**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ НАУКИ
ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ
ИМ. В.П. ИВАННИКОВА
РОССИЙСКОЙ АКАДЕМИИ НАУК**

На правах рукописи

Белеванцев Андрей Андреевич

МНОГОУРОВНЕВЫЙ СТАТИЧЕСКИЙ АНАЛИЗ ИСХОДНОГО КОДА ДЛЯ ОБЕСПЕЧЕНИЯ КАЧЕСТВА ПРОГРАММ

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

Диссертация на соискание ученой степени
доктора физико-математических наук

Москва – 2017

Оглавление

Qu’on ne dise pas que je n’ai rien dit de nouveau, la disposition des matières est nouvelle. Quand on joue à la paume c’est une même balle dont joue l’un et l’autre, mais l’un la place mieux.

J’aimerais autant qu’on me dît que je me suis servi des mots anciens. Et comme si les mêmes pensées ne formaient pas un autre corps de discours par une disposition différente, aussi bien que les mêmes mots forment d’autres pensées par leur différente disposition[1].

Les Pensées de Blaise Pascal. Chap. XXIX - *Pensées morales*: 1678 n° 19 p. 275.

[1] «Пусть не корят меня за то, что я не сказал ничего нового: ново само расположение материала; игроки в мяч бьют по одному и тому же мячу, но с неодинаковой меткостью.
С тем же успехом меня можно корить за то, что я употребляю давным-давно придуманные слова. Стоит по-иному расположить одни и те же мысли – и получается новое сочинение, равно как, если по-иному расположить одни и те же слова, получится новая мысль.»

Цит. по: Блез Паскаль. Мысли. СПб. Азбука-Классика, 2000. Пер. с фр. Э. Линецкой.

ВВЕДЕНИЕ

В современном мире связь между ошибками в программах и качеством программ не нуждается в доказательстве – ошибки влияют на надежность выполнения программ, их производительность и безопасность. Если пятьдесят лет назад ошибки искались вручную или с помощью предупреждений компилятора, то сейчас применяется множество дополняющих методов поиска – статический и динамический анализ, фаззинг, верификация моделей, тестирование на проникновение и др. Разнообразие методов возросло соразмерно сложности задачи – современные программные системы содержат десятки миллионы строк кода, дистрибутивы ОС – сотни миллионов. При этом распространение сетевых и облачных технологий увеличивает цену ошибки, так как велика вероятность превращения ошибки в уязвимость безопасности, через которую можно получить неавторизованный доступ к системе. Повсеместное использование открытого программного обеспечения (ПО) многократно тиражирует ошибки – так, единственный дефект в коде OpenSSL, известный как HeartBleed, привел к уязвимости полумиллиона сайтов.

Созданы стандарты разработки безопасного программного обеспечения, например, Microsoft Security Development Lifecycle и ГОСТ Р 56939-2016, описывающие способы применения инструментов поиска ошибок в жизненном цикле ПО. Все они нацелены на то, чтобы в ходе разработки и внедрения ПО как можно раньше найти возможно большее количество ошибок.

Одним из распространенных подходов к поиску ошибок является статический анализ исходного кода программы, позволяющий проверить все пути выполнения программы и найти ошибки на редко выполняющихся путях, для которых сложно составить тесты либо выявить их динамическим анализом. Для промышленного применения статический анализатор должен обладать рядом свойств, важнейшими из которых являются: способность находить часто встречающиеся виды ошибок, обработка кода на промышленных языках программирования, подробное объяснение сути найденных ошибок, тесная интеграция в процесс разработки. Среди нефункциональных требований к анализаторам можно отметить масштабируемость на уровне выполнения анализа десятков миллионов строк кода за несколько часов, высокая точность (малое количество ложных срабатываний, полностью от которых невозможно избавиться из-за ограничений технологии), расширяемость анализатора для поиска новых типов ошибок.

За последние 10-15 лет требования к статическим анализаторам постоянно расширяются, и для их удовлетворения необходимо привлекать новые подходы. Спектр используемых подходов весьма широк – от внутрипроцедурного анализа потока управления и данных до межпроцедурного анализа на основе аннотаций функций, чувствительного к путям выполнения анализа, символьного выполнения и определения выполнимости формул в теориях (SMT-

решателям). Технологии статического анализа активно разрабатываются коммерческими компаниями, в результате чего создан ряд инструментов, подходящих под указанные требования (анализаторы Coverity Prevent, Klocwork K11, HP Fortify), однако привлекаемые ими модели программы и алгоритмы анализа закрыты, их подробное описание не опубликовано.

Общеизвестно, что качественный анализ требует применения набора межпроцедурных алгоритмов анализа, обеспечивающих контекстную чувствительность и чувствительность к путям выполнения, которые при этом для достижения нужной масштабируемости должны выполнять нестрогий анализ, пропуская возможные ошибки. Однако этим дело не исчерпывается. Классы ошибок, которые требуется искать, настолько разнообразны, что обойтись единственной моделью программы и возможно точными алгоритмами анализа ее свойств на основе этой модели невозможно. Для таких ошибок, как, например, нарушения правил безопасного кодирования, неверное использование интерфейсов стандартных библиотек, требуется анализ абстрактного синтаксического дерева (АСД) и максимально детальная информация об исходном коде программы. Существуют и примеры ошибок, занимающие промежуточное положение между анализом уровня АСД и чувствительным к путям анализом. В работах многих ученых, таких как Д. Энглер, П. Кузо и Р. Кузо, А. Айкен, Т. Кременек, Ф. Логоццо, П. Годефруа, Т. Диллиг, Ф. Иванчич и другие, исследуются различные аспекты искомых моделей программы и алгоритмов анализа, однако организация единого набора методов анализа всех нужных уровней не предложена.

Актуальной научной проблемой, на решение которой направлена данная работа, является задача разработки методологии статического анализа для поиска дефектов в исходном коде программ на современных императивных языках, а также составляющих эту методологию наборов методов анализа, алгоритмов поиска дефектов и программных средств.

Для решения этой проблемы представляется необходимым работать по следующим направлениям. Во-первых, требуется разработать модели программы, пригодные для популярных императивных языков программирования (Си, Си++, Java, C#), которые включают в себя модель памяти программы, единую для различных уровней анализа и настраиваемую для учета особенностей различных языков. Разработанные модели должны давать возможность вычислять необходимую информацию о значениях переменных программы с линейной масштабируемостью. Кроме того, нужно предложить и математически обосновать алгоритмы анализа для построения моделей на следующих уровнях анализа: внутрипроцедурном анализе, межпроцедурном контекстно-чувствительном анализе всей программы, чувствительном к путям анализе. При этом вычисленная на предыдущих уровнях информация должна быть доступна для использования на последующих уровнях.

Во-вторых, на основе созданных моделей требуется разработать алгоритмы поиска десятков классов ошибок (детекторы) – ошибок кодирования, неверного использования стандартных интерфейсов, критических ошибок и т.д., доставляющие высокое качество анализа. При этом для сложных типов ошибок из-за требуемой масштабируемости и нестрогости алгоритмов статического анализа может понадобиться несколько различных детекторов (до десятка), отвечающих за поиск разного рода "ситуаций" в модели программы, которые указывают на наличие ошибки.

Наконец, нужно разработать программные средства, обеспечивающие работу предложенных методов анализа и алгоритмов поиска ошибок в промышленном окружении для сверхбольших программ. Для этого сначала требуется разработать архитектуру системы анализа, которая полностью поддерживает весь процесс анализа от построения необходимых внутренних представлений и проведения анализов всех уровней до показа выданных предупреждений анализатора пользователю, обеспечивая при этом полностью автоматический анализ, понятный графический интерфейс, возможность использования в системах непрерывной интеграции (CI) при рецензировании кода. Потребуются следующие компоненты системы анализа: полностью автоматический (прозрачный для пользователя) сбор нужной информации о программе с помощью мониторинга сборки исходной программы, хранилище собранных данных, которое переносимо на другие машины для организации удаленного анализа, быстрый повторный анализ только лишь изменившейся части программы (инкрементальный анализ). Система показа выданных предупреждений для анализатора, постоянно применяющегося на этапе разработки, должна предусматривать хранение результатов периодических запусков анализа, перенос пользовательской разметки между запусками, скрытие предупреждений, однажды помеченных как ложные.

После создания архитектуры системы анализа нужно разработать и реализовать программную систему, которая управляет работой набора анализаторов, обеспечивая возможность единообразно просматривать найденные ими ошибки для разных языков программирования, а также позволяет подключать новые анализаторы, конфигурировать ход анализа. Дело в том, что на практике анализаторы младших уровней поддерживают только один язык или семейство языков, и для покрытия ряда требуемых языков нужно комбинировать несколько анализаторов. Анализаторы, использующие глубокий межпроцедурный чувствительный к путям выполнения анализ, добиваются масштабируемости эвристическими алгоритмами, вносящими нестрогость в ход анализа. Как следствие, из-за разницы в применяемых эвристиках выдаваемые такими анализаторами ошибки для одной и той же программы пересекаются незначительно – на 20-30% (если разработчики анализатора не тратят на это усилий). Поэтому возможность работы с набором анализаторов является не прихотью, а необходимостью, возникающей в реальном промышленном окружении.

Объектом исследования являются инструменты статического анализа программного обеспечения на языках программирования Си, Си++, Java, C#. **Предметом исследования** являются методы статического анализа исходного кода программ, в том числе методы межпроцедурного анализа, методы чувствительного к путям выполнения анализа, а также модели программы и модели памяти, предназначенные для статического анализа.

Цель и задачи работы. Создание методологии проведения статического анализа исходного кода программ для поиска ошибок в программах, состоящей: в разработке и реализации набора моделей программы и вычисляющих эти модели методов статического анализа; в разработке алгоритмов поиска ошибок (детекторов) на основе предложенных моделей; в разработке архитектуры программных средств, обеспечивающих совместную работу этих методов и алгоритмов для программ в десятки миллионов строк кода на популярных языках программирования (Си, Си++, Java, C#) и высокий процент истинных срабатываний анализатора (не менее 60%).

Для достижения поставленной цели необходимо решить следующие задачи:

- Разработка моделей программы, модели памяти и соответствующих алгоритмов анализа, позволяющих выполнять поиск ошибок, для которого требуются различные уровни анализа и представления программы (анализ на уровне АСД и внутрипроцедурный анализ, межпроцедурный анализ, чувствительный к путям выполнения анализ);
- Разработка алгоритмов поиска популярных типов ошибок (детекторов) на основе предложенных моделей и методов анализа;
- Разработка архитектуры системы анализа, поддерживающей весь ход анализа с использованием предложенных алгоритмов и детекторов, а также обеспечивающей единообразную работу с набором анализаторов разных уровней;
- Реализация разработанных моделей и алгоритмов, системы анализа для промышленных языков программирования Си, Си++, Java, C#.

Методы исследования. Для решения поставленных задач использовались методы теории множеств, теории графов, теории решеток, абстрактной интерпретации, теории компиляции, в том числе анализа потока данных, символьного выполнения.

Научная новизна. В диссертации получены следующие новые результаты, которые **выносятся на защиту**:

- Методология проведения статического анализа исходного кода программ для поиска ошибок в программах, заключающаяся в проведении многоуровневого статического анализа с помощью набора моделей программы и методов анализа с общей моделью памяти на уровнях анализа АСД, внутрипроцедурного анализа, межпроцедурного

контекстно-чувствительного анализа, чувствительного к путям выполнения анализа. Предложенные модели и алгоритмы математически обоснованы, имеют линейную масштабируемость и пригодны для популярных императивных языков программирования, а также позволяют переиспользовать вычисленную информацию с предыдущих уровней анализа на следующих уровнях;

- Алгоритмы поиска конкретных ошибок в программе (детекторы) на основе предложенных методов, которые выполняют поиск популярных классов ошибок: ошибок кодирования, неверного использования стандартных интерфейсов, критических ошибок (разыменование нулевого указателя, переполнение буфера, ошибки управления памятью и ресурсами, использование неинициализированных переменных, ошибки многопоточных примитивов, недостижимый код и др.). Детекторы позволяют искать заданный тип ошибки на разных уровнях анализа и не выдавать ошибку на последующих уровнях, если она уже была найдена на предыдущих;
- Архитектура программной системы, обеспечивающая автоматическую работу всех предложенных методов на протяжении всего процесса анализа, а также управление набором анализаторов для различных языков и единообразный показ их результатов. Разработанные компоненты системы анализа включают: автоматическое построение внутренних представлений для анализа на основе прозрачной для пользователя контролируемой сборки; единое переносимое хранилище собранной для анализа информации и результатов анализа, обеспечивающее запуск анализа на любой машине; подсистема просмотра и разметки результатов анализа, которая обеспечивает перенос выполненной пользователем разметки между результатами анализа программы; инкрементальный анализ только лишь изменившейся части программы.

Теоретическая и практическая значимость. Теоретическая значимость заключается в разработанной методологии выполнения статического анализа исходного кода, состоящей из набора моделей и методов анализа, алгоритмов поиска ошибок, архитектуры системы анализа, которые пригодны в целом для сверхбольших программ на современных императивных языках и доставляют необходимое качество анализа.

Практическая значимость работы определяется тем, что на базе разработанных методов в ИСП РАН реализовано программное средство Svace, включающее в себя пять анализаторов разных уровней для Си, Си++, Java и C# и демонстрирующее требуемые от промышленных анализаторов характеристики масштабируемости и качества анализа. Анализатор Svace внедрен в цикл промышленной разработки компании Samsung Electronics с 2015 года, а также используется в НИЦ «Курчатовский институт». Разработанное средство Svace может

применяться в жизненном цикле разработки безопасного ПО согласно ГОСТ Р 56939-2016, несмотря на отсутствие в настоящий момент методической документации, регламентирующей требования к статическим анализаторам по этому ГОСТ, и использоваться как модельный инструмент анализа для разработки безопасного ПО, реализующий все необходимые методы анализа.

Автором опубликовано более 40 научных печатных трудов по теории компиляции и анализу программного кода, а также получено 9 свидетельств [11-19] о регистрации программ для ЭВМ в Федеральной службе по интеллектуальной собственности, патентам и товарным знакам. В том числе по материалам диссертации опубликовано 12 работ, из них 10 статей [1-10] опубликовано в изданиях, входящих в список изданий, рекомендованных ВАК РФ, 4 статьи [2, 5, 6, 9] опубликованы в изданиях, индексируемых Web of Science. Основные результаты диссертационной работы обсуждались на конференциях и семинарах различного уровня, в том числе 4 доклада на международных конференциях и 5 на всероссийских. Работа по теме диссертации проводилась в соответствии с планами исследований по проектам, поддержанными грантом РФФИ, контрактами в рамках Программы фундаментальных исследований Президиума РАН «Фундаментальные проблемы системного программирования», а также договорами с компанией Samsung Electronics.

Личный вклад. Выносимые на защиту результаты получены соискателем лично. В опубликованных совместных работах постановка и исследование задач осуществлялись совместными усилиями соавторов под руководством и при непосредственном участии соискателя. Статья [9] полностью принадлежит автору. В статье [1] автором написаны разделы 1-3, 5, 6, 11, в статье [3] – разделы 1-3. Статья [2] содержит созданное автором общее описание инструмента Svace и постановку задачи. В статье [4] автору принадлежат разделы 1, 2, 5; в статье [5] – постановка задачи, общее описание анализатора, исследования по межпроцедурному анализу (разделы 5-6). В статье [6] автор руководил разработкой межпроцедурного анализа для поиска ошибок переполнения буфера и выполнил общую постановку задачи. В статье [7] автор написал разделы 1-4 и участвовал в разработке системы контролируемой сборки. Статья [8] содержит написанные автором разделы 1-2 и 5. В статье [10] автор руководил разработкой обеих инфраструктур анализа помеченных данных.

Диссертация состоит из введения, пяти глав, заключения и списка литературы. Общий объем диссертации составляет 229 страниц. Диссертация содержит 39 рисунков и 17 таблиц. Список литературы содержит 196 наименований.

Первая глава представляет собой обзор современного состояния методов статического анализа всех используемых в работе уровней, известных анализаторов, а также опыта применения инструментов анализа в промышленности.

Вторая глава посвящена предлагаемой методологии многоуровневого статического анализа исходного кода. Она содержит методы анализа уровня АСД, предлагаемую модель памяти программы и алгоритмы ее построения, модель программы для межпроцедурного контекстно-чувствительного анализа и алгоритмы ее построения, алгоритмы чувствительного к путям анализа в варианте, расширяющем предыдущий уровень межпроцедурного анализа, а также непосредственно использующие методы символьного выполнения.

В третьей главе описана архитектура предлагаемой системы анализа, реализованная в семействе инструментов Svace, которая обеспечивает все этапы работы анализатора, и её соответствующие компоненты: подсистема контролируемой сборки программы; разработанные на основе открытых систем компиляторы для создания представления программы для анализа; основная фаза анализа с возможностью параллельной работы, а также инкрементального анализа; хранение и просмотр результатов анализа.

Четвертая глава заключает в себе описание детекторов ошибок, разработанных на основе предложенных в главе 2 алгоритмов и реализованных в анализаторах семейства Svace. Рассмотренные детекторы принадлежат всем уровням анализа (от АСД до чувствительного к путям) и предлагают методы поиска ошибок для Си/Си++, Java и C#.

Пятая глава содержит экспериментальные результаты применения инструмента Svace к промышленному исходному коду, которые показывают необходимую масштабируемость инструмента (6 часов для анализа ОС Android из 12 млн. строк кода, 15 часов для анализа 27 млн. строк кода дистрибутива ОС Tizen) и качество анализа (60-80% истинных срабатываний).

В заключении формулируются основные результаты диссертационной работы и предлагаются направления дальнейших исследований.

1. СОВРЕМЕННЫЕ МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА ПРОГРАММ

Статический анализ программ появляется одновременно с первыми оптимизирующими компиляторами как необходимая часть обеспечения корректности выполняемых трансформаций кода [23, гл. 1], а его применение для поиска ошибок в коде традиционно датируется концом 1970-х с появлением инструмента lint [24] в операционной системе Unix V7. С тех пор появились десятки статических анализаторов, посвященных обнаружению ошибок, с разными подходами и уровнями технологии, для различных языков программирования и всевозможных типов ошибок. Исчерпывающий обзор этих инструментов и подходов был бы весьма амбициозным исследованием, и его создание выходит за заданные рамки настоящей работы. Целью данной главы является описание наиболее релевантных методов анализа, которое дается на фоне панорамы общего состояния дел в этой области, что, можно надеяться, позволит читателю получить вполне четкое представление об актуальных исследованиях. Кроме того, выделяются вопросы о том, чего ждут пользователи от статического анализатора, каков опыт внедрения таких анализаторов в крупных компаниях, и какие прорывные направления анализа, не связанные с традиционными подходами, наиболее перспективны.

Изложение материала организовано следующим образом. Разделы 1.1, 1.2 и 1.3 посвящены методам и инструментам, соответствующим выделенным в главе 2 уровням анализа – анализу на абстрактном синтаксическом дереве и методам внутрипроцедурного потока данных, межпроцедурному анализу, и чувствительному к путям анализу соответственно. Раздел 1.4 посвящен вопросу о том, что, собственно, является предметом поиска анализатора – как определить, имея в своем арсенале методы предыдущих подразделов, что в коде наличествует ошибка определенного типа. Из-за принципиальных ограничений на точность статического анализа этот вопрос отнюдь не очевиден и, например, искать ошибки переполнения буфера "в лоб" по непосредственному определению в сколь-нибудь реальной программе окажется невозможным.

Наличие ложных срабатываний у статического анализатора проявляет себя и по-другому. Возникает желание максимально сократить затраты времени программиста, который должен просмотреть список генерируемых анализатором предупреждений и оценить их истинность. Этого можно добиваться, пытаясь сортировать выдачу инструмента так, чтобы первыми шли те предупреждения, вероятность истинности которых максимальна. С другой стороны, для некоторых типов ошибок сам анализатор может предлагать исправления и, после проверки программистом, применять их к программе автоматически. Таким подходам посвящен раздел 1.5.

Очень важным вопросом является обратная связь от пользователей инструмента. Это воистину проблема "курицы и яйца" – невозможно построить промышленный анализатор без учета мнения пользователей, которое можно получить только при реальном внедрении, но само

это внедрение непросто организовать, если статический анализатор нельзя применять промышленно. Тем ценнее редкие публикации, проливающие свет на реальные проблемы, возникающие при внедрении, от таких компаний, как Google и Microsoft, и от крупнейшего производителя анализатора Coverity. Обзор таких публикаций и выводы из него представлены в разделе 1.6.

Можно заметить, что все методы разделов 1.1-1.3 так или иначе выросли из компиляторных технологий. Однако существуют и новые подходы, которые нельзя отнести к одному из выделенных нами уровней анализа либо их комбинации. Они еще не доказали свою применимость в промышленной обстановке, но являются перспективным заделом на будущее и, вполне возможно, представляют из себя основу области через 5-10 лет. Это новые способы моделирования памяти программы, применение методов больших данных, представление программ в виде графовых баз данных, метасистемы анализа. Обзору наиболее интересных систем такого рода посвящен раздел 1.7.

Прежде всего перечислим кратко основные из используемых далее терминов. *Статический анализатор* исходного кода программы использует в своей работе алгоритмы, не требующие запуска программы и, соответственно, подготовки для нее входных данных. Анализатор, ищущий ошибки в программе, в результате работы предъявляет список мест в исходном коде, в которых найдены ошибки, с указанием типа ошибки, сообщения о ее характере и, возможно, некоторой дополнительной информации о том, при каких условиях (входных данных, выполнении конкретных переходов и т.п.) ошибка может произойти.

По поводу того, что называть ошибкой, в литературе нет единого мнения. Обсудим один из возможных используемых вариантов. Обычно под *ошибкой* понимают некоторое место (оператор, конструкцию) исходного кода программы, из-за которого на определенных входных данных программа может аварийно завершиться либо вывести некорректные выходные данные. Тем самым наличие ошибки подразумевает возможность "доказать" ее, предъявив эти входные данные[2]. *Дефектом* называют место, указывающее на недостаток исходного кода, который не обязательно приведет к некорректной работе программы, но может ухудшить ее эксплуатационные характеристики (например, утечка памяти). *Уязвимостью* называют ошибку в программе, которая может быть использована атакующим для намеренного краха программы, выполнения произвольного кода, утечки конфиденциальных данных либо других нарушений безопасности системы.

При реальном использовании статического анализатора такая формальная классификация немедленно доставляет сложности. Во-первых, часто сложно бывает сказать, является ли

[2] Может потребоваться еще и соответствующим образом настроенное окружение, например, для ошибок многопоточности, которые часто могут проявиться только в специальных условиях.

предупреждение анализатора ошибкой или дефектом, т.е. возможно ли найти те входные данные, на которых ошибка проявится. В сложной программе даже ее авторы могут затрудниться с ответом на этот вопрос, а задача автоматического подтверждения предупреждений статического анализатора, например, с помощью динамического анализа в настоящий момент отнюдь не решена. Во-вторых, в программе может иметься дефект, по стечению обстоятельств не приводящий к ошибке, например, использование операции `sizeof(pointer)` вместо `sizeof(*pointer)`, когда размеры соответствующих типов совпадают (про такой детектор см. подробнее в разделе 4.1.1.1). Однако это не значит, что указанный дефект не надо исправлять – он может превратиться в ошибку на другой аппаратной платформе. В-третьих, еще сложнее установить, будет ли являться найденная ошибка уязвимостью, даже если входные данные, на которых она проявляется, уже предъявлены.

Как видим, с точки зрения статического анализатора границы между ошибкой, дефектом и уязвимостью достаточно условны. Все это – некоторые *ошибочные ситуации* в программе, то есть участки, дающие возможность предположить о нарушении семантики, неточности или неконсистентности модели программы, построенной анализатором. Мы будем рассматривать постановку задачи анализа, в которой для заданного типа ошибочных ситуаций и заданной программы анализатор пытается найти все ошибочные ситуации, которые есть в этой программе (то есть пропустить как можно меньше ситуаций, или false negatives), при этом по возможности не делая ложных предупреждений о не существующих в программе ошибочных ситуациях (false positives).

Заметим, что для пользователя ложным срабатыванием обычно является такое предупреждение, которое не является ошибкой, даже если оно является истинной ошибочной ситуацией в модели анализатора. В настоящее время вместо термина "истинное срабатывание" может применяться термин "требующее исправления срабатывание" (actionable issue), чтобы учесть случаи вроде упомянутого выше, когда предупреждение является дефектом, но не ошибкой, и в любом случае указанное место в исходном коде программы требует исправления.

Возможны и другие постановки, например, доказать отсутствие в программе определенного класса ошибочных ситуаций (верификация), либо гарантированно не пропустить ни одной потенциальной ошибочной ситуации (инструменты, помогающие в проведении аудита кода), либо выдать предупреждения только об истинных ошибочных ситуациях. Эти постановки не рассматриваются в данной работе по причине их ограниченного применения на практике.

В связи с вышесказанным, далее мы будем использовать термины "ошибка" и "дефект" как синонимы, понимая под ними ошибочную ситуацию в модели анализатора, и трактуя истинные и ложные срабатывания по отношению к этой же модели. При необходимости

описать ошибку в приведенном значении ситуации, реализуемой на некоторых входных данных, мы будем оговаривать это отдельно.

Наконец, важно отметить, что как критерий качества анализатора обычно выступает процент истинных предупреждений в его выдаче. Процент пропущенных ошибок можно оценить только на заранее подготовленных тестах, в которые вручную внесены известные исследователю ошибки, либо (косвенно) в сравнении с другим статическим анализатором, результаты которого для данной программы известны и просмотрены экспертами. Оба метода имеют свои недостатки – успешно найденная ошибка в небольшом тесте не означает, что ровно такая же ошибка будет найдена в большой программной системе. При сравнении же результатов с другим анализатором нужно всегда учитывать, что из-за эвристических решений и алгоритмов, используемых в том или ином виде во всех популярных промышленных анализаторах [25], пропущенную одним инструментом ситуацию может быть нелегко найти другим даже существенными усилиями авторов.

