget-1.15. Использовался компьютер machine1, характеристики которого приведены в табл.A.7. Инструмент Saturn запускался с параметрами `-no-fixpoint -timeout 60` (взято из документации). Инструмент Svace без параметров. Как видно из таблицы, время анализа сильно зависит от используемых детекторов. Запуск сразу многих детекторов существенно замедлит анализ. В Svace запуск производился с использованием всех реализованных детекторов (более сотни). Добавление нового типа детекторов в Svace несущественно увеличивает время анализа.

Отличительной особенностью инструмента Saturn является то, что после нахождения ошибки анализ функции прекращается. В итоге, если в функции

Таблица 1.1. Время анализа Saturn и Svace. Время сборки не учитывалось. null.clp – детектор для поиска разыменований нулевых указателей. locking.clp – детектор для поиска ошибок блокировок мьютексов.

Инструмент	Детектор	Проект	Время анализа, с
Saturn 1.2	null.clp	dnprogs-2.62	957
		openssh-6.6p1	6701
		wget-1.15	2629
	locking.clp	dnprogs-2.62	230
		openssh-6.6p1	604
		wget-1.15	317
Svace 1.1.34	Все детекторы	dnprogs-2.62	222
		openssh-6.6p1	843
		wget-1.15	228

есть ещё ошибка, то она не будет найдена. При таком подходе непонятно, как одновременно осуществлять поиск ошибок разных типов.

В инструменте Calysto [34, 35] развиваются идеи анализатора Saturn. В инструменте по сравнению с Saturn была улучшена межпроцедурная чувствительность к путям. Инструмент использует SMT-решатель. По заверениям авторов, Calysto является более точным анализатором, имеет лучшую скорость работы и выдаёт более понятные предупреждения. По приведённым в [34] данным, Calysto в среднем вдвое быстрее Saturn (20875 секунд вместо 40226 для набора из 10 проектов, общим размером 685 тысяч строк кода). В отличие от Saturn, инструмент недоступен для использования, что автор объясняет отсутствием желания поддерживать инструмент и делать его более дружелюбным к пользователям. Тем не менее, даже двойного ускорения недостаточно для достижения масштабируемости на большие программы.

1.5.6. Инструмент ARCHER

Инструмент ARCHER [36] не пытается найти все возможные предупреждения, а сосредотачивается на поиске только ошибок доступа к памяти. Целью является проверка всех выполнимых путей внутри функции, но на практике проверяются в том числе и невыполнимые пути, а также могут пропускаться некоторые реальные пути внутри функции. В инструменте используется симуляция выполнения путей, инструмент не рассматривает все пути внутри циклов.

Анализ имеет чувствительность к путям и не рассматривает пути, для которых удалось определить их невыполнимость.

Поиск межпроцедурных ошибок основан на *триггерах* (представление ограничений, при которых в функции может произойти ошибка). Во время анализа функции создаётся множество триггеров ошибок. В точке анализа вызова функции каждый триггер проверяется с учётом контекста вызова, если триггер выполняется, то выдаётся предупреждение.

Таким образом, используемый анализ для ошибок доступа к памяти является межпроцедурным контекстно-чувствительным (за счёт применения триггеров в разных контекстах) и имеет чувствительность к путям.

1.5.7. Итеративный анализ потока данных

В качестве анализа отдельной функции можно использовать классический итеративный анализ потока данных, заключающийся в поиске фиксированной точки для уравнений потока данных. Найденное решение будет описывать свойства всех возможных путей внутри функции.

В [37, 38] описывается чувствительный к путям и потоку анализ указателей. Используется внутреннее представление IPSSA, являющееся расширением SSA [39]. Анализ указателей использует неконсервативное предположение, что все указатели, переданные в процедуру с помощью параметров, глобальных переменных и ячеек памяти, достижимых через входные параметры и глобальные переменные, не являются алиасами друг друга. Авторы аргументируют, что предположение хорошо согласуется с тем, как пишутся программы. Большое количество алиасов затрудняет поддержку программы, поэтому для большей простоты семантика большинства функций не предполагает использование алиасов.

На основе анализа указателей реализован детектор поиска уязвимостей использования непроверенных данных из внешнего источника (файлы, пользовательский ввод). Анализ запускался на множестве проектов, самым большим из которых был проект pcre размером 13 тысяч строк. Таким образом, недостаточно данных об использовании описанного анализ для анализа больших проектов.

1.5.8. Анализаторы Coverity и Klocwork

Статический анализатор от фирмы Coverity [40–43] занимает лидирующие позиции на рынке статических анализаторов. Анализатор осуществляет поиск множества дефектов, в том числе разыменования нулевых указателей, утечки ресурсов, использования памяти после освобождения, использования неинициализированных переменных, переполнения массивов.

Инструмент имеет высокие показатели производительности, масштабируемости и процента истинных срабатываний. Согласно отчёту Coverity [44], опубликованному в 2006 году, для анализа 12 тысяч строк кода достаточно 15 секунд. Такая скорость была существенно выше, чем скорость остальных анализаторов, которые оценивали авторы отчёта. Согласно тому же отчёту, инструмент имеет от 12.7% до 35.7% ложных срабатываний. Детали используемых алгоритмов неизвестны. Но известно, что производится межпроцедурный анализ потока данных. Создаётся резюме, достаточное для контекстно-чувствительного анализа. Устраняется часть путей, которые не могут произойти во время выполнения, т. е. инструмент имеет чувствительность к путям. Благодаря чувствительности к путям устраняется часть ложных срабатываний.

Анализатор Klocwork [45] является вторым по доле рынка промышленно используемым статическим анализатором. Анализатор имеет сравнимые с Coverity качество и архитектуру. Детали используемых алгоритмов также неизвестны.

В обоих инструментах анализ алиасов реализован лишь частично, значения глобальных переменных могут игнорироваться. Оба инструмента поддерживают поиск ошибок в исходном коде языков Си, Си++, Java и Си#.

В статье [46] приводится сравнение различных статических анализаторов, в том числе инструмента от Coverity и инструмента Klocwork K7. Согласно отчёту, оба инструмента могут пропускать реальные дефекты. Пересечение множества найденных дефектов меньше 20%. На основе опыта использования инструментов инженерами Ericsson были сделаны следующие выводы:

- Инструменты легко устанавливать.

- Они находят ошибки, которые было бы тяжело найти без статического

анализа.

- Возможно анализировать проекты размером в несколько миллионов строк кода за время, сравнимое с временем сборки.
- Даже для больших проектов уровень ложных срабатываний невысокий.
- Некоторые пользователи ожидали, что инструменты найдут больше дефектов. Некоторые пользователи были удивлены, что инструменты смогли найти сложные дефекты в хорошо оттестированных приложениях.
- Многие найденные дефекты не приводят к серьёзным ошибкам, но незначительное изменение кода может привести к тому, что дефект станет серьёзным.

На основании приведённой информации можно сделать вывод, что высокие показатели истинных срабатываний достигаются за счёт пропуска дефектов. Для пользователей важна высокая скорость анализа (сравнимая со скоростью сборки) и низкое количество ложных срабатываний.

Анализаторы Coverity и Klocwork выполняют все требования, предъявляемые к промышленным анализаторам (масштабируемость, высокий уровнь истинных предупреждений, поддержка популярных языков программирования, возможность расширения анализа новыми дефектами и спецификациями), но имеют значительную стоимость, их алгоритмы анализа подробно не описаны.

1.6. Анализатор Svace

Инструмент Svace разрабатывается в Институте системного программирования РАН. В настоящее время инструмент выполняет поиск дефектов для программ, написанных на языках Си, Си++, Java и C#. Целью анализа является найти как можно больше дефектов при приемлемом уровне ложных срабатываний за время, сравнимое со временем сборки проекта. В инструменте реализованы разные анализы: лексический, синтаксический, семантический, статистический.

Описание архитектуры инструмента можно найти в [2–4, 6]. Другие работы по Svace [1, 5, 7, 47–49]. Описание анализатора для Си и Си++, реализованного в компиляторе Clang, приведено в [50, 51]. Описание семантического анализатора для C# находится в [52].

На рис. 1.2 схематически показана архитектура Svace. Утилита build_capture получает на вход команду сборки проекта и осуществляет перехват запусков компилятора и компоновщика. Собранная информация используется для компиляции исходных файлов без оптимизаций. Целью такой компиляции является сохранение максимального количества информации об исходной программе и генерация внутреннего представления. Для языков Си и Си++ используется модифицированный компилятор Clang [53] и утилита компоновки. Для Java используется модифицированный компилятор JavaC [54], и для C# используется модифицированный компилятор Roslyn [55]. Список модификаций каждого компилятора исчисляется сотнями.

В перечисленных выше компиляторах реализованы синтаксические детекторы для поиска дефектов в исходном коде. Кроме этого, компиляторы используются для генерации промежуточного представления для семантических анализаторов.

Для Си и Си++ промежуточным представлением являются биткод LLVM, для Java – байткод виртуальной машины, для C# – абстрактное синтаксическое дерево. Используются два вида семантических анализаторов: анализатор для C# и анализатор для Си/Си++/Java. Результатом работы анализаторов является список предупреждений, который затем можно экспортировать в сервер истории предупрждений. Сервер истории хранит всю историю предупреждений и их разметки пользователями. Для доступа к серверу истории используется Web-клиент. Результаты анализа также можно посмотреть с помощью плагина для Eclipse [56].

В данной работе рассматривается задача улучшения анализатора семантического уровня, используемого для анализа программ, написанных на языках Си и Си++. Решается задача сохранения высокого качества анализа при переходе от средних и больших проектов размером сотни строк кода к гигантским проектам размера десятки миллион строк кода. Для этого необходимо

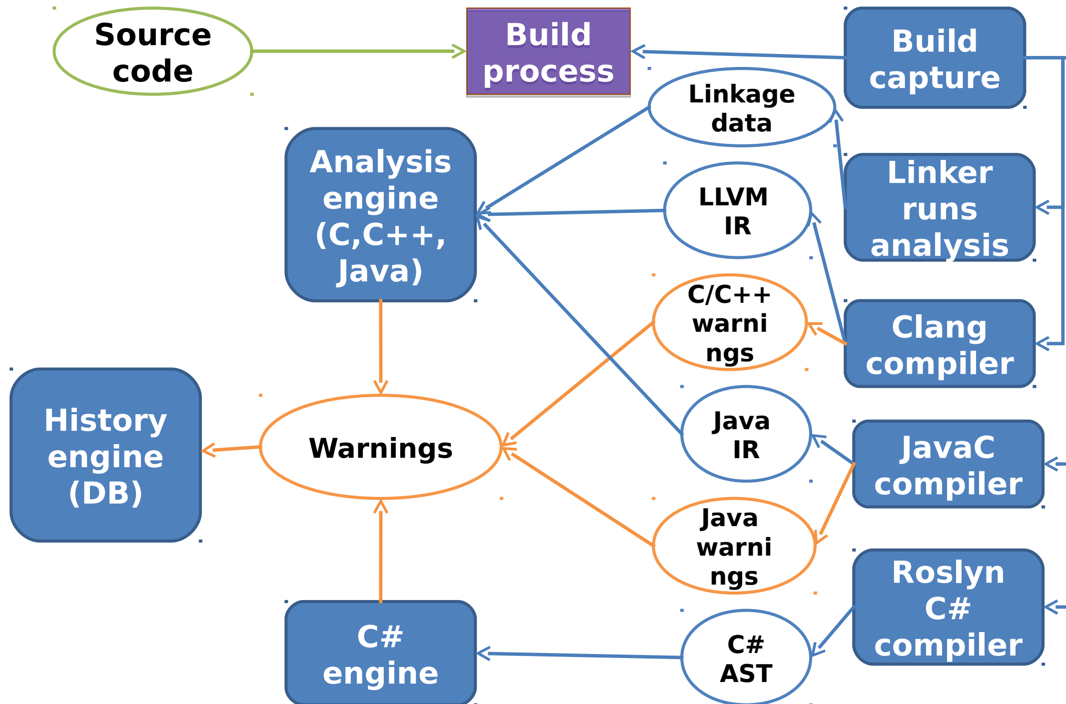

Рис. 1.2. Архитектура Svace

увеличить количество выдаваемых предупреждений при сохранении высокого уровня истинных срабатываний и масштабируемости анализа, достаточной для использования анализатора во время ночной сборки.

Количество выдаваемых ложных срабатываний для разных проектов не является случайной величиной и зависит от используемых в проекте идиом и конструкций языка. Если проект содержит сложную конструкцию, которую анализатор моделирует недостаточно точно, то результатом будет большое количество ложных срабатываний. То есть процент истинных срабатываний сильно зависит от используемого проекта. В больших проектах увеличивается количество используемых конструкций и соответственно повышается вероятность того, что встретится плохо моделируемая конструкция. Поэтому с ростом размера проекта увеличивается количество ложных срабатываний. Требуется переработка применяемых анализом эвристик для сохранения хорошего качества анализа.

В 2012 году согласно [49] среднее количество истинных срабатываний инструмента составляло 50%, при этом для некоторых детекторов семантического

уровня оно было 70-90%. Анализ проводился для проектов размером в сотни тысяч строк кода. При переходе в 2014 году к анализу больших проектов (операционные системы Tizen и Android) уровень истинных срабатываний для многих детекторов упал ниже 15%. Такой уровень был недостаточен для возможности использования инструмента на больших проектах. Анализ ложных срабатываний показал, что основными их причинами является недостаточно точное моделирование памяти; использование передаточных функций для инструкций записи в память через указатель, изменяющих значение всех указываемых ячеек памяти, что приводит к потери предыдущих значений; неточный анализ циклов; ошибки при сопоставлении формальных и фактических параметров функции при обработке вызовов в ходе межпроцедурного анализа.

Одним из условий внедрения инструмента Svace также является увеличение количества находимых дефектов. При этом недопустимо повышать процент ложных срабатываний. Пропуск некоторых дефектов был следствием фильтрации предупреждений для случаев, когда не было достаточно уверенности, что предупреждение будет истинным. Т. е. повышение точности анализа могло улучшить количество находимых дефектов.

Дополнительно на основе обзора инструментов статического анализа, приведённом в разделе 1.5, были сделаны следующие выводы:

- Консервативный анализ, покрывающий все возможные пути выполнения, не позволяет обеспечивать высокий уровень истинных срабатываний. Достижение высокого показателя истинных срабатываний возможно за счёт рассмотрения только частичного поведения программы и пропуска части дефектов.

- Резюме описывает только частичное поведение функции для сокращения его размеров. При создании резюме допустимо производить округление таким образом, чтобы это не привело к выдаче ложных срабатываний.

- При анализе программы приемлемо делать предположения об отсутствии алиасов среди параметров функции.

- Выбор путей, рассматриваемых анализатором, определяется эвристика-

ми.

- Все описанные анализаторы имеют чувствительность к контексту.
- В большинстве инструментов реализована чувствительность к путям.
- Для промышленных анализаторов важен анализ проектов в миллионы строк кода за приемлемое время.

На основании обзора и анализа предупреждений в работе доработан внутрипроцедурный анализ и анализ указателей (глава 3), предложены механизмы написания детекторов на основе разработанных алгоритмов анализа, в том числе критерий выдачи предупреждений детекторами и способ учета отдельных путей выполнения (глава 4), улучшена контекстная чувствительность детекторов (глава 5). Все предложенные алгоритмы реализованы в инструменте Svace, произведена оценка времени работы анализатора и качества выдаваемых предупреждений (глава 6).

Глава 2

Анализируемый язык

2.1. Промежуточное представление Svace

В анализаторе Svace разбор исходного кода, написанного на языках Си и Си++, осуществляется модифицированным компилятором Clang [53], генерирующим биткод LLVM [57]. Компиляция осуществляется без использования оптимизаций, и сгенерированные LLVM-файлы содержат только базовое подмножество инструкций LLVM. Целью такой компиляции является сохранение максимального количества информации об исходной программе. Код LLVM преобразуется во внутреннее представление анализатора. Выделение внутреннего представления, отличного от LLVM, позволяет анализировать байткод виртуальной машины Java, а также даёт возможность поддержки других языков.

Язык svace0 описывает упрощённое внутреннее представление анализатора. Внутреннее представление Svace включает больше типов (целые числа различной размерности, числа с плавающей точкой, строки), констант (булевы константы, константы преобразования типов, строковые константы, константы сравнения, константы для результата сдвига указателя и другие) и инструкций (побитовые сдвиги, побитовое и, побитовое или, взятие остатка от деления, вызов функции по указателю, тернарная операция, инструкции для работы с исключениями Си++ и Java, псевдоинструкции для обозначения точек возврата из функции и другие). Для целей формального описания анализа достаточно рассмотреть инструкции svace0.

2.2. Описание языка svace0

В данном разделе будет приведено неформальное описание анализируемого языка. Формальное описание можно найти в следующем разделе.

В языке svace0 все переменные всех функций имеют уникальное имя. В языке нет операции взятия адреса переменной, кроме этого, все переменные

имеют только одно присваивание. Представление программы, где все переменные имеют одно присваивание называют SSA[1] [39]. Константы языка имеют два вида: целочисленные константы и указатели на ячейки памяти.

2.2.1. Типы

Язык поддерживает следующие типы: целые числа *int*, указатели *ref*, структуры *struct* и массивы []. Все структуры должны быть описаны в начале файла до определения любых переменных и функций. На имя структуры можно ссылаться во всём файле, в том числе до определения структуры.

Примеры типов:

```
struct SName {
  int a;
  ref ref int p;
  ref struct SName next;
}
int x;
ref int p;
int[10] arr;
```

2.2.2. Частичная SSA-форма

LLVM использует вид SSA, известный как частичная SSA-форма (partial SSA form)[58, 59], в котором все переменные программы разбиваются на два класса . В один класс помещают переменные, на которые не ссылается ни один указатель, эти переменные переводятся в SSA-форму. Второй класс содержит переменные, на которые могут ссылаться указатели, эти переменные не переводятся в SSA-форму. Переменные первого класса называются *переменными верхнего уровня* (top-level variables), переменные второго класса называются *переменными со взятым адресом* (address-taken variables). Переменные верхнего уровня содержатся в потенциально бесконечном множестве псевдорегистров LLVM. Операндами всех инструкций LLVM являются псевдорегистры. Т. к.

[1] От Static Single Assignment

переменные второго класса не находятся в SSA-форме, то они могут иметь множество определений. LLVM не именует переменные второго класса, но каждая переменная со взятым адресом имеет как минимум один ассоциированный псевдорегистр, который указывает на эту переменную. Такие переменные не хранятся в памяти и могут быть доступны с помощью инструкций LLVM *load* и *store*, которые принимают в качестве аргументов переменные верхнего уровня

При компиляции без оптимизаций Clang генерирует биткод LLVM для программы в частичной SSA-форме, в которой к переменным верхнего уровня относятся только адреса переменных исходного языка и создаваемые псевдорегистры. При таком представлении в программе нет необходимости генерировать ϕ-функции, т. к. все переменные LLVM обозначают либо неизменяемые адреса переменных, либо неизменяемые псевдорегистры.

В языке svace0 также используется частичная SSA-форма. Переменные языка svace0 будем называть псевдорегистрами, чтобы избежать путаницы с переменными исходного языка.

2.2.3. Инструкции

Для удобства примем следующие соглашения об именовании: целочисленные константы обозначаются как n_i или n, константы как c_i или c, переменные как s_i или s, термы (переменные или константы) как t_i или t, где $i \in \mathbb{N}$. Переменные, являющиеся возвращаемым значением, будем обозначать r. Используем p или p_i для обозначения переменных и указателей на ячейки памяти.

$r = alloca\, T()$ – выделить память, вернуть указатель на выделенную память. T – тип выделяемой памяти, может быть пропущен для целочисленных переменных. Возвращается указатель, который не имеет алиасов среди всех созданных до этого указателей.

$r = load\, p$ – прочесть значение ячейки, на которую указывает переменная или константа p, результат присвоить переменной r.

$store\, t, p$ – переменную или константу t поместить в ячейку памяти, на которую указывает p.

Инструкция $shift$ позволяет сдвигать указатель на структуру или массив "вглубь" структуры. Обратный сдвиг не разрешён. Инструкция является упро-

щённой версией LLVM-инструкции getelementptr.

$r = shift\, p, f$ – сдвинуть указатель p, имеющий тип “указатель на структуру с полем f”, на смещение поля относительно начала структуры. Вернуть указатель на поле f внутри структуры.

$r = shift\, p, s$ – сдвинуть указатель p, имеющий тип “указатель на массив на s элементов”. Вернуть указатель на элемент s внутри массива.

Арифметические инструкции:

$r = t_1 + t_2$ – сложить t_1, t_2.

$r = t_1 - t_2$ – вычитание.

$r = t_1 * t_2$ – умножение.

$r = t_1 / t_2$ – деление.

Для простоты язык является структурированным и не имеет инструкций безусловного перехода. if – инструкция ветвления, $while$ – инструкция для выполнения цикла.

$if(bexp, code_{true}, code_{false})$ – если $bexp$, то выполнить $code_{true}$, иначе $code_{false}$. При этом $bexp$ может иметь форму: $a = b$, $a \neq b$, $a > b$, $a < b$, $a \geq b$, $a \leq b$.

$while(bexp, code)$– до тех пор, пока bexp, выполнять code.

$r = t$ – инструкция присваивания.

Отдельные инструкции разделяются точкой с запятой:
$instr_1; instr_2$ – последовательность из двух инструкций, где $instr_1$ выполняется до $instr_2$.

Определение функции, возвращающей значение:

$$def\, func(a_1, .., a_n) = body; ret\, a_r$$

Определение функции, не имеющей возвращаемого значения:

$$def\, func(a_1, .., a_n) = body; ret,$$

где $body$ – последовательность инструкций, $func$ – имя определяемой функции. Каждая функция имеет только одну инструкцию возврата управления.

$call\, func(t_1, .., t_n)$ – инструкция вызова функции $func$, передача параметров по значению.

$r = call func(t_1, .., t_n)$ – инструкция вызова функции $func$, имеющей возвращаемое значение.

Выполняемая программа состоит из последовательности объявленных функций, после чего следует последовательность инструкций для выполнения.

Рис. 2.1 содержит описание синтаксиса языка в виде БНФ [60].

$\mathbf{Program} ::= \mathbf{StructDef}^{*}\ \mathbf{FuncDef}^{*}\ \mathbf{Instructions}$
$\mathbf{StructDef} ::= struct\ \mathbf{Name}\ \{\mathbf{StructField}^{*}\}$
$\mathbf{StructField} ::= \mathbf{Type}\ \mathbf{Name}$
$\mathbf{Type} ::= int \,|\, ref\ \mathbf{Type} \,|\, struct\ \mathbf{Name} | Type[Number]$
$\mathbf{FuncDef} ::= def \mathbf{Name}\,(\mathbf{Arguments}\,) = \mathbf{FuncFullBody}$
$\mathbf{FuncFullBody} ::= \mathbf{Instructions}\,; ret \,|\, \mathbf{Instructions}\,; ret\ \mathbf{Name}$
$\mathbf{Arguments} ::= \mathbf{Name} \,|\, \mathbf{Arguments}, \mathbf{Name}$
$\mathbf{Instructions} ::= \mathbf{Instruction} \,|\, \mathbf{Instructions}, \mathbf{Instruction}$
$|\, if\ (\mathbf{Bexp}, \mathbf{Instructions}\ , \mathbf{Instructions})$
$|\, while\ (\mathbf{Bexp}\ , \mathbf{Instructions})$
$\mathbf{Bexp} ::= \mathbf{Name}\ \mathbf{Op}\ \mathbf{Name}$
$\mathbf{Op} ::= = \ |\ \neq\ |\ >\ |\ <\ |\ \geq\ |\ \leq$
$\mathbf{Name} ::= \mathbf{Letter} \,|\, \mathbf{Name} \,|\, \mathbf{Letter}$
$\mathbf{Letter} ::= a..z \,|\, A..Z \,|\, \mathbf{Digit}$
$\mathbf{Digit} ::= 0..9$
$\mathbf{Instruction} ::= \mathbf{Name}\ = alloca() \,|\, \mathbf{Name}\ = alloca\ \mathbf{Name}()$
$|\, \mathbf{Name}\ = load\ \mathbf{Name} \,|\, store\ \mathbf{Name}, \mathbf{Name} \,|\, \mathbf{Name}\ =\ \mathbf{Name}$
$|\, shift\ \mathbf{Name}, \mathbf{Name} \,|\, shift\ \mathbf{Name}, \mathbf{Number}$
$|\, \mathbf{Name}\ = \mathbf{Name}\ +\ \mathbf{Name} \,|\, \mathbf{Name}\ = \mathbf{Name}\ -\ \mathbf{Name}$
$|\, \mathbf{Name}\ = \mathbf{Name}\ *\ \mathbf{Name} \,|\, \mathbf{Name}\ = \mathbf{Name} \,/\, \mathbf{Name}$
$|\, call\ \mathbf{Name}\,(\,\mathbf{Arguments}\,)\ |\, \mathbf{Name}\ = call\ \mathbf{Name}\,(\,\mathbf{Arguments}\,)$

Рис. 2.1. БНФ синтаксиса языка svace0

2.3. Структурная операционная семантика языка

Рассмотрим фиксированную процедуру. Пусть R – множество переменных процедуры (пседоврегистров), C – бесконечное множество констант. $M \subset C$ – подмножество констант, которое будет использоваться для обозначения ячеек памяти. Ячейки памяти могут создаваться только с помощью инструкции $alloca$.

Константа undef $\in$ C является специальной константой для обозначения неинициализированной памяти. Константа $m.f$ используется для обозначения ячеек памяти для полей структур, где f – поле структуры. Для обозначения ячеек памяти для элементов массивов используется константа $m.c$, где c – номер элемента массива. Предполагается, что определена функция $offset : M \times M \to \mathbb{N}$, которая для указателя на композитный тип (структуру или массив) и указателя на элемент композитного типа вернёт смещение в байтах второго указателя относительно первого.

Конкретное состояние программы описывается двумя частичными функциями: $\nu_R : R \to C$ и $\nu_M : R \to C$. Функция ν_R возвращает значение для псевдорегистра, функция ν_M возвращает значение для ячейки памяти. Для более краткой записи вместо функций ν_R и ν_M будет использоваться функция $\nu : R \cup M \to C$, определённая следующим образом:

$$\nu(a) = \begin{cases} \nu_R(a), & \text{если } a \in R; \\ \nu_M(a), & \text{если } a \in M. \end{cases}$$

Будем называть подстановкой c в t для функции f функцию:

$$f[t \mapsto c](x) = \begin{cases} c, & \text{если } t = x; \\ f(x), & \text{иначе.} \end{cases}$$

Будем обозначать несколько подстановок следующим образом:
$f[t_1 \mapsto c_1, t_2 \mapsto c_2] = f[t_1 \mapsto c_1][t_2 \mapsto c_2]$.

Инструкции программы модифицируют состояние программы. Семантическая функция инструкции для каждого состояния до выполнения инструкции возвращает состояние программы после выполнения инструкции. Опишем семантику инструкций языка svace0 с помощью суждений вида $v_1, code_1 \longmapsto v_2$ и $v_1, code_1 \longmapsto v_2, code_2$, где v_1 описывает состояние программы до выполнения кода $code_1$, а v_2 – состояние после выполнения [61, 62]. Суждения первого вида описывают ситуацию, когда все инструкции программы были выполнены.

$$\frac{m - fresh\ m \in M}{\nu, r = alloca() \longrightarrow \nu[m \mapsto undef, r \mapsto m]}$$

После выполнения инструкции *alloca* регистр r будет указывать на вновь созданную ячейку памяти, имеющую неопределённое значение.

$$\frac{\nu(p) = m \ \ \nu(m) = c}{\nu, r = load\, p \longrightarrow \nu[r \mapsto c]}$$

При выполнении инструкции *load* значение c, содержащееся в ячейке памяти m, на которую указывает регистр p, будет присвоено регистру r.

Для описания инструкции *store* потребуется вспомогательная частичная функция *value*, определённая следующим образом:

$$value(t, \nu) = \begin{cases} t & \text{если } t \in C; \\ \nu(t) & \text{если } t \in dom(\nu). \end{cases}$$

Запись $dom(\nu)$ используется для обозначения области определения функции ν.

$$\frac{\nu(p) = m \ \ value(t, \nu) = c}{\nu, store\, t,\ p \longrightarrow \nu[m \mapsto c]}$$

После выполнения инструкции pmove в ячейку m, на которую указывает регистр p, будет помещено значение t. При этом t может быть как константой, так и регистром.

Выполнение инструкции *shift* описывается двумя правилами: для структур и для массивов.

$$\frac{\nu(p) = m \ field(p, f)}{\nu, r = shift\ p, f \longrightarrow \nu[r \mapsto m.f]}$$

$$\frac{\nu(p) = m \ c = value(t, \nu) \ elem(p, c)}{\nu, r = shift\ p, t \longrightarrow \nu[r \mapsto m.c]}$$

Предикат *field* истинен, если указатель p имеет тип структуры с полем f. Предикат *elem* истинен, если размер массива достаточен для доступа к элементу с индексом c.

$$\frac{c_r = value(t_1, \nu) + value(t_2, \nu)}{\nu, r = t_1 + t_2 \longrightarrow \nu[r \mapsto c_r]}$$

Описание инструкций вычитания, умножения и деления аналогично.

$$\frac{c_t = value(t, \nu)}{\nu, r = t \longrightarrow \nu[r \mapsto c_t]}$$

Выполнение последовательности инструкций:

$$\frac{\nu_1, code_1 \longrightarrow \nu_2}{\nu_1, code_1; code_2 \longrightarrow \nu_2, code_2}$$

$$\frac{\nu_1, code_1 \longrightarrow \nu_2, code_1'}{\nu_1, code_1; code_2 \longrightarrow \nu_2, code_1'; code_2}$$

$$\frac{istrue(bexp, \nu)}{\nu, if(bexp, code_{true}, code_{false}) \longrightarrow \nu, code_{true}}$$

Частичная функция $istrue$ определена следующим образом:

$$istrue(a \oplus b, \nu) = \{v_a \oplus v_b,\ \text{где}\, v_a = value(a, \nu)\ v_b = value(b, \nu)\}$$

и $\oplus$ принимает значения из множества: $\{=, \neq, >, <, \geq, \leq\}$.

Если $expr$ истинно, то состояние после выполнения всего условного выражения будет эквивалентно состоянию после выполнения $code_{true}$.

$$\frac{not\, istrue(bexp, \nu)}{\nu, if(bexp, code_{true}, code_{false}) \longrightarrow \nu, code_{false}}$$

$$\frac{istrue(bexp, \nu)}{\nu, while(bexp, code) \longrightarrow \nu, code; while(a = b, code)}$$

$$\frac{not\, istrue(bexp, \nu)}{\nu, while(bexp, code) \longrightarrow \nu}$$

$$\frac{def func(a_1, .., a_n) = body; ret\, a_r}{\nu, r = func(t_1, .., t_n) \longrightarrow \nu, a_1 = t_1; ..; a_n = t_n; body; r = a_r}$$

Из всех инструкций состояние памяти изменяют только инструкции $alloca$ и $store$. Чтение из памяти осуществляет только инструкция $load$.

2.4. Функция на языке svace0

Программа на языке Си может быть оттранслирована в программу на языке svace0. Рассмотрим функцию на языке Си:

```
int calc(int f) {
    int a = 1, b = 2;
    int*p = f? &a : &b;
    int x = *p;
    *p = 3;
    int y = *p;
    return a + x + y;
}
```

Эта функция в зависимости от параметра f возвращает $3 + 1 + 3 = 7$ или $2 + 1 + 3 = 6$. Рис. 2.2 содержит функцию на языке svace0, эквивалентную вышеприведённой Си-функции.

```
 1: def calc(f) =
 2:     a = alloca();
 3:     b = alloca();
 4:     p = alloca ref int();
 5:     store 1, a1;
 6:     store 2, b1;
 7:     if(f!=0,
 8:         store a1, p1,
 9:         store b1, p1
10:     );
11:     t1 = load p1;
12:     x1 = load t1;
13:     store 3, t1;
14:     t2 = load p1;
15:     y1 = load t2;
16:     t3 = load a1;
17:     t4 = t3 + x1;
18:     t5 = t4 + y1;
19:     ret t5; }
```

Рис. 2.2. Функция на языке svace0

Переменные языка svace0 a, b и p обозначают адреса переменных a, b и p языка Си соответственно. Псевдорегистры $t3$, $t4$ и $t5$ используются для реализации сложения. Переменные $x1$ и $y1$ обозначают переменные x и y Си-программы. Для этих переменных не создаётся память на стеке, т. к. у них не берётся адрес и нет необходимости генерировать ϕ-функции.

Глава 3

Внутрипроцедурный анализ

В главе описывается ядро анализа функции. Ядро вычисляет базовые данные про память и значения переменных. Ядро не занимается поиском ошибок, для этого используются детекторы, которые будут описаны в главе 4. Ядро выделено для масштабируемости и расширяемости анализа.

3.1. Используемый подход

Рассматривается задача анализа только для некоторых состояний на входе в процедуру языка svace0. Разрешёнными состояниями будут такие состояния, где множество всех достижимых значений, начиная с параметров процедуры, не является алиасами.

Будем использовать *абстрактное состояние* (АС) для того, чтобы выражать множество конкретных состояний, являющихся его конкретизацией.

Задача анализа заключается в создании пар: абстрактное состояние и *множество анализируемых путей*, где множество путей – это конечное непустое множество путей на графе потока управления, следующих из точки входа функции в некоторую точку (одну и ту же для всех элементов этого множества).

Результат анализа будет *корректным*, если для любого пути исполнения, начинающегося для разрешённого состояния из точки входа, и принадлежащего множеству анализируемых путей, его конкретное состояние в текущей точке должно принадлежать конкретизации абстрактного состояния.

В работе [37] было показано, что использование эвристик об отсутствии алиасов среди входных параметров является приемлемым для поиска дефектов в исходном коде. Это предположение выполняется для большинства функций. А нарушение предположения не ведёт к росту количества выдаваемых ложных предупреждений.

Конкретные состояния определены в разделе 2.3. Обозначим $\Pi : R \cup M \to C$ – множество конкретных состояний.

Обозначим Ψ – множество абстрактных состояний. Абстрактные состояния будут определяться функциями val_R, val_A и pt, которые будут описаны в разделах 3.2.2, 3.2.3.

Для анализа построим граф потока управления функции на языке svace0. В качестве вершин будем использовать инструкции, а не базовые блоки. Выполняется обход вершин графа потока управления. С каждым ребром графа ассоциируется абстрактное состояние анализа. Анализ продвигает состояние по графу потока управления. Если вершина имеет несколько входящих рёбер, то выполняется операция объединения состояний (раздел 3.7). Ациклические графы обходятся в топологическом порядке.

Анализ циклов рассматривается в разделе 3.4. Для каждого цикла производится несколько итераций обхода тела цикла. Для рёбер, выходящих из цикла, сохраняются все абстрактные состояния, соответствующие отдельным итерациям, и в конце анализа цикла производится объединение этих абстрактных состояний.

Опишем причины, по которым для анализа функции не используются такие подходы, как символьное выполнение программы и итеративный анализ потока данных. При использовании символьного выполнения даже в функции без циклов количество возможных путей растёт экспоненциально относительно числа вершин в графе потока управления. Поэтому в инструментах, использующих символьное выполнение (например, PREfix [27]), количество путей ограничивается некоторой константой, и используются эвристики для выбора путей. В результате даже простой код в функциях, содержащих множество путей, может быть не проанализирован.

Анализ потока данных [63] позволяет проанализировать все пути внутри функции для всех возможных входных данных. Отказ от анализа потока данных был сделан по следующим причинам:

- Текущая версия Svace использует три итерации обхода цикла. Для анализа потока данных в большинстве случаев трёх итераций недостаточно, особенно если используются решётки с большой высотой. Большее количество итераций замедлит анализ.

- Для сходимости анализа потока данных используемые решётки должны иметь небольшую высоту. В используемом алгоритме нет требования к высоте решётки, а также к её конечности.

- Если производить анализ потока данных над определениями переменных, то в передаточных функций необходимо будет при прохождении через определение переменной обновлять все ссылки на эти определения. Обход всех ссылок займёт значительное количество времени. Для более эффективного удаления ссылок необходимо будет запоминать все возможные ссылки, что усложняет анализ. Описываемый анализ основан на идентификаторах значений, имеющих свойство *мобильности* – после создания идентификаторов значений свойства программы можно выражать в виде функций над идентификаторами значений. Нет необходимости изменять ссылки на идентификаторы значений после прохождения через определения переменных, а также в других случаях.

- Для того, чтобы определить, что фиксированная точка достигнута, необходимо сравнить текущее состояние анализа с предыдущим, на что может потребоваться значительное количество времени. Дополнительно необходимо запоминать предыдущее состояние.

Описываемый анализ можно рассматривать как комбинацию символьного выполнения и анализа потока данных.

3.2. Идентификаторы значений и ссылки

3.2.1. Классы эквивалентности переменных

Во многих случаях переменные в программе имеют одинаковые значения. В следующем фрагменте кода на Си

```
a = b + 2;
c = a;
d = b + 2;
```

есть три определения для переменных a, c и d. Несложно увидеть, что после выполнения последней инструкции значения всех трёх переменных будут одни и те же. Определение классов эквивалентности значений переменных называют *нумерацией значений* [64–66]. Нумерация значений, основанная на хэшировании, объединяет значения переменных в один класс эквивалентности, если символьные выражения, определяющие значения переменных, состоят из одних и тех же функций и соответствующие аргументы функций одинаковые. Алгоритм хэширования сопоставляет уникальное число, *номер значения*, деревьям разбора для выражений. Две переменные будут равными, если они имеют одинаковые номера значений.

При этом если переменным сопоставлены разные номера значений, то переменные могут иметь одинаковые значения. Даже для тех случаев, когда переменные всегда имеют одни и те же значения, не всегда просто установить это во время анализа. Например, для фрагмента Си-кода

```
x = -b + (a + b) / 2;
y = (a - b) / 2;
```

переменные x и y имеют всегда одни и те же значения [1], но не всякий анализ сможет это установить.

3.2.2. Идентификаторы значений

Ключевую роль в описываемом анализе играют *идентификаторы значений*. Идентификаторы, так же, как и номера значений, используются для объединения значений переменных в классы эквивалентности. Идентификаторы значений обозначают значения как переменных программы, так и неименованных ячеек памяти, значения которых могут быть получены из значений переменных программы с помощью последовательностей сдвигов и разыменований. Идентификатор значения – это номер, который был присвоен переменной во время симуляции выполнения программы. Идентификаторы значений похожи на номера значений, и для них могут использоваться техники объединения в классы эквивалентности, используемые для номеров значений.

[1] В работе не рассматриваются вопросы переполнения машинных чисел.

Нумерация значений на основе идентификаторов значений имеет следующие отличия от нумерации значений, используемой оптимизируемыми компиляторами:

- Нет задачи проведения максимального объединения переменных в классы эквивалентности. Нумерация значений обычно применяется компиляторами для оптимизации программы. Чем больше переменных будет объединено в классы эквивалентности, тем точнее будет анализ и тем лучше программа будет оптимизирована. При анализе программы для поиска ошибок могут возникнуть сложности с объяснением пользователю, почему некоторая ситуация является ошибкой. Поэтому агрессивная нумерация значений позволит найти больше дефектов, но ухудшит понятность каждого предупреждения. В результате предупреждения могут быть размечены как ложные.

- При обходе определения в цикле анализатор присвоит тот же самый идентификатор значения, только если сможет доказать, что значение переменной не изменилось.

- Нумерация значений проводится для определений переменных. Описываемый анализ может назначить разные номера переменной при повторном прохождении через её определение.

- Описываемый анализ использует некоторые предположения, которые некорректны в общем случае. При использовании излишне агрессивной нумерации значений в случае нарушения предположений ошибка распространится на все переменные одного класса эквивалентности. По этой причине в анализе даже константы не объединяются в один класс эквивалентности.

- Нумерацию значений можно рассматривать как трансформацию промежуточного представления программы. В результате нумерации значений будет получено другое промежуточное представление. В описываемом анализе нельзя выделить фазу нумерации значений. Нумерация значений выполняется одновременно с анализом указателей и эффектов от вызова функций.

Нумерация значений выполняется для присваиваний, сдвигов указателей и арифметических операций. Для хэширования идентификаторов значений будем использовать функции $+$, -, * и $/$, аргументами которых являются идентификаторы значений. Эти функции возвращают один и тот же идентификатор значения для одинаковых аргументов.

Пусть V – множество идентификаторов значений. Для сопоставления идентификаторов значений переменным языка svace0 используется функция $val_R : R \to V$. В главе 2.3 для описания конкретного состояния использовалась функция $\nu_R : R \to C$. Идентификаторы значения объединяют переменные программы в классы эквивалентности. Функция val_R является частью абстрактного состояния. Конкретизацией идентификаторов значений будут константы. Состояние анализа val_R определяет вид конкретных состояний, которые оно описывает. Абстрактное состояние val_R описывает конкретное состояние ν_R, только если из $val_R(a) = val_R(b)$ следует, что $\nu_R(a) = \nu_R(b)$.

Функция $joinval : V \times V \to V$ для двух идентификаторов значений возвращает идентификатор значений, обозначающий свойства обоих. Реализация функции следующая:

$$joinval(v_1, v_2) = \begin{cases} v_1, & \text{если } v_1 = v_2; \\ fresh(), & \text{иначе} \end{cases}$$

Вспомогательная функция $fresh$ создаёт новый идентификатор значения, отличный от всех предыдущих.

С помощью функции $joinval$ разобьём множество V на два непересекающихся множества SV и JV, определённых следующим образом:
$JV = \{joinval(v_1, v_2) | v_1, v_2 \in Val\}$
$SV = V \setminus JV$.

Как видно из определения множеств, $V = SV \cup JV$. Элементы множества JV будем называть *составными идентификаторами значений*, элементы множества SV будем называть *простыми идентификаторами значений.*

3.2.3. Ссылки

Другим важным понятием в анализе является *ссылка*. Ссылка является простым идентификатором значения, обозначающим значения указателей. Для обозначения ссылок будем использовать множество A. При этом $A \subseteq SV$, $A \subseteq V$.

Конкретизацией идентификаторов значений при конкретном выполнении будут константы. Конкретизацией ссылок при конкретном выполнении будут ячейки памяти из множества M (глава 2.3), являющиеся подмножеством констант.

Для ссылок поддерживается свойство непересечения. Две ссылки никогда не обозначают одну и ту же ячейку памяти. Выполнение этого свойства обеспечивается анализом и является одним из условий корректности анализа. Множество ссылок выбирается таким образом, чтобы обеспечить выполнение свойства непересечения. Поэтому не для каждой ячейки памяти будет создана моделирующая её ссылка.

Определим частичную функцию $deref : V \to A$, которая определена для идентификаторов значений, являющихся ссылкой, и возвращающая свой аргумент:

$$deref(v) = v, \text{ если } v \in A$$

Ссылки выполняют в анализе две функции: отслеживание значений ячеек памяти и выполнение анализа указателей.

Анализ отслеживает значения ячеек памяти, сопоставляя ссылкам идентификаторы значения. Для этого используется функцию $val_A : A \to V$. Если $val_A(a) = v$, то значение ячейки памяти, обозначаемой ссылкой a, будет иметь класс эквивалентности, который определяется идентификатором значения v.

В дальнейшем вместо функций val_R, val_A будет использоваться функция $val : R \cup A \to V$, определённая следующим образом:

$$val(x) = \begin{cases} val_R(x), & \text{если } x \in R; \\ val_A(x), & \text{если } x \in A \end{cases}$$

Множество всех функций val будем обозначать как Val.

При использовании ссылок для выполнения анализа указателей отношения указывания моделируются функцией $pt : V \to \wp(A)$, которая для каждого идентификатора значений возвращает множество ссылок, на которые он может указывать. Функция pt ограничивает форму памяти в конкретных состояниях: если $pt(v) = \{a_1, a_2\}$, то указатель, являющийся конкретизацией v, должен указывать на ячейку памяти, являющуюся конкретизацией одной из ссылок из множества $\{a_1, a_2\}$. Множество всех функций отношений указывания pt обозначим как Pt.

Если два указателя указывают на одну и ту же ячейку памяти, то говорят, что они *алиасы* или являются алиасами. Будем называть два идентификатора значения алиасами, если указатели, являющиеся их конкретизацией, могут быть алиасами. Формально идентификаторы значений v_1 и v_2 будут алиасами, если $pt(v_1) \cap pt(v_2) \neq \varnothing$, т. е. два идентификатора значения могут указывать на одну ссылку, обозначающую некоторую ячейку памяти. Идентификаторы значения не являются алиасами, если они указывают на разные ссылки.

Рассмотрим анализ указателей для простого фрагмента Си-кода:

```
int a = 1, b = 2;
int*p = f? &a : &b;
int x = *p;
```

Этот фрагмент может быть оттранслирован в следующий код на языке svace0:

```
a = alloca();
b = alloca();
p = alloca();
store 1, a;
store 2, b;
if(f!=0,
    store a, p,
    store b, p
};
```

```
t1 = load p;
x = load t1;
```

После выполнения этого фрагмента указатель p может указывать либо на переменную a, либо на переменную b. Память переменных a и b может быть изменена косвенно через указатель p. В программе есть следующие алиасы: $(*p, a, t1)$, $(*p, b, t1)$. Т. к. нельзя заключить, что $p = \&a$ или что $p = \&b$, то им должны быть сопоставлены разные идентификаторы значений. Но также нельзя сделать вывод, что $p \neq \&a$ и $p \neq \&b$, поэтому нельзя сопоставить им разные ссылки (свойство непересечения ссылок). Поэтому анализ создаёт ссылки для переменных a и b, но не создаёт ссылку для переменной p.

После анализа этого фрагмента функции val и pt будут иметь следующий вид:
$val(a) = v_a$, $val(b) = v_b$, $val(p) = v_p$,
$val(v_a) = v_1$, $val(v_b) = v_2$, $val(v_p) = v'$,
$pt(v_a) = \{v_a\}$, $pt(v_b) = \{v_b\}$, $pt(v_p) = \{v_a, v_b\}$.

3.3. Передаточные функции

Передаточная функция для каждой инструкции трансформирует входное состояние в выходное состояние. Для обозначения передаточной функции для инструкции $instr$, трансформирующей входное состояние $< val_1, pt_1 >$ в состояние $< val_2, pt_2 >$, будет использоваться следующая нотация:

$$val_1\|instr\| \mapsto val_2$$
$$pt_1\|instr\| \mapsto pt_2$$

Будем обозначать первый элемент состояния анализа $s \in \Psi$ как $s.val$; второй элемент как $s.pt$.

3.3.1. Вспомогательные функции

В данном разделе определим вспомогательные функции, которые будут использоваться для описания передаточных функций.

Функция $const$ создаёт идентификатор значения для константы.

Функции +, -, * и % создают идентификаторы значения для результата арифметических выражений. Их реализация выполняет нумерацию значений на основе хэширования типа инструкции и идентификаторов значений для аргументов. Функция field создаёт идентификатор значения для результата сдвига. Аргументами функции являются идентификатор значения и поле структуры. Реализация основана на хэшировании аргументов.

Вспомогательная функция *getval* возвращает значение идентификатора, используя функции *val*, либо создаёт идентификатор значения.

$$getval(a, val) = \begin{cases} v = val(a), & \text{если } a \in dom(val); \\ fresh(), & \text{иначе} \end{cases}$$

Функция *getpt* возвращает множество указываемых ссылок, используя функции *pt*, либо возвращает множество из единственного элемента – ссылки для аргумента v.

$$getpt(v, pt) = \begin{cases} pt(v), & \text{если } v \in dom(pt); \\ \{deref(v)\}, & \text{иначе} \end{cases}$$

Функция *joinvals* для множества идентификаторов значений создаёт новый, обозначающий их слияние, с помощью функции *joinval*.

Функция *getjoinval* возвращает идентификатор значения для памяти, на которую указывает переменная, обозначаемая v. V_p – обозначает список идентификаторов значений для указываемых ячеек. Если V_p состоит из одного идентификатора значений, то возвращается этот идентификатор значений. Если V_p содержит более одного элемента, то используется функция *joinvals* для выбора идентификатора значения, обозначающего свойства всего множества V_p.

$$getjoinval(v, V_p) = \begin{cases} w, & \text{если } V_p = \{w\}; \\ joinvals(V_p), & \text{иначе} \end{cases}$$

Функция update производит сильное или слабое обновление в зависимости от количества ссылок: сильное обновление делается для одной ссылки, иначе используется слабое обновление с помощью функции *joinval* с аргументами v и предыдущим значением каждой ссылки.

$$update(A_p, v, val) = \begin{cases} val[a \mapsto v], & \text{если } A_p = \{a\} \\ update(A_p \setminus \{a\}, v, val'), & \text{иначе, где } a \in A_p, \\ & val' = val[a \mapsto joinval(v, val(a))] \end{cases}$$

3.3.2. Определение передаточных функций

Передаточная функция для инструкции *alloca*:

$$\begin{array}{l} val\|r = alloca()\| \ \mapsto val[r \mapsto v_r] \\ pt\|r = alloca()\| \ \mapsto pt[v_r \mapsto \{deref(v_r)\}] \end{array}, \text{ где } v_r = fresh().$$

Передаточная функция для инструкции загрузки из памяти *load*:

$$\begin{array}{l} val\|r = load\, p\| \ \mapsto val[p \mapsto v_p, r \to v_r] \\ pt\|r = load\, p\| \ \mapsto pt[v_p \mapsto A_p] \end{array},$$

где $v_p = getval(p, val), A_p = getpt(v_p, pt), v_r = getjoinval(v_p, V_p)$, V_p – значения A_p.

Передаточная функция для инструкции записи в память *store*:

$$\begin{array}{l} val\|store\, t,\ p\| \ \mapsto val'[t \mapsto v_t, p \mapsto v_p] \\ pt\|store\, t,\ p\| \ \mapsto pt[v_p \mapsto A_p] \end{array},$$

где $v_t = getval(t, val), v_p = getval(p, val), A_p = getpt(v_p, pt)$, $val' = update(A_p, v_t, val)$, V_p – новые значения A_p.

Передаточная функция для инструкции присваивания:

$$val\|r = t\| \ \mapsto val[r \mapsto v_t, t \mapsto v_t], \text{ где } v_t = getval(t, val).$$

Передаточная функция для инструкции сдвига указателя на константу (поле структуры):

$$\begin{array}{l} val\|r = shift\, p,\ f\| \ \mapsto val[r \mapsto v_r, p \mapsto v_p] \\ pt\|r = shift\, p,\ f\| \ \mapsto pt[v_r \mapsto \{deref(v_r)\}] \end{array},$$

где $v_r = field(v_p, f)$, $v_p = getval(p, val)$.