В Интернете ведется несколько списков статических анализаторов, хотя никакой из них не является исчерпывающим. Список Маттиаса Эндлера [26] и список Википедии [28], по-видимому, пытаются поддерживать перечень актуальных анализаторов, тогда как список Национального института стандартов и технологий (NIST) [27] концентрируется именно на анализаторах уровня исходного кода (не байт-кода и не бинарного кода), которые могут применяться для поиска потенциальных уязвимостей, и компания-разработчик анализатора может подать заявку на включение в него. Из этих списков наиболее широким по покрытию языков программирования и релевантным, как кажется, является первый, хотя только список NIST упоминает недавно появившийся промышленный анализатор компании CheckMarx [31].

Анализаторы, которые более не актуальны, но когда-то являлись важной вехой на пути развития технологии, не содержатся ни в одном из списков, хотя, например, анализатор Prefix [29], впервые реализовавший идею межпроцедурного чувствительного к путям выполнения на основе аннотаций, опять-таки есть только в списке NIST. Анализатора Saturn [30], одним из первых применившего идеи символьного выполнения в статическом анализе, нет ни в одном из списков.

1.1. Подходы к анализу уровня абстрактного синтаксического дерева

Анализаторы, использовавшие обходы абстрактного синтаксического дерева и внутрипроцедурный анализ потока данных в качестве единственного уровня анализа, разрабатывались в большом количестве 15-20 лет назад, и практически все уже неактуальны. Уже и тогда они предназначались в основном для поддержки ручного аудита безопасности, например, как анализатор RATS [32], потому что в своем выводе помечали все подозрительные

места в коде, генерируя множество ложных срабатываний. Алгоритмов отсеивания таких срабатываний на основе потока данных программы не было, в основном, пользователю давалась возможность пропустить предупреждения в системных заголовочных файлах, в заданных файлах, либо написать спецификации (по типу прагм компилятора) в исходном коде, которые указывали анализатору, что в данном месте программы предупреждение определенного типа выдавать не следует. Такой возможностью, например, обладал один из лучших анализаторов этого типа в то время – открытый анализатор Splint [33]. В настоящее время все еще развивается анализатор Flexelint [34], который его авторы считают старейшим из до сих пор существующих на рынке анализаторов [35] и самым популярным статическим анализатором на планете, хотя последнее утверждение внушает известные сомнения.

Далее опишем современные анализаторы уровня АСД для языков Си, Си++, Java и C#, а после сосредоточимся на подходах, использующих единое внутреннее представление для анализа различных языков, а также методах написания детекторов уровня АСД с помощью специализированных языков запросов. Для Си и Си++ одним из популярных анализаторов является инструмент Clang Static Analyzer (CSA) [38], основанный на компиляторе Clang [37] инфраструктуры LLVM [36]. Детекторы в анализаторе CSA пишутся в виде шаблона "посетитель" (visitor), в котором реализуются собственные обработчики интересующих узлов АСД, а инфраструктура анализатора организует обход АСД и вызывает зарегистрированные обработчики всех детекторов. Написание детекторов облегчает богатое синтаксическое дерево компилятора Clang, которое можно использовать как библиотеку. Кроме того, синхронизация с Clang обеспечивает поддержку новейших стандартов языков Си и Си++ (правда, необходимо заметить, что соответствующие обходчики в анализатор добавляются с некоторым опозданием).

Помимо уровня АСД, анализатор CSA позволяет запустить символьное выполнение отдельных функций или всех функций заданного модуля, строя "расширенный" (exploded) граф потока управления, в котором "раскручиваются" все пути выполнения на заданную глубину, а память моделируется символьно, позволяя представить все особенности языка Си [39]. Это интересный и актуальный пример применения символьного выполнения в открытом статическом анализаторе, однако его межпроцедурность ограничивается рамками одного модуля трансляции, и поэтому в посвященной символьному выполнению разделу обзора мы не будем его упоминать. Впрочем, известны работы по добавлению аннотации к межпроцедурному анализу в инструменте CSA и по организации межмодульного анализа [49].

Для языка Java, несомненно, самым популярным анализатором уровня АСД является инструмент FindBugs [40]. В отличие от схожих анализаторов для других языков, FindBugs анализирует не непосредственно АСД Java, а байткод Java, для чего пользуется известной библиотекой разбора и модификации байткода ASM [47]. Этот подход не позволяет

реализовать детекторы, для которых нужна информация об исходном коде, уже потерянная в байткоде (например, детектор неконсистентного форматирования кода из раздела 4.1.1.7), однако подавляющее количество детекторов можно реализовать и на байткоде. FindBugs сосредоточен на детекторах ошибок, для которых достаточно анализов внутрипроцедурного потока данных и управления, а также межпроцедурной контекстно-нечувствительной информации, при этом уровень ложных срабатываний должен оставаться небольшим. Детекторы реализуются непосредственно на языке Java, а для подавления нежелательных предупреждений пользователь может создавать собственные аннотации в стандартном синтаксисе Java. FindBugs является удобной открытой инфраструктурой для создания легковесных АСД-детекторов для Java, и в качестве такового был внедрен в процесс разработки компании Google [41], а также стал частью универсальных статических анализаторов, таких, как Coverity Prevent или Svace.

Для языка C# существовало некоторое количество инструментов, обеспечивающих в основном стилевые проверки кода. Создание анализаторов упростилось после открытия компанией Microsoft доступа к исходным кодам проекта Roslyn [43] в 2015 году, представляющем из себя промышленный компилятор языка C# и интерфейсы работы с АСД программы на C#. Из популярных АСД-анализаторов на основе Roslyn можно назвать инструмент SonarLint [42], а также компоненту анализа кода утилиты ReSharper [44]. Проверки более серьезных предупреждений, таких, как возможное обращение к нулевому указателю, также выполняются, например, в ReSharper, однако они основаны на собственных аннотациях методов стандартной библиотеки.

Обобщая сказанное, можно отметить некоторые общие черты у популярных промышленных АСД-анализаторов для распространенных языков программирования. Во-первых, все они основаны на открытых компиляторных инфраструктурах для соответствующих языков (в случае FindBugs – на промышленной библиотеке разбора байткода). Тем самым разработчики инструментов сконцентрированы на поддержке детекторов ошибок, а не на разборе программ на языке. Из этого правила есть и исключения, например, инструмент CppCheck [48], содержащий АСД-детекторы для Си и Си++, основан на собственном синтаксическом дереве. Однако его разработка началась более 10 лет назад, и, кроме того, можно заметить, что ресурсы его авторов во многом тратятся на поддержку новых стандартов языков – так, в списке изменений нескольких из последних релизов (1.78, 1.79) можно заметить работы по улучшению поддержки Си++11, стандарта, вышедшего 6 лет назад.

Во-вторых, детекторы ошибок в описанных инструментах реализованы на универсальных языках программирования[3]. Разумеется, в рамках анализаторов реализованы интерфейсы анализа и инфраструктуры, упрощающие написание детекторов, однако не сделано попытки построить некоторый специализированный язык для этой цели. Как представляется, от таких попыток отказались из-за значительного разнообразия ошибок, которые нужно проверять на уровне АСД, в связи с чем их формализация не выходит проще, чем непосредственная запись на языке программирования с использованием интерфейсов доступа к АСД (к схожему выводу пришли и в [50, гл. 1]).

Наконец, отметим, что в Российской Федерации разработаны несколько инструментов статического анализа АСД-уровня – из известных назовем AppChecker [46], использующий обходы АСД и сигнатурный поиск, и PVS-Studio [45], предоставляющий также и некоторый анализ потока данных. Эти инструменты обладают полезным набором детекторов некритических ошибок и имеют удобный пользовательский интерфейс, что не может, однако, заменить отсутствие высших уровней анализа для поиска критических ошибок на всей программе и способности обрабатывать большие программы с приемлемым качеством, а также другие нефункциональные требования к промышленным анализаторам.

Инструменты, использующие метаструктуры данных для объединения разнородных синтаксических деревьев нескольких языков, не столь распространены – в этом случае сложно воспользоваться уже имеющимися парсерами промышленных трансляторов, а собственные разборщики кода не дают нужного качества. Одно из важных применений такого приема в построении инструментов – рефакторинг кода, написанного на нескольких языках, обычно одного основного языка и нескольких специализированных. Например, исследования Майера и Шредера [54, 55, 56] посвящены построению единой системы, позволяющей выполнять исследование и рефакторинг кода в экосистеме Java, написанного с применением инфраструктур и языков типа Hibernate, Spring, HTML. Семантическая модель программы в их системе, реализованной как плагин среды Eclipse, поддерживает подключение различных парсеров, которые выполняют разбор кода и обнаруживают *артефакты* (различные типы узлов АСД), которые связаны между собой в АСД различных языков (например, Java и Spring). Статический анализ используется только для выявления этих связей и последующего построения транзитивного замыкания множества узлов, которое необходимо подвергнуть изменению для автоматического выполнения рефакторинга, но не для поиска каких-либо дефектов.

[3] Интересно, что для всех описанных анализаторов язык реализации детекторов совпадает с проверяемым языком, возможно, потому, что разработчики, для которых небезразлично качество программ на каком-либо языке, чаще всего программируют на нем же.

Интегрированная среда X-Develop [58] предлагается в виде прототипной реализации идеи метамодели программы, написанной на нескольких языках, и объединяющей языково-специфичные АСД. Общая часть модели изначально содержит только синтаксическую информацию, а семантические проверки выполняет парсер каждого языка отдельно и сохраняет результаты в метамодели. Далее общее АСД может использоваться для выполнения операций рефакторинга над всей программой. По утверждениям авторов, X-Develop содержит поддержку языков C#, Java, Visual Basic, ASP, XML, JavaScript и успешно применялся к программам среднего размера (несколько сот тысяч строк кода).

Среда TexMo [57] ставит себе целью поддержать удобную разработку многоязыковых программ. Для этого универсальный АСД содержит только очень общие лексические элементы, а отношения между строками описываются отдельными связями вида "ключ-значения". В прототипе среды модель этих связей строится вручную, в отличие от подхода [54]. Среда не производит никакого существенного анализа, а лишь помогает отслеживать связи между элементами программы, в том числе на разных языках.

Использование обобщенных АСД собственно для анализа кода выполняется в инструменте DMS [52], предназначенном для написания спецификаций и их реализаций на разных языках программирования, а также поддержания связей между ними и выполнения трансформаций кода. Основной структурой данных для представления разнородных АСД являются гиперграфы, позволяющие проводить множество разнотипных связей между вершинами графа. Предоставляется программный интерфейс для манипуляции графом и обхода его вершин посетителем, а также способ написания правил переписывания узлов графа. Система DMS реализована на специализированном параллельном языке Parlanse и была успешно применена к ряду коммерческих приложений на Java и C++, в которых был выполнен поиск клонов кода (а для Си – и их удаления), упрощения директив препроцессора, генерации кода для микроконтроллеров. Инструмент Bauhaus [51] для анализа и обратной инженерии программ также представляет АСД поддерживаемых языков в виде графа, узлы которого формируют единую иерархию классов. Разработчики реализовали базовые консервативные анализы потока данных и управления поверх этого графа, включая анализ указателей, а также выполнили поиск взаимных блокировок и мертвого кода. Зубов [53] предлагает использовать представление на основе языка XML, содержащее основную информацию о классах программы и связях между ними, и получает это представление из парсеров соответствующих языков. В прототипе системы XML-представление генерируется для языков Java и Python и используется для анализа иерархии классов.

Пожалуй, самой развитой инфраструктурой с поддержкой нескольких языков является компилятор ROSE [60] для анализа и преобразований кода типа source-to-source. ROSE содержит общее АСД для языков Си, Си++ и ФОРТРАН, получаемое промышленными

парсерами EDG и Open Fortran Parser. На основе этого АСД построен инструмент Compass [59], реализующий множество детекторов для проверки качества кодирования и идиом безопасности. Проверка правил реализуется на языке Си++ в виде обходов соответствующих узлов АСД. Кроме того, инструменты поддерживают анализ бинарных файлов путем дизассемблирования и генерации того же объединенного АСД.

Запросы к уже построенному абстрактному синтаксическому дереву выполняются обычно двумя способами. Первым способом является написание кода в паттерне "посетитель" на соответствующем языке программирования, на котором реализован инструмент анализа. Такие библиотеки часто реализуются на языке JavaScript, например, для парсера Babylon [69] и Acorn самого JavaScript [70]. Другим способом является построение запросов на декларативных языках, обычно вариантах XPath или XQuery, например, язык запросов ASTq [68], язык запросов в виде XQuery в системе PMD [65], язык запросов на основе REST в PuppetDB [67]. Неполный список таких языков приведен в [66]. Наиболее мощным, как представляется, является язык KAST [62], расширяющий XQuery, который интегрирован в промышленный анализатор Klocwork и используется для значительного количества детекторов уровня АСД. KAST был также расширен для описания простых трансформаций синтаксического дерева [61]. Несколько особняком стоит язык PQL [64], похожий по структуре на SQL, но оперирующий понятиями из семантической модели программы, и язык легковесного анализатора Cobra [63], похожий на простой скриптовый язык, запросы на котором выполняются на потоке токенов программы. В последнем случае невозможно написать сколько-нибудь сложные АСД-детекторы, т.к. структура потока токенов очень простая и специально создана для быстрой обработки.

Основным ограничением всех упомянутых языков является то, что всегда найдутся правила, детекторы для которых неудобно выражать на языке запросов из-за громоздкости понятий, которые необходимо будет в него ввести. В этом случае разумным представляется сосуществование языка запросов для простых детекторов и универсального языка программирования для сложных случаев. Кроме того, язык запросов полезен в случае, когда пользователям анализатора нужно предоставить возможность создавать собственные детекторы, что имеет место в случае языка KAST.

1.2. Подходы к межпроцедурному анализу

Первую общую теорию межпроцедурного анализа потока данных сформулировали Шарир и Пнюэли в 1981 году в книге [71], введя термины для двух подходов к межпроцедурному анализу: первый подход они назвали "функциональным", в котором функции рассматриваются как супероперации, эффект которых на остальную программу

необходимо вычислить (и здесь же впервые появляется аналог аннотации функции), а второй – подходом "строк вызовов", который формирует глобальный граф потока управления из всех функций и использует строки, кодирующие последовательность вызовов, для различения контекстов вызова и учета только тех путей в графе, которые реализуемы (то есть вызовы и возвраты из функций соответствуют друг другу). В дальнейшем функциональный подход, вычисляющий аннотации функции, верные для любых контекстов вызова, стали называть анализом "снизу-вверх", тогда как алгоритм, переанализирующий вызываемую функцию для каждого нового контекста либо группы контекстов вызовов и начинающий работу от точек входа в программу, называют анализом "сверху-вниз". Исследования, работающие над межпроцедурными алгоритмами анализа, шли в основном по этим двум направлениям.

Отдельной постановкой задачи межпроцедурного анализа явилось сведение широкого класса задач анализа потока данных, от которых требовалась конечность множества значений и дистрибутивность передаточных функций, к задаче достижимости на расширенном графе (так называемой IFDS-задаче) [72]. В расширенном графе вершины графа потока управления размножались количество раз, соответствующее мощности множества фактов потока данных, а дуги проводились соответственно графам потока управления анализируемых функций и значениям передаточных функций. Был предъявлен алгоритм решения задачи достижимости на основе динамического программирования, позволяющий решить межпроцедурно поставленную задачу за время, кубическое от количества фактов и линейное от количества ребер в межпроцедурном графе потока управления. Этот алгоритм получил популярность, в частности, для поиска ошибок на основе анализа помеченных данных, как, например, в системе FlowDroid [81] для анализа Android-приложений, экспериментальном анализаторе ИСП РАН на основе биткода LLVM [80], а также в одной из компонент анализатора C# инструмента Svace (см. раздел 3.3.6).

Анализаторы, предназначенные для поиска ошибок и ставящие себе целью масштабирование до программ как минимум в сотни тысяч строк исходного кода, в основном выбирали подход анализа "снизу-вверх", по причине того, что это позволяет анализировать каждую функцию только один раз, так как вычисленная для нее аннотация верна для любого контекста вызова. Кроме того, такой подход распараллеливается более естественно, чем анализ "сверху-вниз", который может посетить функцию несколько раз и либо использовать существующую аннотацию, либо создать новую. Одним из первых таких инструментов являлся анализатор PREfix [74], также начавший выполнять чувствительный к путям анализ с помощью символьного выполнения. Анализатор симулировал пути выполнения функции по отдельности, ограничивая сверху максимальное количество посещаемых путей, после чего для создания аннотации собранные данные о значениях переменных, ошибках и т.п. объединяются с учетом условий на контекст вызова, при которых данная часть аннотации будет справедлива.

Интересно, что экспериментальные результаты авторов показали, что после просмотра 100-200 путей количество найденных ошибок практически не растет, однако пути очень не равноценны – малая доля путей содержит значительную часть ошибок, тем самым выбор путей для отсечки становится важным для точности анализа.

Тем не менее, предлагались и инструменты анализа на основе подхода "сверху-вниз", например, анализатор xgcc на основе специализированного языка metal для написания детекторов [73]. Metal предлагал способ описывать автоматы и их состояния для программиста-неспециалиста в компиляторах, но который является экспертом в собственной программной системе и знает, какие инварианты необходимо проверять и какие состояния системы являются ошибочными. Анализатор также интерпретировал пути в графе потока управления поиском в глубину и использовал кэширование в базовых блоках в случае, если блок снова посещался с тем же состоянием – использовалась запомненная информация о том, как блок повлиял на переходы между состояниями автомата. Межпроцедурный анализ был организован также с помощью кэширования – стартовав от точек входа в программу, анализатор кэшировал вычисленное влияние функции на переходы между состояниями в виде аннотации функции, и если функция посещалась повторно с аналогичным контекстом вызова, то ее анализ не выполнялся, а информация о переходах бралась из аннотации.

В развитии техник межпроцедурного анализа с использованием аннотаций можно выделить три направления. Во-первых, в аннотациях стараются различным способом представить условия наступления интересующих анализатор событий или фактов. Например, в работе [77] упомянутая теория Шарира для конечных решеток фактов потока данных расширена для бесконечных решеток построением наиболее слабых предусловий на входные переменные, которые гарантируют выполнение некоторых общих постусловий на выходное состояние функции. В работе [75] предлагается межпроцедурный анализ для вычисления модели динамической памяти программы, для чего аннотация функции представляет из себя граф указательных отношений между ячейками памяти, дуги которого аннотированы условиями, при которых данное отношение выполняется. Такая структура данных позволяет выполнять сильные обновления значений в ячейках памяти при совместности соответствующих условий из аннотации и текущих условий из контекста вызова, что, по мнению авторов, критическим образом влияет на точность анализа. Построением аннотаций через символьное выполнение, вычисляющее пред- и постусловия в тройках Хоара для описания модели динамической памяти в разделяющей логике (separation logic) занимаются авторы в [76].

Во-вторых, исследователями комбинируются подходы к межпроцедурному анализу, заключающиеся в обходах графа вызовов программы "сверху-вниз" и "снизу-вверх". Так, в работе [78] предлагается выполнять несколько анализов функции при обходе "сверху-вниз",

после чего, если функция генерирует слишком много различных аннотаций, выполнять ее анализ методом "снизу-вверх" и строить соответствующую аннотацию, верную для всех контекстов вызова. В этой аннотации сохраняется только некоторое количество условий, которые участвуют в абстрактных состояниях функции, наиболее часто встречавшихся при анализе "сверху-вниз". Таким образом, при дальнейшем анализе в случае удачного выбора сохраненных в аннотации "снизу-вверх" условий эта аннотация сможет повторно использоваться для большого количества контекстов вызова, что, по данным авторов, позволяет ускорить комбинированный анализ в разы по сравнению с чистым анализом, обходящим граф только в одну из сторон.

Наконец, разрабатываются так называемые "разреженные" (sparse) анализы, целью которых является хранение только необходимой в данной точке части абстрактного состояния программы и ее передача непосредственно от определений переменных к использованиям (по def-use цепочкам). Примером инфраструктуры для построения таких анализов по имеющимся классическим анализам, сформулированным в терминах абстрактной интерпретации, является работа [79]. Результаты авторов показывают, что разреженные анализы, выполненные в их парадигме, позволяют достичь на порядок большей масштабируемости, чем исходные анализы.

Нужно отметить, что, несмотря на упомянутые исследования, промышленные анализаторы в настоящий момент по-прежнему следуют схеме выполнения межпроцедурного анализа с однократным посещением функций в основной фазе анализа при обходе "снизу-вверх" и созданием аннотации для всех контекстов вызовов, параметризуемой входными параметрами и глобальной памятью функции.

1.3. Чувствительность к путям и SMT-решатели

Уже к концу 1990-х с развитием статических анализаторов стало понятно, что одним из основных источников ложных срабатываний является учет анализом нереализуемых путей в программе – например, места возникновения и проявления ошибки выполняются при несовместных условиях, следовательно, ошибка является ложной и не должна выдаваться. *Чувствительный к путям* анализ, то есть анализ, вычисляющий различные данные о программе для разных путей выполнения или групп путей выполнения, начал входить в требования при разработке новых анализаторов. Одним из первых таких инструментов стал уже упоминавшийся PREfix, авторы которых выдвинули список требований, актуальных для поиска дефектов и сейчас: применимость к реальным программам на Си и Си++ (а, значит, способность обрабатывать массивы, структурные типы, объединения, адресную арифметику, битовые операции, приведения типов и т.п.); отсутствие требований пользовательских спецификаций (могут применяться для моделирования неизвестных анализу функций, но для

функций с исходным кодом анализатор обязан строить необходимые модели самостоятельно); учет потока данных только с выполнимых путей (то есть чувствительный к путям анализ); вычисление информации о программе не только с целью поиска ошибок, но и для того, чтобы как можно лучше объяснить пользователю причину возникновения ошибки. Чувствительность к путям в PREfix выражается в симуляции отдельных путей выполнения в функции и в сохранении в аннотации функции условий, при которых могут возникнуть ошибки либо произойти определенные факты во время выполнения функции.

В последующих инструментах можно выделить несколько способов построения чувствительного к путям анализа. Первым способом является отслеживание предикатов – условий, при которых происходят интересующие анализ события – и последующее отсеивание событий, условия возникновения которых несовместны с условиями их использования (как правило, использование заключается в определении ошибочной ситуации либо в применении аннотации функции, и соответствующим условием является необходимое условие достижимости рассматриваемой точки программы). Предикатные домены использовались еще в работах по абстрактной интерпретации (например, в [82]). Вторым способом является применение символьного выполнения и SMT-решателей, на чем мы остановимся подробно ниже. Третьей группой методов являются подходы к статической верификации программ, основными из которых являются ограничиваемая проверка моделей (BMC) и уточнение абстракции по контрпримерам (CEGAR). Эти методы также используют в своей работе SMT-решатели и предикатные абстракции [83]. В текущем виде их производительности и ограничений на анализируемое окружение недостаточно для поиска ошибок в больших программных системах, поэтому мы остановимся на них кратко.

Символьное выполнение заключается в построении искомой модели программы (результатов выполнения функций, значений выходных переменных, памяти) в виде формул над символьными значениями переменных и символьными выражениями (в противоположность конкретным значениям). В ходе символьного выполнения поддерживается символьное состояние, отображающее переменные и память на соответствующие символьные значения или выражения, и символьный предикат пути, являющейся логической формулой над символьными выражениями. В классическом случае каждое разветвление потока управления порождает два символьных выполнения с соответствующими предикатами пути, поэтому одной из основных проблем является взрывной рост количества путей, которые необходимо обойти. В зависимости от конечной цели применения символьного выполнения по окончании выполнения, падении программы или нарушения некоторого предиката ошибки с помощью SMT-решателя можно сгенерировать входные данные либо их часть, на которых произошла интересующая ситуация.

Большая часть исследований посвящена динамическому символьному выполнению, целью которого является создать тестовые входные данные для найденной ошибочной ситуации. Недавние обзоры и состояние дел в этой области можно найти в [84] и [86]. Библиография исследований по символьному выполнению, к сожалению, редко пополняющаяся после 2013 года, ведется в [85]. Из известных динамических инструментов можно назвать KLEE [87] и S2E [89]. KLEE проводит символьное выполнение биткода LLVM, моделируя память с точностью до бита, и реализует ряд оптимизаций по кэшированию запросов к SMT-решателю, эвристикам выбора путей для символьного выполнения. Например, из-за значительного пересечения предикатов пути по условиям переходов кэширование запросов позволяет быстро проверить, что запрос с добавленными предикатами по сравнению с кэшированным, возможно, все еще может быть решен кэшированным ответом, а для запроса с убранными предикатами всегда подходит кэшированный ответ как решение более узкого запроса. KLEE, как кажется, является самым популярным открытым инструментом, на базе которого ведется множество исследований, например, система UC-KLEE [94], позволяющая проверять программные изменения (патчи) на внесение дополнительных ошибок, а также реализовывать общие детекторы типа поиска неинициализированных переменных или утечек памяти поверх инфраструктуры символьного выполнения.

Система S2E производит смешанное выполнение, переключаясь между символьным и конкретным выполнением по мере необходимости (техника, известная как concolic testing, от concrete + symbolic): например, если для некоторого детектора ошибок нет необходимости символьно выполнять код некоторой библиотеки, то инфраструктура инструмента обеспечит переход на конкретное выполнение на входе в библиотеку и обратное переключение на символьное на выходе из нее. При этом поддерживается полносистемное символьное выполнение, то есть возможно выполнять анализ и кода операционной системы. Это достигается комбинацией динамической бинарной трансляции на основе полносистемного эмулятора QEMU и символьного выполнения на базе KLEE, причем каждый инструмент был модифицирован для одновременной поддержки символьного и конкретного состояния программы (например, записи в память и чтения из памяти в QEMU также передаются в KLEE). Система S2E является чрезвычайно перспективной для построения анализаторов, выборочно применяющих символьное выполнение, однако является непростой в работе и модификации.

Переходя к статическим анализаторам, одним из широко известных открытых проектов являлся анализатор Saturn [90]. В нем объединены идеи, позволяющие получить масштабируемый и качественный статический анализатор, а именно: используется символьное выполнение с объединением состояний (что позволяет избежать проблемы взрывного роста путей за счет, во-первых, огрубления формул при объединении и, во-вторых, перекладывании

части работы на SMT-решатель); межпроцедурный анализ делается на основе аннотаций функций; разрывается рекурсия в графе вызовов программы; обходится только часть итераций цикла; делаются нестрогие предположения, например, об отсутствии алиасов или пропускаются некоторые конструкции языка Си. Память моделируется с точностью до бита, при этом для указательных отношений между ячейками памяти отслеживаются предикаты, при которых они становятся истинными. Тем не менее, основной анализ выполняет только анализ указателей и отслеживание значений целых типов, а стратегии обхода циклов и построения аннотации полностью зависят от конкретного детектора.

При этом инструмент является открытым и доступен для исследований, и в работе [96] удалось сравнить его время работы с временем работы одной из предыдущих версий анализатора Svace. Оказалось, что Saturn от 5 до 10 раз медленнее со включенными двумя детекторами, тогда как в Svace работали все детекторы. Как видно, масштабируемости инструмента для программ современного размера недостаточно.

Инструмент Calysto [91] заявлялся как дорабатывающий идеи Saturn, однако авторы не сделали его открытым, что позволяет судить о нем только по публикациям. Calysto также проводит символьное выполнение, записывая аннотацию функцию в виде структуры *максимально разделяемого графа*: все части символьных выражений, которые могут быть общими, переиспользуются как вершины этого графа, а дуги указывают на связи по данным между выражениями. При необходимости подстановки аннотации в ходе обработки вызова соответствующая символьная переменная связывается с аннотацией, но она раскрывается лениво по необходимости проверки условия ошибки или уточнения реализуемости какого-либо условия перехода, чем, по утверждению авторов, достигается масштабируемость инструмента.

Ряд инструментов концентрируется на поиске заданного типа ошибки вместо создания общей инфраструктуры для поиска многих типов. Например, инструмент Archer [92] ищет ошибки переполнения буфера, отслеживая как отношения между целочисленными переменными и константами, так и символьные связи между переменными с неизвестными значениями. Аннотации функций содержат условия возникновения ошибок доступа к буферу, которые проверяются с учетом контекста вызова в момент обработки инструкции вызова. Идея полностью символьных интервалов для целочисленных переменных при поиске переполнения буфера развивается в работе [95]. Авторы [104] также ищут переполнения с символьными интервалами переменных, однако их алгоритм вычисляет и уточняет символьные интервалы вдоль связей между определениями и использованиями переменных по запросу (by demand) для достижения эффективности, а также учитывает границы циклов, обрабатывая аффинно изменяющиеся индукционные переменные вместо разворачивания циклов на фиксированное количество итераций. Тем не менее, как можно судить, их алгоритм является внутрипроцедурным. В работе [93] инфраструктура символьного выполнения анализатора CSA

расширяется созданием аннотаций для поиска утечек памяти, однако построенный таким образом инструмент ограничен межпроцедурным анализом в рамках одного модуля трансляции. Напротив, работа [49] посвящена расширению инфраструктуры CSA для организации и вычисления, во-первых, аннотаций функции и межпроцедурного анализа на их основе для любых детекторов, пользующихся символьным выполнением, а, во-вторых, для полного межмодульного анализа сохранением необходимых участков аннотаций и АСД. Сложности возникают из-за значительного размера АСД в компиляторе Clang при их сериализации на диск, а также из-за необходимости доработки имеющихся детекторов для корректной работы с аннотациями.

Инструменты, проводящие символьное выполнение, используют SMT-решатели для проверки совместности логических формул, полученных в ходе вычислений. SMT-решатели, проверяющие, верна ли формула в предположениях некоторой *теории*, основаны на произошедшем за последнее десятилетие поистине взрывном прорыве в технологиях SAT-решателей, которые в свою очередь занимаются базовой задачей совместности формул алгебры логики с кванторами существования. Разработано больше десятка теорий, связанных с неинтерпретируемыми функциями, вычислений с битовыми векторами, моделированием массивов, целочисленной и плавающей арифметики, которые позволяют отслеживать большинство интересующих возможностей языка. Недавний обзор результатов в этой области можно найти в [97]. Развитие решателей подстегнуло создание единого формата запросов SMTLib [99], который повлек возможность организовать соревнования решателей SMT-comp [100]. Список актуальных решателей ведется в [98], а из наиболее популярных можно назвать Z3 [101], CVC4 [102] и Yices2 [103], регулярно занимающие высокие места на этих соревнованиях.

Верификация программ с помощью ограниченной проверки моделей заключается в построении по программе некой "суперфункции", в которой на заданную глубину выполнено встраивание функций и развертывание циклов. В полученном графе управления представлены все пути выполнения программы, причем их количество конечно из-за введенного ограничения на глубину развертки. Тем самым решаемой становится задача кодирования выполняемых на этих путях операций в логические формулы и их предъявление SMT-решателю вместе с дополнительными условиями ошибок. Совместность полученных формул означает наличие ошибки в программе. Примером такого инструмента является LLBMC [110], особенность которого в обработке компиляторного представления (биткода LLVM) вместо операций исходного языка. Тем самым упрощается проверка программ на Си и Си++ с некоторым риском не обнаружить ошибку из-за того, что она "спрятана" компилятором при генерации биткода.

Подход уточнения абстракции по контрпримерам (CEGAR) заключается в определении того, достижима ли заданная точка в программе (как правило, для поиска ошибок программа инструментируется рядом проверок, выполнимость которых и означает наличие ошибки). Строится упрощенная модель программы, заключающаяся в некотором наборе абстрактных состояний, записываемых через предикаты на значения программы, и проверяется совместность условий, описывающих достижимость ошибочных состояний. В случае положительного ответа проверяется соответствующая цепочка условий для пути из реальной программы (strongest path precondition). Совместность этих условий означает найденную ошибку, а несовместность – построенный контрпример, на основе предикатов которого изначальная абстракция может быть уточнена, и весь цикл анализа повторен. Хорошим введением в область CEGAR является статья [105].

Подход CEGAR реализован в ряде инструментов – CPAChecker [108], SLAM [109], BLAST [111]. Одна из основных сложностей в его применении заключается в необходимости построения полной модели окружения программы в случае проверки некоторой части программной системы, а также высокие требования к ресурсам. В этой связи наиболее успешные инструменты специализированы для проверки особых программ, например, драйверов операционной системы, размер которых относительно невелик, и значительными усилиями компании можно построить модель окружения, как в инструменте от Microsoft Static Driver Verifier [174], наследнике SLAM. Даже в этом случае сильно ограничены возможности языка, которые разрешено использовать в драйвере, подвергаемом верификации, и требуется проводить активные исследования по улучшению производительности верификатора, например, с помощью переиспользования доказательств решателя с предыдущих попыток верификации [113].

Необходимо отдельно остановиться на очень важной статье [114], которая сравнивает применимость методов статического анализа и проверки моделей для поиска ошибок. Особенно ценно то, что ее авторы известны и как создатели успешных статических анализаторов, и как разработчики методов проверки моделей. Авторы описывают свой опыт применения обеих техник поиска ошибок к нескольким программам различных размеров (до нескольких десятков тысяч строк кода). Их вывод заключается в том, что для ошибок, которые могут быть найдены обоими подходами, статический анализ применяется более успешно. По сути, это ошибки, возникающие из-за свойств языка программы и окружения, в котором она выполняется, а не из-за специфических свойств алгоритмов самой программы. Статический анализ может обрабатывать миллионы строк кода, находит больше ошибок, его ложные срабатывания проще определить вручную и соответственно отладить инструмент. Верификаторы завязаны на корректную модель окружения программы, и ее правильное построение является чрезвычайно сложной задачей, которая, во-первых, недооценивается

исследователями, а, во-вторых, становится неподъемной при увеличении размера верифицируемой программы. Авторы описывают свой опыт верификации модуля стека протоколов TCP в ядре Linux, в ходе которого оказалось, что проще добиться помещения всего ядра в область верификации, чем пытаться выделить минимально возможный набор интерфейсов модуля с остальным ядром и создать соответствующую модель окружения. Ошибки в этих моделях ведут к ложным срабатываниям, которые сложно отладить. Тем не менее, рекомендация авторов в том, чтобы применять верификаторы моделей в тех областях, где они сильны, то есть при проверке ошибок вроде неверных протоколов взаимодействия, нарушения программных инвариантов, которые сложно сформулировать в виде ситуаций, пригодных к поиску статическим анализом. В этом случае, если необходимость проверки некоторых свойств программы превышает потребные затраты ресурсов на создание моделей окружения, применение верификаторов более чем оправдано.

Подытоживая вышесказанное, необходимо заметить, что основные анализаторы на рынке в данный момент подходят под описанную категорию чувствительных к путям анализаторов, выполняющих межпроцедурный анализ на основе аннотаций, и ставящих себе целью найти как можно больше истинных ошибок в программе, возможно, за счет пропуска некоторых ошибок (т.е. выполняющих нестрогий анализ). Это анализаторы Coverity Prevent [115], Klocwork K11 [116], HP Fortify [117]. Верификаторы присутствуют в значительно меньшей степени в виде инструмента Varvel [106] и его предшественника DC2 [107], а также Polyspace Verifier [112]. Анализатор Goanna [175], комбинировавший метод CEGAR и интервальный анализ, был куплен компанией Synopsys в 2015 году и с тех пор не присутствует на рынке. При этом, например, в Varvel, масштабируемости на большие программы, по словам авторов, удалось добиться за счет того, что используется абстрактная интерпретация для проверки простых свойств переменных, а также ряд легковесных анализов и оптимизаций, выполненных по аналогии с компиляторными, которые упрощают код и модели (предикаты) перед применением ограниченной проверки моделей. Даже в этом случае неочевидно, сколько детекторов реализовано авторами и насколько сложно добавить поиск новых ошибок в систему. Тем не менее, направление исследований по внесению большего числа компиляторных анализов в верификаторы для уменьшения сложности анализа выглядит одним из самых многообещающих в развитии этой группы инструментов.

Самым известным анализатором, насколько можно судить, является Coverity Prevent. Открытой информации по нему, как и по остальным перечисленным анализаторам, не так много. Можно предположить, что анализатор просматривает некоторое конфигурируемое количество путей в функции, использует принцип существования контекста выполнения (либо комбинация точек программы означает ошибку из-за неконсистентности, либо некоторый код недостижим – см. также главу 4), комбинирует анализы различных уровней. Анализатор HP

Fortify, так же, как и относительно недавно набравшие известность CheckMarx [31] и AppScan [118], в основном нацелены на ошибки, которые могут указывать на уязвимости, и для выявления которых обычно достаточно анализа АСД и межпроцедурного анализа.

1.4. Классификация ошибок и формализации понятия ошибки в программе

Этот подраздел посвящен нескольким вопросам о понятии ошибки в программе, возникающем в области статического анализа. Во-первых, кратко перечислим общедоступные классификации ошибок с точки зрения пользователей, то есть прикладных программистов, и соответствующие базы данных и тестов. Во-вторых, упомянем работы по формализации ошибок и идеям поиска ошибочных ситуаций. Наконец, еще раз кратко остановимся на вопросе строгости и полноты анализа для поиска ошибок.

Наиболее известной и полной попыткой каталогизировать виды ошибок в программах является проект CWE [124] (Common Weakness Enumeration), который ведет компания MITRE совместно с сообществом исследователей по безопасности. Изначально составлялась база данных конкретных ошибок в программах, теперь известная как CVE (Common Vulnerabilities and Exposures), для публичного оповещения о найденных уязвимостях и способах защиты от них. По мере накопления информации ошибки систематизировались с учетом популярных классификаций типа "семи зловредных царств" [136] (seven pernicious kingdoms) и других подходов. Получившийся каталог является общепринятым, причем компания MITRE может присвоить инструменту анализа статус "совместимого с CWE" в том случае, если найденные этим инструментом предупреждения будут корректно классифицироваться согласно каталогу CWE. Ошибки выстроены в иерархическую структуру, когда для каждого типа ошибки известен как его супертип (более широкий класс), так и более конкретные варианты проявления этого типа ошибки (подтипы). Ценно то, что не навязывается единственной иерархии, а возможно построить несколько так называемых "видов", которые группируют конкретные типы ошибок по-разному. Одним из используемых видов является "программистский" (ошибки выделены по концепциям, применяемым прикладным разработчиком, например, исключения, работа со временем, обработка ошибок внешних интерфейсов и т.п.), а другим – "исследовательский" (ошибки в вычислениях, в доступах к индексируемым ресурсам, в обработке потока управления и т.п.).

Для разработчиков анализаторов каталог CWE в исходном виде неудобен, в основном потому, что приводимая классификация не соответствует тому, как именно анализаторы (как статические, так и динамические) ищут ошибки соответствующего типа. Основное назначение каталога все-таки прикладное, которое позволяет судить об ошибке с точки зрения программиста приложения. Кроме того, часто уровни иерархии бывают либо слишком

мелкими (т.е. различные подтипы ошибок должны искаться одним и тем же детектором, причем даже те, у которых непосредственные супертипы тоже различны), либо слишком крупные (в один тип сливаются ошибки, требующие разных детекторов).

Например, для переполнения буфера основным типом является CWE-118/119 (неверное ограничение доступа в границах буфера), однако среди его вариантов можно найти как различные виды собственно переполнения (CWE-120, 125, 466, 786-788, 805, 823), так и переполнение из-за использования неинициализированного указателя (CWE-824), и переполнение из-за некорректного указателя, например, указывающего на уже освобожденную память (CWE-825, 416). Как правило, последние две разновидности переполнения будут искаться анализаторами именно как использование неинициализированной переменной либо использование указателя после его освобождения – хотя формально сам доступ является переполнением (доступ за пределами выделенной памяти), но не это его основная характеристика, по которой его пытаются найти. И наоборот, различение переполнения из-за доступа ниже нижней границы или выше верхней границы памяти (CWE-786, 788), доступ по чтению или записи (CWE-125, 787) существенной роли не играет, так как отслеживание размера буфера, индекса доступа, потенциальных целей указателей будет выполняться совершенно аналогично. Тем не менее, некоторые узкие подтипы могут использоваться детекторами АСД-уровня, если требуется определять переполнение, происходящее из-за ошибки, сделанной ровно заданным способом (например, CWE-806 – доступ к буферу, в который производится копирование данных, с использованием размера буфера-источника), либо если уровень анализа недостаточен для качественного поиска более общих ошибочных ситуаций.

Важно отметить, что общепринятой публичной классификации ошибок именно с точки зрения разработчика анализаторов пока не существует, поскольку для этого требуется обобщение опыта создания промышленных инструментов анализа, который может рассматриваться разработчиками как их "ноу-хау". Однако определенные подвижки в эту сторону имеются, и есть надежда, что в обозримом будущем такой каталог будет создан.

Среди других попыток классификаций можно отметить отраслевые стандарты кодирования, например, MISRA-C [119] и MISRA-C++ [121] – стандарты применения языков Си и Си++ в критических системах, а также правила безопасного кодирования CERT C [122] и создания эффективных программ на Си++ [120]. Следование этим стандартам для инструмента обычно является ориентиром для потребителей в соответствующей отрасли, например, автомобильной. Для разработчика анализатора эти правила в основном являются либо предметом разработки АСД-детекторов или даже предупреждений компилятора (такая попытка делалась, например, в компиляторе GCC при создании опции `-Weffc++`), либо реализации семейства детекторов, потому что часто правила могут быть сформулированы

слишком общо, например, "не допускайте формирования индексов, выходящих за границы массива" – правило ARR30-C стандарта SERT C Coding Standard). Также для ориентирования программистов создаются рейтинги уязвимостей, например, список OWASP Top Ten для веб-приложений [125].

Отдельным способом классификации является создание наборов тестов, на которых анализаторы должны продемонстрировать нахождение определенного вида ошибок либо, наоборот, не выдать ложных срабатываний. Например, набор тестов Juliet [123] содержит примеры для ошибок по классификации CWE, а набор тестов компании Toyota [133] – примеры для ряда критических ошибок, однако выбор типов этих ошибок также является некоторой произвольной классификацией. Задача общедоступного набора тестов для объективного сравнения анализаторов является очень важной, однако, как представляется, может быть решена только после того, как будет создана классификация ошибок с точки зрения разработчиков инструментов анализа, о которой было сказано выше.

Переходя к формализации понятия ошибки, заметим, что исследований с некими общеприменимыми рецептами по разработке критерия искомой ошибочной ситуации в программе не так много. Как правило, предлагается некий формализм анализа (модель программы, ее памяти и пр.), а затем во введенных терминах определяются ошибки, которые анализатор будет искать. Тем не менее, можно выделить один из принципов формулировки определения ошибок, который в разных видах появляется в статьях по статическому анализу и верификации программ. Как представляется, самой ранним его проявлением является статья Энглера [127], в которой по исходному коду предлагается вычислять набор "представлений" (beliefs) о программе, например, разыменование указателя свидетельствует о представлении "указатель отличен от нуля", а сравнение его с нулем при переходе по истинной ветке, наоборот, говорит о представлении "указатель равен нулю". Далее можно искать противоречия в представлениях, наличие каковых говорит о потенциальной ошибке, при этом, собственно, неважно, какое именно представление оказалось неверным – важно, что нашелся контекст выполнения, в котором они противоречат друг другу.

Аналогично, в статье [126] идея неконсистентных представлений о коде развивается как наличие в программе реализуемого пути (в терминах совместности формулы, собранной по предикатам условий), на котором представления о значениях, вычисляемых программой, содержат противоречия, причем для избежания проблемы алиасинга (в разных точках программы представления формулируются об одном значении, которое находится в разных ячейках памяти) вводится понятие конгруэнтности переменных вдоль некоторого пути выполнения, схожее с нашим понятием класса значений (см. раздел 2.2.2). Делается важное наблюдение о близости задачи обнаружения противоречий в представлениях о программе для поиска в ней ошибок и задаче вывода типов (type inference) в слабо типизированных языках

программирования, для которых наличие противоречия означает ошибку типа (например, одна и та же переменная трактуется как число и как строка).

Как видим, применение принципа поиска противоречий в программе базируется на следующих соображениях. При статическом анализе в том или ином виде всегда наличествует неизвестное анализатору окружение, являющееся источником неопределенных значений переменных программы при вызове одной из анализируемых функций: программа на входе анализатора может быть неполной, либо, например, даже если предъявлена вся программа, то при достаточно больших ее размерах в ходе межпроцедурного анализа неизбежно произойдет потеря точности вычисляемых данных, приводящая к тем же неопределенностям. Консервативный анализатор, выдающий предупреждение о возможной ошибке всякий раз, когда к этому его вынуждает неполнота знаний об окружении, генерирует слишком много сообщений, которые редко будут являться истинными с точки зрения пользователя. Следовательно, задачей анализатора неизбежно становится сделать некоторые разумные предположения о неизвестном окружении так, чтобы они приблизительно соответствовали происходящему и, как следствие, выданные ошибки рассматривались программистом как действительно требующие исправления. Для узкоспециализированного верификатора можно переложить решение этой задачи на пользователя и предложить ему выразить недостающие предположения об окружении в виде спецификаций той или иной формы. Однако для автоматического статического анализатора больших программ это неприемлемо, и искомые предположения нужно извлекать непосредственно из исходного кода.

Критерием наличия ошибки в программе становится противоречивость собранных данных о некоторых значениях в предположении, что найдется такой контекст или группа контекстов выполнения программы (то есть входные данные и состояние неизвестного окружения), при котором это противоречие реализуется. Часто в эту группу включают все варианты выполнения, проходящие через некоторую точку программы, что означает, что если предположение о реализуемости этих контекстов является неверным, то анализатор обнаружил мертвый код (одна из частей противоречия на самом деле не выполняется), а иначе в программе найдена ошибка. Например, вычисление индекса буфера в одном месте программы с участием сравнения с константой 10 дает основание предположить, что индекс может быть больше 10, а использование этого индекса для доступа к буферу размера 8 в другом месте означает предположение, что индекс не может быть больше семи. Если больше о значении индекса у анализатора нет никаких данных, и есть основания считать, что найдется реализуемый путь выполнения, проходящий через обе точки, то можно выдать предупреждение о возможной ошибке переполнения буфера. Если же на самом деле путь нереализуем, то какая-то часть вычислений является условно мертвым кодом (может быть перемещена в другую точку программы, где будет выполняться на реализуемых контекстах), а

если некоторая точка программы такова, что все пути, проходящие через нее, также проходят через упомянутые точки вычисления индекса и обращения к буферу, то эта точка является мертвым кодом.

В зависимости от точности проводимого анализа и реально встречающихся в программах ситуаций можно по-разному строить приближения к решению вопроса о существовании реализуемого контекста выполнения. Так, в инструменте Svace для некоторых межпроцедурных детекторов без чувствительности к путям критерием является уже упомянутое наличие такой точки в программе, что проход потока управления через нее непременно влечет противоречие [96] (она может не совпадать ни с одной из точек, вычисления в которых являются частью возникшего противоречия). В статье [128] вопрос о предъявлении контекста сформулирован как задание набора абстракций, на котором всегда происходит ошибка, и рассмотрены варианты составления этого набора – как одна или несколько критических точек, как путь выполнения, как совместность логической формулы, в которой элементарные предикаты условий в потоке управления рассматриваются независимо (т.е. предполагается, что управление может быть направлено в любую из сторон для каждого условия). Статья [129] вводит понятие *обреченной* (doomed) точки программы, проход выполнения через которую означает непременное возникновение ошибки, и предлагает алгоритм поиска обреченных точек через переписывание программы в форму без циклов и вызовов и проверку совместности построенных условий в решателе в рамках верификатора. Статья [132] ставит себе задачу фильтровать результаты верификации, показывая только те ошибки, для которых удалось построить предусловия на неизвестное окружение, нарушающие заданные пользователем "ангельские предположения" – те, для которых предполагается существующим хотя бы один контекст выполнения, их нарушающий. Здесь задача выведения таких предположений возложена на пользователя, что, как мы видели, неприемлемо для масштабируемого статического анализа. При организации тестирования в статье [131] делается попытка по возможности уточнить контекст возникновения ошибки, поставив задачу удовлетворить максимальному числу предикатов из формулы ошибки при ее возникновении, что локализует причины возникновения противоречий к оставшейся части формулы. В статье [130] делается попытка кластеризовать функции, исходя из связей по их интерфейсам, так, что получившиеся разбиения функций можно тестировать по отдельности – для статического анализа такая задача схожа с анализом программной системы по частям так, что каждую из частей можно считать самостоятельным проектом.

В связи с определением ошибок выше уже говорилось о строгости (консервативности) анализа, то есть должен ли анализатор пропускать ошибки либо выдавать предупреждения о любых возможных ошибках (потенциально ложные). В короткой заметке [134] Годефруа высказывает мысль о том, что нельзя недооценивать роль must-анализов при

поиске ошибок – важнее обеспечить строгость самих сообщений об ошибках (т.е. отсутствие ложных срабатываний), чем строгость анализатора (т.е. консервативность в выдаче возможных ошибок, не пропуская ни одной). Его комментарии, однако, относятся более к динамическим инструментам анализа, которые могут построить подтверждающие ошибку входные данные, и к верификаторам, область применения которых, как мы видели, ограничена. В более современной статье-манифесте [135] ряда известных исследователей в мире анализа замечается, что ни один из известных промышленных статических анализаторов является полностью строгим, но каждый из них делает некоторые нестрогие предположения (в зависимости от анализируемого языка, применяемых методов анализа и т.п.); при этом исследовательское сообщество чрезвычайно трепетно относится к строгости разрабатываемых методов. Авторы призывают всех исследователей в области анализа принять существующее положение дел с частично строгими анализаторами, предлагая термин "soundy" – "почти строгий" анализ (за неимением лучшего перевода), и в дальнейших работах четко формулировать источники нестрогости алгоритмов анализа, стараясь оценивать возможные последствия для точности и полноты его результатов.

1.5. Ранжирование выданных предупреждений и автоматическое исправление кода

Выше уже говорилось о важности задачи уменьшения ручного труда программиста, который требуется потратить на просмотр выданных анализом предупреждений, отсеивание ложных, принятие решения о том, как реагировать на истинные. Существует целый пласт исследований, посвященных этой теме. Одним из вариантов является уточнение выданных предупреждений с помощью верификации или динамического анализа (направленного тестирования, символьного выполнения, фаззинга и т.п.), который далее мы рассматривать не будем. Из оставшихся двумя важными направлениями является ранжирование ошибок (т.е. выдача их в такой форме, что первыми будут просмотрены предупреждения, которые с наибольшей вероятностью являются истинными) и облегчение просмотра предупреждений (например, построение слайсов программы для ускорения навигации по коду) вплоть до автоматического исправления некоторых видов предупреждений. Остановимся подробнее на этих направлениях.