Передаточные функции для арифметических операций:

$$\begin{array}{l} val\|r = t_1 + t_2\| \ \mapsto val[r \mapsto v_1 + v_2, t_1 \mapsto v_1, t_2 \mapsto v_2] \\ val\|r = t_1 - t_2\| \ \mapsto val[r \mapsto v_1 - v_2, t_1 \mapsto v_1, t_2 \mapsto v_2] \\ val\|r = t_1 * t_2\| \ \mapsto val[r \mapsto v_1 * v_2, t_1 \mapsto v_1, t_2 \mapsto v_2] \\ val\|r = t_1 \% t_2\| \ \mapsto val[r \mapsto v_1 \% v_2, t_1 \mapsto v_1, t_2 \mapsto v_2] \end{array}, \text{ где } v_1 = getval(t1, val),$$

$v_2 = getval(t2, val)$. Здесь символы $+, -, *$ и $\%$ используется для обозначе-

ния инструкций в передаточных функций и для обозначения вспомогательных функций, оперирующих с идентификаторами значений.

Передаточные функции инструкций ветвления не изменяют состояние анализа, поэтому описываться не будут.

3.4. Анализ циклов

На вход анализатору подаётся граф потока управления. В графе выделяются сильно *связанные компоненты графа*, а затем части графа, обозначающие циклы программы. Для выделения сильно связанных компонент используется алгоритм из [67]. После чего производится выделение *естественных циклов*[2] [68]. Естественные циклы с одним заголовком рассматриваются как один цикл.

Анализ производит несколько итераций каждого цикла. В текущей версии анализатора каждый цикл обходится три раза. На первой итерации игнорируются обратные рёбра, на всех последующих итерациях игнорируются прямые рёбра. Для ребёр, выходящих из цикла, на каждой итерации сохраняется состояние анализа. После завершения обходов цикла производится объединение состояний на рёбрах, выходящих из цикла, для всех итераций цикла.

Каждый цикл обходится отдельно, поэтому тело внутреннего цикла для вложенных циклов будет обходиться N^D раз, где N – количество обходов цикла, а D – глубина вложенности циклов. В программах редко пишут циклы большой вложенности, но даже для циклов небольшой вложенности (не больше 5) количество обходов может достигнуть $3^5 = 243$, что замедлит анализ. В целях ускорения анализа для циклов вложенности больше двух количество итераций уменьшается на единицу. В этом случае для циклов с вложенностью 5 количество обходов будет $3^2 * 2^3 = 72$.

С помощью трёх обходов цикла можно промоделировать поведение функций, в которых циклы выполняются не более трёх раз. В реальных программах циклы могут иметь значительно больше итераций. Чтобы анализировать циклы, имеющие больше трёх итераций, на первой итерации не используется точ-

[2] Естественный цикл имеет один входной узел, называемый *заголовком цикла*, доминирующий над всеми узлами цикла, и в цикле существует обратное ребро, ведущее в заголовок цикла.

ная нумерация значений. В результате для переменных цикла создаются новые идентификаторы значений, имеющие произвольные свойства. На последующих итерациях их свойства будут уточнены. Дополнительно выполняется анализ индуктивных переменных цикла, для которых вычисляется шаг изменения в цикле. Рассмотрим анализ следующего цикла:

```
for(i=0;i<10000;i++) {
        if(i==500)
                break;
}
```

В конце первой итерации для переменной i будет установлен новый идентификатор значений. При прохождении через условия $(i < 10000)$ и $(i == 500)$ анализ ограничит диапазон возможных значений переменной. У цикла будет два выходящих ребра. Для одного из них анализ установит, что $(i >= 10000)$, а для другого – что $(i == 500)$. Используя информацию о шаге изменения индуктивной переменной, можно установить, что переменная i после выполнения цикла имеет значение 500 либо 10000. Таким образом, чтобы проанализировать циклы с большим количеством итераций, может быть достаточно трёх итераций.

Но не все циклы возможно проанализировать таким образом. Для следующего цикла

```
int* a = &array;
for(i=0;i<10000;i++) {
        a = b;
        b = c;
        c = d;
        d = 0;
}
```

анализ не сможет установить ни то, что указатель a на выходе из цикла всегда имеет нулевое значение, ни даже то, что указатель a может иметь нулевое значение. Для получения такой информации требуется хотя бы 4 итерации. Как правило, при написании циклов программисты не создают слишком сложные зависимости по данным с предыдущими итерациями циклов (loop carried

dependencies), поэтому использование ограниченного обхода итераций цикла будет достаточным для анализа большинства написанного кода.

Определим функцию, возвращающую для графа потока управления и количества итераций множество рассматриваемых путей. Для этого пометим все обратные рёбра в графе потока управления меткой, означающую глубину вложенности цикла. Обратные рёбра в одном цикле будут иметь одинаковые метки. Пусть L – множество меток, E – множество рёбер, функция $label : E \to L$ возвращает метку для обратного ребра и ноль для остальных рёбер. Тогда анализ рассматривает пути, для которых справедливо, что количество рёбер с непустыми метками l не больше, чем N^l, где N – количество обходов. Функция $numberl$ для пути возвращает количество рёбер, имеющих определённую метку. Тогда функция рассматриваемых путей будет определена как
$paths(C, N) = \{p \,|\, numberl(e, l) \leq N^l \;, e \in p, \, p \in C\}$.

Таким образом, множество рассматриваемых путей внутри функции определяется функцией обхода циклов.

3.5. Корректность анализа

3.5.1. Семантика абстрактных состояний

Частичные функции val_R, val_A и pt определяют абстрактное состояние анализа. Абстрактные состояния описывают форму памяти и равенство значений переменных и ячеек памяти.

Определим функцию конкретизации абстрактных состояний. Конкретизацией абстрактного состояния будет множество конкретных состояний, имеющее ту же форму памяти и равенство значений переменных и ячеек памяти, и удовлетворяющее дополнительным условиям. Введём предикат Corr для тестирования конкретных состояний на соответствие абстрактным состояниям, который опишем ниже. Предикат возвращает истину, если данное абстрактное состояние описывает данное конкретное состояние. Будем говорить, что абстрактное состояние s *корректно* описывает состояние программы ν, если $Corr(s, \nu)$.

Тогда функция конкретизации абстрактного состояния $\gamma : \Psi \to \wp(\Pi)$ будет определена следующим образом:

$$\gamma(s) = \{\nu | Corr(s, \nu)\}$$

.

Предикат тестирования конкретных состояний Corr возвращает истину, если существует функция конкретизации идентификаторов значений $conc : V \to C$ и выполнены следующие условия:

- Значение для каждой переменной в конкретном состоянии должно соответствовать тому, что функция conc вернёт для идентификатора значения этой же переменной. Т. е. $conc \circ val_R = \nu_R$. (*Свойство равенства для регистров*).

- Для каждого ссылочного составного идентификатора значения абстрактное состояние указывает множество ссылочных простых идентификаторов значений, которым он может быть равен. Для составного идентификатора значения функция конкретизации должна вернуть значение одного из идентификаторов, который составляет этот идентификатор значения. Т. е. для $v_p \in dom(pt) : conc(v_p) \in conc \circ pt(v_p)$ (*Свойство частей*).

- Ссылки соответствуют разным конкретным указателям. $conc(a_1) \neq conc(a_2)$, если $a_1 \neq a_2$, где $a_1, a_2 \in A$. (*Свойство непересечения ссылок*).

- Рассмотрим произвольную ячейку памяти $m \in M$. Функция конкретизации и АС накладывают ограничения на значения в ячейке m. В АС будут идентификаторы значений, конкретизация которых вернёт указатели, указывающие на ячейку m. Функция conc должна быть выбрана таким образом, что ячейке либо не соответствует ни одной ссылки, либо соответствует только одна. С помощью функции val_A определим идентификатор значения v, сопоставленный этой ссылке. Должно выполняться, что $\nu(m) = conc(v)$. Т. е. значение m должно быть конкретизацией одного из идентификаторов значений, указанных абстрактным состоянием. Пусть $conc(a) = m$, тогда $conc(val(a)) = \nu(m)$. (*Свойство равенства для ячеек*).

Т. е. требуется, чтобы если в абстрактном состоянии $val_R(r) = val_R(q)$ для некоторых регистров r и q, то и в конкретном состоянии $\nu_R(r) = \nu_R(q)$. Конкретизация составного идентификатора значений должна равняться конкретизации одной из его частей. Ссылки должны обозначать непересекающиеся области памяти. Если ссылки a_1 и a_2 обозначают ячейки m_1 и m_2 соответственно, то из $val_A(a_1) = val_A(a_2)$ должно следовать, что $\nu_R(m_1) = \nu_R(m_2)$.

Пример. Рассмотрим конкретные состояния $\nu_1 = \nu[a \mapsto 7, b \mapsto 7, c \mapsto 3]$, $\nu_2 = \nu[a \mapsto 99, b \mapsto 99, c \mapsto -4]$ и $\nu_3 = \nu[a \mapsto 0, b \mapsto 0, c \mapsto 0]$. Абстрактное состояние $s_1 =< val[a \mapsto v_1, b \mapsto v_1, c \mapsto v_2], \emptyset >$ корректно описывает конкретные состояния ν_1, ν_2 и ν_3.
Абстрактное состояние $s_2 =< val[a \mapsto v_1, b \mapsto v_1, c \mapsto v_1], \emptyset >$ корректно описывает только состояние ν_3.

3.5.2. Сохранение корректности

Пусть $f : \Psi \to \Psi$ – передаточная функция, $\pi : \Pi \to \Pi$ – семантическая функция. Тогда будем говорить, что передаточная функция f *сохраняет корректность* относительно семантической функции, если для любых $s \in \Psi$, $\nu \in \Pi$ верно $Corr(s, \nu) \Rightarrow Corr(f(s), \pi(\nu))$.

Для $s =< val, pt >$ обозначим $def(s) = dom(val) \cup dom(pt) \cup ran(val) \cup ran(pt)$, где dom – область определения функции, ran – область значений функции.

Лемма 1. *Пусть $a \in A$, $s =< val, pt >$ и $Corr(s, \nu)$. Пусть $val' = val[a \mapsto v_f]$, $s' =< val', pt >$, причём $v_f \notin def(s)$. Тогда $Corr(s', \nu)$.*

Доказательство. Т. к. $a \in A$, то $val'(r) = val(r)$ для всех $r \in R$, поэтому свойство равенства для регистров не будет нарушено.

Т. к. pt не меняется, то свойство частей также будет выполнено.

Абстрактное состояние s корректно описывает конкретное состояние ν, следовательно, существует функция конкретизации идентификаторов значений $conc$, удовлетворяющая требованиям корректности. Абстрактное состояние s' будет описывать конкретное состояние ν, если будет существовать функция конкретизации $conc'$, удовлетворящая требования корректности. Выберем $conc' =$

$conc[v_f \mapsto \nu(conc(a))]$. Обозначим $m = conc(a)$. В этом случае $conc'(v_f) = \nu(m)$, и, следовательно, $conc'(val(a)) = \nu(m)$, т. е. выполнено свойство равенства для ячеек памяти.

Т. к. v_f не находится в множестве $def(s)$, то условие непересечения ссылок также не будет нарушено. □

Лемма 2. *Пусть $a \in A$, $s =< val, pt >$ и $Corr(s, \nu)$. Пусть $val' = val[a \mapsto getval(a, val)]$, $s' =< val', pt >$. Тогда $Corr(s', \nu)$.*

Доказательство. Абстрактное состояние s корректно описывает конкретное состояние ν, следовательно, существует функция конкретизации идентификаторов значений $conc$, удовлетворяющая требованиям корректности. Докажем, что $conc$ также является функцией конкретизации значений для s'.

Если $a \in dom(val)$, то $getval(a, val) = val(a)$ и, следовательно, $val[a \mapsto getval(a, val)] = [a \mapsto val(a)] = val$, т. е. $s' = s$.

Если $a \notin dom(val)$, то функция $getval$ вернёт новый идентификатор значения, который не будет в $def(s)$. Тогда $Corr(s', \nu)$ выполняется по лемме 1. □

Лемма 3. *Пусть $v \in V$, $s =< val, pt >$ и $Corr(s, \nu)$. Пусть $pt' = pt[v \mapsto getpt(v, pt)]$, $s' =< val, pt' >$. Тогда $Corr(s', \nu)$.*

Доказательство. Пусть $conc$ – функция конкретизации для s. Покажем, что она также является функцией конкретизации для s'. Если $v \in dom(pt)$, то $pt[v \mapsto getpt(v, pt)] = pt[v \mapsto pt(v)] = pt$, т. е. $s' = s$.

Если $v \notin dom(pt)$, то $pt(v) = \{v\}$. Свойства равенства и непересечения ссылок при этом не могут быть нарушены. Если $w \neq v$ и $w \in dom(pt)$, то $conc(w) \in conc \circ pt(w)$ (свойство частей), т. к. $Corr(s, \nu)$. Для v свойство частей выполняется тривиально, т. к. $pt(v) = \{v\}$. □

3.5.3. Состояние на входе в функцию

Анализ производится для конкретных состояний, где множество всех достижимых (начиная от параметров процедуры) значений не является алиасами.

Определим достижимость значений и опишем абстрактное состояние на входе в функцию. Шагом доступа будет смещение либо разыменование (обозначим как ref). Путь доступа будет последовательностью шагов доступа для

некоторой переменной. Пример: $p, ref, buf, 7, ref, val$ – путь доступа, начинающийся с переменной p.

Функция $StepA$ для переменной и пути доступа создаёт уникальную ссылку. Функция $StepM$ для переменной и пути доступа создаёт ячейку памяти.

Выделим подмножество *свободных ссылок* FA из множества ссылок A. Свободная ссылка полностью определяется путём доступа, начинающегося с параметра процедуры.

Абстрактное состояние описывает множество конкретных состояний, где ячейки памяти, соответствующие свободным ссылкам, не пересекаются.

Если ссылка a является свободной, то и ссылка $StepA(a, s)$ будет свободной для некоторого пути доступа:

$$a \in A \Rightarrow Step(a, s) \in A$$

Доопределим функцию $getval$ для работы с ссылками.

$$getval(a, val) = \begin{cases} v = val(a), & \text{если } a \in dom(val); \\ v = fresh(), & \text{где } v \in FA, \text{ если } a \in FA; \\ v = fresh(), & \text{где } v \in (A \setminus FA), \text{ если } a \notin FA; \end{cases}$$

Функция $getpt$ возвращает множество указываемых ссылок, используя функцию pt, либо возвращает множество из единственного элемента – ссылки для аргумента v.

На входе в функцию выберем абстрактное состояние такое, что для любого пути доступа s справедливо:

$$conc(StepA(p, s)) = StepM(p, s), \text{где } p \text{ – параметр процедуры.}$$

Лемма 4. *Пусть* $s \in \Psi$, $\nu \in \Pi$. *И существует функция конкретизации* $conc$, *что* $conc(StepA(p, s)) = StepM(p, s)$, *где* p *– параметр процедуры.. Причём* $def(s) \subseteq FA$. *Тогда* $Corr(s, \nu)$.

Доказательство. Т. к. все ссылки, используемые в s, являются свободными

ссылками, то для них выполняется свойство непересечения. Свойство равенства выполняется, т. к. для любого пути доступа, начинающегося с параметра процедуры, определён уникальный идентификатор значения с помощью функции $StepA$. □

3.5.4. Корректность слияния путей

Рассмотрим точку слияния путей. Пусть $s_1, s_2 \in \Psi$ – абстрактные состояния на входных рёбрах в точку, ν_1, ν_2 – конкретные состояния на этих рёбрах. При этом выполняется, что $Corr(s_1, \nu_1)$ и $Corr(s_2, \nu_2)$. Необходимо выбрать s_3 такое, что будет выполняться $Corr(s_3, \nu_1)$ и $Corr(s_3, \nu_2)$. Это абстрактное состояние будет использоваться для выходного ребра точки слияния.

Пусть состояние s_1 использует больше ссылок, чем состояние s_2. Создадим расширенное выходное абстрактное состояние s', содержащее столько же ссылок, сколько есть в s_1. Пусть произвольным образом определены инъективные функции $rename_1 : A \to A$ и $rename_2 : A \to A$, которые для каждой ссылки в s_1 и s_2 соответственно возвращают ссылку из s'.

Построим функции val и pt, используя только ссылки из s'. Пусть $f : V \times V \to V'$ – функция, определяющая для каждой пары идентификаторов значений уникальный идентификатор значений из множества V', где $V' \subset V$ и $V' \cap (def(s_1) \cup def(s_2)) = \emptyset$.

Для $r \in R$ определим $val(r) = f(val_1(r), val_2(r))$.

Для $a \in s'$ обозначим $a_1 = rename_1^{-1}(a)$, $a_2 = rename_2^{-1}(a)$, $v_1 = val_1(a_1)$ и $v_2 = val_2(a_2)$.

Для $a \in s'$ определим $val_A(a) = f(rename_1(v_1), rename_2(v_2))$.

Для $v \in ran(rename_1) \cup ran(rename_2)$ определим
$pt(v) = rename_1(pt_1(a_1)) \cup rename_2(pt_2(a_2))$.

Теорема 1. *Пусть $s =< val_R, val_A, pt >$ – абстрактное состояние, построенное с помощью процедуры, определённой выше. Если $Corr(s_1, \nu_1)$ и $Corr(s_2, \nu_2)$, то $Corr(s_3, \nu_1)$ и $Corr(s_3, \nu_2)$.*

Доказательство. Докажем $Corr(s_3, \nu_1)$. Абстрактное состояние s_3 будет корректно описывать корректное состояние ν_1, если существует функция конкре-

тизации идентификаторов значений $conc_3$, удовлетворяющая условиям корректности. Т. к. выполняется $Corr(s_1, \nu_1)$ по условиям теоремы, то существует функция конкретизации идентификаторов значений $conc_1$, демонстрирующая это.

Пусть $v_1 = val_1(r)$ и $conc_1(v_1) = c_1$, $v_1' = rename_1(v_1)$, где $r \in R$. Функция val_R определена так, что $val_R(r) = f(v_1', w) = v_3$, где w – некоторый идентификатор значений. Определим функцию конкретизации $conc_3$ следующим образом: $conc_3(v_3) = conc_1(v_1)$. Тогда $conc_3(val_R(r)) = conc_3(v_3) = conc_1(v_1) = \nu(r)$. Т. е. свойство равенства для регистров выполняется. Поэтому $Corr(s_3, \nu_1)$.

Для доказательства свойства равенства для ссылок возьмём $conc_3(a_3) = conc_1(a_1)$, где $a_3 = rename_1(a_1)$.

Свойство непересечения ссылок выполняется, т. к. ссылки в s_1 не пересекаются по условиям теоремы, и функция переименования $rename_1$ инъективная.

Доказательство $Corr(s_3, \nu_2)$ производится аналогично. □

Теорема 1 утверждает, что описанная выше процедура объединения абстрактных состояний будет корректной, т. е. результирующее абстрактное состояние будет описывать конкретные состояния на всех входных рёбрах в точку слияния. При этом можно использовать произвольное сопоставление ссылок в абстрактных состояниях. В теореме не утверждается, что результирующее АС будет наиболее точным. В разделе 3.7 будет описана реализация слияния, основанная на процедуре объединения и выборе функций переименования.

3.5.5. Доказательство корректности анализа

Лемма 5. *Пусть instr – инструкция языка svace0, состояние s_0 – абстрактное состояние перед выполнением инструкции, состояние s_1 – абстрактное состояние после выполнения инструкции, $F(instr)$ – передаточная функция, ν_0 – состояние анализа на входном ребре инструкции, ν_1 – состояние анализа на выходном ребре инструкции. Тогда $Corr(s_0, \nu_0) \Rightarrow Corr(s_1, \nu_1)$.*

Доказательство. Для доказательства рассмотрим все передаточные функции для инструкций программы. Обозначим $val_1 = s_1.val$, $val_0 = s_0.val$. Обозначим $pt_1 = s_1.pt$, $pt_0 = s_0.pt$. Пусть $conc_0$ – функция конкретизации идентификаторов значений перед выполнением инструкции. Корректность не будет нарушена,

если удастся подобрать функцию конкретизации $conc_1$ для выходного ребра инструкции, удовлетворяющую условиям корректности.

Инструкция $r = alloca()$. $\nu_1 = \nu_0[m \mapsto undef, r \mapsto m]$, где m – новая ячейка памяти. $val_1 = val_0[r \mapsto v_r]$, где $v_r = fresh()$. Выберем функцию конкретизации идентификаторов значений $conc_1 = conc_0[v_r \mapsto m]$. Т. к. r – регистр, а v_r – простой идентификатор значения, то достаточно доказать свойство равенство для регистров и свойство непересечения ссылок. Идентификатор v_r – новый создаваемый идентификатор значения, который не находится в множестве значений val, ячейка m – также новая ячейка памяти, которая не находится в s_0. Т. е. свойство равенства будет выполняться. Свойство непересечения ссылок выполняется, т. к. $m \notin dom(\nu_0)$.

Инструкция $r = t$. Если t – константа, то $\nu_1 = \nu_0[r \mapsto t]$. Для констант анализ создаёт новые идентификаторы значения, поэтому можно выбрать функцию конкретизации такую, что её значение для созданного идентификатора значений будет равно t. Если t – регистр, то $\nu_1 = \nu_0[r \mapsto \nu_0(t)]$. При этом $val_1 = val_0[r \mapsto v_t, t \mapsto v_t]$, где $v_t = getval(t, val)$. Выберем $conc_1 = conc_0(v_t \mapsto \nu_0(t))$. Т. е. $val_1(r) = val_1(t)$ и $\nu_1(t) = \nu_1(r)$, то корректность не нарушается.

Инструкция $r = t_1 + t_2$. $\nu_1 = value(t_1, \nu_0) + value(t_2, \nu_0)$. Функция value возвращает константу, если аргумент является константой, и $\nu_0(t)$ в остальных случаях. $val_1 = val_0[r \mapsto add(v_1, v_2), t_1 \mapsto v_1, t_2 \mapsto v_2]$, где $v_1 = getval(t_1, val_0)$, $v_2 = getval(t_2, val_0)$.

Функция add возвращает значение на основе хэш-функции для своих аргументов. $add(a, b) = add(c, d)$, только если $a = b$ и $c = d$. Поэтому $v_r = add(v_1, v_2)$ совпадёт с уже определённым идентификатором значения, только если v_1 и v_2 будут равны некоторым x и y. Но по условиям теоремы, если $val_0(t_1) = val_0(x)$, то и $\nu_0(t_1) = \nu_0(x)$ для некоторого x. Аналогично для v_2 и y. Т. к. $t_1 + t_2 = x + y$, если $t_1 = x$ и $t_2 = y$, то корректность будет сохранена.

Доказательство для инструкций $r = t_1 - t_2$, $r = t_1 * t_2$ и $r = t_1 \% t_2$ аналогично.

Инструкция $r = shift\ p, f$. Инструкция создаёт идентификаторы значения для p и r. Для создания идентификатора значения для переменной p используется функция getval, используя лемму 2, можно доказать, что корректность

не будет нарушена. При создании идентификатора значения для r используется функция кэширования $field$. Доказательство корректности в этом случае аналогично доказательству корректности для сложения.

Инструкция $r = load\, p$. Передаточная функция инструкции использует функции $getval$, $getpt$ и $getjoinval$. Сохранение корректности для функций, использующих функции $s_1 =< val_0[a \mapsto getval(a, val)], pt_0 >$, была доказана в лемме 2. Сохранение корректности для функций $s_1 =< val_0, pt_0[v \mapsto getpt(a, val)] >$ была доказана в лемме 3. Функция $getjoinval$ используется для выбора идентификатора значения v_r для результата инструкции r. Одним из аргументов функции является множество идентификаторов значений $V_p = val(A_p)$, где множество ссылок $A_p = getpt(v_p, pt)$. В простом случае множество A_p содержит только одну ссылку a. Причём справедливо $val(p) = a$. Созданный идентификатор значения ассоциируется с регистром r. Таким образом, может быть нарушено только свойство равенства для регистров. Передаточная функция создаст новую функцию val, где $val(r) = val(p)$. В конкретном состоянии $\nu_1(r) = \nu_1(p)$. Поэтому свойство равенства не будет нарушено для этого случая. Для второго случая используется функция $joinvals$. Эта функция возвращает новый идентификатор значения, если её аргументы разные, и первый аргумент, если они одинаковые. Т. к. вновь созданный идентификатор значения будет составным, то свойство непересечения не будет нарушаться. Для обеспечения свойства частей можно выбрать $conc_3(v_r) = conc_3(v_1)$, где $v_r \in V_p$, v_1 – составляет v_r.

Инструкция $store\, t, p$. Для конкретного состояния после выполнения инструкции справедливо, что $\nu_1(t) = \nu_1(m)$, где $m = \nu_1(p)$. Передаточная функция инструкции использует функции $getval$, $getpt$ и $update$. Для функций $getval$, $getpt$ доказательство аналогично инструкции $load$. Функция $update$ создаёт функцию val_1, где обновлены значения ссылок A, на которые указывает идентификатор значения v_p для регистра p. Если множество ссылок A_p состоит из одного элемента a, то в результирующем абстрактном состоянии $val_1(v_t) = val_1(a)$, $a = pt_1(v_p)$. Для этого абстрактного состояния корректность будет сохранена. Если множество ссылок A_p содержит более одного элемента, то создаётся новый (составной) идентификатор значения, накладывающий более слабые ограниче-

ния на вид конкретного состояния. Поэтому свойства равенства и непересечения будут выполнены. Для доказательства свойства частей можно подобрать функцию конкретизации $conc_3(a) = conc_3(a_1)$, где $a_r \in A_p$, a_1 – составляет a_r. □

Лемма 5 постулирует, что передаточные функции сохраняют корректность абстрактного состояния. Используя теорему 1, можно доказать корректность всего анализа. Будем использовать функцию рассматриваемых путей из раздела 3.4, возвращающую анализируемые пути для графа потока управления и заданного количества итераций.

Теорема 2. *Для заданного количества итераций циклов N и графа потока управления C рассмотрим множество путей внутри функции $pathset = paths(C, N)$. Рассмотрим множество конкретных состояний перед началом функции, принадлежащее множеству разрешённых состояний. Пусть после анализа инструкции на некоторой итерации анализ сопоставил выходному ребру инструкции абстрактное состояние s и множество анализируемых путей $p \subseteq pathset$. Тогда для любого пути $l \in p$, имеющего разрешённые конкретные состояния в точке входа в процедуру, его конкретное состояние ν в текущей точке должно принадлежать конкретизации абстрактного состояния: $\nu \in \gamma(s)$.*

Доказательство. Докажем корректность по индукции по длине выполняемого пути. Корректность выполняется на входе в функцию согласно лемме 4. Сохранение корректности анализа для инструкций доказано в лемме 5. Сохранение корректности на путях, проходящих через точки слияния в графе потока управления, доказано в теореме 1. □

Теорема 2 постулирует корректность анализа. При этом абстрактное состояние корректно описывает конкретное состояние на каждом шаге анализа. Это свойство применяется детекторами (глава 4), использующими абстрактное состояние по ходу анализа, не дожидаясь его завершения.

3.6. Сильные и слабые обновления

3.6.1. Кэш значений

Обновление ячейки памяти называется *сильным*, если операция присваивания значения ячейки памяти моделируется удалением всех предыдущих присваиваний. Если предыдущие присваивания не удаляются, то обновление называется *слабым*.

В описываемом анализе память моделируется ссылками. Единственная инструкция записи в память в языке – это *store*. Рассмотрим инструкцию записи в память с последующим чтением:

```
store p, x;
y = load p;
```

Обновление будем называть *сильным*, если в состоянии анализа после инструкции *load* выполняется $val(y) = val(x)$, и *слабым* во всех остальных случаях.

Рассмотрим инструкцию $r = load\, p$. Пусть $v_p = val(p)$, множество $A_p = pt(v_p)$. Если множество A_p содержит только одну ссылку, то можно использовать значение, ассоциированное с этой ссылкой. Если же множество A_p содержит больше одной ссылки, то с помощью функции *joinval* будет создан новый идентификатор значения, обозначающий свойства всех идентификаторов, ассоциированных со ссылками в A_p.

Рассмотрим инструкцию $store\, p, v$. Пусть $v_p = val(p)$, множество $A_p = pt(v_p)$. Если множество A_p содержит только одну ссылку a_p, то можно сделать *сильное обновление* и заменить предыдущий идентификатор значения, ассоциированный с a_p. Передаточные функции:

$$val[p \mapsto v_p]\|store\, p, v\| \ \mapsto val[p \mapsto v_p, a_p \mapsto val(v)]$$
$$pt[v_p \mapsto \{a_p\}]\|store\, p, v\| \ \mapsto pt[v_p \mapsto \{a_p\}]$$

Если же множество A_p содержит больше одной ссылки, то необходимо сделать *слабое обновление*, т. к. неизвестно, какая именно ячейка могла измениться. Для этого с каждой ссылкой a ассоциируется идентификатор значения $joinval(val(v), val(a))$, обозначающий свойства нового значения и предыдущего значения.

Для более точного анализа используется нумерация значений для разыменования указателя на основе использования функции $cache : \wp(V) \rightarrow V$. Для указателя p анализ формирует пару $< v_p, V_p >$, где $v_p = val(p)$ – идентификатор значения указателя p, множество $V_p = val(pt(v_p))$ – упорядоченное множество идентификаторов значений для ссылок A_p, на которые указывает p, и сопоставляет ей идентификатор значения $v_n = val(v)$ для $*p$. После анализа инструкции $store$ вычисляется $cache_{out} = cache_{in}[< v_p, V_p > \mapsto v_n]$. При следующем доступе к памяти через указатель вновь вычисляется пара $< v'_p, V'_p >$, где $v'_p = val'(p)$, $V'_p = val(pt(v'_p))$. Если функция $cache$ определена для этой пары, значит, не было изменений в указываемых ячейках памяти, и можно использовать предыдущее значение $v_n = cache(< v'_p, V' >)$.

В разделе 3.8 будет показано использование функции $cache$ на примере.

3.6.2. Расширение абстрактных состояний

Расширим абстрактное состояние, добавив туда функцию $cache$. Теперь абстрактным состоянием анализа будет тройка $< val, pt, cache >$.

Необходимо доопределить передаточные фунции. Кэш значений влияет только на передаточные функции для инструкций $load$ и $store$. Поэтому для передаточных функции остальных инструкций можно использовать идентичную функцию: $cache\|instr\| \mapsto cache$.

Будем использовать вспомогательную функцию $getcacheval$, которая является расширением функции $getjoinval$ и использует кэш значений.

Функция $getcacheval$ возвращает идентификатор значения для памяти, на которую указывает переменная, обозначаемая v. V_p – обозначает список идентификаторов значений для указываемых ячеек. Если V_p состоит из одного идентификатора значений, то возвращается этот идентификатор значений.

$$getcacheval(v, V_p, cache) = \begin{cases} w, & \text{если } V_p = \{w\}; \\ cache(< v, V_p >), & \text{если } < v, V_p > \in dom(cache); \\ joinvals(V_p), & \text{иначе} \end{cases}$$

Передаточная функция для инструкции загрузки из памяти *load*:

$$\begin{array}{rl} val\|r = load\, p\| & \mapsto val[p \mapsto v_p, r \rightarrow v_r] \\ pt\|r = load\, p\| & \mapsto pt[v_p \mapsto A_p] \\ cache\|r = load\, p\| & \mapsto cache[< v_p, V_p > \mapsto v_r] \end{array},$$

где $v_p = getval(p, val), A_p = getpt(v_p, pt), v_r = getcacheval(v_p, V_p, cache)$, V_p – значения A_p.

Передаточная функция для инструкции записи в память *store*:

$$\begin{array}{rl} val\|store\, t,\, p\| & \mapsto val'[t \mapsto v_t, p \mapsto v_p] \\ pt\|store\, t,\, p\| & \mapsto pt[v_p \mapsto A_p] \\ cache\|store\, t,\, p\| & \mapsto cache[< v_p, V_p > \mapsto v_t] \end{array},$$

где $v_t = getval(t, val), v_p = getval(p, val), A_p = getpt(v_p, pt)$,
$val' = update(A_p, v_t, val)$, V_p – новые значения A_p.

3.7. Слияние состояний

В разделе 3.5.4 приведено описание процедуры выбора абстрактного состояния, обозначающего свойства двух других абстрактных состояний, и приведено доказательство корректности выбора. При этом использовались произвольные функции переименования ссылок в абстрактных состояниях. Выбор разных функций переименования приведёт к построению разных выходных абстрактных состояний.

При анализе инструкций создаётся выходное состояние на основе входного состояния. При этом переиспользуются ссылки, созданные до разделения путей на графе потока управления (определяются по значениям *val* для регистров). Поэтому в точках слияния часть ссылок в абстрактных состояниях на входных рёбрах будет одинаковая. Для таких ссылок функции переменования $rename_1$ и $rename_2$ будут выбраны согласованно, т. е. $rename_1(a) = rename_2(a) = a$.

Среди оставшихся ссылок рассмотрим только достижимые через функции *val* и *pt*, начиная от общих ссылок. Если $val_1(a) = a_1$ и $val_2(a) = a_2$, где a – общая ссылка, то и функции $rename_1$ и $rename_2$ выберем таким образом, чтобы $rename_1(a_1) = rename_2(a_2)$. Аналогично, если $q \in pt_1(v)$ и $q \in pt_2(v)$.

Рассмотрим слияние состояний на примере рис. 3.1. После анализа инструкции на строке 1 состояние будет следующим: $val[p \mapsto v_p]$, где v_p ссылка

для обозначаения созданной ячейки памяти.

Для первой ветки условия на строке 5 абстрактное состояние будет иметь вид: $val[p \mapsto v_p, t1 \mapsto v_{t1}, v_p \mapsto v_{t1}, v_{t1} \mapsto v_{99}]$

Для второй ветки условия на строке 9 абстрактное состояние будет иметь вид: $val[p \mapsto v_p, t2 \mapsto v_{t2}, v_p \mapsto v_{t2}, v_{t2} \mapsto v_{77}]$.

```
1:   p = alloca ref int();
2:   if(f > 0,
3:       t1 = alloca();
4:       store 99, t1
5:       store t1, p
6:     ,
7:       t2 = alloca();
8:       store t2, p;
9:       store 77, t2
10:  )
```

Рис. 3.1. Фрагмент кода с ветвлением

Абстрактные состояния имеют общую ссылку v_p. В первом состоянии есть ссылка v_{t1}, во втором состоянии – ссылка v_{t2}. Т. к. $val_1(v_p) = v_{t1}$ и $val_2(v_p) = v_{t2}$, то выберем функции переименования такие, что $rename_1(v_{t1}) = rename_1(v_{t2}) = v_{t1}$. В результате выходное состояние будет:

$val[p \mapsto v_p, v_p \mapsto v_{t1}, v_{t1} \mapsto v_j]$, где $v_j = joinval(v_{99}, v_{77})$.

3.8. Пример анализа

Рассмотрим работу алгоритма для Си-функции на рис. 2.2. На рис. 2.4 приведён код svace0. Если значение f истинно, то функция вернёт $3 + 1 + 3 = 7$, иначе $2 + 1 + 3 = 6$. Процесс анализа показан на рис. 3.2. Состояние анализа показано в виде комментариев. Обозначение $< val_n, pt_n, cache_n >$ используется для состояния анализа для n-й строки.

Использовались следующие обозначения:

$v_a = fresh()$, $v_b = fresh()$, $v_p = fresh()$,

$a_a = deref(v_a)$, $a_b = deref(v_b)$, $a_p = deref(v_p)$,

```
1:  def calc(f) = //< val_1, pt_1, cache_1 >=< [], [], [] >
2:      a = alloca();
//< [a ↦ v_a], [v_a ↦ {a_a}], [] >
3:      b = alloca();
//< val_2[b ↦ v_b], pt_2[v_b ↦ {a_b}], [] >
4:      p = alloca ref int();
//< val_3[p ↦ v_p], pt_3[v_p ↦ {a_p}], [] >
5:      store 1, a;
//< val_4[a_a ↦ v_1], pt_3, [] >
6:      store 2, b;
//< val_5[a_b ↦ v_2], pt_3, [] >
7:      if(f!=0,
8:          store a, p,
            //< val_6[a_p ↦ v_a], pt_3, [] >
9:          store b, p
            //< val_6[a_p ↦ v_b], pt_3, [] >
10:     );
//< val_6[a_p ↦ v_ab], pt_3[v_ab ↦ {a_a, a_b}], [< v_ap; v_a, v_b >↦ v_12] >
11:     t1 = load p;
//< val_10[t1 ↦ v_ap], pt_10, cache_10 >
12:     x = load t1;
//< val_11[x ↦ v_12], pt_10, cache_10 >
13:     store 3, t1;
//< val_6[a_a ↦ v_13, a_b ↦ v_23], pt_10, [< v_ap; v_a, v_b >↦ v_3] >
14:     t2 = load p;
//< val_12[t2 ↦ v_ap], pt_10, cache_13 >
15:     y = load t2;
//< val_13[y ↦ v_3], pt_10, cache_13 >
16:     t3 = load a;
//< val_14[t3 ↦ v_13], pt_10, cache_13 >
17:     t4 = t3 + x;
//< val_15[t4 ↦ v_4], pt_10, cache_13 >
18:     t5 = t4 + y;
//< val_16[t5 ↦ v_5], pt_10, cache_13 >
19:     ret t5; }
```

Рис. 3.2. Анализ функции `calc`.

$v_1 = const(1), v_2 = const(2), v_3 = const(3),$

$v_{ab} = joinval(v_a, v_b),$

$v_{12} = joinval(v_1, v_2),\ v_{13} = joinval(v_1, v_3),\ v_{23} = joinval(v_2, v_3),$

$v_4 = v_{13} + v_{12}$, $v_5 = v_4 + v_3$.

Если раскрыть значение возвращаемого регистра t5, то получим: $val(t5) =$
$v_5 = v_4 + v_3$
$= (v_{13} + v_{12}) + const(3)$
$= (joinval(v_1, v_3) + joinval(v_1, v_2)) + const(3)$
$= (joinval(const(1), const(3)) + joinval(const(1), const(2)) + const(3)$.

Для читаемости заменим joinval на ϕ. Получим $val(t5) = \phi(1,3)+\phi(1,2)+3$. Т. е. описывается множество возвращаемых значений {6, 7, 8}, которое включает в себя множество возвращаемых значений, возможных при выполнении: {6, 7}.

3.9. Массивы, объединения, приведения типов

3.9.1. Массивы

Доступ к массивам может производиться как по константному индексу, так и с помощью переменной. Анализ массивов организован следующим образом. Для всех ячеек памяти массива, явно упомянутых в программе, создаются ссылки. При этом создаётся ссылка с псевдоиндексом *, обозначающая свойство всех ячеек массива. При записи в ячейку $p[i]$ производится сильное обновление ссылки $p[i]$ и слабое обновление ссылки $p[*]$. При чтении используется техника, описанная выше, на основе функции *cache*. Если с помощью *cache* не удалось получить точное значение ячейки памяти, то используется значение на основе значений $p[i]$ и $p[*]$: $joinval(p[i], p[*])$.

Благодаря такой реализации можно учесть потенциальные алиасы между массивами. Для следующего кода:

```
void foo(int*p, int i, int j) {
1:     p[2] = 7;
2:     p[i] = 10;
3:     int x = p[2];
4:     p[j] = 20;
5:     int y = p[i]; }
```

будет установлено, что переменная x имеет значение 7 или 10, а переменная y – значения 7, 10 или 20. Процесс анализа фрагмента кода показан в табл. 3.1. Значение ссылки $p[*]$ изменяется каждый раз при записи в память с использованием указателя p (строки 1, 2, 4). Значения остальных ссылок изменяются только при записи в память с использованием определённого индекса. В таблице значения ссылок указаны с учётом влияния функции *cache*, т. е. те значения, которые будут получены с помощью использования функции *getcacheval*. Вычисляются только идентификаторы значений, которые используются: для ссылки $p[2]$ сопоставлен идентификатор значения v_1, это значение будет обновлено на новое значение $joinval(v_1, val(p[*]) = joinval(v_1, v_2)$, если встретится инструкция загрузки ячейки $p[2]$.

Таблица 3.1. Анализ массивов. Идентификаторы значений c_7, c_{10} и c_{20} служат для обозначения соответствующих констант. $v1 = joinval(c_7, c_{10})$, $v2 = joinval(v_1, c_{20})$, $v3 = joinval(v_2, c_{10})$.

Код	Ссылки			
	p[2]	p[i]	p[j]	p[*]
1: p[2] = 7;	c_7	-	-	c_7
2: p[i] = 10;	c_7	c_{10}	-	v_1
3: int x = p[2];	v_1	c_{10}	-	v_1
4: p[j] = 20	v_1	c_{10}	c_{20}	v_2
5: int y = p[i]	v_1	v_3	c_{20}	v_2

3.9.2. Приведение типов

Язык Си поддерживает приведение типов. Рассмотрим приведение типов с последующей записью через указатель

```
T1* a1;
T2* a2 = (T2*)a1;
a2->f = u;
```

Для адреса $a2$ используется тот же самый идентификатор значения v, что и для адреса $a1$. Т. к. типы разные, то будет создано множество ссылок вида $deref(v).f_1^1, deref(v).f_2^1..$ и $deref(v).f_1^2, deref(v).f_2^2..$, где $f_1^1, f_2^1..$ – поля структуры $T1$, а $f_1^2, f_2^{12}..$ – поля структуры $T2$. Для всех полей известны смещения.

При записи через указатель $a2$ будет сделано обновление для всех ссылок для полей $T1$, для которых смещения могут пересекаться со смещеним f.

3.9.3. Объединения

Язык Си поддерживает структуру данных "объединение". Обработка объединений языка Си аналогична обработке инструкций приведения типа. Рис. 3.3 содержит два фрагмента кода, эквивалентных друг другу, один из них использует объединения, а другой – преобразования типов.

```
long val = 733;
char*buf = (char*)&val;
char ch2 = buf[3];
```

```
typedef union {
  char buf[8];
  long val;
} U;

U u;
u.val = 733;
char ch = u.buf[3];
```

Рис. 3.3. Объединения и преобразования типов

Глава 4

Детекторы

Настоящая глава посвящена детекторам для поиска ошибок.

4.1. Атрибуты

4.1.1. Разделение анализа на ядро и расширения

Для большей гибкости анализ разделён на ядро и расширения. Ядро выполняет анализ указателей, нумерацию значений, создание идентификаторов значений и ссылок, и предоставляет возможность отдельным детекторам ассоциировать произвольные атрибуты с идентификаторами значений, ссылками, а также с точками графа потока управления. Конкретные детекторы реализуются в виде расширений. Все свойства анализа, необходимые детектору, представляются в виде *атрибутов*. Значения атрибута должны образовывать полурешётку[1]. Для каждой функции выполняется симуляция её выполнения, как описано в предыдущей главе. Ядро анализа оповещает расширения о всех событиях, расширения реагируют на события и могут создавать и читать атрибуты. Для детектора анализируемая программа состоит из событий, генерируемых ядром анализа, а реализация детекторов заключается в описании передаточных функций для событий. Выдача предупреждений происходит в момент анализа событий.

Все детекторы оповещаются одновременно во время обхода графа потока управления. Не производится отдельных проходов для некоторых детекторов. Такая организация анализа позволяет немного улучшить скорость анализа за счёт выделения общих действий в ядро анализа. Кроме этого, т. к. состояние анализа не требуется после анализа инструкции внутри базовых блоков (предупреждения выданы, выходное состояние создано, сравнивать состояния на разных итерациях не требуется), то его не обязательно хранить. Благодаря это-

[1] Значения атрибута являются частично упорядоченным множеством с оператором объединения, определяющим для каждой пары элементов наименьший верхний элемент.

му можно существенно уменьшить потребление памяти.

Атрибуты расширяют абстрактное состояние анализа. К тройке val, pt и $cache$ добавляются функции для каждого атрибута. Областью определния этих функций могут быть идентификаторы значений, ссылки и точки графа потока управления.

Для хранения атрибутов используется структура данных, отображающая тип атрибута в значение атрибута. Для большей эффективности каждый атрибут имеет значение по умолчанию, которое явно не хранится в структуре. Отсутствие записи об атрибуте означает, что атрибут имеет значение по умолчанию.

Для реализации расширенного абстрактного состояния анализа используется структура данных, описанная Маликовым в [69]. Принцип компактного представления основан на том, что каждая инструкция лишь незначительно меняет состояние анализа. Для более эффективного хранения данных состояние на выходном ребре инструкции содержит ссылку на состояние входного ребра и описывает разницу между двумя состояниями. В данном анализе ссылки на предыдущее состояние используются только для разных базовых блоков. В пределах одного базового блока используется только одно состояние, которое “продвигается” анализом от инструкции к инструкции.

4.1.2. Атрибуты и решётки

Пусть S – множество значений атрибутов, $\sqsubseteq_A$ – отношение упорядочивания (рефлексивное, антисимметричное и транзитивное). Частично упорядоченное множество $< S, \sqsubseteq_A >$ будет называться *полурешёткой*, если для всех элементов определён оператор объединения $\sqcup_A$, возвращающий для любых двух элементов множества наименьший верхний элемент. Множество будем называть *решёткой*, если дополнительно определён оператор сбора $\sqcap_A$, возвращающий для любых двух элементов множества наибольший нижний элемент [70].

Для верхнего элемента решётки $\top$, если он существует, справедливо $s \sqsubseteq_A \top$, где $s \in A$. Аналогично для нижнего элемента решётки $\bot$, если он существует, справедливо $\bot \sqsubseteq_A s$. Если в решётке не существует верхнего элемента, то можно его добавить и рассмотреть множество $S \cup \{\top\}$ с модифицированным

отношением упорядочивания. Аналогично для нижнего элемента.

В описываемым анализе требование, чтобы множество значений атрибутов образововывало полурешётку, не является необходимым, т. к. не вычисляется фиксированная точка, как в анализе потока данных. Но представление значений атрибутов в виде полурешётки позволяет рассуждать об атрибутах по аналогии с известными анализами в теории компиляторов. В точках слияния путей для вычисления атрибута на выходном ребре используется оператор объединения $\sqcup$ для атрибутов на входных рёбрах. Для оператора объединения известно, что его результат корректно приближает свойства, обозначаемые атрибутами на входных рёбрах, и что это будет самое точное приближение для рассматриваемой полурешётки. Свойства отношения упорядочивания определяются семантикой атрибута. Должно выполняться правило, что если значение a может быть корректно заменено на значение b в соответствии с семантикой атрибута, то $a \sqsubseteq_A b$.

Рассмотрим задачу определения того, что указатель был передан в функцию освобождения памяти. Будем использовать простое множество значений атрибута: $free$ – указатель был освобождён, $notfree$ – указатель не был освобождён. Рассмотрим два детектора, которым требуется атрибут с такими значениями: поиск утечек памяти и поиск двойного освобождения памяти. Целью анализа является нахождение дефекта без выдачи ложного срабатывания. Несмотря на то, что анализируемые значения совпадают для двух ситуаций, необходимо использовать разные атрибуты, т. к. отличается их семантика. Для задачи утечек памяти по умолчанию надо считать, что функция освобождает память, иначе будет выдано ложное срабатывание для ситуаций, когда функция может освободить память. Для поиска двойного освобождения памяти, наоборот, по умолчанию надо считать, что функция не освобождает память. Атрибут для утечек памяти будет использовать отношение упорядочивания $notfree_1 \sqsubseteq_1 free_1$, где $free_1$ будет обозначать, что память могла быть освобождена. Атрибут для поиска двойного освобождения памяти будет использовать отношение упорядочивания $free_2 \sqsubseteq_2 notfree_2$, где $free_2$ будет обозначать, что память точно была освобождена.

Для анализа функции на рис. 4.1 для точки (*) с помощью описанных вы-

ше значений невозможно точно установить, при каких условиях память будет освобождена. Поэтому необходимо корректно приблизить эти условия. Для первого атрибута такое приближение будет $free_1$, а для второго атрибута $notfree_2$.

```
struct Lst {
   struct Data*data;
   struct Lst*next;
};

void freeif(struct Lst*l) {
    if(!empty(l) && l->data==NULL)
        free(l);
}//(*)
```

Рис. 4.1. Пример функции, условно освобождающей память

4.2. Критерий выдачи

При написании функции программисты часто подразумевают некоторый контекст использования, в котором возможно вызывать функцию. Множество всех контекстов, где функция может быть вызвана без ошибок, будем называть *контрактом* функции. Функция на рис. 4.2 имеет три аргумента x, y и z, и использует два глобальных указателя p и q. Ничего не известно про взаимосвязь параметров, возможно, не все их комбинации разрешены контрактом. Поэтому предупреждения о разыменовании нулевых указателей на путях, проходящих через точки (1)-(3) и (2)-(4), могут оказаться ложными, если контракт функции не позволяет выполнение таких путей.

Предположим, что для каждого отдельного пути внутри функции есть критерий определения того, что путь содержит ошибку. Рассмотрим некоторую инструкцию функции такую, что все пути, проходящие через данную инструкцию, содержат ошибку. При этом ошибка может произойти в любом месте, а не обязательно в рассматриваемой инструкции. Неизвестно, включает ли контракт эти пути. Если ни один из этих путей не допустим контрактом функции, то тогда инструкция не может быть выполнена ни в одном допустимом контексте вызова. Инструкция будет бесполезной, и предполагается, что программисты не

```
void foo(int x, int y, int z) {
    if(x) {
        p = 0;// (1)
        if(y) q = 0; //(2)
    }
    if(z)*p = 1; //(3)
    *q = 0;// (4)
}
```

Рис. 4.2. Иллюстрация контракта функции.

пишут таких инструкций преднамеренно. Следовательно, можно считать эту ситуацию ошибкой в исходном коде. Если же контракт допускает один из таких путей, то ошибка может произойти в одном из потенциальных контекстов вызова функции или даже во время выполнения программы. В этом случае также нужно выдать предупреждение об ошибке.

Рассуждая аналогично, для инструкций ветвления должно быть возможным выполнение как по истинным путям, так и по ложным. Иначе инструкция будет лишней, а описанная ситуация – ошибочной. Переформулируем критерий, заменив инструкцию на ребро в графе потока управления: *предупреждение выдаётся, если все пути, проходящие через некоторое ребро в графе потока управления функции, содержат ошибку*. Недостижимое ребро будет соответствовать либо недостижимой инструкции, либо лишней инструкции ветвления.