Теме обработки результатов статического анализа для улучшения точности посвящены недавние обзоры [138], [139] и в особенности [137], которые рассматривают ряд техник, включая применение последующего, более точного анализа, динамического анализа и т.п. Нас интересуют направления, посвященные ранжированию предупреждений, среди которых в соответствии с упомянутыми обзорами можно выделить следующие:

- Кластеризация результатов (т.е. объединение предупреждений в группы) с той целью, чтобы при оценке программистом истинности одного из предупреждений в группе можно было сделать вывод о том, что и остальные предупреждения той же истинности. Кластеризация выполняется либо на основе схожести элементов предупреждения (сообщения, места, названия функции или переменной, трассы и т.п.), что может выполняться как анализатором, так и отдельным инструментом непосредственно перед выдачей результатов пользователю, либо на основе связей между данными предупреждений, вычисленными внутри анализатора (например, в [141] ошибки переполнения буфера группируются на основе зависимостей по управлению и связей между значениями индекса так, чтобы если основная ошибка из группы истинная либо ложная, то и остальные тоже). Нужно отметить, что часто такого типа эвристики по выдаче или подавлению предупреждений встроены в анализатор изначально.
- Ранжирование по статистике – выдача предупреждения согласно вероятности его истинности, вычисленной по некоторой статистической модели. Классическим примером является исследование по z-ranking [140], в котором для некоторых видов ошибок (например, проверки результатов работы функций) делается предположение о независимости сделанных ошибок друг от друга, после чего на основании поведения большинства контекстов, в которых может быть сделана ошибка, делается вывод о том, какое именно поведение является правильным, а какое ошибочным. Например, если в 90% случаев указатель, возвращаемый некоторой неизвестной функцией, проверяется на равенство нулю, а в остальных – нет, то можно сделать вывод, что оставшиеся 10% – возможные ошибки. Это наблюдение используются анализаторами достаточно широко, в частности, в инструменте Svace на его основе реализован ряд особых статистических детекторов.
- Предсказание истинности предупреждения на основании истории изменений кода пользователем или истории разметки предупреждений с применением либо статистических моделей, например, логистической регрессии [143], либо машинного обучения на основе разнообразных статических параметров предупреждения и участка исходного кода, в котором оно выдано [142], либо насколько быстро предупреждение было исправлено [144]. Отметим, что сложность применения таких методов заключается в том, что в промышленном окружении при внедрении статического анализатора может быть сложно организовать доступ к истории изменений соответствующего исходного кода, т.к. из-за соображений безопасности обработка исходного кода компании, в особенности репозиториев, может следовать

особым правилам. Доступ к истории разметки предупреждений, как правило, организовать проще.

- Использование внешнего анализа, например, оценка вероятности достижимости места предупреждения в программе в ходе ее выполнения с учетом зависимостей по управлению и данным [145].
- Слияние результатов нескольких инструментов [146]. Здесь практическая сложность заключается в том, что, как уже упоминалось, инструменты одного уровня, выполняющие глубокий межпроцедурный анализ, будут иметь небольшое количество общих предупреждений из-за эвристических подходов в выборе участков нестрогости анализа. Скорее наоборот, для критического кода рекомендовано использование нескольких независимых анализаторов, и слияние их результатов необходимо для единообразной работы с ними, но не для подтверждения предупреждений одного анализатора другим, хотя, разумеется, такие общие предупреждения будут иметь наибольший приоритет.

Нужно отметить, что отдельно развивается направление по применению машинного обучения не для ранжирования предупреждений, а непосредственно для поиска ошибок. Например, в работе [147] предлагается использовать машинное обучение для выделения участков кода, скорее всего не содержащих ошибку (в зависимости от конкретного детектора, например, циклы, редко переполняющие буфер) и опасных участков. Первые, по предположению авторов, должны объединяться некоторыми типовыми характеристиками, в связи с чем возможно провести машинное обучение, предъявляя только положительные примеры (т.е. типовые). Далее предлагается использовать нестрогий анализ для вероятно безопасных участков и строгий анализ для остальных, опасных участков. Представляется неочевидным, что такой частично нестрогий анализ будет так же эффективен на больших программных системах, как и промышленные нестрогие анализаторы. Кроме того, неясно, будет ли селективное применение нестрогости по результатам машинного обучения лучше, чем такое же с учетом особенностей конкретного анализатора и детекторов. В случае использования машинного обучения для выявления особенностей не случайных и стохастических процессов, а объектов, имеющих внутреннюю структуру (программ на высокоуровневом языке), крайне маловероятно получение знаний, которые не могут быть созданы экспертами отрасли, т.е. исследователями компиляторных технологий.

Для автоматического исправления предупреждений также существует только что вышедший замечательный обзор [149], рассматривающий множество различных способов исправлений. Нас касаются методы исправления, в которых оракулом (источником знаний) выступает статический анализ (а не, например, контракты методов, записанные в виде

предусловий и постусловий). Такие методы, как правило, либо ограничивают неизвестные контракты (контексты использования кода), явно вводя необходимые предусловия, либо (в зависимости от детектора) могут определить неверную ошибочную ситуацию и, исходя из ее специфики, предложить исправление как на уровне АСД, так и на межпроцедурном уровне. Например, детектор потенциально бесконечных циклов может выявить ситуацию сравнения индуктивной переменной с верхней границей цикла и уменьшения ее на каждой итерации (вместо увеличения) и соответственно предложить в качестве исправления увеличивать переменную. Другим примером является исправление ошибок доступа к буферу на один элемент (т.н. off-by-one error).

В работе [151] исправлением считается изменение кода, которое максимально уменьшает количество "плохих" трасс выполнения и соответствующее увеличение "хороших", и изменения выполняются в контексте абстрактного интерпретатора. В работе [148] детально разбирается задача исправления выражений с целочисленной арифметикой, которая может переполняться, для примера линейных целочисленных выражений, для которых известны интервалы значений, а также логических выражений с участием таких линейных выражений. В работе [150] анализируется исправление ошибок переполнения буфера в системе рефакторинга с помощью двух техник – замены использования опасных библиотечных функций Си безопасными и введением типов, явно отслеживающих длину буфера (например, структуры с явной длиной строки вместо указателя на символьный тип). Используются преобразования исходного кода на основе современных алгоритмов анализа указателя. В работе [152] исправляются утечки памяти консервативным алгоритмом вставки операций освобождения в точках программы, в которых соответствующая память гарантированно более не будет использована и ранее не была освобождена. Реализация выполнена на основе LLVM с применением алгоритмов обратного анализа потоков данных и анализа указателей и смогла исправить 29% ошибок.

Отметим, что автоматическое исправление ошибок является одним из наиболее многообещающих направлений развития инструментов статического анализа. Как видно, все исправления чрезвычайно зависят от конкретных ошибочных ситуаций, и наиболее удобно выполнять их на уровне АСД, когда можно достаточно свободно построить преобразование, исправляющее ошибку, на основе найденных частей АСД. Библиотека компилятора Clang уже содержит интерфейсы для преобразования исходного кода. Для анализов последующих уровней потребуется вычисление дополнительной информации и поддержание очень точной связи между внутренним представлением анализа и исходным кодом.

1.6. Опыт применения промышленных статических анализаторов

Одной из первых работ, проливающих свет на то, чего разработчики ждут от инструментов статического анализа, является исследование [158], в котором двадцати программистам-респондентам задавались вопросы о том, какими статическими анализаторами они пользовались в работе, что им понравилось и не понравилось, что можно было бы улучшить и т.п. В ответах программистов подчеркивалась необходимость малого количества ложных срабатываний, лучшей интеграции в рабочий процесс (собственный редактор, системы непрерывной интеграции), детальные объяснения ошибки, по возможности, с предложениями автоматического или полуавтоматического исправления. Основные тенденции развития и проблемы внедрения в исследовании схвачены верно, однако можно отметить некоторые недостатки в работе. Во-первых, респонденты выбирались методом анкетирования достаточно случайно и, как следствие, немногие из них используют статический анализатор, поскольку это входит в правила разработки, заданные в компании (что часто бывает в крупных компаниях, закупивших лицензию на анализатор). Во-вторых, большая часть разработчиков имели дело только с анализаторами с открытым исходным кодом, которые в большинстве своем не поднимаются выше уровня АСД-анализа. Лишь несколько разработчиков использовали инструмент Coverity и один – инструмент Klocwork. Поэтому результаты работы не слишком репрезентативны. Кроме того, для промышленных инструментов разработчики могли не иметь законной возможности комментировать качество инструмента либо проблемы в работе с ним, будучи связанными договорами о неразглашении.

Без всякого сомнения, основным исследованием по данной теме является статья [153], в которой разработчики Coverity делятся своим опытом от внедрения инструмента во многих компаниях. Статья является редким случаем честного обсуждения внутренней "кухни" разработчика анализатора. По словам Энглера, одного из авторов статьи, это самая скачиваемая статья с сайта ACM, по мнению ассоциации[4] [159]. В статье подчеркивается важность ряда технических вопросов, в первую очередь прозрачного для пользователя перехвата сборки, а также поддержки компиляции всевозможных диалектов языков программирования для построения внутреннего представления анализатора. В исходном коде, про компиляцию которого анализатору ничего не известно, а если известно, то собственный компилятор анализатора не смог этот код разобрать, ошибок найдено не будет. Разнообразие встречаемых в компаниях методов сборки кода, странных конструкций кода, предназначаемых для устаревших более десяти лет компиляторов часто недооценивается.

Подход по организации контролируемой сборки и доработки компиляторного разборщика-парсера схож с применяемым в анализаторе Svace с той разницей, что мы

[4] Точная цитата: "According to ACM this is their most downloaded paper ever".

используем открытый компилятор Clang в качестве основы, тогда как в Coverity начинали значительно раньше и в результате лицензировали парсер EDG [160]. Кроме того, одним из методов компиляции несовместимого со стандартом кода у Coverity является написание трансформеров кода в форму, которая понимается EDG, и только в ряде случаев приходится напрямую изменять EDG. В компиляторе Svace роль трансформеров выполняет нечеткий режим компиляции, а в остальных случаев парсер Clang изменяется непосредственно.

Но важнейшей темой, поднимаемой в статье, является не технические проблемы внедрения, а социальные сложности, которые также сложно предсказать заранее. Во-первых, разработчиков непросто убедить в том, что их программы содержат ошибки. Любые непонятные сообщения об ошибках либо ошибки со сложной причинно-следственной связью, в которой программист не может разобраться сходу, помечаются при разметке предупреждений как ложные. Помочь может как можно более четкое объяснение происхождения ошибки в интерфейсе просмотра, причем такое, которое разработчик может без особых проблем "прокрутить" в голове.

Во-вторых, любые улучшения анализатора (например, выход новой версии) могут быть встречены "в штыки", поскольку появление новых ошибок в, казалось бы, уже многократно проверенном коде, является неприятным сюрпризом для менеджеров. Создателям Coverity пришлось ограничивать для каждого релиза объем разницы в ошибках, которая позволительна по сравнению с предыдущим релизом. Более того, некоторые методы анализа, которые выдавали сложные для пользователя предупреждения, пришлось отключать в промышленной версии, и авторы констатируют, что исследовательский прототип анализатора был более "продвинут" в определенных областях, чем последующий коммерческий продукт. Проблема ограничения алгоритмов анализа для генерации более понятных предупреждений встречалась нам и в инструменте Svace.

Другая важная статья [154] инженеров компании Google описывает инфраструктуру анализа Tricorder, внедренную в компании (помимо разработчиков Chrome и Android, у которых свои инструменты). Опыт использования различных статических анализаторов внутри Google, в том числе коммерческих, показал, что они, как правило, не масштабируются на размеры программ Google, плохо встраиваются в процесс разработки и оставляют программистов недовольными ложными срабатываниями. Поэтому было принято решение разработать инфраструктуру для запуска анализаторов, которая полностью интегрирована с системами распределенной сборки и непрерывной интеграции (рецензирования кода), принятой в компании. Такая инфраструктура должна запускать различные анализаторы как плагины, при этом отдельные инструменты анализа должны иметь минимум ложных срабатываний (в пределах 10%), а также давать возможность пользователям писать

собственные детекторы ошибок логики, специфичной для проекта (например, проверять корректное использование библиотеки ее потребителями).

Tricorder запускает анализаторы для каждого посланного программистом на рецензию множества изменений (changeset), первый раз – для изменившихся файлов для быстрых текстовых анализаторов (которым не требуется полный разбор программы), второй раз – когда система сборки определяет множество зависимых от изменившихся файлов, которые необходимо пересобрать, и третий раз – когда доступно полное дерево разбора. В заданных ограничениях возможно использовать в основном анализаторы уровня АСД и простейшие межпроцедурные анализаторы. Результаты работы показываются в интерфейсе рецензирования кода, и для каждого предупреждения предусмотрен вариант предоставления обратной связи разработчику ("пожалуйста, исправьте это срабатывание") либо пометка о том, что предупреждение не представляет интереса. Как видно, для компании Google важно исправить как можно больше ошибок еще на этапе включения кода в репозиторий, для чего используются достаточно легковесные анализаторы, а далее упор делается на тестирование и динамический анализ фаззерами и инструментами на основе Google Sanitizers [161].

Компания Microsoft реализует схожую с описанной инфраструктуру CloudBuild [156], которая организовывает распределенную сборку приложений в облаке в том числе с возможностью запускать анализаторы. В статье [157] предлагаются результаты опроса программистов Microsoft о том, какими анализаторами они пользуются и что хотели бы в них видеть, подобно исследованию [158], упоминавшемуся в начале этого раздела. Результаты снова содержат пожелание низкого уровня ложных срабатываний, инкрементального анализа и встраивания анализатора в процесс разработки (большинство программистов предпочитают видеть результаты в своем редакторе кода). Кроме того, большая часть респондентов готова на более долгий анализ в обмен на лучшее качество предупреждений (меньше ложных).

Интересно, что авторы статьи классифицировали причины реальных критических ошибок в сервисах Microsoft за определенный период 2016 года и сравнили результаты с предпочтениями разработчиков о том, какие виды ошибок им важнее всего видеть поддерживаемыми в инструментах анализа. Реальные критические ошибки невысоко расположены в списке приоритетов разработчиков, что не особо удивительно, учитывая значительное расстояние между программистами внутри компании круглосуточной поддержкой. Важны также комментарии по использованию сторонних инструментов анализа в Microsoft, из которых, например, видно, что инструмент CheckMarx сравним с HP Fortify, имея более понятный интерфейс, но проводя не такой глубокий анализ.

Отдельным источником информации о возможностях анализаторов являются статьи-сравнения, однако в них, как правило, либо приводятся уже устаревшие на настоящий момент данные (широко известна статья [163] 2008 года), либо не участвуют известные

промышленные анализаторы, авторам которых не всегда интересно раскрытие информации об их качестве (статья [162] о наборе тестов для оценки ранжирования предупреждений), либо сами тесты известны, но не дают полного представления о возможностях анализаторов (статья [164] об оценке инструментов на основе упоминавшихся ранее тестов по безопасности SAMATE). Еще одним подходом является оценка на основе ошибок, раскрытых как уязвимости в реальной жизни в ходе эксплуатации программ [155]. Представляется, что необходимо оценивать анализаторы как на основе общедоступной базы данных тестов, созданной именно с учетом классификации с точки зрения авторов анализа, про создание которой речь шла выше, так и на основе анализа открытых проектов большого размера. Такое исследование до сих пор, насколько нам известно, в открытых источниках не опубликовано.

1.7. Современные подходы вне классической парадигмы

В настоящем разделе остановимся на ряде инструментов анализа, которые не используют методы компиляторных технологий и верификации программ – это подходы на основе нового типа моделирования памяти, графовых баз данных, микрограмматик и больших данных. Примером первого подхода является инструмент Infer [165], применяемый в компании Facebook для проверки своего кода. Инструмент, по заявлению его авторов, является строгим и основан на разделяемой логике (separation logic): это метод моделирования кучи, в котором при анализе функции выводятся (отсюда название Infer) предусловия и постусловия на динамическую память такие, что выполнение функции завершается без ошибок. Далее эту информацию можно использовать для выдачи предупреждений. Важно, что выведенные пред- и постусловия можно построить, зная только вызываемые в функции другие функции, и тем самым становится возможным обычный межпроцедурный анализ снизу вверх по графу вызовов с сохранением аннотаций функции. В статье [166] кратко описывается опыт применения Infer в Facebook – он встроен в систему постоянной интеграции и рецензирования кода, но также может использоваться и как отдельный инструмент по желанию разработчика. Инструмент развивается, в частности, к нему был недавно добавлен прототип детектора переполнения буфера InferBo [176].

В статье [167] предлагается способ построения детекторов на основе микрограмматик, которые пропускают при разборе программы те ее части, которые нерелевантны для данного детектора. Разборщики микрограмматик могут комбинироваться друг с другом, разбор каждого нетерминала происходит отдельно. В результате разбор программ на Си, достаточный для восстановления ее структур управления потоком, выполняется примерно в 150 строк кода на Haskell. Для потоково-чувствительного детектора разыменования нулевых указателей, основанном на поиске противоречий в исходном коде (один и тот же указателей разыменовывали, но впоследствии сравнивали с нулем), требуется разборщик выражений Си

еще на 30 строк. Такой простой анализатор, содержащий на два порядка меньше кода, чем коммерческая система, находит более 200 ошибок в ядре Linux с уровнем ложных срабатываний всего в 5%. Разборщики, нужные для отдельных детекторов, могут переиспользоваться в других и даже в детекторах других языков (например, детектор для Си был с легкостью портирован для Java и Си++). В настоящий момент система не поддерживает межпроцедурный анализ, но авторы упоминают, что его можно достаточно просто добавить. Конечно, она не служит полноценной заменой промышленного анализатора, однако в ней легко экспериментировать над новыми детекторами, а также использовать, как дополнительный сравнительно легковесный анализ, не требующий строгого разбора программы.

Статья [168] является первым примером применения подхода к анализу больших данных для поиска ошибок. Метод авторов заключается в построении представления программы в виде семантического графа, содержащего все необходимые связи по данным и по управлению, так, чтобы можно было применять уже известные методы решения задач анализа сведением к задаче достижимости на графах. Авторы предложили собственную систему Graspan работы с такими графами в параллельном режиме в стиле подходов больших данных, особенность которой в том, что после построения исходного графа можно задать грамматику, управляющую динамическим добавлением ребер и узлов в граф в ходе анализа (например, для анализа указателей появляются ребра, характеризующие связи между указателями и их целями). Такого свойства, по словам авторов, не было у известных параллельных систем работы с большими графами.

В ходе анализа исходный граф программы генерируется модифицированным компилятором Clang. Graspan содержит несколько популярных детекторов (разыменование нулевого указателя, использование после освобождения, ошибки работы с многопоточными примитивами), которые пользуются анализом указателей или потока данных, сводимым к задаче достижимости. Система позволяет проанализировать код ядра Linux за 12 часов и найти около 100 ошибок с уровнем ложных срабатываний примерно в 20%. Это очень важный шаг на пути использования аппарата больших данных для поиска ошибок.

Существует еще несколько подходов к поиску ошибок, которые представляют детекторы ошибок в виде запросов к некоторой базе данных, разнясь в способах организации запросов и самой базы. Уже упоминавшийся анализатор компании CheckMarx [31] использует собственный язык запросов, напоминающий C#, для поиска ошибок безопасности в основном по типу помеченных данных. Важным свойством анализатора является открытость пользователю всех запросов, составляющих поставку, и возможность их изменять. Тем самым, например, если программист добавил собственную функцию как очищающую (sanitized) помеченные данные, то все стандартные детекторы ошибок будут пользоваться этим знанием.

Авторы языка QL [169], представляющего из себя объектно-ориентированный язык запросов к такой же базе данных и компилирующийся в Datalog, реализовали ряд простых детекторов на языке Java из инструмента ErrorProne [173] на QL, достигнув трехкратного сокращения в объеме кода детекторов. Наконец, в работах [170, 171, 172] предлагается искать уязвимости безопасности, построив графовую базу данных, которая объединяет различные графовые представления программы (АСД, зависимости по данным и по управлению) и создавая запросы к этой базе на специализированном языке Gremlin. В работе удалось найти 18 неизвестных ранее ошибок в ядре Linux версии 3.10.

Подытоживая сказанное, можно констатировать, что перспективными направлениями развития статических анализаторов будут следующие:

- на уровне анализа АСД – введение языков запросов, на которых удобно писать хотя бы ряд типичных детекторов, возможно, с применением микрограмматик;
- на уровне основного анализа – новые методы моделирования памяти либо другие новшества из области верификации, которые доказывают на деле масштабируемость на сверхбольшие программы;
- на уровне инфраструктуры анализа – создание инфраструктуры "метаанализа", в которую удобно добавлять отдельные инструменты анализа и интегрировать результаты с процессом разработки;
- на уровне отдельных движков анализа – развитие применения методов больших данных для построения графовых видов программных систем и обработки запросов к ним, снимая ряд проблем масштабируемости, которые есть у традиционных компиляторных подходов.

2. МНОГОУРОВНЕВЫЙ СТАТИЧЕСКИЙ АНАЛИЗ ИСХОДНОГО КОДА ПРОГРАММ ДЛЯ ПОИСКА ДЕФЕКТОВ

В данной главе предлагается методология многоуровневого статического анализа исходного кода, заключающаяся: в объединении моделей программы и алгоритмов анализа в единую схему выполнения анализа на трех уровнях – уровне абстрактного синтаксического дерева, уровне межпроцедурного контекстно-чувствительного анализа и уровне чувствительного к путям анализа; в разработанных моделях программы и алгоритмах анализа соответствующих уровней; в алгоритмах поиска конкретных ошибок (детекторах), изложенных подробнее в главе 4. Математическое обоснование предложенных алгоритмов состоит в формулировке и доказательстве теорем об оценках сложности и о корректности алгоритмов. Предложенные методы анализа пригодны для построения промышленных анализаторов, которые могут быть применены в цикле разработки безопасного ПО.

При разработке методов необходимо удовлетворить следующим требованиям:

- Функциональные:
  - Поддержка распространенных статически типизированных языков программирования (Си, Си++, Java, C#);
  - Поддержка актуальных классов дефектов (в заданных языках) различной степени критичности;
  - Возможность свободно добавлять поддержку новых классов дефектов;
- Нефункциональные:
  - Масштабируемость до проведения анализа миллионов строк кода за несколько часов;
  - Не менее 60% истинных срабатываний;
  - Пропуск незначительного количества дефектов.

Анализ известных классов дефектов, рассматриваемых в общедоступных базах данных CWE, рекомендаций стандартов CERT Coding Standards, методических документов ФСТЭК, дефектов, которые находятся популярными коммерческими инструментами анализа Coverity Prevent, Klocwork K11 и другими, показывает, что обойтись единственным уровнем анализа («движком») для поиска всех этих дефектов невозможно. С одной стороны, наиболее критические дефекты требуют для своего обнаружения межпроцедурного анализа, который поддерживает чувствительность к контексту и к путям выполнения. С другой стороны, большинство дефектов, связанных с нарушениями правил безопасного кодирования, отраслевых стандартов типа MISRA, рекомендованных языковых практик кодирования, требуют анализа уровня абстрактного синтаксического дерева (т.н. AST-уровня) и максимально детальной информации об исходном коде. На этом же уровне чаще всего требуется искать новые дефекты. Наконец, существует некоторое количество дефектов,

занимающих промежуточное положение между этими двумя категориями – для них требуется межпроцедурный анализ, чувствительный к контексту выполнения, но, как правило, не требуется чувствительность к путям. Тем самым нет необходимости привлекать символьное выполнение, SMT-решатели и подобные техники. Примером таких дефектов могут служить специфичные ситуации разыменования нулевого указателя (безусловное разыменование указателя после предшествующего сравнения с нулем или присваивания нуля), консистентное использование операторов `new/new[]` – `delete/delete[]` в языке Си++ и некоторые другие.

Предлагается выполнять многоуровневый статический анализ исходного кода, схема которого представлена на рисунке. Первый уровень анализа выполняется для представления по типу абстрактного синтаксического дерева отдельно для каждого из поддерживаемых языков. Помимо детекторов, использующих те или иные обходы дерева, имеется возможность использования внутрипроцедурных анализов потока данных и управления. Для этого строится модель памяти, поддерживающая языки Си и Си++, которая отражает операции с указателем для массивов и структур. Необходимый анализ возможных значений (в первую очередь целочисленных переменных и указателей) для внутрипроцедурных детекторов можно выполнять как в виде анализа потока данных (например, над интервалами значений), так и с использованием более сложных абстрактных доменов, в том числе предикатного домена.

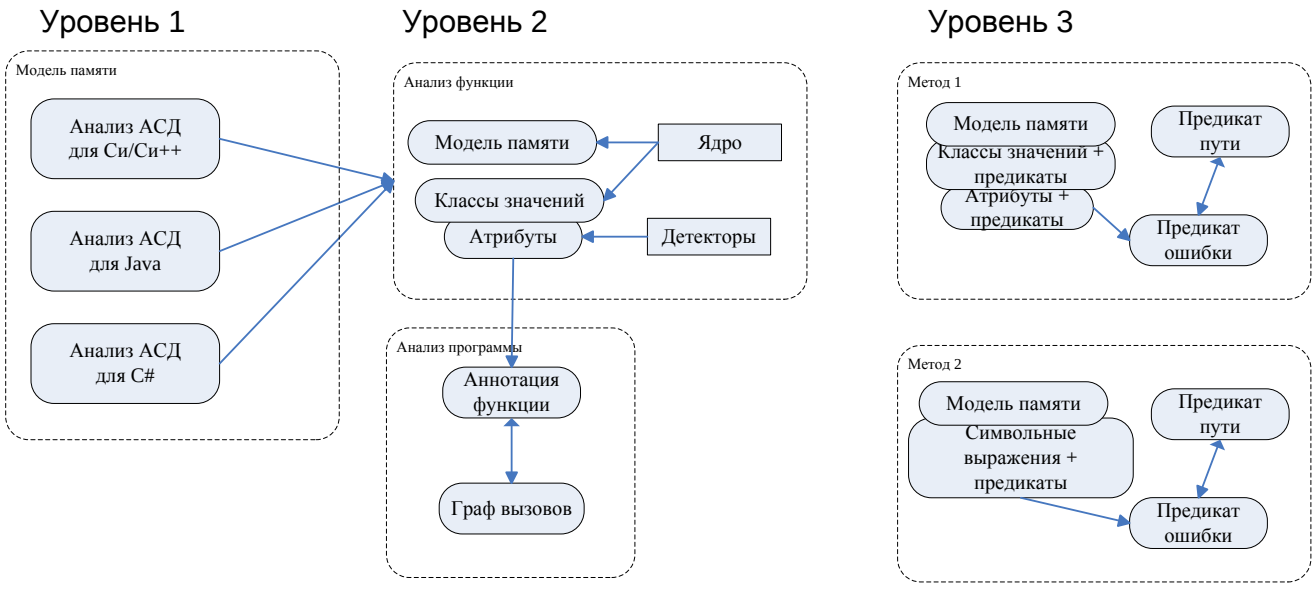

Рисунок 2.1. Схема многоуровневого анализа исходного кода.