Критерий применим для всех рёбер графа потока управления, которые были сгенерированы для инструкций, написанных программистом. Критерий нельзя применять к инструкциям, сгенерированным препроцессором Си для макросов. Макросы могут содержать проверки, которые при подстановке в тело конкретной функции будут лишними.

Функция foo на рис. 4.2 имеет два разыменования указателей в инструкциях (3) и (4). При этом все пути, проходящие через инструкцию (2), проходят через инструкцию (4) и содержат разыменование нулевого указателя. Поэтому согласно критерию следует выдать предупреждение.

Для путей, проходящих через некоторое ребро, не обязательно иметь ошибку в одном и том же месте, ошибка может быть в любом месте пути. На рис. 4.3

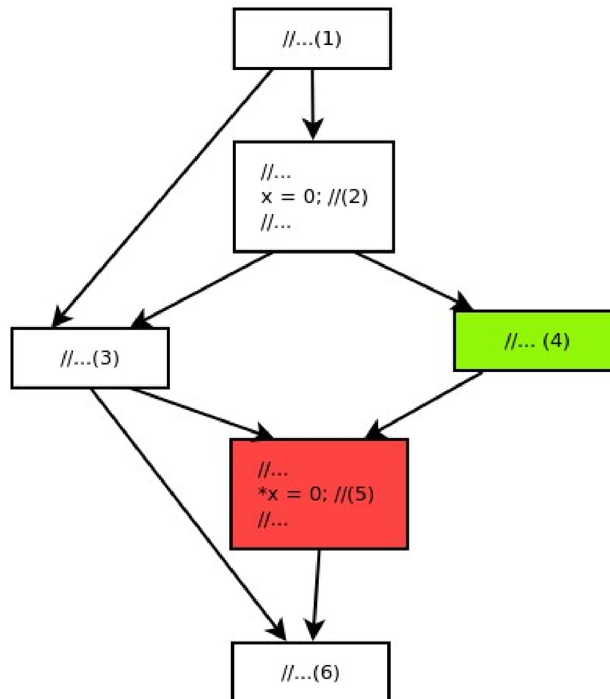

Рис. 4.3. Граф потока управления функции, для которой можно выдать предупреждения согласно предложенному критерию

показан граф потока управления, в котором путь, проходящий через инструкции (2) - (5), присваивает переменной p нулевое значение и разыменовывает его. В точке присваивания (2) ошибка не обязательно произойдёт, т. к. существует путь (3) - (6) в обход инструкции разыменования. В точке разыменования (5) указатель не обязательно имеет нулевое значение, возможно, инструкция была достигнута через точки (1) - (3). Но для точки (4) все пути содержат ошибку. Поэтому можно выдать предупреждение.

4.3. События

В разделе описываются события, генерируемые ядром анализа во время обхода графа потока управления. Рис. 4.4 содержит перечень генерируемых событий. Аргументами у событий являются идентификаторы значений и константы.

Событие $deref$ генерируется для инструкций, непосредственно разыменовывающих указатели: *load* и *store*. Если разыменование выполняется внутри

$$const(v, c) \quad deref(v) \quad assume(a = b) \quad assume(a \neq b) \quad assume(a \geq b)$$

$$assume(a \leq b) \quad assume(a > b) \quad assume(a < b) \quad shift(r = p, f) \quad r = \phi(a, b) \quad unknown(v)$$

Рис. 4.4. Виды событий. $v \in V; c \in C; a, b \in V \cup C$.

вызываемой функции, то событие не генерируется. Событие $deref(v_x)$ генерируется для инструкций $r = load\, x$ и $store\, x,\ y$, где $v_x = val(x)$.

Событие $assume$ генерируется для каждого ребра инструкций ветвления.

Событие $const(v, c)$ генерируется для уведомления детекторов, что значение идентификатора значения v равно константе c. Событие $shift$ создаётся для уведомления детекторов, что идентификатор значения r содержит смещение относительно идентификатора значения p.

Событие $unknown$ генерируется для значений переменных, адрес которых был передан в неизвестную функцию. Это событие позволяет детектору учесть то, что функция могла изменить значение и свойства переменной.

Реализация детекторов заключается в написании передаточных функций для событий. Будет использоваться следующая нотация для записи передаточной функции для события $event$ атрибута $Attr$:
$Attr_1 \| event \| = Attr_2$.

При обходе текущей инструкции всем детекторам доступно состояние анализа на входном ребре в инструкцию. Все детекторы могут читать это состояние, и никакие детекторы не могут изменять это состояние. Все изменения атрибутов, которые делают детекторы, сохраняются в специальном буфере. На основе этих изменений после оповещения всех детекторов будет изменено абстрактное состояние. Таким образом, последовательность оповещения детекторов не влияет на результат анализа, т. к. каждый детектор читает только неизменённое состояние.

4.4. Вспомогательные атрибуты

Атрибуты используются детекторами для представления свойств анализа. Часть свойств может разделяться между детекторами. Так как все детекторы запускаются одновременно, то они могут читать свойства друг друга. Анализ позволяет создавать расширения, которые не выполняют поиск дефектов, но анализируют некоторые свойства программы и сохраняют их в виде вспомогательных атрибутов.

Примером вспомогательных атрибутов является атрибут ValueInterval для отслеживания возможного интервала целочисленных значений. Для краткости будем писать VI. *Если в некоторой точке программы* $VI(v_x) = [a; b]$, *где* $v_x = val(x)$, *то во время выполнения программы значение переменной* x *будет в интервале* $[a; b]$.

Передаточные функции ValueInterval для некоторых инструкций:

$VI\|const(x,k)\| = VI[x \mapsto [k;k]]$

$VI\|assume(x = k)\| = VI[x \mapsto [k;k]]$

$VI\|assume(x \geq k)\| = VI[x \mapsto [k;+\infty] \cap VI(x)]$

$VI\|assume(x \leq k)\| = VI[x \mapsto [-\infty;k] \cap VI(x)]$

$VI\|unknown(x)\| = VI[x \mapsto [-\infty;+\infty]]$

4.5. Разыменование нулевого указателя

Опишем поиск дефектов на примере разыменования нулевого указателя. Рассмотрим только ситуации, где есть инструкция сравнения указателя с нулевым значением. Ситуацию, когда указателю было присвоено нулевое значение, рассматривать не будем.

Для анализа создадим двоичный атрибут $Null$, имеющий значения: $null$ – указатель был сравнен с нулём, и результат сравнения является истиной, any – указатель может иметь любое значение. Решётка атрибута $null \sqsubseteq any$. Передаточная функция для события $assume(p = 0)$ ассоциирует с идентификатором значения p атрибут $null$. Дальнейшим распространением атрибута занимается ядро анализа: для инструкций присваивания идентификатор значения копиру-

ется вместе с этим атрибутом. В точках слияния путей используется оператор слияния, т. е. если оба идентификатора значения имеют значение $null$, то и результат получит такое значение. Если для события разыменования с указателем ассоциирован идентификатор значения, которому сопоставлено значение $null$, то выдаётся предупреждение, т. к. если инструкция разыменования достижима, то произойдёт разыменование указателя, который может иметь только нулевое значение.

Передаточные функции детектора показаны на рис. 4.5. Передаточные функции этого детектора используют вспомогательный атрибут $ValueInterval$ для отслеживания нулевого значения константы в $assume$. Вспомогательная функция to_null, используемая в $assume(p = v)$, проверяет значение v на ноль с помощью атрибута $ValueInterval$. Атрибут $Null$ устанавливается в $null$, если v может иметь только нулевое значение. Назначение вспомогательной функции to_any – убрать значение $null$ для указателя, если сравниваемая переменная могла иметь нулевое значение. Такая реализация передаточных функций нацелена на уменьшение ложных срабатываний за счёт пропуска части дефектов. Если выполнение программы прошло через инструкцию разыменования программы и не произошла ошибка, то указатель имеет ненулевое значение. Поэтому передаточная функция для $deref$ устанавливает значение атрибута в any.

Атрибут $Null$ означает, что указатель имеет нулевое значение. В Си, как правило, разыменование указателя, полученного смещением нулевого значение на небольшую константу, также приведёт к ошибке. Описываемый детектор не осуществляет поиск таких ситуаций. Для его поиска следует изменить семантику атрибута так, чтобы он означал, что указатель точно имеет значение, которое некорректно разыменовывать. Можно использовать описанные передаточные функции и добавить функцию для события $shift$:
$Null\|shift(r = p, f)\| \mapsto Null[r \mapsto Null(p)]$, которая установит для результата смещения такое же значение, как и для оригинального указателя.

Несмотря на простоту, данный детектор позволяет находить истинные срабатывания на реальных проектах. На рис. 4.6 показан пример срабатывания, найденный с помощью описанного дефекта в проекте tizen/framework-uifw. Ме-

$$Null\|assume(p = 0)\| \mapsto Null[p \mapsto null]$$
$$Null\|assume(p! = 0)\| \mapsto Null[p \mapsto any]$$

$$Null\|assume(p = v)\| \mapsto Null[p \mapsto to_null(v)]$$
$$Null\|assume(p! = v)\| \mapsto Null[p \mapsto to_any(p, v)]$$

$$Null\|deref(p)\| \mapsto Null[p \mapsto any]$$

$$Null\|r = \phi(a, b)\| \mapsto Null[r \mapsto Null(a) \sqcup Null(b)]$$

$$Null\|unknown(p)\| \mapsto Null[p \mapsto any]$$

$$to_null(v) = \begin{cases} null & \text{если } ValueInterval(v) = [0; 0] \\ any & \text{иначе} \end{cases}$$

$$to_any(p, v) = \begin{cases} any & \text{если } 0 \in ValueInterval(v) \\ Null(p) & \text{иначе} \end{cases}$$

Рис. 4.5. Передаточные функции для детектора разыменования нулевых указателей.

сто разыменования находится всего через 2 строки от проверки на ноль. Не стоит недооценивать такое предупреждения. Если функция NextIndicatorName вернёт нулевой указатель, то пользователь вместо сообщения об ошибке получит аварийное завершение программы.

```
260| old = new
261| new = NextIndicatorName(info);
262| if (!new)
263| {
264|     WSGO1("Couldn't␣allocate␣name␣for␣%d\n", new->ndx);
265|         ACTION("Ignored\n");
266|         return False;
267| }
```

Рис. 4.6. Дефект разыменования нулевого указателя.

4.6. Обратный анализ

4.6.1. Назначение

Детектор, описанный в предыдущем разделе, находит инструкции, разыменовывающие переменные, которые на всех путях могут иметь только нулевые значения. На рис. 4.7 приведены графы потока управления для различных ситуаций присваивания указателю нуля и разыменования указателя. Описанный детектор позволяет найти дефекты для ситуаций а, б, г.

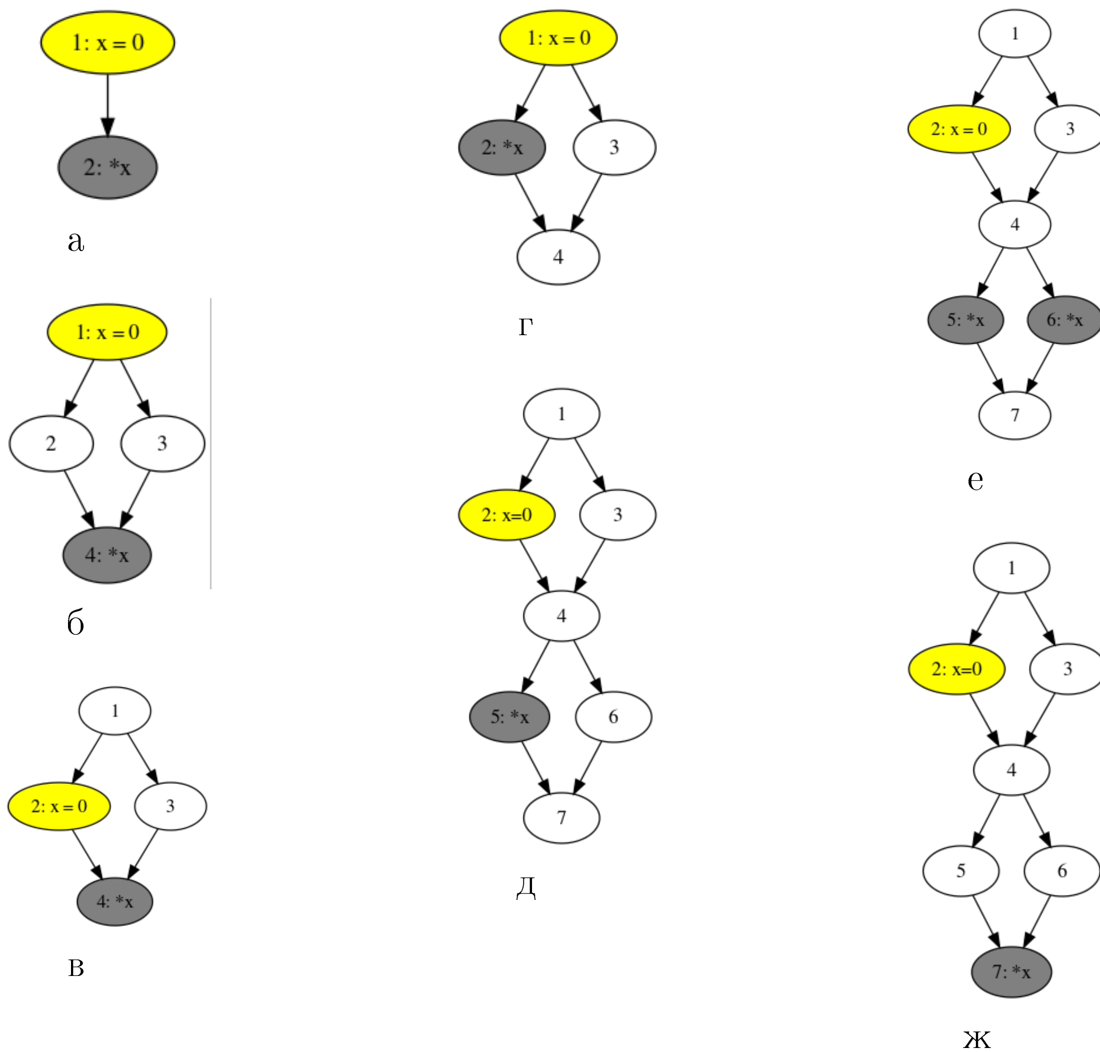

Рис. 4.7. Использование обратного анализа

Критерий выдачи предупреждений позволяет осуществлять поиск более сложных дефектов. Согласно критерию, необязательно, чтобы выполнение конкретной инструкции всегда приводило к ошибке. Достаточно, чтобы все пути, проходящие через некоторое ребро, содержали ошибку. При этом ошибка может

происходить в разных инструкциях программы. Предупреждение также можно выдать для ситуаций в, е, ж, где разыменование обязательно произойдёт в будущем. Для поиска таких дефектов используется обратный анализ.

Обратный анализ позволяет ответить на вопрос, что произойдёт после выполнения некоторой инструкции. Передаточные функции для обратного анализа на вход получают состояние на выходном ребре инструкции и возвращают состояние для входного ребра инструкции. В точках разветвления графа потока управления обратный анализ для состояний на выходных рёбрах создаёт состояние на входном ребре.

В работе предлагается простая схема обратного анализа, когда во время прямого анализа для каждой инструкции создаются передаточные функции для обратного анализа. Создание передаточных функций зависит от детекторов. После прямого анализа выполняется обратный, который использует созданные передаточные функции. Обратный анализ не создаёт новых идентификаторов значений, а распространяет атрибуты для идентификаторов значений, созданных во время прямого анализа. Производится всего одна итерация обратного анализа. Одного обхода недостаточно для полного анализа поведения функции, но экспериментально было установлено, что одного обхода хватает на практике для большинства ситуаций.

4.6.2. Расширение детектора разыменования нулевых указателей

Добавим атрибут $Deref$ со значениями $deref$ – указатель точно будет разыменован и $other$ – остальное.

Во время прямого анализа детектор создаёт передаточные функции, которые будут использоваться во время обратного анализа. Для всех инструкций, которые разыменовывают переменную с идентификатором значения v, создаётся передаточная функция $backderef(Deref) = Deref[v \mapsto deref]$. Такими инструкциями будут инструкции $load$, $store$, а также инструкция вызова функции, если анализом было установлено, что функция разыменовывает некоторую переменную.

Передаточная функция $backderef$ используется обратным анализом для распространения атрибута $Deref$. Тогда в точках графа потока управления,

где с указателем будут ассоциированы значения атрибутов $null$ и $deref$, будет выдано предупреждение, т. к. в этой точке указатель может иметь только нулевое значение, и в дальнейшем указатель точно будет разыменован.

Подобный анализ позволяет находить несогласованное использование указателей. Предупреждение, найденное описываемым детектором, показано на рис. 4.8. На строке 111 указатель vc проверяется на ноль, т. е. в данной точке предполагается, что указатель vc может иметь нулевое значение. В строке 119 производится разыменование указателя vc без проверки на ноль. По всей видимости, вызов функции memset необходимо вставить после строки 117. Возможно, проверка на ноль является лишней, но в этом случае её лучше убрать, т. к. она только запутывает код функции.

```
110|void vorbis_comment_clear(vorbis_comment *vc){
111|  if(vc) {
112|    long i;
113|    for(i=0;i<vc->comments;i++)
114|      if(vc->user_comments[i]) _ogg_free(vc->user_comments[i]);
115|    if(vc->user_comments) _ogg_free(vc->user_comments);
116|    if(vc->comment_lengths) _ogg_free(vc->comment_lengths);
117|    if(vc->vendor) _ogg_free(vc->vendor);
118|  }
119|  memset(vc,0,sizeof(*vc));
120|}
```

Рис. 4.8. Несогласованное использование указателей, найденное для Android 5.0.2, файл treminfo.c

Другое срабатывание детектора показано на рис. 4.8. Здесь сравнивается значение переменной mds, а разыменовывается значение переменной md. Ядро анализатора сопоставляет этим переменным один и тот же идентификатор значения.

4.7. Сопоставление атрибутов ссылкам

До сих пор атрибуты сопоставлялись только идентификаторам значений. Анализ позволяет сопоставлять атрибуты также ссылкам. Может показаться, что сопоставление атрибутов ссылкам является лишним, т. к. ссылки являются

```
static void exif_mnote_data_samsung_load (ExifMnoteData *md,
   const unsigned char *buf,
   unsigned int buf_size) {
161:ExifMnoteDataSamsung *mds = (ExifMnoteDataSamsung *) md;
    ExifShort entry_count;
    size_t i, tcount, off, datao;
    size_t val_size;
166:if (!mds || !buf || !buf_size) {
167:   exif_log (md->log, EXIF_LOG_CODE_CORRUPT_DATA,
           "ExifMnoteDataSamsung", "Short MakerNote");
      return;
    }
```

Рис. 4.9. Несогласованное использование указателей, найденное для tizen/framework-multimedia, файл exif-mnote-data-samsung.c

видом идентификаторов значений. Ниже будет определена операция сопоставления атрибута ссылкам и показан пример её использования.

Функция pt из состояния анализа отслеживает все ссылки, на которые может указывать некоторый идентификатор значения. Если два указателя могут быть алиасами, то анализ сопоставит им такие идентификаторы значения v_1 и v_2, что пересечение $pt(v_1) \cap pt(v_2)$ является непустым. Последнее можно переформулировать следующим образом: $\exists a\, a \in pt(v_1) \wedge a \in pt(v_2)$. Т. е. для идентификаторов значений, являющихся алиасами, всегда существует ссылка, на которую указывают оба этих идентификатора значения. Это свойство можно использовать для анализа сразу всех алиасов некоторого идентификатора значения. Для этого достаточно ассоциировать атрибут со всеми ссылками, на которые указывает идентификатор значения. В результате все алиасы будут указывать на ссылку, помеченную этим атрибутом.

Для переменной s сопоставление значения a_v атрибута $Attr$ для идентификатора значения будет функцией $Attr[val(s) \mapsto a_v]$. Под сопоставлением значения a_v атрибута $Attr$ для ссылки будем понимать функцию $Attr[pt(val(s)) \mapsto a_v]$. При этом функция pt возвращает множество ссылок, и всем этим ссылкам будет сопоставлено значение атрибута a_v.

Рассмотрим анализ утечек памяти для фрагмента кода на рис. 4.10. Инструкция (1) выделяет в куче память и сохраняет указатель в переменной q. Инструкция (2) освобождает память, на которую указывает переменная q.

```
void foo(char*p, int flag) {
    char*q = 0; char* r;

    if(flag & 0x10) {
        q = (char*)malloc(100); //(1)
        r = q;
    } else {
        r = p;
    }

    use(r);
    free(q); //(2)
    //(3)
}
```

Рис. 4.10. Фрагмент кода с условным выделением памяти

Будем использовать атрибут HeapV для обозначения выделенной в куче памяти, которая не была освобождена. Значения атрибута: *heap* – память, возможно, выделена и не была освобождена, *maybeheap* – память была выделена на одном из путей и не была освобождена, *other* – остальное. $heap \sqsubset maybeheap$, $maybeheap \sqsubset other$. Если ассоциировать атрибут с идентификатором значений, то передаточная функция для функции освобождения памяти (2) изменит только значение атрибута для q. И в точке (3) получим, что $HeapV(v_r) = maybeheap$, $HeapV(v_q) = other$. Т. е. указатель r, возможно, указывает на память, которая была выделена и не была освобождена.

Если же ассоциировать атрибут $HeapA$ с ссылками, тогда в точке (3) получим, что $HeapV(a_q) = other$, что точнее описывает поведение функции.

4.8. Чувствительность к путям

4.8.1. Мотивация для чувствительности к путям

Анализ, имеющий чувствительность к путям, различает пути, по которым выполнение могло достигнуть некоторой точки, и способен отсеять несуществующие пути, зависящие от предикатов, которые не могут одновременно выполняться.

В ситуации “д” на рис 4.7 наличие ошибки зависит от условий, которые определяют переходы в графе потока управления. Такую ситуацию часто называют “двойным ромбом” из-за формы графа потока управления.

На рис 4.11 показан пример кода, граф потока управления которого содержит “двойной ромб”. В примере имеется два условия и четыре возможных пути выполнения. Если не рассматривать условия, от которых зависит выполнения программы, то нельзя определить, содержит ли данный код ошибку разыменования нулевого указателя. Поэтому анализатору без чувствительности к путям остаётся либо выдавать предупреждение об ошибке, что может привести к ложным срабатываниям, либо не выдавать предупреждение о потенциальной ошибке, т. к. недостаточно данных. При использовании более сложных алгоритмов можно определить, что разыменование нулевого указателя произойдёт, только если истинна формула $(\overline{(a > b)} \wedge (a > b+1)$. Предупреждение выдаётся, только если формула разрешима, что в данном случае неверно[2].

```
int g;
void func(int a, int b) {
    int*p = 0;
    if(a > b) { p = &g; }
    if(a > b + 1) {*p  = 6;}
}
```

Рис. 4.11. “Двойной ромб”

4.8.2. Представление формул

Свойства анализа формулируются в виде формул логики высказываний и представляются с помощью атрибутов. Формулы имеют вид, показанный на рис. 4.12, где Val – идентификаторы значений, $Const$ – константы. Для каждой литеральной формулы $Atom$ определена инструкция отрицания, возвращающая другую литеральную формулу.

Для представления формул используется ациклический граф, узлами которого являются операции конъюнкции и дизъюнкции, а листьями – литеральные

[2] В примере ошибка будет, если учитывать возможность переполнения машинных чисел.

$\bowtie ::= > | < | = | \neq | \leq | \geq$
$\oplus ::= + | \text{-} | * | /$
$\textbf{Op} ::= \textbf{Val} \mid \textbf{Const}$
$\textbf{Atom} ::= \textbf{True} \mid \textbf{False} \mid \textbf{Op} \bowtie \textbf{Op}\ \textbf{Op} = \textbf{Op} \oplus \textbf{Op}$
$\textbf{Conj} = \textbf{Atom} \mid \textbf{Conj} \wedge \textbf{Conj}$
$\textbf{CFormula} ::= \textbf{Conj}$
$\textbf{SFormula} ::= \textbf{Conj} \wedge \overline{\textbf{Conj}}$
$\textbf{FFormula} ::= \textbf{Atom} \mid \textbf{FFormula} \wedge \textbf{FFormula} \mid \textbf{FFormula} \vee \textbf{FFormula}$

Рис. 4.12. Используемые условия.

формулы. Используется алгоритм хеширования, гарантирующий, что одинаковым формулам будет сопоставлен один граф. При такой реализации граф более сложной формулы ссылается на графы её частей, что позволяет сократить потребление памяти (общие части всех формул хранятся только один раз).

Были добавлены новые типы атрибутов для представления формул: $CFormula$, $SFormula$ и $FFormula$. Самый простой атрибут описывает формулы $CFormula$, являющиеся конъюнкцией литеральных формул. Атрибут $SFormula$ описывает формулы, состоящие из конъюнкции и отрицания конъюнкции, что позволяет описывать условия в функциях, которые имеют условную инструкцию выхода:

```
void func(char*p, int f) {
        if(!p && f)
                return;//p̄ ∧ f


        //‾(p̄ ∧ f)
        if(f)
                *p = 0;//f ∧ ‾(p̄ ∧ f)
}
```

Условие $f \wedge \overline{\overline{p} \wedge f}$ может быть выражено с помощью атрибута $SFormula$, но не может быть выражено с помощью атрибута $CFormula$, т. к. условие невозможно представить в виде конъюнкции с отрицанием только литеральных формул. Атрибут на основе $FFormula$ позволяет описать ещё более сложные условия, но и требует более сложного анализа для определения разрешимости

формулы. Выбор конкретного типа атрибутов зависит от детектора.

4.8.3. Вспомогательные атрибуты

На основе описанных выше типов атрибутов были реализованы расширения анализа, выполняющие поиск необходимых условий для каждого ребра графа потока управления. Необходимые условия достижимости ребра на графе потока управления – это условия, которые следуют из того факта, что ребро было достигнуто.

Посчитанные необходимые условия доступны для каждого детектора. Создаётся несколько атрибутов разных типов для вычисления необходимых условий, что позволяет использование этих атрибутов разными детекторами, в том числе детекторам вообще без чувствительности к путям.

Введём операцию $F \bigtriangledown A$, заменяющую в формуле F формулу A на истину. Обозначим $F^+ = F \bigtriangledown A$.

Теорема 3. $\forall A, F \in FFormula : F \Rightarrow F \bigtriangledown A$.

Доказательство. Доказательство будем проводить по индукции по структуре дерева разбора формулы. Индуктивное предположение $F \Rightarrow F \bigtriangledown A$.

Для литеральных формул предположение справедливо. Пусть F – литеральная формула, A – некоторая формула. Если $F \neq A$, то $F^+ = F \bigtriangledown A = F$ и $F \rightarrow F$. Если $F = A$, то $F^+ = F \bigtriangledown F = True$ и $F \Rightarrow True$.

Сложные формулы имеют вид $F \vee G$ и $F \wedge G$.

Докажем, что $(F \vee G) \Rightarrow (F \vee G) \bigtriangledown A$, по индуктивному предположению $F \Rightarrow F \bigtriangledown A$ и $G \Rightarrow G \bigtriangledown A$. Используя свойство $((X \Rightarrow X') \wedge (Y \Rightarrow Y')) \Rightarrow ((X \vee Y) \Rightarrow (X' \vee Y'))$, получим, что $(F \vee G) \Rightarrow (F \bigtriangledown A) \vee (G \bigtriangledown A)$.

Доказательство $(F \vee G) \Rightarrow (F \vee G) \bigtriangledown A$ получается аналогично, используя свойство $((X \Rightarrow X') \wedge (Y \Rightarrow Y')) \Rightarrow ((X \wedge Y) \Rightarrow (X' \wedge Y'))$.

□

Согласно теореме 3, формула F^+ является приближением формулы F, т. е. формула F^+ является более слабой. Теорема позволяет выбирать приближённую формулу для описания точного условия.

4.8.4. Детекторы с чувствительностью к путям

Общий подход поиска дефектов заключается в следующем: условие ошибки формулируется в виде формулы на основе одного из перечисленных выше атрибутов. Анализ строит формулу по возможности точно. Если не удалось доказать отсутствие ошибки, то выдаётся предупреждение о подозрительной ситуации.

Рассмотрим уже описанную задачу поиска разыменований нулевых указателей: возможно ли при выполнении программы, что произойдёт разыменование указателя, который был сравнён с нулём. На этот раз поиск ошибки будет реализован с чувствительностью к путям.

Условие будем ассоциировать с идентификаторами значений. Атрибут NullCond для идентификатора означает ассоциированные условия того, что значение указателя было сравнено с нулём и оказалось равным нулю. Атрибут RchCond обозначает необходимые условия.

$$NullCond\|assume(p = 0)\| \ \mapsto NullCond[p \mapsto True]$$

$$NullCond\|assume(p! = 0)\| \ \mapsto NullCond[p \mapsto False]$$

$$NullCond\|r = \phi(a_0, .., a_k)\| = \bigcup(NullCond_{in_k}(a_k) \wedge RchCond(in_k))$$

Для инструкции разыменования $*q$ условие ошибки описывается формулой $NullCond_p(v_q) \wedge RchCond(p) \wedge v_q = 0$. Т. е. предупреждение будет выдано, если для ребра не было установлено, что оно недостижимо, и указатель может иметь нулевое значение.

Глава 5

Межпроцедурный анализ

5.1. Резюме функции

Используется анализ на основе резюме [68]. При таком анализе каждая функция представлена кратким описанием поведения функции (“резюме”). Резюме используется для анализа инструкции вызова функции и позволяет избежать повторного анализа тела функции. Описываемая работа использует вариант анализа, при котором функции программы обходятся по графу вызовов, начиная с листьев, таким образом, чтобы вызываемая функция обходилась до вызывающей. Циклы в графе вызовов разрываются.

Резюме функции описывает эффект от вызова функции и является набором частичных функций val, pt и $Attr$ для каждого атрибута. Резюме содержит информацию об изменении отношений указывания, а также информацию, специфичную для отдельных детекторов.

Процесс переноса эффектов от вызова функции, сохранённых в резюме, в контекст вызова, будем называть *трансляцией резюме*. Все значения в функции делятся на два вида: внешние и внутренние. Внешние значения будем называть *предзначениями*, это фактические значения из произвольного контекста вызова, свойства которых неизвестны.

Внутренние значения создаются внутри функции, и на стороне вызывающей функции им не соответствуют никакие объекты. Поэтому в резюме необходимо поместить все предзначения и только те внутренние значения, до которых можно “дотянуться” через последовательность разыменований и сдвигов, начиная от предзначений. Последнее будет соответствовать ситуации, когда внутри функции было создано значение и записано в память, указываемую формальным аргументом функции.

В главе будут использоваться следующие обозначения:
$v = pr(a)$ – идентификатор значения v является предзначением для ссылки a.
InV – множество входных параметров

RV – множество возвращаемых параметров

5.2. Создание резюме функции

5.2.1. Граф формы памяти

На основе функций функций val и pt можно создать *граф формы памяти*. В графе формы памяти вершинами являются идентификаторы значений и ссылки. Ребро от идентификатора значения v к ссылке a добавляется, если $a \in pt(v)$. Ребро от ссылки a к идентификатору значения v добавляется, если $val(a) = v$.

Пусть функция $Len(a, b)$ возвращает минимальную длину пути от a к b в графе формы памяти. Элементы будем называть *связанными*, если $Len(a, b) = 1$. Граф является направленным, и $Len(a, b)$ не обязательно равно $Len(b, a)$.

5.2.2. Множество добавляемых элементов

Для создания резюме функции используется состояние в единственной инструкции выхода из функции. В резюме добавляются идентификаторы значения переменных, которые будут видимы в контексте вызова. Значения большинства локальных переменных не добавляются в резюме. Исключением является ситуация, когда значение локальной переменной было записано в память, видимую в контексте вызова. Добавление идентификаторов значений в резюме начинается с входных параметров функции. После чего производится несколько обходов состояния (конкретная реализация описана в разделе 6.4), на каждом из которых в резюме добавляются идентификаторы значений, связанные с уже добавленными.

Процесс создания резюме параметризуется количеством обходом состояния. Пусть N – количество обходов состояния, G – граф формы памяти. В резюме будут добавлены только элементы, для которых длина пути в графе отношений от входных параметров и возвращаемого значения меньше N. Обозначим это множество S. Формально $S = \{a \,|\, Len(a, b) < N, b \in InV \cup RV\}$. Для локальных переменных, значения которых не были записаны в память,

на которую указывают значения, длина пути на графе G от любого входного параметра будет бесконечной, поэтому они не будут добавлены в резюме.

Параметр N ограничивает размер множества S. Но в некоторых случаях этого может быть недостаточно для создания компактных резюме. С целью минимизации используемой памяти используется дополнительное ограничение на количество добавляемых элементов в резюме. Пусть M описывает это ограничение. Чем больше будут параметры N и M, тем точнее будут резюме, но и тем больше памяти потребуется для хранения резюме. Выбор N и M в нашей реализации описан в разделе 6.4.

Листинг 1 содержит алгоритм добавления идентификаторов значений и ссылок в резюме. Вначале создаётся множество r, обозначающее множество всех элементов, которые будут добавлены в резюме. Множество входных параметров и возвращаемые значения добавляются в множество r. Затем производится ряд итераций обхода состояния, которые завершаются либо когда превышен лимит количество итераций, либо когда превышен лимит количества элементов в резюме, либо не было изменений. Множество v' обозначает множество вновь добавленных идентификаторов значений на каждой итерации. Множество a' обозначает множество вновь добавленных ссылок. На каждой итерации добавляются связанные элементы с элементами из множеств v' и a'. На основе множества r формируются ограничения[1] для функций val и pt. Т. е. результатом будут две частичные функции, которые определены только на множестве r.

Определённую сложность представляет анализ массивов. Возможны ситуации, когда информация дублируется для разных элементов массива. Сохранённая информация не всегда будет использоваться. В результате в резюме будет добавлена малополезная информация об элементах массива. Для предотвращения этого используется порог на максимальное количество элементов массива в резюме. Если в резюме добавлено элементов массива больше этого порога, то все они исключаются из резюме. Экспериментально было установлено, что сохранение информации об элементах массива не приводит к существенно более точному анализу, зато сильно увеличивает потребление памяти. Такой резуль-

[1] Ограничением функцииF для множества s будет $f \restriction s = \{< a, b > | f(a) = b, a \in s\}$.

Алгоритм 1 Алгоритм добавления элементов в резюме

function CREATE_SUMMARY($val, pt, inval, rval$)
  $r \leftarrow inval \cup rval$
  $v' \leftarrow r$
  for $i \leftarrow 1$ to N **do**
    if $sizeof(r) < M \wedge (a' \neq \emptyset \vee v' \neq \emptyset)$ **then**
      $a' \leftarrow \bigcup\{a|a \in pt(v),\ v \in v'\}$
      $v' \leftarrow \bigcup\{val(a)|a \in a'\}$
      $r \leftarrow r \cup v' \cup a'$
    end if
  end for
  $val' \leftarrow val \upharpoonright r$
  $pt' \leftarrow pt \upharpoonright r$
  return $< val', pt' >$
end function

тат объясняется тем, что программисты, как правило, работают с элементами массивами единообразно, и свойства разных элементов массива не различаются. Если необходимо различать элементы массива, то удобнее использовать структуры.

5.2.3. Добавление атрибутов

Включение атрибутов в резюме зависит от конкретного детектора. Для поддержки межпроцедурного анализа генерируется событие *annotate*. Событие $annotate(d, s)$ вызывается при создании резюме функции. Событие вызывается только для элементов, добавленных функцией $CREATE_SUMMARY$ алгоритма 1. При этом d – идентификатор значения в резюме, а s – идентификатор значения для состояния анализа в единственной инструкции возврата из функции.

Внутрипроцедурные детекторы могут просто игнорировать событие *annotate*. Для межпроцедурных детекторов необходимо проверить свойства s и при необходимости изменить свойства d. Реализация передаточной функции для *annotate* зависит от детектора, но должны выполняться два принципа:

1. Атрибут, записываемый в резюме, должен быть достаточно компактным. Нежелательно запоминать свойства программы, представление которых

потребует значительного количества памяти. Игнорирование этого принципа приведёт к росту размера резюме. В отличие от состояний анализа, которые удаляются после завершения анализа функций, резюме могут храниться значительное время. В текущей версии Svace размер резюме, как правило, не превышает 100Кб. Множество атрибутов необходимо выбирать таким образом, чтобы не превышать это значение.

2. Не следует сохранять свойства, которые не будут использованы дальнейшим анализом. Например, если функция может разыменовать указатель, но неизвестно, при каких именно условиях, то для анализа не имеет смысла сохранять эту информацию, т. к. в точке вызова функции предупреждение всё равно не будет выдано (с целью минимизации количества ложных срабатываний).

5.2.4. Объединение элементов в резюме

При создании резюме может иметь избыточную структуру вида: $< val[a \mapsto v_x, b \mapsto v_y], pt[v_p \mapsto \{a, b\}], .. >$, где a, b, v_x, v_y являются внутренними значениями. Граф формы памяти показан на рис. 5.1.

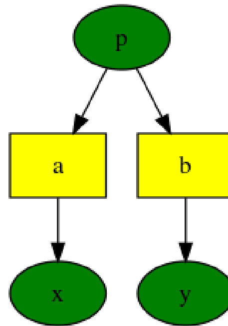

Рис. 5.1. Избыточная структура в резюме

Неизвестно, на какую именно ячейку памяти указывает p – на ячейку, описываемую ссылкой a, или на ячейку, описываемую ссылкой b. Поэтому в момент трансляции резюме свойства идентификатора значения, обозначающего разыменование p, должны включать в себя свойства и v_x и v_y (будет использоваться функция $joinval$ из раздела 3.7).

Если нет других ссылок на a и b, то нет необходимости производить эту операцию в момент трансляции резюме, можно выполнить все действия в мо-

мент создания резюме. Для этого создаётся ссылка, обозначающая свойства a и b, и идентификатор значения $v_{xy} = joinval(v_x, v_y)$. В результате состояние анализа упрощается до $< [ab \mapsto v_{xy}][v_p \mapsto \{ab\}], .. >$. Затем анализ вызывает оператор объединения для всех атрибутов, ассоциированных с v_x и v_y. Если $Attr(v_x) \sqcup Attr(v_y) = \top$, то такой атрибут не добавляется в резюме, т. к. не содержит информации[2].

5.3. Трансляция резюме в контекст вызова

5.3.1. Карта трансляции

При трансляции резюме в контекст вызывающей функции происходит сопоставление формальных и фактических параметров. В процессе трансляции резюме используется функция $trans : V \to \wp(V)$, сопоставляющая каждому элементу в резюме множество соответствующих элементов в состоянии контекста вызова.

Отношения соответствия элементов – многие ко многим, а не один к одному. Ситуация, когда $trans(v) = \{x, y\}$, возникает, если одно и то же значение в вызываемой функции присваивается нескольким переменным вызывающей функции:

```
void foo(int*a, int*b) {
        int x = bar();
        *a = x;
        *b = x;
}
```

Ситуация, когда $trans(v) = \{x\}$ и $trans(w) = \{x\}$, возникает, если одинаковые переменные передаются как фактические параметры функции:

```
foo(z, z);
```

В последнем примере нарушается предположение об отсутствии алиасов среди параметров функции. Корректность в этом случае не гарантируется, но

[2] Значение $\top$ является верхним элементом решётки и абстрагирует любые свойства. Информацию о том, что переменная может иметь любые свойства, можно не хранить явно.

экспериментально было установлено, что такие случаи редко приводят к ложным срабатываниям.

Элемент x назовём *родительским* для элемента y, если $y = pr(x)$ или $y = field(x, f)$ для некоторого f.

На рис. 5.2 приведены правила создания функции $trans$ в виде индуктивных высказываний. Для построения функции trans вначале в карту соответствия добавляются формальные и фактические параметры. Затем в карту добавляются идентификаторы значений, у которых нет родительских элементов. После чего, если в карте есть родительский элемент, то осуществляется поиск соответствующего родительского элемента в контексте вызова, и для него создаётся новый элемент.

$$\frac{x \text{ -- формальный параметр, } v \text{ -- фактический параметр}}{v \in trans(x)}$$

$$\frac{\nexists a,y,f. x{=}field(y,f) \vee x{=}pr(a) \quad v{-}fresh}{trans(x){=}\{v\}}$$

$$\frac{m \in trans(a) \; x{=}pr(a)}{pr(m) \in trans(x)} \quad \frac{v \in trans(x) \; w{=}field(v,f)}{field(x,f) \in trans(w)}$$

Рис. 5.2. Правила создания функции trans

5.3.2. Изменение состояния анализа в контексте вызова

Помимо создания новых идентификаторов значений и ссылок, во время трансляции резюме в контекст вызова происходит обновление существующих ссылок в контексте вызова. Если функция изменяет состояние памяти программы при выполнении, то и анализ должен учитывать это. В описываемом анализе свойства памяти моделируются с помощью функций val и pt.

Функция pt обозначает, на какие ссылки указывает идентификатор значения. Обновления этой функции происходят только для вновь создаваемых идентификаторов значения. Функия val моделирует значения, которые могут быть у псевдорегистров и ячеек памяти.

Резюме создаётся для точки выхода из функции. Поэтому, если в этой точке $val(a) = v$, то ячейки памяти, обозначаемые a, будут иметь значение, моделируемое v.

Резюме будем обозначать как $r =< val^R, pt^R >$. Если ссылка a описывает

ячейки, видимые на стороне вызова, то значение попадёт в резюме: $val^R(a) = v$. Для обновления состояния в контексте вызова достаточно найти соответствия для a и v на стороне вызова с помощью функции $trans$ и обновить состояние анализа в точке вызова:
$val_{out} = val_{in}[trans(a) \mapsto trans(v)]$. Сложность возникает, только если $trans(v)$ содержит более одного элемента. В этом случае делается слабое обновление нового значения с предыдущим в val_{in}.

Алгоритм изменения отношений указывания приведён на листинге 2. На вход алгоритм получает состояние анализа, резюме и функцию трансляции. Возвращается функция val для состояния после инструкции вызова. Для реализации сильных и слабых обновлений используется вспомогательная функция $joinvals$, определённая следующим образом:

$$joinvals(v_p, vals) = \begin{cases} v, & \text{если } vals = \{v\}; \\ joinval(\{v_p\} \cup vals), & \text{иначе} \end{cases}$$

Алгоритм 2 Алгоритм обновления АС в контексте вызова

function UPDATE_CONTEXT_STATE($s, r, trans$)
    $val' \leftarrow s.val$
    for $a^R \in dom(r.val)$ **do**
        $v^R \leftarrow r.val(a^R)$
        for $a \in trans(a^R)$ **do**
            $v \leftarrow joinvals(val(a), trans(v^R))$
            $val' \leftarrow val'[a \mapsto v]$
        end for
    end for
    return val'
end function

5.3.3. События для межпроцедурного анализа

Событие $translate(d, s)$ генерируется в момент трансляции идентификаторов значения из резюме в контекст вызова функции. Межпроцедурные детекторы могут распространять анализируемые свойства в контекст вызывающей функции либо сообщить об ошибке, если для этого достаточно оснований. Со-

бытие *translate* генерируется только для элементов резюме, для которых определена функция *trans* из раздела 5.3.1.

5.4. Отложенное слияние для параметров функций

Рассмотрим ссылку a в точке слияния путей. Пусть $v_1 = val_1(a)$ – идентификатор значения ссылки на одном пути, а $v_2 = val_2(a)$ – идентификатор значения на другом. Пусть $v_3 = joinval(v_1, v_2)$ – идентификатор значения на выходном ребре. Будет вызван оператор объединения для всех атрибутов, ассоциированных с v_1 и v_2. Если v_1 является предзначением, то ничего неизвестно про его свойства до начала выполнения функции. Поэтому свойства атрибутов будут описываться верхним элементом решётки $\top$. В этом случае для любого атрибута результатом слияния будет $\top \sqcup Attr(v_2) = \top$, т. е. произойдёт потеря информации. В случае внутрипроцедурного анализа такой анализ приемлем, т. к. рассматриваются все контексты вызова функции. Проблема возникает в том случае, когда результат, описываемый идентификатором v_3, записывается в память, видимую на стороне вызова, где значения предзначений будут известны.

На рис. 5.3 показан пример кода, где происходит потеря информации. Атрибут $ValueInterval$ используется для анализа интервала целочисленных значений (раздел 4.4). При анализе функции set ничего не известно про значения $*p$. Поэтому значение $ValueInterval$ для $*p$ в начале функции будет $[-\infty;+\infty]$. В точке слияния путей (1) новое значение для $*p$ будет $[-\infty;+\infty] \sqcup [11;11] = [-\infty;+\infty]$, т. е. информация о том, что в функции переменной $*p$ может быть присвоено значение 11, будет потеряна. В результате с идентификатором значения для переменной x в точке (2) будет ассоциировано значение интервала $[-\infty;+\infty]$.

Для более точного анализа предложен механизм *отложенного слияния*. Если дополнительно запоминать слияния, произошедшие с предзначениями, то можно вызывать оператор объединения для атрибутов для фактических параметров и значений, используемых внутри функции. Благодаря этому улучшается степень контекстной чувствительности алгоритма.

```
void set(int*p, int flag) {
    if(flag)
        *p = 11;
    //(1)
}
```

```
void use(int a) {
    int x = 10;
    set(&x, a);
    //(2)...
}
```

Рис. 5.3. Мотивация для отложенного слияния идентификаторов значений

Будем использовать непересекающиеся множества JV и SV, определённые в разделе 3.2.2. Множество JV используется для обозначения составных идентификаторов значения, созданных с помощью функции joinval. Множество SV обозначает простые идентификаторы значений.

Выделим подмножества из множества Val: PV для обозначения предзначений.

Будем использовать функцию *parts*, которая для каждого идентификатора значения возвращает множество идентификаторов значений, из которых он был создан:

$$parts(v) = \begin{cases} \{v\} & \text{если } v \in SV; \\ part(v_1) \cup part(v_2) & \text{если } v = joinval(v_1, v_2). \end{cases}$$

Для механизма отложенного слияния потребуется две функции: *delay* и *ival*. Функция *ival* используется для создания идентификатора значения, обозначающего свойства всех внутренних значений.

$$ival(joinval(v_1, v_2)) = \begin{cases} ival(v_1) & \text{если } v_1 \notin PV, v_2 \in PV; \\ ival(v_2) & \text{если } v_2 \notin PV, v_1 \in PV; \\ joinval(ival(v_1), ival(v_2)) & \text{если } v_1 \notin PV, v_2 \notin PV. \end{cases}$$

Функция *delay* требуется для запоминания всех предзначений, используемых в точке слияния: $delay(v) = parts(v) \cap PV$.

Пусть *trans* обозначает карту трансляции идентификаторов значения из резюме в сопоставленные идентификаторы значения вызывающей функции. В момент трансляции резюме в контекст вызова запомненные предзначения заменяются на соответствующие идентификаторы значения на стороне вызова:

$delay'(v) = \{trans(w)|w \in delay(v)\}$. Объединение атрибутов производится для атрибутов соответствующих элементов контекста вызова $delay'$(v) и атрибутов резюме для $ival(v)$.

Это позволяет более точно производить объединение атрибутов для отдельных детекторов. Рассмотрим механизм на примере рис. 5.3. При использовании отложенного слияния для функции set будет создан идентификатор значения со ссылками на предзначение v_{dp} для *p и v_{11}, при этом $ValueInterval(v_{dp}) = [-\infty; +\infty]$, $ValueInterval(v_{11}) = [11; 11]$. В контексте вызова функции set $ValueInterval(v_x) = [10; 10]$, новое значение для x будет $r = joinval(v_x, v_{11})$. Значение $ValueInterval$ для r будет вычислено как $ValueInterval_{use}(r) = ValueInterval_{use}(v_x) \sqcup ValueInterval_{set}(v_{11}) = [10; 10] \sqcup [11; 11] = [10; 11]$. При этом, для операции слияния используются атрибуты как контекста функции use, так и резюме для функции set. Для данного примера, где неизвестны возможные значения параметра a, значение интервала для переменной x, равное $[10; 11]$, будет наиболее точным. Удаление 10 или 11 из интервала делает анализ некорректным.

5.5. Трассы атрибутов

От статического анализатора требуется не только найти ошибку, но и объяснить, почему найденная ситуация является ошибкой. Для этой цели используются *трассы* атрибутов. Трасса атрибута является односвязным списком, в котором каждый элемент хранит точку в программе, где менялось значение атрибута, и поясняющее сообщение. Детектор принимает решение о выдаче предупреждения на основе состояния атрибутов. Для этих атрибутов детектор дополнительно показывает трассу, т. е. точки в программе, где атрибут распространялся.

Трассы полезны и для внутрипроцедурных детекторов, но особенно важны для поиска межпроцедурных дефектов. На рис. 5.4 показан пример предупреждения разыменования нулевого указателя, найденный для проекта Android 5.0.2 (external/linux-tools-perf/perf-3.12.0/tools/perf/builtin-script.c:914).

Если функция script_desc__new вернёт нулевой указатель на строке 905,

```
875|static void
script_desc__delete(struct script_desc *s) {
877|    free(s->name);
878|    //...
881|}

898|static struct
script_desc *script_desc__findnew(const char *name)
899|{   //...
905|    s = script_desc__new(name);
906|    if (!s)
907|        goto out_delete_desc;
909|    script_desc__add(s);
911|    return s;
913|out_delete_desc:
914|    script_desc__delete(s);
916|    return NULL;
917|}
```

Рис. 5.4. Разыменование указателя внутри функции

то управление перейдёт на строку 914, где произойдёт разыменование (нулевого) указателя вызовом функции script_desc__delete. Без дополнительной информации непонятно – действительно ли функция *script_desc__delete* разыменовывает без проверки свой аргумент. Трасса для атрибута разыменования содержит точку, где атрибут был создан, – 877. Благодаря трассе пользователь может быстро проверить, что функция *script_desc__delete* действительно безусловно разыменовывает свой аргумент.

5.6. Параметризованные атрибуты

Для улучшения контекстной чувствительности могут использоваться *параметризованные* атрибуты: такие атрибуты, которые ссылаются на идентификаторы значений. С помощью этих атрибутов сохраняется информация о взаимосвязи идентификаторов значений. Во время создания резюме и его трансляции в контекст вызова ссылки на формальные параметры в атрибутах будут заменены на фактические параметры в соответствии с картой трансляции *trans*.