Второй уровень анализа является межпроцедурным с контекстной чувствительностью и применением аннотаций функции (англоязычный термин – summary, также используется термин «резюме»). Анализу предъявляется вся программа, в которой тела функций

представлены в трехадресном представлении среднего уровня (по типу биткода LLVM или GCC GIMPLE), общем для всех поддерживаемых языков. Память моделируется так же, как и на первом уровне анализа, отслеживаются возможные значения переменных и множества ячеек памяти, на которые могут ссылаться указательные переменные. Дополнительно для отслеживаемых значений выполняется объединение одинаковых значений в классы эквивалентности (т.н. классы значений, или КЗ), аналогичные номерам переменных в алгоритме нумерации значений. В ходе вычислений при этом делается ряд предположений, ведущих к нестрогости анализа, т.е. потенциальному пропуску ошибок.

Детекторы конкретных дефектов вычисляют интересующие их свойства классов значений или ячеек памяти (мы будем называть эти свойства *атрибутами*), задавая для них множества возможных значений, передаточные функции, и функции объединения значений для точек слияния потока управления. Условием выдачи предупреждения об ошибке является истинность предиката ошибки, при построении которого используются вычисленные значения атрибутов.

По окончании внутрипроцедурного анализа создается *аннотация* функции, которая компактно описывает состояние анализа после выполнения инструкции возврата из функции. Межпроцедурный анализ состоит в однократном обходе графа вызовов программы от вызываемых функций к вызывающим. При обработке вызова функции ячейки памяти, соответствующие в её аннотации формальным параметрам, заменяются ячейками, моделирующими фактические параметры, и часть аннотации, которая построена для памяти вызываемой функции, достижимой из вызывающей через параметры и глобальные переменные, переносится в контекст вызывающей функции.

Третий уровень анализа обеспечивает чувствительность к путям выполнения в проводимом межпроцедурном анализе. Этот уровень можно организовать двумя способами. В первом способе в дополнение к отслеживанию классов значений внутрипроцедурный анализ использует символьное выполнение с объединением состояний. Классы значений являются в том числе символьными переменными, из предикатов с которыми анализ формирует булеву формулу, описывающую условие попадания в заданную точку функции (т.е. *предикат пути*). В модели памяти программы целочисленные значения отслеживаются, как и прежде, консервативно для определения необходимых ячеек памяти для массивов и указательной арифметики. Вдобавок к этому к значениям указателей и целых чисел приписываются предикаты, которые отражают условия, при которых будут приниматься эти значения, и выполняют изменения этих предикатов по мере прохождения по графу потока управления. Атрибуты, дополнительно заводимые детекторами ошибок, также получают предикаты. Выдача предупреждения об ошибке выполняется в результате проверки истинности предиката ошибки и его совместности с предикатом пути с помощью SMT-решателя. Для организации межпроцедурного анализа сформированные предикаты попадают в аннотацию функции, и их

логические атомы, выражающиеся в терминах классов значений и ячеек памяти вызываемой функции, переписываются через контекст вызывающей функции при обработке инструкции вызова, совершенно аналогично основной части аннотации.

Другой способ организации чувствительного к путям анализа применим к языкам без указательной арифметики и других операций над указателями (C#, Java). Так как нет необходимости отслеживания целочисленных значений для отслеживания смещений при моделировании памяти, то в предлагаемой модели используемые смещения всегда известны. Значения ячеек памяти сразу являются символьными переменными, классы значений и их атрибуты не отслеживаются. Объединение состояний анализа, предикаты пути и ошибки строятся сразу над этими переменными. По сути, часть функциональности анализа, выполнявшегося на втором уровне и выделявшего классы значений, переносится на SMT-решатель. Организация межпроцедурного анализа схожа с первым способом.

Далее в разделе 2.1 формулируется первый уровень анализа – уровень абстрактного синтаксического дерева, описывается модель памяти программы. В разделе 2.2 предлагается межпроцедурный анализ с контекстной чувствительностью на основе аннотаций, использующий в качестве внутрипроцедурного анализа комбинацию анализа потока данных и абстрактной интерпретации с выделением классов значений. В разделе 2.3 вводится чувствительный к путям анализ с использованием классов значений на основе формализма анализа второго уровня (раздел 2.3.1), а далее предлагается межпроцедурный анализ, непосредственно использующий символьное выполнение для достижения чувствительности к путям с привлечением модели памяти раздела 2.1, но без выделения классов значений (раздел 2.3.2). В разделе 2.4 описываются принципы построения детекторов на основе инфраструктуры анализов разделов 2.2 и 2.3.

2.1. Статический анализ уровня абстрактного синтаксического дерева

Первым уровнем статического анализа является уровень абстрактного синтаксического дерева (АСД). Мы будем относить дефекты к проверяемым на этом уровне при следующих условиях:

1. для обнаружения дефекта достаточно выполнить либо обход АСД функции, либо найти заданный шаблон на этом АСД. Широко распространенными примерами таких дефектов являются:
    - нарушения правил кодирования, например, стандарта MISRA;
    - неверное использование возможностей языка программирования, например, написание кода, результат выполнения которого зависит от порядка вычисления фактических параметров функции;

- небезопасное использование программных интерфейсов, в том числе криптоинтерфейсов;

2. для обнаружения дефекта необходимо выполнить внутрипроцедурный анализ потока данных или управления, схожий по сложности с анализами, используемыми в компиляторах. Примерами таких дефектов являются использование неопределенного поведения либо поведения, заданного реализацией (в терминах стандарта языка Си), т.е. потенциальное целочисленное переполнение, деление на ноль, использование указателей неверного типа (type punning).

Разберем отдельно организацию анализа на уровне АСД для обоих упомянутых случаев.

2.1.1. Организация обходов АСД

Для детекторов дефектов, выполняющих обход АСД, программа предъявляется в виде набора модулей трансляции, анализ каждого из которых выполняется отдельно. Каждый модуль представляется в виде пары $M = <S, F>$, где S – таблица символов модуля, а F – множество функций (методов) модуля, каждая из которых представлена своим АСД. Таблица символов S является иерархической, то есть может быть представлена как дерево: корнем является глобальная таблица символов (глобальные определения и объявления функций и переменных, пространств имен, классов – в зависимости от языка), а далее связи между таблицами «предок-потомок» соответствует связям между областями видимости в модуле трансляции.

Так как и иерархическая таблица символов, и АСД функций по сути являются древовидными структурами, то можно не разделять таблицы символов и АСД, представляющие код программы, а считать, что модуль трансляции представлен единым АСД, содержащим как узлы с информацией о символах, так и узлы с кодом программы. Мы разделяем АСД с кодом и таблицу символов для удобства классификации детекторов-обходчиков.

Итак, детекторы могут выполнять обходы как таблицы символов, так и АСД функций. Природа выполняемых детекторами проверок может быть совершенно различной, и, как представляется, для формализации стоит остановиться только на вопросе масштабируемости анализа. Для масштабируемости нужно потребовать, чтобы сложность алгоритмов детекторов была линейной от размера представления программы, т.е. от количества узлов АСД и элементов таблиц символов. Имея в виду это соображение, введем следующую классификацию детекторов:

- **Детекторы 1 типа** выполняют обход только таблицы символов, но не АСД функций. В таблице символов выполняется поиск всех элементов заданного типа,

после чего для каждого найденного элемента его «потомки» – элементы вложенных в него таблиц символов – могут посещаться не более одного раза. Как пример можно привести детекторы Си++-специфичных правил кодирования – о том, что классы должны содержать конструктор по умолчанию; в классе не должно быть публичных и приватных полей одновременно; в публичных методах не должно быть параметров типа обобщенного указателя (`void *`) и т.п.

- **Детекторы 2 типа** выполняют обход АСД функций в поисках узлов-операторов заданного типа, после чего для каждого найденного узла выполняется посещение не более чем константного количества других узлов АСД. Как представляется, данный тип детекторов наиболее распространен – так, для безопасного кодирования часто используется правило о запрещении использования выделенного набора «опасных» функций, в языковых правилах кодирования – запрещение сравнивать переменные вещественного типа на точное равенство, запрещение использовать выражения с побочным эффектом внутри оператора `sizeof`, запрет на оператор `goto` и т.п.
- **Детекторы 3 типа** выполняют обход АСД функций в поисках узлов-операторов заданного типа, после чего для каждого найденного узла выполняется посещение всех его подузлов, но не более одного раза. Типичным примером такого детектора является проверка ограничений, накладываемых на запись некоторого выражения – запрет использовать ассоциативность операций отношения (выражения вида `a > b > c`, `a == b == c`), проверка потенциально ошибочных логических операций с одинаковыми аргументами (выражения вида `a || a`, `b && b`) и т.п. При их реализации ищутся все операции нужного типа, а далее обходятся все подузлы АСД, формирующие операнды для этих операций, и для этих подузлов проверяются заданные ограничения.
- **К детекторам 4 типа** относятся все детекторы, не подошедшие под условия одного из предыдущих типов. Как правило, такие детекторы после обнаружения в АСД узлов-операторов заданного типа просматривают подузлы найденного узла более одного раза либо просматривают также и ряд узлов АСД, не являющиеся потомками найденного узла – родительские узлы, узлы-соседи того же уровня иерархии и т.д. В качестве примера можно привести проверку того, что индуктивные переменные цикла обновляются в теле цикла не более одного раза – количество посещений узлов АСД, относящихся к телу цикла, соответствует количеству индуктивных переменных. Другим примером является проверка запрета на переопределение виртуальных методов невиртуальными в Си++ – для

этого необходимо выполнить обход всей иерархии классов в поисках нарушающих ограничение переопределений функции.

Интуитивно детекторы выделенных типов либо работают за линейное время, либо для достижения линейной сложности необходимо дополнительно ограничить количество просматриваемых ими узлов АСД. Сформулируем это соображение более строго в виде следующей теоремы:

Теорема 2.1. Для данного модуля $M = <S, F>$:

- детекторы 1 типа выполняются за время $O(s)$, где s – количество символов во всех таблицах символов модуля, тогда, когда при обработке элемента таблицы заданного типа в любой вложенной таблице символов дополнительно каждый символ просматривается детектором не более чем $O(1)$ раз;
- детекторы 2 типа выполняются за время $O(n)$, где n – количество узлов в АСД данной функции;
- детекторы 3 типа выполняются за время $O(n)$, где n – количество узлов в АСД данной функции, тогда, когда при обработке узла заданного типа дополнительно в поддереве этого узла в АСД каждый узел просматривается детектором не более чем $O(1)$ раз;
- детекторы 4 типа выполняются за время $O(n)$, где n – количество узлов в АСД данной функции, тогда, когда каждый поиск интересующего узла АСД занимает не более чем $O(1)$ шагов, при этом искомых детектором узлов не более $O(f(n))$, после чего количество дополнительно просматриваемых узлов АСД составляет не более чем $O(n/f(n))$, где $f(n)$ – функция, характеризующая частоту встречаемости в функции искомых узлов.

Доказательство следует непосредственно из определения типов детекторов. Для детекторов первого типа сложность поиска заданного элемента по типу в данной таблице символов при использовании хеш-таблицы в качестве структуры данных составляет $O(1)$, а дальнейший просмотр вложенных таблиц для поиска элемента потребует $O(t)$, где t – количество таблиц, для первого элемента заданного типа из вложенной таблицы и $O(1)$ для остальных элементов. Тем самым сложность поиска элемента заданного типа для всех элементов составит $O(s+t)$.

После этого может быть дополнительно просмотрено еще $O(s_i)$ элементов, где s_i – количество элементов в i-й таблице символов. Т.к. дополнительно каждый элемент просматривается не более одного раза, то для всех искомых элементов дополнительно будет просмотрено не более $\sum_i s_i = s$ элементов, что составляет еще $O(s)$ шагов. Итого имеем общую сложность в $O(s+t)$ шагов. В наихудшем случае $t=O(s)$, тем самым сложность составляет $O(s)$.

Для детекторов второго типа обход АСД с поиском узлов нужного типа составит не более $O(n)$ шагов, а т.к. на каждом успешном поиске дополнительно посещается не более $O(1)$ узлов, то общая сложность составит $O(n)$, где n – количество узлов АСД.

Для детекторов третьего типа рассуждения схожи с таковыми для детекторов первого типа. Посещение всех нужных узлов займет не более чем $O(n)$ времени, тогда как для каждого i-го узла дополнительный просмотр его подузлов займет не более $O(n_i)$ шагов, а всего – не более чем $\sum_i n_i = n$ шагов, т.е. $O(n)$.

Для детекторов четвертого типа, как оставшихся от классификации, сложно построить разумное ограничение на время выполнения. Можно лишь заметить, что если некоторая функция $f(n)$ задает количество узлов АСД, которые должны быть протестированы детектором, то для достижения линейной сложности, во-первых, на поиск узла нужно тратить в среднем не более $O(1)$ шагов, а далее на каждый тест нужно расходовать в среднем не более чем $O(n/f(n))$ шагов. Так как для проведения теста, как правило, нужно просмотреть некоторое количество других узлов, а сложность обработки одного узла константна, то по существу данное ограничение по времени работы ограничивает количество узлов, которые можно дополнительно просмотреть.

Для быстрого поиска узла заданного типа в некотором поддереве АСД предложим следующий алгоритм посещения узлов нужного типа:

Алгоритм 2.1. Пусть $f(n)$ значительно меньше n, тогда будем дополнительно хранить в каждом узле:

1) индекс данного узла в порядке обхода в глубину, а также наименьший и наибольший индексы узлов этого поддерева (схожим образом также делается в [178]),
2) ссылку на следующий узел в АСД (в порядке обхода в глубину) того же самого типа,
3) ссылки на следующие узлы в АСД (в порядке обхода в глубину) для остальных требуемых для детекторов типов узлов.

Тогда для обработки всех подузлов заданного типа:

- проверим хранимую ссылку на следующий узел искомого типа;
- пока индекс следующего узла лежит в пределах, заданных наименьшим и наибольшим индексами для узлов данного поддерева, будем обрабатывать необходимые узлы, переходя между ними по ссылкам 2) на следующий узел того же типа (аналогично, например, делается в прошитом двоичном дереве [177]).

В случае, когда количество искомых узлов $f(n)$ сопоставимо с n, более подходящим вариантом поиска будет обычный обход АСД, не требующий дополнительных расходов памяти. При этом дополнительно для проведения теста детектор может просмотреть не более $O(1)$ узлов. ■

Отдельно отметим, что описанный в доказательстве алгоритм 2.1 поиска подузлов заданного типа требует значительного количества памяти, если детекторы четвертого типа требуют поиска многих типов узлов: вообще говоря, для каждого подузла необходимо хранить ссылки на следующие в порядке обхода в глубину узлы всех требуемых типов, то есть дополнительно в каждом узле тратится линейный от количества поддерживаемых типов узлов объем памяти. Можно улучшить ситуацию следующими способами:

- Хранить не ссылки на узлы, а индексы следующих узлов нужного типа в порядке обхода в глубину. Для этого нужно дополнительно поддерживать отображение (хеш-таблицу), позволяющую получить по индексу ссылку на узел за $O(1)$ времени. Для АСД функции в подавляющем большинстве случаев будет достаточно двух байт на индекс, тогда как ссылка в современных 64-битных системах займет 8 байт.
- Воспользоваться тем, что для соседних узлов ссылки на следующие узлы нужного типа часто будут одинаковыми. Тем самым можно хранить эти ссылки только в некоторых узлах АСД (назовем такие узлы *базовыми*), а в остальных узлах хранить единственную ссылку на ближайший родительский базовый узел. Тогда возможно следующее улучшение алгоритма 2.1:

  Алгоритм 2.1'. Для получения ссылки на следующий узел данного типа при хранении индексов в базовых узлах необходимо:

  1. получить (за $O(1)$) ссылку на базовый узел;
  2. получить ссылку на следующий узел данного типа для родительского узла;
  3. если индекс следующего узла находится в необходимых границах между наименьшим и наибольшим индексом для подузлов текущего узла, то можно итерироваться по узлам нужного типа по ссылкам 2), как в алгоритме 2.1;
  4. если индекс меньше наименьшего индекса, то нужно пропустить ряд узлов, проходя по ссылкам 2) до тех пор, пока индекс очередного узла не окажется в необходимых границах;
  5. если же индекс больше наибольшего индекса, то это означает, что подузлов искомого типа у текущего узла нет.

  Базовыми узлами целесообразно сделать узлы, отвечающие за организацию потока управления (узлы ветвлений, циклов, операторов выбора) и узлы составных операторов. Если провести аналогию с графом потока управления, то нужная информация будет храниться для каждого базового блока. Так как в типичных программах средний размер базового блока небольшой (5-10 инструкций [179]), то количество дополнительных итераций по ссылкам 2) до попадания в интервал допустимых индексов узлов для этих случаев будет составлять $O(1)$.

Можно также заметить, что оба предложенных метода легко комбинируются при хранении в базовых узлах АСД индексов узлов вместо ссылок.

2.1.2. Модель памяти программы и внутрипроцедурный анализ уровня АСД

Детекторы, для реализации которых требуется применение внутрипроцедурного анализа потока управления и данных, используют в качестве входных данных ту же пару $M = <S, F>$, где S – таблица символов модуля, а F – множество функций модуля, но в этом случае представление функции в виде АСД дополняется графом потока управления: $F = \{ f:<A, G> \}$, где A – АСД функции, а G – ее граф потока управления. Базовые блоки графа потока сформированы из узлов АСД, соответствующих операторам языка программирования, не содержащим в себе переходов управления. При этом ссылки на родительские узлы АСД позволяют судить, например, о том, какие базовые блоки составляют данный узел-цикл или условие.

Для построения детекторов необходимо описать используемую ими модель памяти программы и её семантику, т.е. интерпретацию операторов языка в её терминах. Мы сконцентрируемся на описании подмножества операторов языка Си и модели памяти, поддерживающей все операции Си. Тем самым предлагаемая модель будет применима и к более высокоуровневым языкам (Си++, C#, Java – для последних двух языков ситуация упростится из-за отсутствия адресной арифметики и операции взятия адреса). При этом анализ программ на различных языках на уровне АСД может быть как полностью раздельным, так и опираться на некоторое «объединённое» АСД, узлами которого могут быть операторы всех поддерживаемых языков [58]. Этот вопрос ортогонален организации модели памяти для анализа.

Для моделирования памяти мы используем понятие *ячеек памяти*, которые принадлежат некоторым *областям памяти* (схоже с [39]). Концепция областей памяти следует идее классов памяти стандартов Си и Си++, и для целей анализа достаточно различать *статическую*, *автоматическую* (стек) и *динамическую* область памяти. Ячейки памяти принадлежат к одной из описанных областей памяти и составляют четверку $M = <R, S, O, P>$, где:

- R является областью памяти;
- P является *родительской* ячейкой памяти (например, ячейка для всего массива по отношению к элементу массива или ячейка полной структуры для поля структуры). Родительская ячейка также задает *базовый адрес* для данной ячейки памяти, т.е. либо виртуальный адрес начала стека для данной функции, глобальный указатель, либо начало динамически выделенного участка памяти. Поэтому отдельно хранить сущность, аналогичную базовому адресу, нет необходимости, достаточно ограничиться заданием родительской ячейки;

- S и O задают размер ячейки памяти (в битах) и смещение относительно базового адреса.

При построении модели памяти предполагается, что в языковом АСД уже произошло распределение локальной памяти и вычисление размеров всех типов данных. Тем самым все смещения локальных переменных относительно начала фрейма функции, смещения полей структуры относительно ее базового адреса, размеры элементов массивов известны.

Опишем правила создания ячеек памяти для локальных, глобальных, и динамических переменных. Ячейки памяти создаются «лениво» в момент первого доступа к ним. Необходимо рассмотреть следующие случаи:

1. Создается ячейка памяти, базовый адрес которой, размер и смещение относительно базового адреса для ее класса памяти известны статически (во время анализа);
2. Размер и/или смещение определяются динамически. Примерами таких ситуаций для языка Си с заранее неизвестным размером ячейки являются массивы переменной длины (VLA), ячейки, выделенные функцией `alloca`, внешние массивы, динамически выделенная память. Примеры заранее неизвестного смещения – это доступы к массивам с динамически вычисляемым индексом.

   Для таких ситуаций сначала нужно предложить способ моделирования значений переменных программы (в основном целочисленных), чтобы иметь возможность оценивать размер и смещение ячейки.
3. Базовый адрес ячейки памяти статически не определяется однозначно. Примерами таких ситуаций являются доступы к полям структуры по указателю, который может указывать на несколько потенциальных ячеек памяти для структур. В этом случае, помимо моделирования значений целочисленных переменных, необходимо также моделировать возможные значения указательных переменных, например, определяя для каждой множество ячеек памяти, на которые данная переменная может указывать (т.н. points-to множества).

Сначала мы рассмотрим случай 1) известного базового адреса, размера и смещения, потом предложим способы моделирования целочисленных значений и базовых адресов, а потом вернемся к рассмотрению создания новых ячеек памяти для случаев 2) и 3). Для простоты изложения, не теряя общности, будем рассматривать следующее множество выражений и операторов языка:

$$Expr := Operands \mid ArithExpr \mid AssignExpr \mid CompExpr \mid AddrExpr \mid FuncExpr$$
$$Operands := Constant \mid Variable$$

$$Variable := Scalar \mid Pointer \mid Array \mid Structure$$
$$ArithExpr := Expr + Expr \mid Expr - Expr$$
$$AssignExpr := Variable = Expr$$
$$CompExpr := Expr > Expr \mid Expr < Expr \mid Expr == Expr \mid Expr\ != Expr$$
$$AddrExpr := \&Expr \mid *Pointer \mid Var[Expr] \mid Var.Field \mid (type *)\ Expr \mid malloc\ (Expr)$$
$$FuncExpr := Variable(Expr_1, \dots, Expr_n) \mid Pointer(Expr_1, \dots, Expr_n) \mid return\ Expr$$
$$Stmt := Expr; \mid \{\ Stmt_1; \dots; Stmt_n;\ \} \mid if\ (Expr)\ Stmt;\ else\ Stmt; \mid while\ (Expr)\ Stmt;$$

Рисунок 2.2. Модельный язык.

Такое множество выражений дает нам возможность выразить все необходимые варианты именования памяти, включая динамическую память и приведение указательных типов, и минимум арифметических выражений (сложение/вычитание) и организующих поток управления операторов (ветвления, циклы), который нужен для формальных рассуждений.

Итак, для полностью известных параметров ячеек памяти возможны следующие случаи:

1. Доступ к локальной переменной. Для каждой функции *foo* изначально создается ячейка памяти *MLFrameFoo*, представляющая фрейм функции в целом. В ячейке *MLFrameFoo*:
   - R = Local,
   - S – вычисленный компилятором минимальный размер фрейма функции (без учета массивов переменного размера),
   - O = 0,
   - P = *MLFrameFoo*.

   Далее, для ячейки памяти *MLLocal*, создаваемой для обрабатываемого доступа:
   - R = Local,
   - S – вычисленный компилятором размер типа переменной,
   - O – вычисленное компилятором смещение данной локальной переменной относительно начала фрейма функции,
   - P = *MLFrameFoo*.

   Выставление смещения локальной переменной относительно начала фрейма функции позволяет при желании отслеживать доступы, попадающие за ее пределы, понимая, какая именно память фрейма функции затрагивается этими доступами (т.е. классифицировать переполнения буфера на стеке).
2. Доступ к параметру функции. Создается ячейка памяти *MLParam* по аналогии с ячейками памяти *MLLocal* для локальных переменных.

3. Доступ к возвращаемому значению функции. Создается ячейка памяти *MLReturn* по аналогии с ячейками памяти *MLLocal*, отдельно для каждого вызова.
4. Доступ к глобальной переменной. Поля ячейки памяти *MLGlob* заполняются следующим образом:
    - R = Static,
    - S – вычисленный компилятором размер типа переменной,
    - O = 0,
    - P – эта же ячейка памяти.
5. Выделение динамической памяти (через вызовы функции `malloc` или подобных ей). Считается, что каждый новый вызов функции динамической памяти возвращает новую (не пересекающуюся с ранее выделенной) память. Поля ячейки памяти *MLHeap* заполняются следующим образом:
    - R = Dynamic,
    - S – размер участка памяти, соответствующий значению параметра функции `malloc`;
    - O = 0,
    - P – эта же ячейка памяти.
6. Доступ к элементу составной структуры данных (полю структуры или элементу массива). Сначала создается ячейка памяти для всей структуры данных. Далее создается ячейка *MLElement* для элемента составного типа следующим образом:
    - R – класс памяти родительской ячейки,
    - S – вычисленный компилятором размер типа переменной;
    - O – вычисленное компилятором смещение относительно родительской ячейки (смещение поля структуры либо смещение элемента массива относительно начала массива – произведение размера типа элемента на значение индекса),
    - P – ячейка памяти для всей структуры.

    Если для полей структур моделирование ячеек памяти выполняется точно, то для элементов массива задается константа *MaxSize*. В случае, когда индекс превышает значение *MaxSize*, новой ячейки памяти не создается, вместо этого используется специально заведенная ячейка *MLUndefElement*, в которой поле смещения *O* = -1. Это дает возможность ограничить точность моделирования для больших массивов.

Теперь для обработки случаев неизвестных статически базовых адресов и значений размеров и индексов введем отслеживание абстрактных значений для целочисленных и указательных переменных. Для целочисленных переменных будем пользоваться абстрактным

доменом из интервалов значений `[a, b]`, в которых границы интервалов – это целые числа или ±∞, и хранить для каждой ячейки памяти абстрактное значение *Val* из этого домена. Для указательных переменных будем хранить множества *PtTo* ячеек памяти, на которые могут указывать переменные.