Простейший параметризованный атрибут содержит только одну ссылку. Обозначим простой атрибут как $Param$. Между идентификаторами значения устанавливается взаимосвязь, если идентификатор значения v_1 имеет значение атрибута, равное v_2, т. е. $Param(v_1) = v_2$. Семантика взаимосвязи зависит от конкретного детектора. Более сложные параметризованные атрибуты могут содержать множество ссылок.

Параметризованные атрибуты удобно использовать для обработки кодов ошибок функций. Многие функции в случае неудачного завершения возвращают некоторый код ошибки. Если возвращаемое значение функции равно этому коду ошибки, то некоторое действие не было произведено функцией.

Например, функция $pthread_mutex_lock$ стандартной библиотеки Си изменяет состояние семафора на заблокированное. В случае удачного завершения функция возвращает нулевое значение. Ненулевое значение обозначает код ошибки, в этом случае семафор не был заблокирован. На рис. 5.5 показан типичный пример использования функций блокировки семафора. Если результат res вызова функции $pthread_mutex_lock$ не равен нулю, то семафор $mutex$ не был заблокирован. Поэтому в точке, помеченной на рисунке как (*), семафор находится в разблокированном состоянии. Анализ, не учитывающий кодов возврата, будет считать, что в точке (*) семафор заблокирован. Это приведёт к некорректному анализу, в частности, может быть выдано ложное срабатывание об отсутствии вызова $pthread_mutex_unlock$ для всех путей внутри функции use.

Для решения проблемы введём атрибут $ErrLock$, который будем ассоциировать с переменной для кода возврата из функции. Значение атрибута будет обозначать блокируемый семафор. Для кода из примера выше $ErrLock(v_{res}) = v_{mutex}$. В точке сравнения результата с нулём анализ переведёт семафор в разблокированное состояние.

5.7. Удаление резюме

Для хранения резюме используются компактные структуры данных, но общее количество анализируемых функций может быть значительным. В ре-

```
void use(pthread_mutex_t *mutex, int flag) {
    int res = pthread_mutex_lock(mutex);

    if(res!=0) {
        log("Error");
        return;//(*) error of mutex creation
    }

    critical_func(flag);
    pthread_mutex_unlock(mutex);
}
```

Рис. 5.5. Функция, использующая pthread_mutex_lock

зультате не все резюме могут поместиться в оперативной памяти. Для решения этой проблемы используется два метода:

1. Сериализация резюме (может быть как включена, так и выключена). Сериализация снижает требования к необходимой памяти, но замедляет анализ из-за операций чтения-записи на диск. Необходимость использования сериализации зависит как от размера проекта (анализ больших проектов потребует больше памяти), так и от размера оперативной памяти.

2. Удаление резюме после того, как они не нужны. Резюме используется только при анализе функций, вызывающих данную функцию. Поэтому после того, как все функции, вызывающие данную функцию, проанализированы, резюме удаляется.

5.8. Анализ конструкторов и деструкторов

В языке Си++ при создании объекта вызывается конструктор, осуществляющий его инициализацию. При удалении объекта будет вызван деструктор для очистки используемых ресурсов. Деструктор должен гарантировать, что все созданные в конструкторе ресурсы будут освобождены при уничтожении объекта. Несогласованность между конструкторами и деструкторами может приводить к утечке ресурсов, их двойному освобождению, либо освобождению невыделенных ресурсов.

Для поиска таких дефектов в конце анализа запускается фаза парного анализа конструкторов и деструкторов. Эта фаза была разработана таким образом, чтобы максимально переиспользовать возможности основного анализатора. Для этого в код деструктора в качестве первой инструкции вставляется вызов конструктора. Затем производится обычный анализ полученного кода, и при обработке вызова конструктора используется его резюме так же, как и при межпроцедурном анализе. Различные детекторы при этом могут проверить несогласованность в конце анализа деструктора. Если у класса есть несколько конструкторов, то такой анализ проводится для каждого конструктора в отдельности. Также парный анализ выполняется для операторов присваивания.

Ошибки несогласованности могут быть найдены и обычным анализом. При определении переменной компилятор сгенерирует вызов конструктора, деструктор будет сгенерирован для точки, где завершается область видимости переменной. Анализ найдёт ошибку, к которой приводит несогласованность конструкторов и деструкторов, и покажет её пользователю. Выделение отдельной фазы для анализа несогласованностей имеет следующие преимущества:

- Ошибка заключается не в использовании конструкторов и деструкторов, а в их несогласованном определении. Об ошибке можно сообщить даже, если класс или конкретный конструктор нигде не используются.

- Переменные класса не обязательно объявляются только на стеке. Часто все переменные создаются в куче, при этом точки создания и удаления могут быть значительно удалены друг от друга. Такие ошибки просто не будет найдены без использования специальной фазы.

- Общий анализ более сложный, т. к. требует две трансляции резюме для конструктора и деструктора. В результате не будут найдены ошибки в ситуациях, где не хватает контекстной чувствительности.

- Возникает сложность объяснения ошибки пользователю. Ошибки несогласованности конструкторов и деструкторов проще объяснить, так как достаточно показать только их код. При общем анализе также необходимо показать вызывающий код.

- Если класс имеет несогласованные конструкторы и деструкторы, и программа использует много переменных данного класса, то общий анализ выдаст множество предупреждений, соответствующих только одной ошибке в коде. Такая ситуация нежелательна, т. к. сообщения дублируют друг друга, и, как минимум, потребуется их фильтрация.

Например, детектор MEMORY_LEAK.CTOR выдаёт предупреждение, если в конструкторе выделяется память для некоторого члена класса, а в деструкторе отсутствует освобождение выделенной памяти. Для поиска ошибок в конце анализа кода деструктора, модифицированного вставкой вызова конструктора, детектор запускает стандартный анализ утечек памяти, осуществляющий поиск достижимых ячеек памяти. Если в результате поиска находится утечка памяти, то это означает, что после вызова деструктора не все ресурсы, выделенные в конструкторе, будут освобождены. Пример ошибки для проекта Android 5.0.2 (AST.cpp:612) приведён на рис. 5.6.

```
611| FinallyStatement::FinallyStatement()
612|            :statements(new StatementBlock) { }
   //Код деструктора :
616| FinallyStatement::~FinallyStatement()
617|{ }
    //Определение FinallyStatement (FrameworkListener.h) :
280| struct FinallyStatement : public Statement
281| {
282|     StatementBlock* statements;
```

Рис. 5.6. Несогласованность конструктора и деструктора

Глава 6

Реализация

В главе описывается реализация предложенных алгоритмов в инструменте Svace, оценивается производительность и результаты анализа.

6.1. Инструмент Svace

Инструмент Svace осуществляет статический поиск ошибок в программах, написанных на языках Си, Си++, Java и Си#. Поиск ошибок для каждого языка осуществляется двумя способами:

- с помощью просмотра абстрактного синтаксического дерева и поиска подозрительных шаблонов (реализовано в соответствующих компиляторах для каждого языка);
- с помощью межпроцедурного анализа на основе резюме (основной анализатор).

Описанные в работе алгоритмы были реализованы в семантическом анализаторе, осуществляющем анализ на основе резюме для языков Си и Си++. Часть ошибок в программах на Си и Си++ ищется с помощью разработанных в ИСП РАН детекторов на основе анализатора Clang Static Analyzer (CSA) [53, 71]. В этих детекторах не используются описываемые в работе алгоритмы, поэтому далее результаты подсистемы анализа на основе CSA оцениваться не будут.

Основной анализатор Svace при обработке программ на Си и Си++ получает на вход файлы биткода LLVM и информацию о компоновке программы. Биткод LLVM генерируется компилятором, основанным на Clang версии 3.4. По сравнению с открытой версией Clang выполнено более 500 модификаций, направленных на поддержку максимального количества диалектов и расширений языков Си и Си++, обусловленной разнообразием компиляторов, а также на

максимальное сохранение информации об исходном коде в генерируемом биткоде LLVM. Далее инструкции биткода преобразуются в язык svace0, описанный в разделе 2. Информация о выполненных компоновках при сборке программы используется для разрешения имён при построении графа вызовов анализатора.

Для получения хорошего качества анализа желательным является наличие спецификаций хотя бы для стандартных библиотек языков Си и Си++, т. к. эти библиотеки практически всегда используются в бинарном виде, и анализатор не имеет доступа к их исходному коду. Спецификации пишутся на языке Си в виде заглушек, в которых указаны только важные для анализа операции. Перед началом анализа для спецификаций также создаются резюме.

Инструмент Svace осуществляет анализ на основе резюме. Такой анализ имеет высокий потенциал для распараллеливания. Две функции анализируются независимо, если в графе вызова нет пути от одной функции к другой. Практически не требуется никакой синхронизации между разными потоками.

При запуске Svace можно задать максимальное количество используемых потоков. Анализатор создаёт требуемое количество потоков. Для выбора анализируемых функций используется диспетчер выбора функций, который уведомляет потоки о том, что для анализа доступна следующая функция. Диспетчер гарантирует, что каждая вызываемая функция будет проанализирована до вызывающей (циклы в графе вызовов разорваны). При выборе функций диспетчер решает следующие задачи:

- Минимизация количества резюме, которые требуется одновременно держать в памяти.

- Повышешие локальности анализа функций относительно модуля, где они определены. Благодаря этому повышается попадание в кэш, содержащий глобальную информацию для каждого модуля.

- Поддержка множества кандидатов для анализа достаточно большим, чтобы можно было загрузить все потоки.

- Избежание блокировок при чтении модуля. Нет смысла дублировать чтение одного и того же файла разными потоками. Если один поток осуществ-

ляет загрузку модуля для функции, а второй поток получит для анализа функцию из этого же модуля, то второй поток будет простаивать, ожидая завершения операции загрузки.

6.2. Трансляция Си и Си++-кода в LLVM

Был выполнен ряд изменений, направленный на сохранение информации об исходном коде, которая теряется при обычной трансляции исходного кода Си и Си++ в биткод LLVM. В разделе будут описаны наиболее интересные из таких изменений.

Первым шагом было отключение всех оптимизаций в транслируемой программе. Даже при отключенных оптимизациях компилятор Clang проводит некоторые простые оптимизации (замену инструкций на более быстрые, подстановку тел маленьких функций и др.). Для отключения этих оптимизаций потребовалась модификация кода компилятора.

Значительные сложности анализа были связаны с макросами. Программисты оперируют макросами так же, как и функциями. Поэтому макросы часто содержат проверки на случай использования в разных контекстах. Макросы заменяются препроцессором перед началом компиляции, и их тело вставляется в функции. С точки зрения анализатора, макрос – это некоторая последовательность инструкций. Поэтому если в макросе содержится проверка, которая в некоторой функции лишняя, то в соответствии с критерием выдачи, описанном в главе 4.2, будет выдано предупреждение, которое окажется ложным.

Для борьбы с такими срабатываниями в ранних версиях инструмента просто производилась фильтрация по строке, содержащей проверку. Если строка не содержала условных инструкций языка, то предупреждение не выдавалось. В текущем компиляторе, используемом Svace, помечаются все инструкции, сгенерированные из макросов. Инструмент не выдаёт предупреждение, если проверка находится в макросе.

Другой проблемой было определение точек возврата из процедуры. Генерируемая программа на языке LLVM содержит одну точку возврата из процедуры. Если программа на Си содержит несколько точек возврата, то в LLVM-коде они

будут заменены на безусловные переходы на базовый блок с единственной точкой возврата. При этом возвращаемое значение записывается в специальную переменную, которая будет прочитана перед инструкцией возврата. Рис. 6.1 содержит пример Си-функции с двумя инструкциями возврата и соответствующий код на языке LLVM. В функции *foo* произойдёт утечка выделенной памяти в строке (1), если входной параметр `q` равен нулю. Проблема в том, что в сгенерированном LLVM-коде есть только одна инструкция возврата, и невозможно определить, какой оригинальный Си-код соответствует LLVM-коду. Определение точной места происхождения утечки критично для правильного описания дефекта, особенно для функций большого размера, в которых разные точки возврата могут быть сильно удалены друг от друга. Неправильная идентификация точек возврата приводит к тому, что предупреждение будет размечено как ложное.

```
void foo(int m, char**q) {
  char*p = (char*)malloc(100);
  if(!q)
    return; //(1)
  *q = p;
  return;
}
```

```
define void @foo(i8** %q) {
  %p = alloca i8*
  store i8** %q, i8*** %2
  %31 = call i8* @malloc(i64 100)
  store i8* %1, i8** %p
  %2 = icmp ne i8** %q, null
  br i1 %2, label %3, label %4
; <label>:3
  br label %10
; <label>:4
  %5 = load i8** %p
  store i8* %5, i8** %q
  br label %6, !dbg !24
; <label>:6
  ret void, !dbg !25
}
```

Рис. 6.1. Точки возврата в Си и LLVM

С целью исправления ситуации были внесены изменения в используемый компилятор Clang. Для всех точек возврата добавлена генерация специальной функции *__sf_svace_return*, благодаря чему можно идентифицировать точки возврата из функции.

Похожая проблема была с операторами *break*, *continue* и метками *case* в инструкциях *switch*. Если перед этими конструкциями нет инструкций, то нет необходимости создавать пустой базовый блок, соответственно, нет отладочной

информации, указывающей место расположения этих операторов в исходном коде. Исправление компилятора также заключалось в добавлении специальной функции, описывающей недостающую информацию.

Часть условных проверок добавляется компилятором. Если добавленное компилятором условное выражение является лишним, то инструмент выдаст предупреждение, которое будет ложным. Например, семантика оператора *delete* языка Си++ допускает использование нулевого указателя в качестве аргумента. Компилятор Clang для оператора *delete* генерирует проверку аргумента на ноль и вызывает функцию $_ZdlPv$ для удаления указываемой памяти. Инструмент будет предполагать, что программист считает, что указатель может иметь нулевое значение. Для безусловного разыменования указателя будет выдано предупреждение, которое программист посчитает ложным. Для исправления ложных предупреждений все условные инструкции, которые не присутствуют в исходном коде, но были сгенерированы компилятором, помечаются специальным образом. Анализатор выдаёт предупреждения только для условных инструкций, которые не были помечены.

6.3. Вспомогательные алгоритмы

Описываемый анализ позволяет своё расширение для проведения вспомогательных анализов. Взаимодействие между вспомогательными анализами и детекторами производится посредством атрибутов. Полученные свойства, найденные дополнительными анализами, сохраняются в виде атрибутов, которые доступны для чтения всем остальным расширениям, в том числе и детекторам.

В разделе опишем наиболее важные вспомогательные анализы, которые были добавлены для улучшения качества детекторов.

6.3.1. Вызов неизвестных функций

Неизвестной будем называть функцию, у которой нет сопоставленного резюме. Резюме может отсутствовать в следующих случаях:

- исходный код функции недоступен анализатору;

- вызов функции производится по указателю, и анализатор не может выбрать резюме конкретной функции.

Функция может иметь множество побочных эффектов. Игнорирование этих эффектов приводит к ложным срабатываниям. В следующем фрагменте кода

```
char*p = 0;
init(&p);
p[10] = 0;
```

указатель на переменную p передаётся в функцию $init$. Функция может инициализировать значение переменной. Если у функции нет резюме, то анализатору недоступна информация о поведении функции. Игнорирование побочных эффектов функции приведёт к выдаче потенциально ложного срабатывания об разыменовании нулевого указателя.

Анализатор при обработке инструкции вызова функции для всех переменных, адрес которых был передан в неизвестную функцию, генерирует событие $unknown$ (раздел 4.3). Детектор может подписаться на это событие и изменить анализируемые свойства.

6.3.2. Анализ сохранённых значений

При анализе утечек памяти и ресурсов предупреждение выдаётся, если была выделена область памяти, которая не была освобождена в конце функции, и на которую указывают только локальные переменные. Значение считается *сохранённым*[1], если оно было присвоено глобальной переменной либо переменной, доступной вызывающей функции. Сложность анализа заключается в том, что сохранёнными надо помечать все значения, которые могут быть доступны через цепочки разыменований и смещений переменных с сохранёнными значениями.

Для анализа сохранённых значений используется атрибут $EscapeValue$, который устанавливается в следующих случаях:

- значение было присвоено глобальной переменной;

[1] escape

- значение было передано в неизвестную функцию, вызванную по указателю;
- значение было получено смещением другого значения, которое помечено атрибутом $EscapeValue$;
- значение было получено чтением памяти, на которую указывает указатель, помеченный атрибутом $EscapeValue$.

Значение переменной, переданной в неизвестную функцию, не помечается как сохранённое. Эта эвристика в некоторых случаях приводит к выдаче ложных срабатываний, но экспериментально было установлено, что такое поведение позволяет найти больше предупреждений при незначительном увеличении ложных срабатываний.

6.3.3. Анализ логических переменных

Анализ используется для фрагментов кода вида:

```
bool flag = (a != 0);
if(flag) {
    *a = 0;//(1)
}
```

Анализатор запоминает с помощью атрибута, что значение переменной $flag$ зависит от результата сравнения переменной a с нулём. Во время анализа инструкции сравнения ядро анализа использует записанный атрибут и генерирует событие $assume(a \neq 0)$. Благодаря чему в точке (1) доступна информация, что указатель a не равен нулю.

Такой анализ является примером тесного взаимодействия ядра анализа и расширений. Не только расширения зависят от ядра анализа, но и ядро анализа может использовать результаты, полученные расширениями.

6.4. Создание резюме

В разделе 5.2 описан алгоритм создания резюме, параметризованный количеством обходов состояния в точке возврата из функции. В инструменте используется следующая реализация этого алгоритма:

- выполняется 3 итерации обхода состояния от входных параметров;
- добавляются глобальные переменные;
- выполняется ещё 2 итерации обхода состояния от всех добавленных в резюме элементов.

Максимальное количество используемых в резюме элементов – 250. Такое соотношение было подобрано эмпирически. При меньшем количестве функции в резюме для больших функции могут не попасть элементы, от которых зависит выдача предупреждения, что ведёт к пропуску дефектов. Большее количество не используется с целью минимизации размеров резюме.

6.5. Оценка времени анализа

В данном разделе произведена оценка времени анализа с целью показать масштабируемость анализа для проектов в миллионы строк кода.

Анализатор Svace имеет множество режимов работы для различных ситуаций. По умолчанию инструмент оптимизирован для анализа огромных проектов размером в миллионы строк кода на машинах с большим количеством оперативной памяти (64-128 Гб). При этом инструмент использует настолько много памяти, насколько возможно, если при этом будет улучшение скорости анализа. Все резюме хранятся в оперативной памяти, не используется их сериализация.

При запуске Svace можно задать максимальное количество потребляемой памяти. Также можно включить сериализацию резюме и существенно снизить требования к памяти. Но время анализа при этом возрастёт.

Тестирование инструмента проводилось на 37 проектах с открытым исходным кодом. Проекты были условно разделены на три большие группы: Си-проекты, Си++-проекты и огромные проекты (ядро ОС Linux, ОС Tizen, ОС

Android). Для проектов средних размеров запуск анализа проводился на пользовательской машине с 12 Гб оперативной памяти, при этом использование памяти было ограничено 4 Гб опцией запуска Svace. Для огромных проектов тестирование проводилось на серверах с большим объёмом оперативной памяти (74-264 Гб). Табл. A.7 содержит описание всех используемых машин.

К Си-проектам было отнесено ПО, написанное преимущественно на языке Си. Табл. A.1 содержит названия проектов, размер их исходного кода, размер сгенерированного биткода LLVM и время анализа. Размер биткода приведён, т. к. скорость анализа во многом зависит от размера внутреннего представления, которое генерируется из биткода LLVM. Размер сгенерированного биткода не пропорционален размеру Си-файлов и зависит от используемых конструкций языка (особенно макросов).

Здесь и далее при приведении данных о размере исходного кода для определения количества строк кода использовалась утилита sloccount [72]. Подсчитывался только размер файлов, компиляция которых при сборке исходной программы была перехвачена Svace (некоторые временные файлы, например, генерируемые скриптами `configure` служебные файлы для определения свойств машины пользователя, не анализируются). Округление производилось до тысяч строк кода отбрасыванием дробной части.

Табл. A.2 содержит аналогичное описание для Си++-проектов. Для большинства Си- и Си++-проектов время анализа не превышает 30 минут. Общий размер исходного кода для Си-проектов составил более 5.5 миллионов строк кода, для Си++-проектов – более 2 миллионов строк кода. При этом для Си++-проектов было сгенерировано более чем в два раза больше биткода (2313 Мб против 909 Мб). Средняя скорость анализа составила 705 строк кода в секунду для Си-проектов и 246 строк для Си++-проектов.

В табл. A.5 приводится замер времени анализа для Си- и Си++-проектов на разных машинах.

Табл. A.3 содержит данные для используемых огромных проектов (ядро Linux 3.17, ОС Tizen 2.2, ОС Android 4.4, ОС Android 5.0.2, ОС Android 6.0). Размер каждого проекта превышает 5 миллионов строк кода и составляет 8.5 миллионов строк кода для Android 5.0.2. Результаты замера времени анализа

приведены в табл. A.4.

Как видно из результатов, анализ даже таких огромных проектов, как ОС Android 5.0.2, занимает менее 5 часов, т. е. анализ можно использовать во время ночной сборки. Средняя скорость анализа для огромных проектов ниже, чем скорость для анализа Си-проектов, но выше, чем скорость анализа Си++-проектов. На основе приведённых данных можно сделать вывод, что время анализа растёт линейно с ростом размера проекта. Во многом скорость анализа определяется соотношением Си и Си++-кода. Проект Android 5.0.2 содержит значительную часть Си++-кода, и его анализ производится медленнее, чем анализ Tizen 2.3, где доля Си++-кода ниже.

В табл. A.6 содержатся результаты замеров скорости анализа Tizen 2.3 при использовании разного количества потоков выполнения. Использование 16 потоков вместо двух ускоряет анализ почти в 4 раза (3,89). Использование 32 потоков вместо 16 даёт дополнительный прирост производительности в 7%. Анализ производился на сервере server4-32-264 с 16 ядрами процессора, реализующего технологию HyperThreading.

6.6. Оценка выданных предупреждений

В разделе приводится оценка количества выданных предупреждений для 32 проектов размером в сотни тысяч строк кода и 5 проектов размером в миллионы строк кода. Методика оценивания приведена в 6.6.1.

6.6.1. Методика

Для оценки количества истинных срабатываний использовалась следующая методика. Оценка производилась независимо по разным группам предупреждений.

В одну группу предупреждений включались срабатывания, выданные похожими детекторами. Для каждой группы производилась случайная выборка предупреждений. Применялся следующий алгоритм:

1. для каждого предупреждения вычислялся хэш на основе имени файла,

номера строки и типа предупреждения;

2. все предупреждения упорядочивались по хэшу;

3. выбирались первые N элементов из получившегося упорядоченного списка, где N – требуемый размер выборки. Если несколько предупреждений имеют одинаковый хэш, то все предупреждения включались в выборку;

4. все элементы выборки оценивались вручную на истинные, ложные и непонятные;

5. вычислялся процент истинных срабатываний как $\frac{T}{N}$, где T – количество истинных.

Описанная выше методика имеет следующие преимущества:

- не обязательно оценивать все предупреждения, достаточно оценить только часть;

- постоянный хэш для предупреждения удобно использовать, если анализатор меняется. В этом случае при повторном запуске изменённого инструмента большинство уже размеченных предупреждений снова попадёт в выборку. При сравнении двух версий инструмента общие предупреждения размечаются только один раз.

6.6.2. Результаты для ОС Tizen и ОС Android

Оценка результатов, выданных для операционной системы Tizen 2.3, показана в табл. Б.1. Оценка результатов, выданных для операционной системы Android 5.0.2, показана в табл. Б.2. Для всех предупреждений с количеством больше 50 оценивалась случайная выборка, содержащая минимум 50 предупреждений. Для большинства детекторов более 70% срабатываний являются истинными.

Можно выделить следующие основные причины ложных срабатываний:

- Ошибка находится в макросе. Просматривая тело макросов, сложно понять, в чём именно заключается ошибка. Инструмент использует имена

переменных в промежуточном представлении, по которым не всегда угадывается параметр макроса.

- Макрос, проверяющий некоторое условие и вызывающий функцию завершения программы, при сборке был раскрыт в пустую строку. В итоге ошибка есть в промежуточном представлении, но её нет в исходном коде.
- Ошибка связана с цепочкой присваиваний переменных, которым Svace сопоставляет один идентификатор значения. Т. к. трасса связана не с переменной, а с идентификатором значения, то в ней не отражены присваивания. Пользователю может быть непонятно, что две переменные имеют одно и тоже значение.
- Нетривиальные зависимости между функциями и методами класса. Анализатор не знает, что функции `vector::push_back` и `vector::size` связаны. После добавления элемента в вектор его размер увеличивается. Это может использоваться при написании кода.
- Недостаточная чувствительность к контексту. Внутри функции значения переменных могут зависеть друг от друга. Эту зависимость важно сохранить после трансляции резюме в контекст вызова.
- Отсутствие девиртуализации вызова Си++-функций.

Наиболее плохой результат у детекторов поиска утечек памяти (58%). Основная причина ложных срабатываний была связана с недостаточно точным моделированием ситуаций, когда функция возвращает структуру данных, одно из полей которой описывает код ошибки. Если значение поля равно некоторому выделенному значению, то внутри функции произошла ошибка, и память не была выделена.