Запишем передаточные функции для введенных нами операторов языка. Для этого введем функцию пересечения интервалов *Intersect* и объединения интервалов *Unify*:

$$\frac{Val_1 = [a,b], Val_2 = [c,d]}{Intersect(Val_1, Val_2) = [a,b] \cap [c,d] = [\max(a,c), \min(b,d)]}$$

$$\frac{Val_1 = [a,b], Val_2 = [c,d]}{Unify(Val_1, Val_2) = [a,b] \cup [c,d] = [\min(a,c), \max(b,d)]}$$

Если при вычислении функции *Intersect* левая граница интервала становится больше правой (т.е. $\max(a,c) > \min(b,d)$), то результатом является пустой интервал $\emptyset$.

- Инициализация: если ячейка памяти ML встретилась в первый раз (только что создана), то $Val(ML) = [-\infty, +\infty]$.
- Константа:

$$\frac{a = const}{Val\big(ML(a)\big) = [const, const]}$$

- Арифметические операции:

$$\frac{a = b + c, Val\big(ML(b)\big) = [b1, b2], Val\big(ML(c)\big) = [c1, c2]}{Val\big(ML(a)\big) = [b1 + c1, b2 + c2]}$$

$$\frac{a = b - c, Val\big(ML(b)\big) = [b1, b2], Val\big(ML(c)\big) = [c1, c2]}{Val\big(ML(a)\big) = [b1 - c2, b2 - c1]}$$

- Присваивание:

$$\frac{a = b, Val\big(ML(b)\big) = [b1, b2]}{Val\big(ML(a)\big) = [b1, b2]}$$

- Сравнение в условии перехода (условного оператора, цикла):

$$\frac{if\ (b < c), Val\big(ML(b)\big) = [b1, b2], Val\big(ML(c)\big) = [c1, c2]}{\begin{array}{c} Val_{b<c}\big(ML(b)\big) = Intersect([b1, b2], [-\infty, c2 - 1]) \\ Val_{b<c}\big(ML(c)\big) = Intersect([c1, c2], [b1 + 1, +\infty]) \end{array}}$$

$$\frac{if\ (b == c), Val\big(ML(b)\big) = [b1, b2], Val\big(ML(c)\big) = [c1, c2]}{Val_{b==c}\big(ML(b)\big) = Val_{b==c}\big(ML(c)\big) = Intersect([b1, b2], [c1, c2])}$$

Здесь $Val_{cond}\big(ML(a)\big)$ обозначает значение *Val* для ячейки памяти, соответствующей переменной *a* на пути, на котором выполняется *cond*. Приведены передаточные функции, которые применяются для пути с выполненным условием (переход к then-части условного оператора или успешно выполненный вход в цикл). Функции для перехода к else-части (невыполненное

условие) строятся аналогично. Единственной тонкостью является то, что в случае перехода по условию неравенства значения *Val* не изменяются, т.е. $Val_{a!=b}\big(ML(a)\big) = Val(ML(a))$. Дело в том, что в домене интервалов значений нельзя представить ситуацию “выколотой точки”, т.е. тот факт, что искомое значение лежит в некотором диапазоне значений, за исключением некоторого конкретного значения.

Можно добиться поддержки таких ситуаций, изменив используемый абстрактный домен, например, на домен множества непересекающихся интервалов (тем самым выколотая точка может разбить один из интервалов множества на два), либо использовать так называемые “анти-интервалы” [180]. Другим вариантом является оставить домен интервалов значений, завести дополнительный атрибут ячеек памяти, отслеживающий равенство либо неравенство некоторой константе в дополнение к интервалу значений, выписать и вычислить передаточные функции для этого атрибута. Тем не менее, уточнение анализа предлагаемой модели памяти для легковесного анализа на уровне АСД не соответствует нашему построению многоуровневого статического анализа, и потому в данном разделе мы довольствуемся абстрактной интерпретацией на интервалах значений. Если клиентскому анализу для поиска искомых дефектов недостаточно предоставляемой точности на уровне АСД, необходимо повысить уровень до межпроцедурного анализа, где данная проблема решена.

- Обработка точек слияния потока управления:

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Pred(b), Val_{b_i}\big(ML(a)\big) = \big[a_i^l, a_i^r\big]}{Val_b\big(ML(a)\big) = Unify_{b_i}^{i=1,n}\big(Val_{b_i}\big(ML(a)\big)\big)}$$

Здесь функция $Unify_{b_i}$ выполняет попарное объединенине интервалов для ячейки памяти переменной a, взятых из блоков-предшественников:

$$Unify_{b_i}^{i=1,n}\big(Val_{b_i}\big(ML(a)\big)\big) = \begin{cases} Unify\big([a_1^l, a_1^r], [a_2^l, a_2^r]\big), n = 2 \\ Unify\big(Unify_{b_i}^{i=1,n-1}\big(Val_{b_i}\big(ML(a)\big)\big), [a_n^l, a_n^r]\big), n > 2 \end{cases}.$$

- Разыменование:

$$\frac{a =* b, PtTo(b) = \{ML_i\}}{Val\big(ML(a)\big) = Unify_i(Val(ML_i))}$$

Аналогично предыдущему случаю, здесь функция *Unify* объединяет все возможные интервалы значений ячеек памяти, на которые может указывать ячейка памяти для переменной b.

Теперь запишем передаточные функции для множеств *PtTo*:

- Инициализация: если ячейка памяти ML встретилась в первый раз (только что создана), то $PtTo(ML) = Undef$.

  Значение *Undef* выделяется для неизвестного множества (данная переменная может указывать куда угодно). Дополнительно выделяется значение $\emptyset$, которое означает, что переменная никуда не указывает (например, это нулевой указатель).

- Арифметические операции:

$$\frac{a = p + const, PtTo(p) = \{ML_i\}}{PtTo\big(ML(a)\big) = \{ML_j : ML_j = Copy(ML_i) \wedge S_j = S_i + 8 \times const \times \text{sizeof}(*p)\}}$$

  Здесь $Copy(ML_i)$ означает функцию, копирующую ячейку ML_i. Другими словами, новое множество *PtTo* состоит из копии множества для указательной переменной *p*, при этом в новых ячейках памяти операция сложения отражена прибавлением к смещению соответствующей константы в битах (с учетом размера типа). Если вторым параметром сложения также является переменная или другое выражение, то оценивается его абстрактное значение *Val*, и результирующее смещение вычисляется в интервальной арифметике:

$$\frac{a = p + e, Val(e) = [a, b], Val(S_i) = [c, d], \text{sizeof}(*p) = sz}{Val\big(S_j\big) = [a + 8 \times sz * c, b + 8 \times sz * d]}$$

  Тем самым смещение (и размер, как мы увидим далее) ячейки памяти может выражаться как целочисленной константой, так и интервалом.

  Для операции вычитания передаточная функция строится аналогично. Для остальных операций считаем, что $PtTo(ML(a)) = Undef$.

- Присваивание:

$$\frac{a = p, PtTo(p) = \{ML_i\}}{PtTo\big(ML(a)\big) = PtTo(p)}$$

$$\frac{a = 0}{PtTo\big(ML(a)\big) = \emptyset}$$

  Для остальных операций присваивания считаем, что $PtTo(ML(a)) = Undef$.

- Сравнение в условии перехода (условного оператора, цикла):

$$\frac{if\ (p == q), PtTo(ML(p)) = \{ML_i\}, PtTo(ML(q)) = \ \{ML_j\}}{PtTo_{p==q}\big(ML(p)\big) = PtTo_{p==q}\big(ML(q)\big) = PtTo(p) \cap PtTo(q)}$$

$$\frac{if\ (p\ != q), PtTo(ML(p)) = \{ML_i | i = \overline{1,m}\}, PtTo(ML(q)) = \ \{ML_j | i = \overline{1,n}\}}{\begin{gathered} m = 1 \rightarrow PtTo_{p!=q}\big(ML(q)\big) = PtTo(q) \setminus PtTo(q) \\ n = 1 \rightarrow PtTo_{p!=q}\big(ML(p)\big) = PtTo(p) \setminus PtTo(q) \end{gathered}}$$

  Другими словами, если две указательные переменные равны, то после перехода по истинной ветви мы можем сделать вывод, что возможные ячейки памяти из

множеств *PtTo* должны были присутствовать в обоих множествах. Если же две переменные не равны, то без хранения дополнительных атрибутов этот факт можно отразить во множествах *PtTo* только в том случае, если для какой-либо переменной множество содержит единственный элемент (т.е. выражает must-отношение). Тогда для другой переменной мы знаем, что ее множество *PtTo* заведомо не содержит этот единственный элемент – иначе переменные были бы равны.

Для остальных операций сравнения мы не изменяем множества *PtTo* для задействованных ячеек памяти.

- Взятие адреса:

$$\frac{a = \&b}{PtTo\big(ML(a)\big) = \{ML(b)\}}$$

- Разыменование:

$$\frac{a = *\, b, PtTo(b) = \{ML_i\}}{PtTo\big(ML(a)\big) = Unify_i(PtTo(ML_i))}$$

- Обработка точек слияния потока управления:

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Pred(b), PtTo_{b_i}\big(ML(a)\big) = \{ML_j^{b_i}\}}{PtTo_b\big(ML(a)\big) = Unify_i(PtTo_{b_i}\big(ML(a)\big))}$$

Здесь функция $Unify_i$ сливает множества *PtTo* с учетом значения *Undef*:

$$Unify(PtTo_a, PtTo_b) = \begin{cases} \{Undef\}, & Undef \in PtTo_a \vee Undef \in PtTo_b \\ PtTo_a \cup PtTo_b, & \text{иначе} \end{cases}.$$

Теперь рассмотрим, как изменяются правила создания ячеек памяти для случаев, когда размер ячейки или ее смещение относительно родительской ячейки не являются константами, т.е. неизвестны точно на момент анализа. Такое поведение возможно для правила 1 и правил 5-6, т.е. для локальных переменных, доступов к элементам массива и динамической памяти. Во всех этих случаях алгоритм создания новой ячейки памяти дополняется так:

- вычисляется абстрактное значение смещения Val_o и/или размера Val_s создаваемой ячейки для данной точки программы;
- просматриваются все ранее созданные ячейки памяти для данной родительской ячейки (начала фрейма функции, массива или заданного вызова функции выделения динамической памяти). Если находится такая ячейка памяти *ML*, что для нее $Intersect(S_{ML} + O_{ML}, Val_s + Val_o) \neq \emptyset$, то новая ячейка памяти не создается, а вместо этого обновляются значения смещения и размера старой ячейки: $S_{ML} = Unify(S_{ML}, Val_s), O_{ML} = Unify(O_{ML}, Val_o)$. Это нужно для того, чтобы избежать возможного пересечения ячеек памяти с одним родителем – тогда

при обновлении их абстрактных значений может требоваться просмотр нескольких ячеек. Вместо этого в модели памяти консервативно заводится единственная ячейка, моделирующая все пересекающиеся доступы к памяти. Если же пересекающихся ячеек с тем же родителем нет, то заводится новая ячейка, и ее размер и смещение устанавливаются равными вычисленным абстрактным значениям.

Для всей функции ячейки памяти, абстрактные значения Val и множества $PtTo$ вычисляются с помощью следующего алгоритма.

Алгоритм 2.2. Входные данные: функция *f*=<*A*, *G*>. Выходные данные: $V = \{Val_i\}$, $P = \{PtTo_i\}$ – вычисленные абстрактные значения Val и множества $PtTo$ для всех ячеек памяти функции в каждой точке (операторе) функции; $M = \{Expr \to ML\}$ – соответствие между выражениями, именующими память, и ячейками памяти.

1. Инициализация: положить $M_{Entry} = \{ML_{static}\}$, где ML_{static} – ячейки памяти для глобальных и статических переменных.
2. Структурный анализ: построить дерево управления, состоящее из выделенных в графе потока управления шаблонов вида *Block*, *IfThen*, *IfThenElse*, *While*. Шаблоны определяются так:
   - *Block*: последовательность регионов (участков графа потока управления) с одним входом и одним выходом (т.н. SESE-регионы), в которой каждый регион является единственным последователем предыдущего региона и единственным предшественником следующего региона;
   - *IfThen*, *IfThenElse*, *While*: регионы с одним входом и одним выходом, соответствующие одноименным узлам абстрактного синтаксического дерева.

   Выделение шаблонов проводится напрямую по узлам-операторам АСД: составный оператор соответствует шаблону *Block*, при этом все последовательные операторы, не содержащие переходов управления, состоят в одном базовом блоке и являются одним регионом, а условные операторы и оператор цикла соответствуют шаблонам *IfThen*, *IfThenElse*, *While* и образуют отдельные регионы. Можно выделять регионы и классическим алгоритмом структурного анализа [181].
3. Основной цикл: обойти дерево управления в глубину, узлы на одном уровне обходятся в порядке следования в функции. Обработка конкретных шаблонов выполняется так:

- *Block*: регионы обрабатываются последовательно. Если регион является базовым блоком, то для каждого оператора *Stmt* базового блока последовательно применяются описанные передаточные функции F_{Stmt}, которые обновляют множество M (для каждого встреченного выражения строят ячейки памяти по описанным правилам), а далее строят множества V_{Out}, P_{Out} в точке после оператора по множествам V_{In}, P_{In} в точке до оператора.
- *IfThen*, *IfThenElse*: обрабатываются регионы, соответствующие условию в операторе, регион *Then* и при наличии регион *Else*, далее проводится слияние вычисленных данных с помощью описанных функций *Unify*. А именно, для всех ячеек памяти *ML* $Val_{IfThen_{Out}}(ML) = Unify\left(Val_{Expr=false}(ML), Val_{Then_{Out}}(ML)\right)$, $PtTo_{IfThen_{Out}}(ML) = Unify\left(PtTo_{Expr=false}(ML), PtTo_{Then_{Out}}(ML)\right)$, здесь $Val_{Expr=false}$ соответствует абстрактному значению в случае невыполнения условия в условном операторе. Аналогично в случае шаблона *IfThenElse* объединяются выходные значения для регионов *Then* и *Else*.
- *While*: обрабатывается условие цикла предложенными передаточными функциями. Выбирается число $k \geq 3$ (количество анализируемых итераций цикла), полагается $i = 1$ (номер текущей итерации). Далее:
  - если $i = 1$, то полагается $Val_{Body_{In}} = Val_{Expr=true}, PtTo_{Body_{In}} = PtTo_{Expr=true}$ и выполняется анализ региона, соответствующего телу цикла, *i* увеличивается на единицу;
  - если $i > 1$, то перед началом анализа тела цикла выполняется слияние абстрактных значений и множеств *PtTo* вдоль обратной дуги цикла: для всех ячеек памяти *ML* $Val_{Body_{In}}(ML) = Unify\left(Val_{Body_{Out}}(ML), Val_{While_{In}}(ML)\right)$, $PtTo_{Body_{In}}(ML) = Unify\left(PtTo_{Body_{Out}}(ML), PtTo_{While_{In}}(ML)\right)$; после этого накладываются ограничения из обработки условий тела цикла (т.е. считаются $Val^{i}{}_{Expr=true}$ и $PtTo^{i}{}_{Expr=true}$ для очередной итерации), проходит анализ тела цикла, и i увеличивается на единицу;
  - при $i = k - 1$ перед началом анализа тела цикла выполняется слияние значений и последующее их ограничение из обработки

условия, аналогично предыдущему случаю. Далее выполняется анализ тела цикла, в котором при вычислении абстрактного значения Val_i для ячейки памяти ML сначала вычисление выполняется обычным образом, а затем применяется расширяющий оператор ∇ [182]: $Val_i(ML) = \nabla(Val_{i-1}, Val_i)$, где

$$\frac{Val_{i-1}(ML) = [a, b], Val_i(ML) = [c, d]}{\nabla(Val_{i-1}, Val_i) = [c < a\ ?\ -\infty : a, d > b\ ?\ +\infty : b]}$$

Аналогично при вычислений множеств $PtTo$ выполняется расширение для них, в котором при несовпадении множеств $PtTo_{i-1}$ и $PtTo_i$ устанавливается $PtTo_i = Undef$. Наконец, i увеличивается на единицу;

- при $i = k$ перед началом анализа тела цикла выполняется слияние значений и последующее их ограничение из обработки условия, аналогично предыдущему случаю. Далее происходит последняя итерация анализа, после которой выполняется окончательное слияние результатов анализа тела цикла и значений, приходящих в точку слияния по дуге, минующей цикл: $Val_{While_{Out}}(ML) = Unify\left(Val_{Body_{Out}}(ML), Val_{While_{In}}(ML)\right)$, $PtTo_{While_{Out}}(ML) = Unify\left(PtTo_{Body_{Out}}(ML), PtTo_{While_{In}}(ML)\right)$ для всех ячеек памяти ML. Наконец, вычисляются окончательные значения ${Val^k}_{Expr=true}$ и ${PtTo^k}_{Expr=true}$ как результат анализа цикла.

Теорема 2.2. В предположении отсутствия псевдонимов в параметрах анализируемой функции f алгоритм 2.2 корректно вычисляет абстрактные значения и множества $PtTo$ ячеек памяти, т.е. множества конкретных значений и конкретных целей указателей являются подмножествами соответствующих вычисленных абстрактных множеств.

Для **доказательства** достаточно заметить, что в предложенной модели инициализация прежде не встречавшихся ячеек памяти выполняется в виде создания новой ячейки памяти. Т.к. параметры функции при распределении локальной памяти получат непересекающиеся участки, то данное предположение означает, что между ними не будет псевдонимов. В случае, если параметры имеют указательный тип, их разыменование аналогичным образом приведет к созданию новых непересекающихся (без псевдонимов) ячеек памяти. Абстрактные значения для новых ячеек памяти являются максимально широкими (верхними элементами решеток), тем самым они консервативно корректны.

Далее, при анализе операторов из базового блока последовательно применяются передаточные функции, каждая из которых сохраняет корректность на выходе из оператора при условии корректности значения на входе: это непосредственно следует из свойств абстрактного домена интервалов значений [183], а для множеств *PtTo* в этом легко убедиться непосредственно. Для условных операторов дополнительно используется лишь слияние абстрактных значений вдоль нескольких путей выполнения, которое также сохраняет корректность, поскольку результирующее абстрактное значение консервативно отражает все возможные варианты значений, полученные вдоль отдельных путей. Наконец, при обработке циклов для получения консервативно корректного значения на предпоследней итерации анализа используется оператор расширения, а последняя итерация анализа тела цикла ограничивает абстрактные значения на входе в цикл, играя роль сужающего оператора, корректность которого для домена интервалов также известна [184]. ■

```
(1)   int foo(int x) {
(2)     int a = 7, b = 10, *p;
(3)     if (x > 4)
(4)       p = &a;
(5)     else
(6)       p = &b;
(7)     return *p;
(8)   }
```

Рисунок 2.3. Пример вычисления ячеек памяти и их атрибутов.

Рассмотрим пример кода на рисунке 2.3. Предполагая, что размер целых чисел и указателей составляет 4 байта (32 бита), при анализе функции будут выполнены следующие действия:

- структурный анализ выделит блок из трех регионов: базового блока с выражениями-инициализаторами на строке 2, условного оператора на строках 3-6, в котором в свою очередь есть блок с условием на строке 3, then-блок на строке 4 и else-блок на строке 6, и базового блока с оператором возврата из функции на строке 7;
- при анализе строки 2 создаются ячейки памяти для переменных a и b, для которых родительской ячейкой является фрейм функции, размер – 32 бита, смещение – 32 и 64 бита соответственно (предполагая порядок выделения памяти на стеке – параметры, локальные переменные). Ячейка памяти для указательной переменной

p сначала не создается, т.к. её первое использование наступает на строке 4. Выполняются передаточные функции для присваивания констант: $Val_2(ML(a)) = [7,7], Val_2(ML(b)) = [10,10]$;

- анализ условного оператора начинается с условия на строке 3. Создается ячейка памяти для параметра x (размер – 32 бита, смещение – 0 бит), инициализируется абстрактное значение ячейки как $Val_3(ML(x)) = [-\infty, +\infty]$. При входе в then-блок применяется передаточная функция для условия, ограничивающая интервал значений для x, тем самым $Val_4(ML(x)) = [5, +\infty]$. Далее в строке 4 создается ячейка памяти для переменной p (размер – 32 бита, смещение – 96 бит), ее *PtTo*-множество инициализируется как *Undef*, а после обработки присваивания становится равным $PtTo_4(ML(p)) = \{ML(a)\}$. Аналогично при входе в else-блок применяется передаточная функция для отрицания условия в строке 3, которая дает нам значение $Val_6(ML(x)) = [-\infty, 4]$, в строке 6 новая ячейка памяти для p не создается, но *PtTo*-множество также ставится в *Undef*, а после присваивания становится $PtTo_6(ML(p)) = \{ML(b)\}$. После окончания анализа оператора выполняется слияние значений для ячеек памяти, при котором мы имеем $PtTo_{if}(ML(p)) = Unify\left(PtTo_4(ML(p)), PtTo_6(ML(p))\right) = \{ML(a), ML(b)\}$;
- анализ последнего базового блока заключается в вычислении передаточной функции для оператора возврата из функции в строке 7. Создается специальная ячейка памяти *MLReturn* для возвращаемого значения, в которую копируются абстрактные значения операции разыменования. Таким образом, $Val_7(MLReturn) = Unify(Val_7(ML(a), Val_7(ML(b)) = [7,10]$.

Теорема 2.3. Алгоритм анализа 2.2 завершается и выполняется за время $O(nk^d)$, где n – количество инструкций в функции, k – количество итераций анализа цикла, d – максимальная глубина цикла.

Доказательство непосредственно следует из построения алгоритма анализа – инструкции внешнего цикла из гнезда циклов будут анализироваться k раз, следующего по вложенности – k^2 раз и так далее, а количество инструкций внутреннего цикла заведомо не превосходит n. ■

В распространенных программах глубина циклов обычно невелика. Так, по нашим данным среднее значение метрики вложенности управляющих операторов (включающей в себя и условные операторы) для исходного кода ОС Tizen 3.0 составляет 2.16, что дает возможность предполагать, что обычно анализ будет выполняться за линейное время относительно

количества инструкций функции ($3^{2.16}$≈10.73). Максимальная глубина циклов может значительно превосходить это значение – так, в [185] приведена статистика тестов SPEC CPU INT 2000, из которой следует, что в 5 из 11 тестов максимальная глубина циклов равна 10.

2.2. Межпроцедурный контекстно-чувствительный анализ

Второй уровень анализа требуется для поиска критических дефектов – таких, как переполнение буфера, разыменования нулевого указателя, ошибки управления динамической памятью, утечки ресурсов – с хорошим качеством. Это означает межпроцедурный анализ, обладающий хотя бы некоторой контекстной чувствительностью (т.е. учитывающий влияние контекста вызова функции на ее выполнение) и чувствительностью к путям выполнения (т.е. отслеживающий, при каких именно условиях, направляющих выполнение по разным путям, будут верны те или иные утверждения о потоке данных программы). Тем не менее, как было сказано в начале главы, межпроцедурный анализ без чувствительности к путям является полезным для некоторых детекторов. Поэтому в данном разделе сначала мы опишем внутрипроцедурный анализ на основе классов значений и организацию межпроцедурного анализа, а варианты привнесения в анализ чувствительности к путям обсудим в разделе 2.3.

Модельным языком для наших построений останется тот же язык на рисунке 2.2. Однако внутреннее представление является более низкоуровневым (трехадресным) – в нем сложные выражения языка понижены до четверок <*Op, VarDest, VarLeft, VarRight*>. Для этого вводятся *псевдорегистры* – неадресуемые временные типизированные переменные с единственным определением (обозначаем их как $@Variable_{index}$). Будем считать, что на этом этапе абстрактное синтаксическое дерево уже недоступно, однако по необходимости сохранена высокоуровневая информация (о типах, соответствия инструкций строкам исходного кода, виртуальности вызовов и т.п.). Анализ получает на вход множество функций *F* = { *f*:<*S, G*> }, где *S* – определение функции (имя, тип возвращаемого значения, имена и типы параметров), *G* – граф потока управления функции из описанных инструкций внутреннего представления, и множество глобальных переменных *Glob* = { *g*:<*S,* I> }, где аналогично *S* – определение переменной, т.е. ее имя и тип, *I* – начальное значение переменной. Заметим, что у различных языков могут быть свои аналоги глобальных переменных, например, в Java множество *Glob* будет содержать статические переменные классов, а множество *F* – состоять из методов классов, однако общая схема рассуждений при этом сохранится.

Опишем сначала алгоритм анализа самого верхнего уровня, а потом обсудим его части в дальнейших подразделах. Целью данного алгоритма является выполнить анализ всех функций алгоритмом 2.2 с учетом эффектов от вызовов функций с контекстной чувствительностью, то есть построить модель памяти программы и вычислить основные факты о ее переменных (абстрактные значения, потенциальные цели указателей) для всех точек программы. Этот

алгоритм составляет т.н. *ядро* анализа. Вопросы написания *детекторов* конкретных ошибочных ситуаций с помощью данных, собранных ядром, и дополнительно вычисленных атрибутов рассмотрены в разделе 2.4.

Алгоритм 2.3. Межпроцедурный анализ на основе аннотаций.

1. Построение графа вызовов программы. В графе вызовов вершинами являются анализируемые функции, а дуги показывают связи по вызовам между ними. Бывают дуги двух типов: *прямые*, соответствующие вызовам по имени, и *косвенные*, соответствующие вызовам по указателю (в том числе для виртуальных функций). Требуется выполнить анализ указателей на функцию для построения достаточно точного графа вызовов, чему посвящен раздел 2.2.1.
2. Разрыв циклов. В графе вызовов определяются сильно связанные компоненты, после чего из каждой компоненты удаляется произвольная дуга (если возможно, косвенная). Это нужно для получения ациклического графа, который можно проанализировать за один проход.
3. Межпроцедурный анализ. Обойти граф вызовов в обратном топологическом порядке (от листьев к верхним уровням). Для каждой обходимой вершины:
   3.1. Если вершина соответствует известной функции (для которой имеется внутреннее представление), то проводится внутрипроцедурный анализ этой функции по алгоритмам, описанным далее в разделе 2.2.2. По окончании анализа создается аннотация функции (результаты ее анализа) согласно подходу, описанному в разделе 2.2.3.
   3.2. Если вершина соответствует функции, для которой нет тела, но она является известной анализу (например, функцией стандартной библиотеки), то ее аннотация также предполагается известной. Запись аннотации стандартных функций мы будем называть *спецификацией функции*. Далее в разделе 3.3.3 будут обсуждены возможные способы описания спецификаций.

2.2.1. Построение графа вызовов программы

Для прямых вызовов (не по указателю) построение графа вызовов не представляет сложностей. Достаточно создать вершину в графе для каждой функции и обойти все тела функций для поиска инструкций вызова. Если вызываемая функция также находится в графе вызовов, то между соответствующими вершинами проводится прямая дуга. Если же функции нет, то создается специальная вершина для функции без тела и помечается как неизвестная функция, а потом проводится дуга.

Вызовы по указателю можно трактовать как вызовы неизвестных функций. Однако точность анализа от этого страдает. Целесообразно провести возможно быстрый анализ для

уточнения потенциальных целей косвенных вызовов. Для этого предлагается обойти тела функций, обращая внимания только на инструкции, оперирующие с указателями на функцию – взятие адреса функций, присваивание указателей, вызовы по указателю, присваивание переменных составных типов (массивов, структур), которые содержат указатели на функцию – и вычислить множества *PtTo* раздела 2.1 только для таких указателей. Далее для вызовов функций по указателю проводятся косвенные дуги до всех функций, попавших в соответствующее множество *PtTo*.

Опишем этот алгоритм более детально.

Алгоритм 2.4. Внутрипроцедурный анализ указателей на функцию.

1. Инициализация ячеек памяти для глобальных переменных. Ячейки памяти создаются как описано в разделе 2.1 со следующими изменениями. Во-первых, ячейки создаются только для тех типов, которые являются указателями на функцию, либо для составных типов, имеющих своими частями указатели на функцию. Во-вторых, для массивов снимается ограничение на максимальное количество моделируемых элементов для того, чтобы поддержать часто встречающиеся таблицы указателей (например, обработчиков событий). В-третьих, для имен функций, используемых как указатели, создаются специальные ячейки памяти с классом памяти *Function* и размером, соответствующим размеру указателя.
2. Обход графа потока управления. Выполняется анализ, аналогичный шагу 3 алгоритма 2.2. Обрабатываются только передаточные функции для множеств *PtTo* для инструкций копирования указателей либо составных типов с указателями, взятия адреса функций. Циклы анализируются по 2 итерации, за исключением циклов с константным числом итераций, в которых оперируют элементами массивов с указателями на функцию. Такие циклы анализируются полностью (в количество итераций, равное числу элементов массива) для избежания потери точности.
3. Фильтрация по сигнатуре функции. Для инструкций вызова по указателю из посчитанных множеств *PtTo* удаляются те элементы, которые, вероятнее всего, не могут описывать выполняемый вызов. Это функции, которые не соответствуют вызову по типу либо по количеству параметров. Известно [186], что некоторые программы на Си могут нарушать это правило, и требование строгого соответствия количества параметров и тех же типов может отсеять корректные функции-кандидаты. В таком случае возможно проведение более слабой фильтрации, в которой требуется иметь в точке вызова хотя бы не меньше параметров, чем у функции-кандидата, и требовать битовой (но не структурной) совместимости по типам между формальными и фактическими параметрами.

4. Проведение косвенных дуг. Для инструкций вызова по указателю в обрабатываемой функции проводятся косвенные дуги во все функции, попавшие в соответствующее множество *PtTo*. Если множество пустое или равно *Undef*, то соответствующий вызов помечается как неизвестный.

Приведенный алгоритм выполняет только внутрипроцедурный анализ. Возможно уточнить его результаты с помощью межпроцедурного алгоритма.

Алгоритм 2.5. Межпроцедурный анализ указателей на функцию.

1. Инициализация ячеек памяти для глобальных переменных. Выполняется инициализация, аналогичная шагу 1 алгоритма 2.4.
2. Построение статического графа вызовов. Строится граф вызовов только из прямых дуг (без учета косвенных вызовов). В нем разрываются циклы аналогично шагу 2 алгоритма 2.3.
3. Межпроцедурный анализ. Выполняется снизу вверх в обратном топологическом порядке по графу вызовов аналогично шагу 3 алгоритма 2.3.
   3.1. Внутрипроцедурный анализ. Каждая функция анализируется аналогично шагу 2 алгоритма 2.4. В точке выхода из функции формируется аннотация – запоминаются множества *PtTo* для всех указателей на функцию. Если таких указателей не было, то помечается, что функция была проанализирована, но ее аннотация пуста.
   3.2. Обработка вызовов. При обработке прямого вызова проверяется, была ли сформирована непустая аннотация для вызываемой функции. В этом случае производится сопоставление формальных и фактических параметров: множества *PtTo* формальных параметров объединяются с множествами *PtTo* фактических параметров, при этом в контекст вызывающей функции также переносятся все ячейки памяти, которые доступны косвенно через множества *PtTo*. Эта операция уточняет посчитанные множества для вызываемой функции. Далее сопоставляются множества *PtTo* из аннотации функции и фактического возвращаемого значения. Это, в свою очередь, определяет множества для результата вызова в контексте вызывающей функции.
4. Проведение косвенных дуг. Выполняется фильтрация полученных множеств *PtTo* по сигнатурам функции аналогично шагу 3 алгоритма 2.4, после чего проводятся косвенные дуги согласно шагу 4 того же алгоритма.

Рассмотрим пример выполнения алгоритма 2.5 для исходного кода на рисунке 2.4. В ходе инициализации будут созданы ячейки памяти для массива `callbacks` на строке 14, и для

ячеек их полей `cb` будут также созданы соответствующие множества *PtTo* из ячеек для функций `bar` и `baz`. При обработке функции `get` в ее аннотации ячейка памяти возвращаемого значения на строке 19 получит множество *PtTo*, состоящее из ячейки памяти для функции `baz`. При обработке функции `go` в цикле для выражения `callbacks[i]` будет создана ячейка памяти, объединяющая оба элемента массива, и для нее множество *PtTo* будет включать обе функции. Далее при обработке вызова функции `foo` на строке 23 произойдет сопоставление ячеек памяти, из которого множество *PtTo* для функции `foo` объединится с посчитанным и станет равным { *bar*, *baz* }. В строке 24 при обработке вызова функции `get` ячейка памяти ее возвращаемого значения сопоставится с ячейкой памяти для указателя `t` и получит множество *PtTo* из функции `baz`. Тем самым на шаге 4 алгоритма при проведении косвенных дуг окажется, что функция `foo` может вызвать функции `bar` и `baz` в строке 2, а функция `go` может вызвать функцию `baz` в строке 25.

```
(1)   int foo(int (*p)(int), int y) {
(2)     p (y);
(3)   }
(4)   int bar (int x) {
(5)     return x;
(6)   }
(7)   int baz (int z) {
(8)     return 0;
(9)   }
(10)  struct cb {
(11)    char *name;
(12)    int (*cb)(int);
(13)  }
(14)  struct cb callbacks[2] = {
(15)    "first", &bar,
(16)    "second", &baz
(17)  };
(18)  int (*)(int) get() {
(19)    return baz;
(20)  }
(21)  int go() {
(22)    for (int i = 0; i < 2; i++)
```

```
(23)      foo (callbacks[i].cb, i);
(24)    int(*t)(int) = get ();
(25)    return t();
(27)  }
```

Рисунок 2.4. Пример исходного кода для анализа функций по указателю.

Нужно также отметить, что описанные алгоритмы предполагают, что в случае прямого вызова функции (т.е. обработки инструкции вызова типа *foo*(*x, y*) из функции *bar*) нам всегда известно, какая именно функция из множества предъявленных анализу вызывается. В реальном инструменте для того, чтобы это узнать, необходимо предпринять дополнительные усилия (например, организовывать компоновку внутреннего представления для анализа по аналогии с компоновкой исполняемых файлов исходной программы), о которых будет подробно написано в разделе 3.3.1.

2.2.2. Внутрипроцедурный анализ

Итак, целью внутрипроцедурного анализа является построение для заданной функции программы модели памяти и вычисления основных фактов о ее потоке данных. Моделью памяти для нас будут являться уже описанные в разделе 2.1.2 ячейки памяти. Правила создания ячеек памяти необходимо дополнить для введенных нами псевдорегистров – будем считать, что создается типизированная ячейка памяти со специальным классом памяти (R = *Pseudo*) и инициализированным размером ($S = 8 \times \mathrm{sizeof}(type)$), нулевым смещением и отсутствием родительской ячейки.

Как и прежде, среди основных интересующих нас данных о ячейках памяти, отслеживание которых поручено ядру анализа, будут абстрактные значения – возможные диапазоны значений для целочисленных переменных и потенциальные цели (множества $PtTo$) для указательных переменных. Но в отличие от предыдущего уровня анализа мы будем отслеживать значения с помощью концепции *классов значений* (КЗ). Классы значений предназначены для отслеживания эквивалентности значений переменных, что помогает повысить точность анализа без привлечения SMT-решателей [187, 96]. Поэтому с каждой ячейкой памяти в каждой точке программы ассоциируется класс значений: $VCs: ML \times I \to VC$, который выполняет роль абстрактного значения. Правила присвоения нового класса значений схожи с правилами заведения номера значений в нумерации значений, основанной на хешировании [188] – одинаковый класс значений для двух ячеек памяти означает, что анализ смог доказать, что в них хранится одно и то же значение. Оценки этого значения для

целочисленных и указательных переменных становятся свойствами классов значений, а не самих ячеек памяти.

Классы значений распространяются через инструкции следующим образом:

- Присваивание константы – заводится класс значений, соответствующий константе, и запоминается в хеш-таблице классов значений.

$$\frac{a = const}{VC\big(ML(a)\big) = vc_{const}}$$

- Присваивание переменной – класс значений ячейки памяти для целевой переменной выставляется по классу ячейки для правой части, в хеш-таблицу заносится новый класс значений для целевой ячейки.

$$\frac{a = b}{VC\big(ML(a)\big) = VC\big(ML(b)\big)}$$

- Арифметические операции.

$$\frac{a = b\ op\ c, op \in \{+, -\}}{VC\big(ML(a)\big) = VLookup(op,\ VC\left(ML(b),\ VC\big(ML(c)\big)\right)}$$

  Здесь функция *VLookup* проверяет наличие записи в хеш-таблице о том, что данная операция с данными классами значений уже была вычислена и ей был присвоен класс значений. В случае положительного ответа возвращается тот же класс значений, иначе заводится новый класс (отличный от всех имеющихся) и запоминается в хеш-таблице:

$$VLookup(op, vc_1, vc_2) = \begin{cases} vc_{old}, & \text{если} < op, vc_1, vc_2 > \in ht \\ vc_{new}, & \text{иначе} \end{cases}$$

- Слияние вдоль потока управления – если вдоль различных путей выполнения, пришедших в данную точку, ячейка памяти имела одинаковый класс значения, то он же и остается после слияния, иначе заводится новый класс.

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Pred(b), VC_{b_i}\big(ML(a)\big) = \{VC_j^{b_i}\}}{VC_b\big(ML(a)\big) = Unify(VC_{b_i}\big(ML(a)\big))}$$

  , где $Unify_{i=1}^{n}(vc_i) = \begin{cases} vc_1, & \text{если } vc_1 = vc_2 = \cdots = vc_n \\ vc_{new}, & \text{иначе} \end{cases}$

- Разыменование переменной – выполняется аналогично слиянию.

$$\frac{a = *b, PtTo(b) = \{ML_i\}}{VC\big(ML(a)\big) = Unify_i(VC(ML_i))}$$

Рассмотрим пример на рисунке 2.5, иллюстрирующий применение техники классов значений. Пусть ячейка памяти для x будет иметь класс значения vc_1 (т.е. $VC\big(ML(x)\big) = vc_1$), тогда на строке 2 для переменной a будет заведен класс значения $VC\big(ML(a)\big) = vc_2$. Этот же

класс значений будет заведен и для переменной c на строке 6 – $VC\big(ML(c)\big) = vc_2$. Для указательной переменной p на строках 4 и 7 будут заведены классы значений vc_3 и vc_4 соответственно. На строке 9 будет заведен новый класс значений vc_5 для p, и его множества *PtTo* будут объединены: $PtTo_9\Big(VC\big(ML(p)\big)\Big) = PtTo_9(vc_5) = \{ML(a), ML(c)\}$. При разыменовании ячейки памяти мы сможем установить, что у обеих ячеек из множества *PtTo* одинаковые классы значений vc_2, тем самым функция всегда возвращает одно и то же значение. Если же интервалы значений являются свойством ячеек памяти, то можно только узнать, что у переменных a и c совпадают интервалы, но не они сами – связь между переменными, возникшая из-за цепочки присваиваний, не отслеживалась.

```
(1) int bar (int x) {
(2)   int a = x + 2, *p;
(3)   if (x > 0)
(4)     p = &a;
(5)   else {
(6)     int c = x + 2;
(7)     p = &c;
(8)   }
(9)   return *p;
(10)}
```

Рисунок 2.5. Пример исходного кода для классов значений.

Итак, интересующие анализ свойства ячеек памяти, прежде всего возможные значения, становятся свойствами классов значений и характеризуют не конкретное место в памяти, а значение, которое там хранится. Такие свойства мы будем называть *атрибутами*. Основными атрибутами, вычисляемыми ядром, являются уже упомянутые интервалы значений *Val* и множества потенциальных целей указателей *PtTo*. Передаточные функции для них уже были приведены в разделе 2.1.2 – они остаются прежними с той лишь разницей, что теперь для их получения по ячейке памяти необходимо сначала узнать класс значений, хранящийся в ячейке, а потом по этому классу получить искомый атрибут. Формулы для передаточных функций получают соответствующие механические изменения. Нужно отметить, что в случае слияния классов значений функцией *Unify*, если все классы значений были одинаковыми, то и атрибуты у них одинаковые – оставляется как старый класс значения, так и прежние атрибуты. Иначе заводится новый класс значений, а его атрибуты получаются слиянием атрибутов

соответствующих классов (пример такой ситуации возникал в ходе анализа исходного кода рисунка 2.5). Так, при обработке разыменований указателей в правой и левой части присваивания (загрузка и запись по указателю соответственно) в случае, когда множество *PtTo* содержит единственный элемент, то атрибуты только его класса значений участвуют в вычислениях. То есть если проводится запись, то обновляется только конкретная ячейка памяти и ее класс значений (т.н. *сильное* обновление). Если же элементов несколько, то при обработке инструкции вида $*p = expr$ выполняется *слабое* обновление – для каждого класса значений, который связан с ячейками памяти – потенциальными целями указателя p – выполняется объединение функцией *Unify* со значением, полученным из *expr*.

Так, на рисунке 2.6 в строке 7 множество *PtTo* будет содержать два элемента, ячейки памяти для переменных a и b. При обновлении посчитанных классов значений при записи по указателю в этой строке для переменной a будут объединены ее класс значений и класс значений для константы 5. Т.к. они одинаковы, то класс значений для a по результатам записи не изменится. А для переменной b будет выполнено объединение, который построит новый класс значений, а интервал для этого класса будет получен как объединение предыдущего и нового интервала, т.е. будет равен [5, 6]. Это требуется, чтобы показать, что после присваивания значение b могло не поменяться, так как указатель p может указывать и на другую переменную.

```
(1) void bar (int x) {
(2)   int a = 5, b = 6, *p;
(3)   if (x > 2)
(4)     p = &a;
(5)   else
(6)     p = &b;
(7)   *p = 5;
(8) }
```

Рисунок 2.6. Слабое обновление.

Классы значений и их атрибуты вычисляются следующим алгоритмом, который является вариантом алгоритма 2.2.

Алгоритм 2.6. Внутрипроцедурный анализ функции с применением классов значений. Входом алгоритма является функция *f*:<*S*, *G*>. Выходом являются посчитанные функции модели памяти $M: LValue \to ML$, классы значений и их атрибуты $VCs: ML \times I \to VC$,

$Attrs: VC \times AttrName \to Attr$ ($AttrName \in \{Val, PtTo\}$, другие атрибуты могут быть добавлены детекторами, как описано в разделе 2.4).

1. Инициализация глобальных ячеек памяти предполагается выполненной межпроцедурным алгоритмом (соответствующий шаг 1 алгоритма 2.2 пропускается).
2. Выделение регионов выполняется обычным структурным анализом (в шаге 2 алгоритма 2.2 используется абстрактное синтаксическое дерево).
3. Обход функции по дереву управления выполняется аналогично шагу 3 алгоритма 2.2 со следующими отличиями:
   3.1. Новые классы значений создаются для новых ячеек памяти.
   3.2. При обработке инструкций сначала обновляются классы значений для ячеек памяти, а потом обновляются их атрибуты через передаточные функции (приведенные в разделе 2.1.2 с учетом упомянутых изменений).
   3.3. При обработке ветвлений и циклов перед слиянием атрибутов (интервалов значений и множеств $PtTo$) сначала сливаются классы значений, а потом их атрибуты, как было описано в алгоритме 2.2.
   3.4. Расширяющий оператор ∇ на предпоследней итерации цикла по умолчанию не применяется (чтобы получить более точные результаты анализа первых k итераций цикла, при этом, возможно, не учтя эффекты поздних итераций). Возможен также вариант алгоритма, когда оператор применяется так же, как и в алгоритме 2.2.

Теорема 2.4. В предположении отсутствия псевдонимов в параметрах анализируемой функции f алгоритм 2.6 корректно вычисляет классы значений, абстрактные значения и множества $PtTo$ ячеек памяти (т.е. множества конкретных значений и конкретных целей указателей являются подмножествами соответствующих вычисленных абстрактных множеств, а классы значений совпадают только тогда, когда совпадают конкретные значения соответствующих ячеек памяти):

- в случае, когда расширяющий оператор не применяется – для всех путей выполнения через функцию, проходящих по циклам не более k раз, где k – количество анализируемых итераций;
- в случае, когда расширяющий оператор применяется – для всех путей.

Доказательство аналогично доказательству теоремы 2.2 с той разницей, что при отсутствии расширяющего оператора анализ не пройдет по путям, которые итерируются по циклам более чем k раз, тем самым вычисленные абстрактные значения могут не отражать конкретных значений, принимаемых переменными на этих путях. Совпадение классов

значений происходят только тогда, когда данные значения были получены с помощью одной и той же цепочки вычислений (что гарантируется процедурой выбора классов значений при обработке инструкций), при этом в случае слияния потока управления эти значения были одинаковы по всем путям выполнения (иначе был бы сгенерирован новый класс значения). ■

2.2.3. Вычисление и использование аннотаций функции

Общепринятым способом организации масштабируемого межпроцедурного анализа является использование аннотаций функций. Смысл аннотации в компактной записи, во-первых, влияния вызываемой функции на внешнюю память (статически и динамически выделенную), и, во-вторых, влияния контекста вызова функции на ее выполнение. При обработке вызова функции используется созданная аннотация, повторного анализа функции не происходит, что позволяет достичь масштабируемости.

Для построения аннотаций необходимо разрешить следующие вопросы:

- что попадает в аннотацию функции в результате анализа,
- каким образом аннотация используется при анализе вызова функции,
- степень контекстной чувствительности, достигаемой аннотациями (т.е. сколько контекстов вызова описывает конкретная аннотация – например, одна аннотация на все контексты, по аннотации на каждый различающийся контекст, либо некоторый компромисс). Этот вопрос по сути является ключевым – ответ на него задает конкретные алгоритмические решения для остальных вопросов.

Выбирая степень контекстной чувствительности анализа, заметим, что создание отдельной аннотации для каждого контекста (каждого пути в графе вызовов) приводит к экспоненциальному росту числа аннотаций. Использование отдельной аннотации для некоторой группы контекстов (например, популярное решение создавать аннотацию для каждой группы путей в графе вызовов, у которых последние $k \geq 2$ вызовов совпадают) – это способ формально описать интуитивное представление о поведении программы: более вероятно, что вызываемая функция будет вести себя по-разному для “значительно” различающихся контекстов. Но зададимся вопросом – что даёт для проводимого нами анализа создание отдельной аннотации для некоторого контекста? Очевидно, что хочется добиться более точного вычисления абстрактных значений и других атрибутов ячеек памяти, создаваемых и модифицируемых функцией. Интуитивно это может происходить в том случае, когда на вход функции подаются различные значения, учет которых при вычислениях приводит к различным ответам для оцениваемых атрибутов. Если удается *параметризовать* влияние контекста вызова в аннотации, то есть выразить зависимость вычисленных значений от входных значений функции, то такая параметризованная аннотация может одна успешно

заменить множество вычисленных обычным способом аннотаций для различных контекстов вызова.

Формализуем это соображение, введя понятие внешних ячеек памяти и предложив учитывающий их алгоритм внутрипроцедурного анализа. *Внешними* ячейками памяти *EML* для данной функции будут ячейки памяти, соответствующие формальным параметрам функции и глобальным переменным (и другой статической памяти). Аналогично, *внешними классами значений* назовем те классы значений, которые хранятся во внешних ячейках памяти перед началом анализа функции: $EVC \in VCs, EVC = VC(EML, f_{Etnry})$. *Внешними абстрактными значениями* (интервалами значений, множествами *PtTo* либо атрибутами детекторов) назовем абстрактные значения внешних классов значений. *Фактическим* значением в данном случае будем называть абстрактное значение, приближающее значение параметра функции или глобальной переменной в данном контексте вызова.

Модифицируем передаточные функции для абстрактных значений следующим образом. Каждый раз, когда в вычислениях передаточной функции участвует внешнее абстрактное значение, и соответствующая функция является ассоциативной, вместо применения конкретного значения по умолчанию символьно подставляется внешнее абстрактное значение. Если же передаточная функция не ассоциативна, то подставляется конкретное значение по умолчанию. Например, если вычисляется передаточная функция для выражения `x+1`, где `x` – это параметр функции, то вместо вычисления интервала $[-\infty, +\infty] + [1,1] = [-\infty, +\infty]$ записывается интервал $Val(EVC(EML(x))) + [1,1]$. Дальнейшее применение передаточных функций к таким значениям выполняет вычисления над конкретными интервалами, если это возможно, а символьные вхождения внешних значений оставляет нетронутыми (например, последующее вычисление `x+2` приводит к интервалу $Val(EVC(EML(x))) + [3,3]$).

Идея данного подхода заключается в том, чтобы сохранить все операции, проведенные передаточными функциями над внешними абстрактными значениями, подставляя их символьно, вместо того, чтобы потерять точность представления значений, оперируя неизвестными значениями по умолчанию. Тогда при сохранении аннотации функции свойства соответствующих классов значений будут вычислены не до конца, а выражены символьно через внешние абстрактные значения. Далее, при обработке аннотации в конкретном контексте вызова символьное внешнее абстрактное значение заменяется на фактическое, и производится довычисление значений, чтобы как можно более точно охарактеризовать наши классы значений и ячейки памяти в точке вызова.

Из приведенных рассуждений непосредственно следует

Теорема 2.5. Если алгоритм 2.6 используется с передаточными функциями, в которых внешние абстрактные значения применяются символьно, и все передаточные функции с

внешними значениями и функции объединения значений (*Unify*) являются ассоциативными, то при применении аннотации функции, подставляя фактическое значение на место внешнего, после вычислений будет получено то же значение, которые было бы вычислено при изначальном применении алгоритма 2.6 при задании этого фактического значения как исходного.

Для **доказательства** достаточно заметить, что окончательные значения, сохраненные в аннотации, будут выражены через некоторую комбинацию $f_1, f_2, \dots, f_n$ передаточных функций и функций объединения, в которых некоторые параметры будут внешними значениями, а некоторые – конкретными значениями, в свою очередь вычисленные над конкретной частью сохраненных значений. Т.к. все примененные функции являются ассоциативными, то изменение порядка их вычисления (сначала – только над конкретной частью значений, потом – над фактическими значениями, подставленными вместо внешних) не влияет на окончательный ответ по сравнению со способом его вычисления в естественном порядке анализа (как если бы все значения были известны заранее). ■

Заметим, что функции объединения для интервалов значений и множеств *PtTo* являются ассоциативными, тем самым введение внешних классов значений помогает восстановить потери точности, происходящие при обработке точек слияния потока управления.

Теперь опишем алгоритмы создания и применения аннотации, учитывая, что внутрипроцедурный анализ выполняется алгоритмом 2.6 с привлечением внешних ячеек памяти и классов значений. Аннотация функции по сути является частью отображения *M* и *VCs* в точке возврата из функции. Однако сохранение всего отображения будет содержать избыточное количество информации – данные, характеризующие локальную для функции память, будут неактуальны в контексте вызова, т.к. вызывающая функция никак не может их использовать. Требуется сохранить только классы значений и их атрибуты для ячеек памяти, которые могут быть доступны через внешние ячейки – те ячейки, которые либо являются внешними, либо дочерними ячейками по отношению ко внешним, либо входят в множества *PtTo* для внешних ячеек через один либо несколько уровней косвенности. Как правило, количество сохраняемых уровней косвенности ограничивают по аналогии с ограничением на точное моделирование элементов массивов в разделе 2.1, чтобы избежать чрезмерного роста аннотации. Эти соображения дают нам следующий

Алгоритм 2.7. Построение аннотации функции. Вход: вычисленные отображения *M* и *VCs*, константа $md \geq 3$. Выход: построенное резюме $Ann \in VCs$.

1. Добавить в *Ann* внешние ячейки памяти, ячейки памяти для возвращаемых значений и их классы значений в точке возврата из функции, положить $iter=0$.
2. Для всех ячеек памяти *ML*, добавленных в *Ann* и не обработанных ранее:

- Пометить ячейку *ML* как обработанную.
- Добавить в аннотацию все ячейки памяти, для которых *ML* является родительской ячейкой, и их классы значений в точке возврата из функции, пометить все эти ячейки как обработанные.
- Если *ML* соответствует указательному типу и имеет непустое множество *PtTo,* то добавить в аннотацию все ячейки из множества *PtTo*(*ML*) и их классы значений в точке возврата из функции, пометить эти ячейки как необработанные.