6.6.3. Оценка результатов применения разработанных алгоритмов для ОС Android 4.4

Целью работы было увеличение количества выдаваемых предупреждений при сохранении высокого уровня истинных срабатываний и масштабируемости

при анализе сверхбольших проектов. Для оценки результатов работы было произведено сравнение предупреждений, выдаваемых текущей версией инструмента Svace (март 2016 года), с предупреждениями, выданными старой версией Svace (март 2015 года). Табл. Б.3 содержит результаты сравнения, выполненного на коде ОС Android 4.4, который анализировался также старой версией. Как видно из таблицы, количество выдаваемых предупреждений существенно возросло, при этом процент истинных срабатываний был сохранён, а в некоторых случаях улучшен. Следовательно, экспериментально подтверждается, что поставленная в работе цель достигнута.

Оценивался общий уровень истинных срабатываний по всем предупреждениям и средний уровень истинных для детекторов. Первая величина получена делением всех истинных предупреждений на общее количество предупреждений. Вторая величина является средним арифметическим уровня истинных предупреждений для каждого детектора.

Средний уровень истинных для детекторов улучшился с 63 до 65 процентов. При этом общий уровень истинных предупреждений ухудшился с 63 до 61 процента из-за того, что стало существенно больше выдаваться сообщений об утечках памяти с относительно низким уровнем истинных предупреждений (46%).

Основные причины ложных срабатываний были такие же, как и для проектов Tizen 2.3 и Android 5.0.2. Предупреждения OVERFLOW_AFTER_CHECK имеют низкие срабатывания на обеих версиях из-за использования макроса `ERREXIT1`, который вызывает функцию по указателю для завершения программы. Анализ подобных конструкций нетривиален в общем случае и потребует дополнительного обхода всех функций, т. к. инициализация указателя происходит до вызова функции.

6.6.4. Результаты

Было просмотрено более 1000 предупреждений для огромных проектов. Процент истинных срабатываний был от 46 до 100. Для большинства детекторов процент истинных срабатываний был выше 70. Количество выдаваемых предупреждений по сравнению со старой версией Svace для проекта ОС Android

4.4 увеличилось более чем в 4 раза.

Также были получены данные с оценкой качества предупреждений от инженеров компании Samsung для мобильной ОС Tizen 2.3 и для внутреннего проекта компании. В соответствии с этими данными, детекторы имеют процент истинных от 50% до 93%. Что подтверждает правильность используемой методики оценки.

Заключение

В данной работе были представлены алгоритмы анализа, которые позволили улучшить масштабируемость инструмента Svace до миллионов строк кода (сохранение высокого уровня истинных срабатываний при увеличении размера проектов) и увеличить количество найденных ошибок. Для этого понадобилось разработать внутрипроцедурные алгоритмы ядра, точнее моделирующие память, увеличить контекстную чувствительность межпроцедурного анализа, улучшить чувствительность к путям, доработать основные группы детекторов для поиска критических ошибок.

Основные результаты диссертационной работы:

- Разработан алгоритм внутрипроцедурного анализа функции, интегрирующий анализ указателей и нумерацию значений и обеспечивающий возможность поиска широкого класса дефектов. Доказана корректность разработанного алгоритма.

- Разработан алгоритм межпроцедурного контекстно- и потоково- чувствительного анализа функций, основанный на алгоритме создания резюме функции, которое обобщает результаты внутрипроцедурного анализа функции, и применения созданного резюме при обработке вызова функций.

- Разработанные алгоритмы реализованы в статическом анализаторе Svace для анализа программ на языках Си и Си++. Экспериментальные результаты анализа больших программных систем подтвердили масштабируемость реализованных алгоритмов при высоком качестве анализа (более 60% истинных срабатываний критических детекторов разыменования нулевых указателей, утечки ресурсов, переполнения буфера, и в разы возросшее количество предупреждений по сравнению с предыдущей версией анализатора).

В дальнейшей работе планируется уменьшать процент ложных срабатываний, а для этого, как показал анализ основных причин ложных срабатываний, необходимо:

- увеличить степень контекстной чувствительности;
- разработать вспомогательные алгоритмы, позволяющие находить взаимосвязи между переменными, в том числе межпроцедурно;
- разработать вспомогательный анализ зависимости возвращаемого значения функции от параметров;
- разработать алгоритм анализа функций, условно завершающих выполнение программы в зависимости от параметра;
- реализовать девиртуализацию вызова виртуальных функций в Си++.

Также планируется улучшить детализацию и понятность предупреждений и лучше показывать точки в программе, соответствующие присваиваниям переменных. В текущей версии отсутствует информация о точках присваивания переменных, что ведёт к ошибочной разметке истинных срабатываний как ложных при большом количестве присваиваний в функции.

Список сокращений и условных обозначений

$\wp(A)$	—	множество всех подмножеств множества A
$\cap$	—	пересечение множеств
$\cup$	—	объединение множеств
$dom(A)$	—	область определения функции A
$ran(A)$	—	область значений функции A
$\vee$	—	дизъюнкция
$\wedge$	—	конъюнкция
$\overline{A}$	—	отрицание
$F \upharpoonright S$	—	ограничение функции F для множества S: $F \upharpoonright S = \{< a, b > \mid F(a) = b, a \in S\}$
$\sqcup$	—	оператор объединения решётки
$\sqcap$	—	оператор сбора решётки
$f[x \mapsto y]$	—	$f'(a) = \begin{cases} y, & \text{если } a = x; \\ f(a), & \text{иначе} \end{cases}$
$f[x \mapsto y, z \mapsto w]$	—	$f[x \mapsto y][z \mapsto w]$
ПО	—	программное обеспечение
АСД	—	абстрактное синтаксическое дерево
АС	—	абстрактное состояние
ОС	—	операционная система

Введённые множества и функции:

C	—	множество констант 2.3
R	—	множество псевдорегистров 2.3
M	—	множество ячеек 2.3
$\nu_R : R \to C$	—	конкретное состояние для псевдорегистров 2.3
$\nu_M : M \to C$	—	конкретное состояние для ячеек памяти 2.3
$\nu : R \cup M \to C$	—	конкретное состояние 2.3
V	—	множество идентификаторов значений 3.2.2
JV	—	множество составных идентификаторов значений 3.2.2
SV	—	множество простых идентификаторов значений 3.2.2
A	—	множество ссылок 3.2.3
val_R	—	абстрактное состояние, значения для регистров 3.2.2
val_A	—	абстрактное состояние, значения для ссылок 3.2.3
$val : R \cup A \to V$	—	абстрактное состояние, значения 3.2.3
Val	—	множество функций значений val 3.2.3
$pt : V \to \wp(A)$	—	абстрактное состояние, отношения указывания 3.2.3
Pt	—	множество функций отношений указывания pt 3.2.3
$cache : \wp(V) \to V$	—	кэш значений 3.6.1
Ψ	—	множество абстрактных состояний 3.1, 3.5.1
Π	—	множество конкретных состояний 3.1
$Corr$	—	корректность абстрактного состояния 3.5.1
InV	—	множество входных параметров 5.1
RV	—	множество возвращаемых параметров 5.1

Список литературы

1. Аветисян А. И., Бородин А. Е. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок // Труды ИСП РАН. 2011. Т. 21. С. 39–54.
2. Аветисян А. И., Белеванцев А. А., Бородин А. Е., Несов В. С. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды ИСП РАН. 2011. Т. 21. С. 23–38.
3. Иванников В. П., Белеванцев А. А., Бородин А. Е. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды ИСП РАН. 2014. Т. 26, № 1. С. 231–250.
4. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code // Programming and Computer Software. 2014. Vol. 40, no. 5. P. 265–275.
5. Бородин А. Е. Статический поиск ошибок повторной блокировки семафора // Труды ИСП РАН. 2014. Т. 26, № 3. С. 103–112.
6. Бородин А. Е., Белеванцев А. А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Труды ИСП РАН. 2015. Т. 27, № 6. С. 111–134.
7. Borodin A. Summary Based Static Analysis for Practical Search for Defects in C Programs and Libraries // Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on / IEEE. Clevland: 2014. P. 231–232.
8. Бородин А. Е. Анализ на основе аннотаций для практического поиска дефектов в программах и библиотеках, написанных на языке Си // Сборник трудов XXI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2014». Москва: 2014.
9. Бородин А. Е. Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Открытая конференция по компиляторным технологиям. Москва: 2015.
10. Landi W. Undecidability of static analysis. // ACM Letters on Programming Languages and Systems. 1992. Vol. 1, no. 4. P. 323–337.

11. ISO. ISO/IEC 9899:2011 Information technology — Programming languages — C. Geneva, Switzerland: International Organization for Standardization, 2011. — December. P. 683 (est.). URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.
12. ISO. ISO/IEC 14882:2011 Information technology — Programming languages — C++. Geneva, Switzerland: International Organization for Standardization, 2012. — Feb. P. 1338 (est.). URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.
13. Younan Y., Joosen W., Piessens F. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. // Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven. 2004.
14. Xie Y., Engler D. Using redundancies to find errors // ACM SIGSOFT Software Engineering Notes. 2002. Vol. 27, no. 6. P. 51–60.
15. Ramalingam G. The Undecidability of Aliasing. // ACM Trans. Program. Lang. Syst. 1994. Vol. 16, no. 5. P. 1467–1471.
16. Steensgaard B. Points-to analysis in almost linear time // Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages / ACM. 1996. P. 32–41.
17. Shapiro M., Horwitz S. Fast and accurate flow-insensitive points-to analysis // Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM. 1997. P. 1–14.
18. Hind M. Pointer analysis: Haven't we solved this problem yet? // PASTE'01. ACM Press, 2001. P. 54–61.
19. Buss M., Brandb D., Sreedharc V., Edwardsa S. A. A novel analysis space for pointer analysis and its application for bug finding // Science of Computer Programming. 2010. Vol. 75, no. 11. P. 921–942.
20. Buss M., Brand D., Sreedhar V., Edwards S. A. Flexible pointer analysis using assign-fetch graphs // Proceedings of the 2008 ACM symposium on Applied computing. – ACM. 12008. P. 234–239.
21. Buss M. Summary-Based Pointer Analysis Framework for Modular Bug Finding: Ph. D. thesis / Columbia University. 2007.
22. Cousot P., Cousot R., Feret J. et al. The ASTREE analyzer // Programming

Languages and Systems. – Springer Berlin Heidelberg. 2005. P. 21–30.

23. Cousot P., Cousot R., Feret J. et al. Varieties of static analyzers: A comparison with ASTREE // Sixth International Symposium on Theoretical Aspects of Software Engineering. 2007. no. 3. P. 3–20.
24. Deutsch A. Static Verification of Dynamic Properties // ACM SIGAda 2003 Conference. 2003.
25. Cousot P., Cousot R., Feret J. et al. Why does Astree scale up // Formal Methods in System Design. 2009. Vol. 35, no. 3. P. 229–264.
26. King J. C. Symbolic execution and program testing // Communications of the ACM. 1976. Vol. 19, no. 7. P. 385–394.
27. Bush W. R., Pincus J. D., Sielaff D. J. A static analyzer for finding dynamic programming errors // Software-Practice and Experience. 2000. Vol. 30, no. 7. P. 775–802.
28. Dilling I., Dilling T., Aiken A. Static error detection using semantic inconsistency inference // ACM SIGPLAN Notices. 2007. Vol. 42, no. 6. P. 435–445.
29. Xie Y., Aiken A. Scalable error detection using boolean satisfiability // ACM SIGPLAN Notices. 2005. Vol. 40, no. 1. P. 351–363.
30. Xie Y., Aiken A. Context-and path-sensitive memory leak detection // ACM SIGPLAN Notices. 2005. Vol. 30, no. 5. P. 115–125.
31. Aiken A., Bugrara S., Dillig I. et al. An overview of the Saturn project // Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. 2007. P. 43–48.
32. Aiken A., Bugrara S., Dillig I. et al. Precise and Scalable Software Analysis. 2006. URL: http://saturn.stanford.edu.
33. Xie Y. Static detection of software errors: Ph. D. thesis / Stanford University. 2006.
34. Babic D., Hu A. J. Calysto // Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. – IEEE. 2008. P. 211–220.
35. Babić D. Exploiting Structure for Scalable Software Verification: Ph. D. thesis / University of British Columbia, Vancouver, Canada. 2008.
36. Xie Y., Chou A., Engler D. Archer: using symbolic, path-sensitive analysis to detect memory access errors // ACM SIGSOFT Software Engineering Notes.

2003. Vol. 28, no. 5. P. 327–336.

37. Livshits V. B., Lam M. S. Tracking pointers with path and context sensitivity for bug detection in C programs // ACM SIGSOFT Software Engineering Notes. 2003. Vol. 28, no. 5. P. 317–326.
38. Avots D., Dalton M., Livshits V. B., Lam M. S. Improving software security with a C pointer analysis. 2005. P. 332–341.
39. Cytron R., Ferrante J., Rosen B. K. et al. Efficiently computing static single assignment form and the control dependence graph // ACM Transactions on Programming Languages and Systems (TOPLAS). 1991. Vol. 13, no. 4. P. 451–490.
40. Bessey A., Block K., Chelf B. et al. A few billion lines of code later: using static analysis to find bugs in the real world // Communications of the ACM. 2010. Vol. 53, no. 2. P. 66–75.
41. Coverity. Coverity Prevent: a static code analysis tool for C, C++, C# and Java. URL: http://coverity.com.
42. Coverity. Coverity Scan: 2013 Open Source Report. URL: http://softwareintegrity.coverity.com/rs/coverity/images/2013-Coverity-Scan-Report.pdf.
43. Coverity. Coverity Scan: 2012 Open Source Report. URL: http://softwareintegrity.coverity.com/rs/coverity/images/2012-Coverity-Scan-Report.pdf.
44. Almossawi A., Lim K., Sinh T. Analysis tool evaluation: Coverity prevent // Pittsburgh, PA: Carnegie Mellon University. 2006.
45. Klocwork. Klocwork: the set of static code analysis tools. URL: http://www.klocwork.com/.
46. Emanuelsson P., Nilsson U. A comparative study of industrial static analysis tools // Electronic notes in theoretical computer science. 2008. Vol. 217. P. 5–21.
47. Маликов О. Р., Несов В. С. Автоматический поиск уязвимостей в больших программах. // Известия ТРТУ, Тематический выпуск «Информационная безопасность». 2006. № 7. С. 114–120.
48. Nesov V. Automatically Finding Bugs in Open Source Programs. // Electronic Communications of the EASST. 2009. Vol. 20.
49. Аветисян А. И. Современные методы статического и динамического анализа

программ для автоматизации процессов повышения качества программного обеспечения: Докторская диссертация / ИСП РАН. 2012.

50. Игнатьев В. Н. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // Труды ИСП РАН. 2012. Т. 22. С. 169–188.
51. Игнатьев В. Н. Формализация ограничений языков C и C++ для их проверки методами статического анализа // Пятый сборник трудов молодых ученых и сотрудников кафедры Вычислительной Техники ИТМО / Под ред. Т. И. Алиева. 2014. С. 17–20.
52. Кошелев В. К., Дудина И., Игнатьев В., Борзилов А. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя // Труды ИСП РАН. 2015. Т. 27, № 5. С. 59–86.
53. Clang: a production quality C, Objective-C, C++ and Objective-C++ compiler. URL: http://clang.llvm.org.
54. The Java programming language Compiler Group. URL: http://openjdk.java.net/groups/compiler.
55. The .NET Compiler Platform (R̈oslyn)̈ provides open-source C# and Visual Basic compilers with rich code analysis APIs. URL: https://github.com/dotnet/roslyn.
56. Eclipse. URL: https://eclipse.org.
57. The LLVM Compiler Infrastructure. URL: http://llvm.org.
58. Cytron R., Gershbein R. Efficient accommodation of may-alias information in SSA form // ACM SIGPLAN Notices. 1993. Vol. 28, no. 6. P. 36–45.
59. Hardekopf B., Lin C. Semi-sparse flow-sensitive pointer analysis // ACM SIGPLAN Notices. – ACM. 2009. Vol. 44, no. 1. P. 226–238.
60. Naur P., Backus J. W., Bauer F. L. et al. Revised report on the algorithmic language ALGOL 60 // Communications of the ACM. 1963. Vol. 6, no. 1.
61. Plotkin G. D. A Structural Approach to Operational Semantics: Tech. Rep. DAIMI FN-19. University of Aarhus: 1981.
62. Nielson H. R., Nielson F. Semantics with Applications: A Formal Introduction. New York, NY, USA: John Wiley & Sons, Inc., 1992.
63. Muchnick S. S. Advanced Compiler Design and Implementation. Morgan Kauf-

mann, 1997. Vol. 1. P. 217–267. ISBN: 9781558603202.

64. Click C. Global code motion/global value numbering // ACM SIGPLAN Notices. 1995. Vol. 30. P. 246–257.
65. Alpern B., Wegman M. N., Zadeck F. K. Detecting equality of variables in programs // Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM. 1988. P. 1–11.
66. Briggs P., Cooper K. D., Simpson L. T. Value numbering // Software-Practice and Experience. 1997. Vol. 27, no. 6. P. 701–724.
67. Sharir M. A strong-connectivity algorithm and its applications in data flow analysis // Computers & Mathematics with Applications. 1981. Vol. 7, no. 1. P. 62–72.
68. Ахо А., Лам М., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструментарий. 2 изд. Вильямс, 2008.
69. Маликов О. Р. Исследование и разработка методики автоматического обнаружения уязвимостей в исходном коде программ на языке Си: Кандидатская диссертация / ИСП РАН. 2006.
70. Davey B. A., Priestley H. A. Introduction to lattices and order. Cambridge university press, 2002. ISBN: 0521784514.
71. Clang Static Analyzer: a source code analysis tool that finds bugs in C, C++, and Objective-C programs. http://clang-analyzer.llvm.org. URL: http://clang-analyzer.llvm.org.
72. SLOCCount. URL: http://www.dwheeler.com/sloccount.

Приложение А

Оценка скорости анализа

Таблица А.1. Замер времени анализа и описание проектов, написанных преимущественно на языке Си.

Проект	**Строк кода, тыс.**		**Размер bc-файлов, Мб**	**Время анализа**
	C	**C++**		
binutils-2.22	1325	80	321	34м10с
busybox-1.18.5	195	1	22	10м32с
cairo-1.12.14	234	3	40	3м32с
dnprogs-2.58ubuntu1	27	10	5	2м44с
glib2.0-2.32.4	362	0	77	13м41с
gnupg-1.4.11	117	0	14	3м42с
gst-plugins-bad0.10	353	22	42	7м23с
gst-plugins-base0.10	214	0	30	4м28с
gst-plugins-good0.10	257	0	43	12м43с
gstreamer0.10-ffmpeg	405	0	3	1м20с
gtk+2_0-2_24_10	572	0	90	6м22с
libxml2-2.7.8.dfsg	128	0	30	7м41с
openssl-1.0.1	273	4	51	9м7с
nss-3.17.4	478	26	40	6м43с
pulseaudio-1.1	188	0	2	32с
xorg-server-1.12.3	377	1	99	8м53с
всего	5505	147	909	2ч13м33с

Таблица А.2. Замер времени анализа и описание проектов, написанных преимущественно на языке Си++

Проект	**Строк кода, тыс.**		**Размер bc-файлов, Мб**	**Время анализа**
	C	**C++**		
fwbuilder-5.0.0	4	380	244	1м54с
glob2-0.9.4.4	0	89	151	19м28с
groff-1.21	10	73	9	11м15с
kde-runtime-4.8.5	2	143	273	4м27с
mm3d-1.3.7	0	92	208	5м58с
muse-2.0 rc2	1	163	334	15м32с
pgadmin3-1.14.0	32	214	145	7м51с
qtiplot-0.9.8.8	5	156	394	19м51с
simutrans-111.0	3	100	62	6м44с
speech-tools-2.1	12	134	53	17м15с
tesseract-3.02.01	1	109	128	11м50с
tinymux-2.6.5.28	0	87	7	3м51с
worker-2.18.1	0	96	159	7м25с
xapian-core-1.2.8	8	100	123	4м34с
xpdf-3.02	0	92	2	51с
yate-2.2.0-1 dfsg	14	122	21	14м4с
всего	94	2150	2313	2ч32м57с

Таблица А.3. Описание больших используемых проектов

Проект	**Строк кода, тыс.**		**Размер LLVM-файлов, Мб**
	C	**C++**	
Android 4.4	2421	3069	8459
Android 5.0.2	3137	5363	20943
Android 6.0.0	2449	3598	18273
Linux 3.17	9088	0	3910
Tizen 2.3	4874	1732	10464

Таблица А.4. Производительность для больших проектов

Проект	Сервер	Время анализа	Пиковое потребление памяти, Гб	Скорость анализа, строк кода/с
Linux 3.17	server3-32-132	1ч.9м.	44	1917
Tizen 2.3	server1-16-74	2ч.54м.	36	632
	server2-16-198	2ч.48м.	55	655
	server3-32-132	1ч.53м.	43	974
	server4-32-264	1ч.52м.	70	983
Android 4.4	server1-16-74	3ч.24м.	37	448
	server2-16-198	3ч.7м.	59	489
	server3-32-132	2ч.15м.	42	677
	server4-32-264	2ч.0м.	69	760
Android 5.0.2	server4-32-264	4ч.55м.	86	480
Android 6.0.0	server4-32-264	4ч.47м.	79	354

Таблица А.5. Замер времени анализа Си и Си++ проектов

Проект	сервер	Время анализа	Скорость анализа, строк кода/с
Си проекты	machine1	2ч.13м.	705
	server1-16-74	1ч.45м.	897
	server2-16-198	1ч.41м.	932
	server3-32-132	1ч.25м.	1108
	server4-32-264	1ч.15м.	1256
Си++ проекты	machine1	2ч.32м.	246
	server1-16-74	2ч.5м.	299
	server2-16-198	1ч.59м.	314
	server3-32-132	1ч.41м.	370
	server4-32-264	1ч.30м.	415

Таблица А.6. Запуск анализа Tizen 2.3 с разным количеством используемых потоков

Количество потоков	Время анализа, мин.
2	467
4	257
8	164
16	120
32	112

Таблица А.7. Описание используемых машин

Название	Оперативной памяти, Мб	Процессор	Размер кэша, Кб	Количество ядер
machine1	12268	Intel(R) Core(TM) i7 CPU 8602.80GHz	8192	8
server1-16-74	74226	Intel(R) Xeon(R) CPUE55302.40GHz	8192	16
server2-16-198	198088	Intel(R) Xeon(R) CPU E55202.27GHz	8192	16
server3-32-132	131998	Intel(R) Xeon(R) CPU E5-2680 02.70GHz	20480	32
server4-32-264	264123	Intel(R) Xeon(R) CPU E5-2650 02.00GHz	20480	32

Приложение Б

Оценка выданных предупреждений

Таблица Б.1. Результаты для Tizen 2.3

Дефект	Всего	Истинных, %
Утечки памяти	1410	56
Утечки ресурсов	213	62
Разыменование нулевых указателей	390	70
Разыменование после сравнения с нулём	1353	68
Сравнение с нулём после разыменования	337	94
Разыменование без проверки результата malloc	1532	100
Некорректное освобождение памяти (new/free, malloc/delete, new[]/delete)	13	100
Использование tainted-целых	113	90
Доступ к массиву с неправильной проверкой индекса	163	70
Неконсистентность конструкторов и деструкторов, приводящая к утечке	13	53

Таблица Б.2. Результаты для Android 5.0.2

Дефект	Всего	Истинных, %
Утечки памяти	1183	48
Утечки ресурсов	274	74
Разыменование нулевых указателей	901	76
Разыменование после сравнения с нулём	1480	68
Сравнение с нулём после разыменования	278	80
Разыменование без проверки результата malloc	704	96
Использование tainted-целых	179	74
Доступ к массиву с неправильной проверкой индекса	185	80
Некорректное освобождение памяти (new/free, malloc/delete, new[]/delete)	54	68
Неконсистентность конструкторов и деструкторов, приводящая к утечке	112	56

Таблица Б.3. Оценка предупреждений для Android 4.4 для двух версий Svace

Дефект	svace март 2015		svace март 2016	
	Всего	Истинных, %	Всего	Истинных, %
Утечки памяти	328	26	1662	46
Утечки ресурсов	93	71	234	87
Разыменование нулевых указателей	143	48	858	54
Разыменование после сравнения с нулём	90	51	955	62
Разыменование без проверки результата malloc	495	97	605	94
Сравнение с нулём после разыменования	64	81	251	74
Некорректное освобождение памяти (new/free, malloc/delete, new[]/delete)	7	85	30	80
Использование памяти после освобождения	14	42	126	44
Ошибка использования memcpy для буферов	8	62	18	50
Использование потенциально отрицательного значения	79	68	449	76
Отсутствие va_end после va_start	23	91	25	96
Использование tainted-целых	90	81	143	80
Использование tainted-строк	27	81	47	87
Использование неинициализированных переменных	31	80	818	58
Использование указателя на переменную как массив	16	43	251	52
Проверка индекса после доступа к массиву	3	100	17	70
Доступ к массиву после проверки индекса	77	30	149	38
Доступ к массиву с неправильной проверкой индекса	71	42	249	67
Удаление памяти не из кучи	10	10	5	20
Неконсистентность конструкторов и деструкторов, приводящая к утечке	28	75	196	74
Среднее	84	63	354	65
Общее	1697	63	7088	61