3. Положить *iter*=*iter*+1. Если *iter*=*md,* то остановиться, иначе перейти к шагу 2.

Для применения аннотации необходимо построить отображение между ячейками памяти и классами значений, с одной стороны, из контекста вызывающей функции, с другой стороны,– сохраненными в резюме. Процесс построения аналогичен алгоритму 2.7 и заключается в параллельном разворачивании ячеек памяти по уровням косвенности в сохраненной аннотации и в контексте вызова и сопоставлении этих ячеек друг другу. На первом шаге алгоритма внешние ячейки памяти из аннотации сопоставляются с ячейками памяти фактических параметров, глобальных переменных и возвращаемых значений в контексте вызова. На следующих итерациях по уровням косвенности сопоставляются дочерние ячейки памяти, для ячеек памяти из множества *PtTo* в вызывающую функцию переносятся новые ячейки, созданные в вызываемой. Если ячейка памяти, для которой ищется сопоставляемая ячейка в аннотации, уже была поставлена в соответствие некоторой ячейке в вызывающей функции, то их классы значений и их атрибуты объединяются функциями *Unify* (в этом случае оказывается нарушенным предположение об отсутствии алиасов между параметрами функции, которое делалось в ходе выполнения межпроцедурного анализа алгоритмом 2.6). После окончания процесса сопоставления происходит перевычисление атрибутов классов значений (интервалов значений и множеств *PtTo*), которые зависели от внешних значений, подставлением их фактических значений и повторным применением передаточных функций и функций объединения.

Рассмотрим пример исходного кода на рисунке 2.7. При анализе функции *baz* создаются внешняя ячейка памяти для переменной p и обычная для *p, внешний класс значений $VC\big(ML(*p)\big) = vc_1$. Далее на строках 3 и 5 классы значений для $ML(*p)$ обновляются (vc_2 и vc_3 соответственно), а в интервалах значений появляется символьное внешнее значение $Val(vc_1)$. Если применяется изначальный алгоритм внутрипроцедурного анализа, то при слиянии интервалов на строке 6 мы получим неопределенный интервал. В нашем же случае в создаваемой аннотации появится интервал $Unify(Val(vc_1), [2,3])$. Сопоставляя при обработке вызова функции *baz* в строке 9 конкретное значение интервала [7,7] внешнему интервалу и

довычисляя окончательное значение, получим, что классу значений для ячейки памяти переменной *k* сопоставлен интервал значений [9,10]. Этот класс значений присваивается ячейке памяти для *k*.

```
(1)   int glob;
(2)   void baz (int *p) {
(3)     *p += 2;
(4)     if (glob > 3)
(5)       (*p)++;
(6)   }
(7)   void faz () {
(8)     int k = 7;
(9)     baz (&k);
(10)      ...
```

Рисунок 2.7. Применение аннотации.

2.3. Межпроцедурный анализ с чувствительностью к путям

На третьем уровне анализа выполняется поиск критических ошибок общего характера (переполнения буфера, утечки ресурсов, разыменования нулевого указателя и т.п.), для которых можно назвать справедливыми следующие два утверждения. Во-первых, такие ошибки сложно искать с помощью «узкоспециализированных» шаблонов в коде, которыми пользуется первый уровень анализа – можно найти ряд вариантов появления таких ошибок, но общие ситуации будут все время ускользать от анализатора (конечно, при условии сохранения высокого уровня истинных срабатываний). Например, анализ на уровне АСД может искать специфические переполнения буфера на один элемент (т.н. off-by-one) при использовании функции `strncat` и известных во время компиляции размеров и длин строк, но не сможет найти переполнение, возникшее при неверном доступе к обычному массиву из-за неверного вычисления неконстантного индекса. Во-вторых, эти ошибки случаются лишь на небольшой доле путей выполнения, тем самым для достижения хорошего качества анализа нельзя консервативно выдавать ошибки при возможно несовместных условиях переходов. Тем самым поддержка чувствительности к путям в анализе становится неизбежной.

2.3.1. Чувствительный к путям анализ с классами значений

Для адаптации анализа раздела 2.2 для чувствительности к путям необходимо рассмотреть следующие изменения:

- Построение предиката пути. Для каждой точки программы внутрипроцедурный анализ будет строить булеву формулу, которая является истинной при условии достижения потоком выполнения этой точки. Предикат пути используется в дальнейшем как часть формулы, совместность которой необходимо проверить для выдачи сообщения об ошибке, и для отслеживания предикатов атрибутов, в том числе основных – множеств *PtTo* и интервалов значений.
- Доработка модели памяти. Для ячеек памяти, составляющих множества *PtTo,* дополнительно будут отслеживаться предикаты, при условии истинности которых указательная переменная будет указывать на данную ячейку памяти. Для интервалов значений сохранится консервативное отслеживание через функцию *Unify* для вычисления неконстантных смещений в ячейках памяти.
- Отслеживание предикатов для классов значений. При объединении классов значений кроме выделения нового класса записываются предикаты, при которых данный класс значений совпадает с некоторым другим. Эти предикаты можно использовать в дальнейшем для уточнения возможного значения, приписанного данному классу значений.
- Отслеживание предикатов для атрибутов детекторов. Если детектор отслеживает некоторый атрибут класса значения, то при прохождении анализом некоторой инструкции детектор может, аналогично предикату пути, конструировать формулу, истинность которой означает, что атрибут принимает заданное значение. Атомами этой формулы являются классы значений, выступающие как символьные переменные (подробнее см. раздел 2.4).

  Основная логика чувствительного к путям анализа заключается в том, каким именно образом конкретный детектор строит формулу ошибки (предикат пути строится анализом централизованно для всех детекторов). В отличие от анализа раздела 2.2, здесь алгоритм детектора играет большую роль, чем алгоритмы ядра.

Итак, обсудим детальнее упомянутые доработки анализа. Предикат пути *Curr* отслеживается дописыванием к нему элементарных формул сравнений классов значений и объединения отдельных предикатов путей в точке слияния потока управления следующим образом:

- При входе в функцию $Curr = true$.

- При переходе через условный оператор вида $if\ (a == b)$ в базовый блок, соответствующий истинному значению условия, предикат модифицируется как $Curr \wedge (VC(a) == VC(b))$, а в блок, соответствующий ложному значению, – $Curr \wedge \neg (VC(a) == VC(b))$.
- При слиянии потока управления предикатов $Pred_1, Pred_2, \dots, Pred_n$ из базовых блоков-предшественников текущего предикат пути записывается как их дизъюнкция $Corr = Pred_1 \vee Pred_2 \vee \dots \vee Pred_n$ и упрощается последовательным применением тождеств вида $a \vee \neg a = true$, $a \wedge \neg a = false$, $a \vee a = a$, $a \wedge a = a$.

Модель памяти программы, использующая ячейки памяти, остается неизменной. Для отслеживания неконстантных смещений дочерних ячеек памяти по отношению к родительским по-прежнему требуется определение целочисленных значений переменных, которое следует описанию раздела 2.2.2. Увеличивается точность отслеживания множеств *PtTo*: для каждой ячейки памяти ML_i, хранящейся во множестве, дополнительно отслеживается предикат $Pred_i$, который является истинным тогда, когда ячейка памяти этого множества указывает на данную ячейку ML_i. Предикаты состоят из выражений над классами значений и строятся непосредственно по следующему набору правил:

- При обработке присваивания и арифметических операций применяются обычные передаточные функции, а предикаты элементов множества не изменяются;
- При инициализации множества *PtTo* ячейкой памяти *ML* в ходе обработки оператора взятия адреса или при инициализации специальным значением *Undef* предикатом элемента становится текущий предикат пути *Curr*:

$$\frac{a = \&b}{PtTo\big(ML(a)\big) = \{Curr : ML(b)\}}$$

- При условном переходе по сравнению двух указателей на неравенство применяются обычные передаточные функции: множества меняются только тогда, когда одно из них выражает must-отношение, предикаты элементов не меняются. При переходе по сравнению на равенство, как и ранее, строится пересечение множеств, и дополнительно предикаты одинаковых элементов объединяются через конъюнкцию:

$$\frac{if\ (p == q), PtTo(ML(p)) = \{Pred_i : ML_i\}, PtTo(ML(q)) = \ \{Pred_j : ML_j\}}{PtTo_{p==q}\big(ML(p)\big) = PtTo_{p==q}\big(ML(q)\big) = PtTo(p) \cap PtTo(q)}$$
$$\frac{ML_k \in PtTo_{p==q}, ML_k = ML_{i \in} PtTo(ML(p)), ML_k = ML_{j \in} PtTo(ML(q))}{Pred_k = Pred_i \wedge Pred_j}$$

- При разыменовании указательной ячейки памяти и при слиянии потока управления множества *PtTo*, как и ранее, объединяются через функцию *Unify*. При этом нет необходимости в специальной обработке элемента *Undef*, т.к. теперь

условие, при котором ячейка указывает на неизвестный элемент, отслеживается соответствующим предикатом: $Unify(PtTo_a, PtTo_b) = PtTo_a \cup PtTo_b$. При построении множеств *PtTo* необходимо обеспечить однократность вхождения в него каждой ячейки памяти, поэтому предикаты соответствующих ячеек объединяются дизъюнкцией.

$$\frac{a =* b, PtTo(b) = \{Pred_i : ML_i\}}{PtTo\big(ML(a)\big) = Unify_i\big(PtTo(ML_i)\big)}$$

$$\frac{ML_k \in PtTo\big(ML_{i_1}\big), ML_k \in PtTo\big(ML_{i_2}\big), \dots, ML_k \in PtTo\big(ML_{i_n}\big)}{Pred_k = \bigvee_{j=1}^{n} Pred_{i_j}}$$

Дополнительно при обработке разыменования можно использовать следующее соображение: передавать функции *Unify* только те множества *PtTo*(*ML*), для ячеек памяти которых хранящиеся предикаты *Pred* совместны с предикатом пути *Curr*. Тем самым варианты отношения указывания, которые не реализуемы на текущем пути выполнения, будут отсечены.

- При слиянии потока управления выполняется аналогичное объединение множеств *PtTo*, но к предикатам элементов множества дополнительно дописывается предикат пути блока, из которого пришло данное множество, а потом они объединяются дизъюнкцией:

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Preds(b), PtTo_{b_i}\big(ML(a)\big) = \{Pred_j^{b_i} : ML_j^{b_i}\}}{PtTo_b\big(ML(a)\big) = Unify_i\big(PtTo_{b_i}\big(ML(a)\big)\big)}$$

$$\frac{ML_k \in PtTo\big(ML_{i_1}\big), ML_k \in PtTo\big(ML_{i_2}\big), \dots, ML_k \in PtTo\big(ML_{i_n}\big) \quad ML_{i_j} : b_j}{Pred_k = \bigvee_{j=1}^{n} (Pred_{i_j} \wedge Curr_{b_j})}$$

Предикаты, отслеживающие, из каких классов значений был сформирован данный класс при разыменованиях и слияниях потока управления, создаются аналогичным образом. Сохраняются предикаты, истинность которых означает, что данный класс значения равен некоторому другому, т.е. хранится множество $VCPreds(VC) = \{Pred_i : VC_i\}, \forall i, j\ i \neq j \vDash VC_i \neq VC_j$. При разыменовании предикаты из множества *PtTo* дописываются через конъюнкцию к предикатам классов значений из *VCPreds*, а потом объединяются через дизъюнкцию. Аналогично происходит при слиянии потока управления, только используются предикаты пути из предшественников текущего базового блока. Сама передаточная функция, генерирующая новый класс значения, не меняется:

$$\frac{a =* b, PtTo(b) = \{Pred_i : ML_i\}, VCPreds(VC_i) = \{Pred_j^i : VC_j\}}{VC\big(ML(a)\big) = Unify_i(VC(ML_i))}$$

$$\frac{VC_k \in VCPreds(VC_1), VC_k \in VCPreds(VC_2), \dots, VC_k \in VCPreds(VC_n)}{Pred_k = \bigvee_{j=1}^{n}(Pred_k^j \wedge Pred_j)}$$

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Pred(b), VC_{b_i}\big(ML(a)\big) = \{Pred_j^{b_i} : VC_j^{b_i}\}}{VC_b\big(ML(a)\big) = Unify(VC_{b_i}\big(ML(a)\big))}$$

$$\frac{VC_k \in VCPreds\big(VC_{b_1}\big), VC_k \in VCPreds\big(VC_{b_2}\big), \dots, VC_k \in VCPreds\big(VC_{b_n}\big)}{Pred_k = \bigvee_{j=1}^{n}(Pred_k^{b_j} \wedge Curr_{b_j})}$$

Соображение об отсечении несовместных конъюнкций из окончательной дизъюнктивной формулы может применяться и здесь. Например, пусть обрабатывается инструкция `k = *p`, где $PtTo\big(ML(p)\big) = \{vc_x > 0 : a, vc_x \leq 0 : b\}$, при этом $VCPreds\big(ML(a)\big) = \big\{vc_x > 5 : vc_1, vc_y = 2 : vc_2\big\}$, $VCPreds\big(ML(b)\big) = \big\{vc_x = 7 : vc_1, vc_y > 5 : vc_2, vc_z = 0 : vc_3\big\}$. Тогда обычная передаточная функция заведет новый класс значений vc_k для ячейки памяти переменной `k`, а предикаты для составных частей этого класса значений будут записаны как $VCPreds(vc_k) = \{vc_x > 5 : vc_1, vc_x > 0 \wedge vc_y = 2 \vee vc_x \leq 0 \wedge vc_y > 5 : vc_2, vc_x \leq 0 \wedge vc_z = 0 : vc_3\}$.

Для внутрипроцедурного анализа, создающего модель памяти, классы значений и их атрибуты, можно применять уже описанные в разделах 2.1 и 2.2 алгоритмы 2.2 и 2.6 соответственно. Действительно, передаточные функции, создающие множества *PtTo* и классы значений, остаются прежними за исключением того, что из множеств *PtTo* удаляются элементы с несовместными предикатами. Однако по построению предиката пути по индукции легко видеть, что, во-первых, генерируемая логическая формула корректно отражает путь выполнения, соответствующий переходам по индивидуальным подформулам. Во-вторых, при выяснении совместности формулы все классы значений трактуются как свободные переменные – единственное ограничение, которое может потенциально нарушить корректность, это то, что все вхождения конкретного класса значений *VC* в формулу обозначают одно и то же значение. Но корректность выбора классов значений алгоритмом 2.6 гарантируется теоремой 2.4 (в заданных условиях просмотра ограниченного числа путей и отсутствия алиасов у параметров функции), а мы не меняем этот алгоритм – введение предикатов состава данного класса значений не влияет на выбор. Поэтому построение множеств *PtTo* и классы значений является корректным, а корректность созданных предикатов следует из корректности построения предиката пути. Тем самым **Доказана**

Теорема 2.6. Алгоритмы 2.2 и 2.6, вычисляющие модель памяти функции и классы значений при внутрипроцедурном анализе с описанными выше изменениями (обозначим их 2.2' и 2.6'), работают корректно в ограничениях теорем 2.2 и 2.4, и предикаты для классов значений и вычисляются правильно.

Осталось обсудить изменения, которые происходят во время межпроцедурного анализа в части создания и использования аннотации функции. Алгоритм 2.7 выделения ячеек памяти и

классов значений, которые попадут в аннотацию, не изменяется, однако дополнительно сохраняются предикаты, вычисленные для множеств *PtTo* и классов значений (*VCPreds*). Кроме того, вводится ограничение *ml* на максимальное количество конъюнкций, которые содержатся в предикате. Если это количество превзойдено, то части «лишних» конъюнкций, которые были добавлены при обработке предиката пути, меняются на истину (другими словами, удаляются из формулы). При этом предикат становится более слабым, но сохраняет корректность – если предикат до удаления был истинен при некоторых значениях свободных переменных, то и после удаления он останется истинным. Будем обозначать этот алгоритм как алгоритм 2.7'.

Внешние классы значений используются по аналогии с теоремой 2.5, но передаточные функции множеств *PtTo* и других абстрактных значений теперь учитывают предикаты, как описано выше. При этом, т.к. исходные передаточные функции являются ассоциативными, и конъюнкция/дизъюнкция предикатов также ассоциативны, то требования теоремы 2.5 выполняются. Наконец, при использовании аннотации алгоритм сопоставления ячеек памяти и классов значений также остается неизменным.

Снова рассмотрим пример кода на рисунке 2.7. Создаются все те же классы значений, включая внешние, но в точке выхода из функции в строке 6 дополнительно записывается, что $VCPreds(vc_{ret}) = \{vc_{glob} > 3 : vc_3, vc_{glob} \leq 3 : vc_2\}$. При применении аннотации для класса значения vc_k получим аналогичное соотношение. Если интервал значений как атрибут класса значения подвергнуть той же процедуре отслеживания предикатов (см. далее раздел 2.4), то получим, что $Val(vc_k) = \{vc_{glob} > 3 : Val(vc_3) = [10,10], vc_{glob} \leq 3 : Val(vc_2) = [9,9]\}$. Можно объединить эти ответы и получить прежнее значение [9,10].

2.3.2. Чувствительность к путям на основе символьных выражений

Описанный выше в разделе 2.3.1 анализ расширяет анализ предыдущего уровня на основе классов значений, которые остаются центральным понятием анализа, и условия того, что ячейка памяти содержит определенное значение, формулируются именно на основе классов значений. Однако с повышением уровня анализа можно заметить, что ценность классов значений как удобного способа отследить эквивалентность значений между переменными снижается. Во-первых, те же самые эквивалентности можно отследить и через запросы к SMT-решателю, если отразить все связи между классами значений через присваивания и выполнения операций в формуле запроса. Действительно, теория равенств и неинтерпретируемых функций (QF_UF, equality and uninterpreted functions) в SMT-решателях может быть использована для проверки утверждений, которые делаются на основе классов значений: распространение одинаковых классов значений через присваивания аналогична

аксиомам равенств, а назначение классов через поиск ранее записанных в хеш-таблице эквивалентностей по сути использует операции над классами значений как неинтерпретируемые функции. Тем самым, если SMT-решателю над теорией QF_UF предъявить два выражения над классами значений, то он может ответить на вопрос, эквивалентны они или нет (т.е. являются ли они одним и тем же классом значений).

Во-вторых, при межпроцедурном анализе с использованием внешних значений уже делается шаг в сторону символьного выполнения, так как внешние значения при вычислениях внутрипроцедурного анализа трактуются как символьные переменные, а их конкретизация происходит при применении аннотации вызываемой функции в вызывающей (см. раздел 2.2.3).

В-третьих, опыт реализации детекторов показывает, что, несмотря на то, что основные вычисляемые атрибуты целочисленных значений и множеств *PtTo* нужны большинству детекторов, эквивалентность значений через равенство классов значений может быть не такой, как требуется детектору. Другими словами, одинаковые классы значений могут иметь разные атрибуты в разных точках программы, хотя значения у них будут совпадать – различной будет информация, которая известна анализу об этом значении. Например, переходя по условию `if (a == 0)`, анализу становится известно, что класс значения, соответствующий ячейке памяти для переменной `a`, внутри этого условного оператора имеет нулевое значение. Однако сам класс значения не изменился, так как записи в переменную не было. В другом примере приведения типа к беззнаковой переменной `int x = -5; unsigned u = (unsigned) x;` классы значений для ячеек памяти переменных `x` и `u` с точки зрения Си совпадают – их битовые представления одинаковы. Однако нам известно, что значение `u` неотрицательно, а `x` – нет. Можно завести новый класс значений для `u`, однако тогда детекторы, интересующиеся значениями конкретных бит, не смогут узнать об одинаковости этих бит без отслеживания дополнительных атрибутов.

Из сказанного естественно вытекает идея организации чувствительного к путям анализа, отличная от подхода раздела 2.3.1. Откажемся от классов значений и будем ставить в соответствие ячейкам памяти напрямую символьные выражения *se*, которые в них хранятся. Такими выражениями будут константы, символьные переменные, одноместные и двуместные операции над символьными выражениями, которые поддерживаются в языке. Отслеживаются предикаты, при истинности которых символьное выражение равно некоторому другому (аналогично классам значений).

Задачей анализа будет определение того, какие символьные выражения хранятся в ячейках памяти программы в каждой ее точке. Дополнительных атрибутов символьных выражений вычислять не будем – вместо этого каждый детектор должен будет строить собственную формулу выполнимости ошибочной ситуации с использованием предикатов над

символьными выражениями (как описано далее в разделе 2.4.2). Совместность формулы будет проверяться с помощью SMT-решателей.

Для эффективного использования такого подхода необходимо также пересмотреть способ построения модели памяти. Если анализируется язык, допускающий адресную арифметику и произвольные смещения (неизвестные статически) дочерних ячеек памяти относительно родительских, то полноценная поддержка символьной памяти будет требовательна к ресурсам. Поэтому целесообразно ограничить применение подхода данного раздела к языкам, в которых указателями нельзя управлять произвольно – например, к Java или C#. Тогда для обработки ситуации, в которой некоторая ячейка памяти может указывать на несколько других, будет достаточно вычислять множества *PtTo*, а интервалы целочисленных значений *Val* можно будет не поддерживать. Для массивов мы по-прежнему будем создавать не более чем *MaxSize* элементов массива (случай 6 создания ячеек памяти в разделе 2.1.2) и моделировать символьный доступ к массиву как последовательность не более чем *MaxSize* условных операторов (ite): `a[i] = i == 0 ? a[0] : (i == 1 ? a[1]: ... (i == MaxSize ? a[MaxSize] : a[undef]) ... )`. Внутри циклов можно отдельно использовать анализ индукционных переменных, показывающий возможные значения линейных выражений с участием этих переменных, для ограничения количества условных операторов.

Итак, ограничим модельный язык рисунка 2.2, запретив арифметические операции с указателями и взятие адреса произвольной ячейки памяти. Указатель становится подобен ссылке на объект, которая может указывать на несколько объектов в зависимости от пути выполнения, но которую можно только присваивать и сравнивать на равенство, а также разыменовывать. Будем вычислять множества *PtTo* в чувствительном к путям выполнения варианте раздела 2.3.1, отслеживая предикаты, при выполнении которых ячейка памяти указывает на ту или иную другую ячейку.

Передаточные функции, обновляющие символьные выражения для ячеек памяти, будут схожи с функциями для классов значений из раздела 2.3.1. Применение двуместных операций к символьным выражениям с предикатами приводит к попарному пересечению предикатов и применению операции к символьным выражениям-частям. При этом несовместные предикаты удаляются из результирующего множества, а предикаты одинаковых выражений объединяются через дизъюнкцию. Аналогично при разыменовании указательной ячейки памяти предикат, при истинности которого ячейка указывает на другую ячейку, приписывается к соответствующим предикатам частей символьного выражения, результирующее множество частей символьных выражений и их предикатов упрощается так же. При слиянии потока управления приписываются соответствующие предикаты путей для базовых блоков-предшественников текущего блока.

$$\frac{a = const}{SE\big(ML(a)\big) = const}$$

$$\frac{a = b}{SE\big(ML(a)\big) = SE\big(ML(b)\big)}$$

$$\frac{a = b \star c,\ \star \in \{+, -, \dots\}}{SE\big(ML(b)\big) = \{< Pred_i : SE_i >\}, SE\big(ML(c)\big) = \{< Pred_j : SE_j >\}}$$
$$SE\big(ML(a)\big) = \{\forall i, j < Pred_i \wedge Pred_j : SE_{ij} = SE_i \star SE_j >\}$$

$$\frac{a =* b, PtTo(b) = \{Pred_i : ML_i\}, SE(ML_i) = \{< Pred_j^i : SE_j^i >\}}{SE\big(ML(a)\big) = \{\forall i, j < Pred_{ij} = Pred_i \wedge Pred_j^i : SE_j^i >\}}$$
$$\exists i, j, k, l : SE_j^i = SE_l^k \vDash Pred_{ij} = \big(Pred_i \wedge Pred_j^i\big) \vee \big(Pred_k \wedge Pred_l^k\big), SE\big(ML(a)\big) \setminus = \{SE_l^k\}$$

$$\frac{\exists i, j : Pred_{ij} \vDash false}{SE\big(ML(a)\big) \setminus = \{SE_j^i\}}$$

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Pred(b), SE_{b_i}\big(ML(a)\big) = \{Pred_j^{b_i} : SE_j^{b_i}\}}{SE\big(ML(a)\big) =}$$
$$\begin{cases} \{\forall i, j < (Pred_i \wedge Curr_{b_1}) \vee (Pred_j \wedge Curr_{b_2}) : SE_{ij} = Unify(SE_{b_1}\big(ML(a)\big), SE_{b_2}\big(ML(a)\big)) >\}, n = 2 \\ Unify(Unify_{b_i}^{i=1,n-1}\left(SE_{b_i}\big(ML(a)\big)\right), SE_{b_n}\big(ML(a)\big)\Big), n > 2 \end{cases}.$$

Алгоритм выполнения внутрипроцедурного и межпроцедурного анализа строится аналогично алгоритму 2.6', описанному в разделе 2.3.1. При обходе графа потока управления обновляются ячейки памяти, множества *PtTo* и символьные выражения, хранящиеся в ячейках, согласно описанным выше передаточным функциям. Строится предикат пути, который необходим для корректного вычисления символьных выражений и детекторов. По-прежнему выполняется анализ ограниченного количества итераций цикла. Расширяющий оператор ∇ не применяется.

При межпроцедурном анализе алгоритм построения аннотации функции остается неизменным – используется алгоритм 2.7', но вместо классов значений вместе с ячейками памяти сохраняются их символьные выражения *se*. Когда аннотация используется, то нет необходимости отдельно трактовать символьно внешние ячейки памяти и выражения, хранящиеся в них, т.к. все вычисления уже выполняются символьно. Если для ячеек памяти, соответствующих формальным параметром, известно, что в них в вызывающей функции содержится символьное выражение $SE\big(ML(param)\big) = \{< Pred_i : SE_i >\}$, то во всех символьных выражениях, где используется некоторая символьная переменная формального параметра se_{param} с предикатом *Pred*, подставляется и раскрывается выражение $\{< Pred \wedge Pred_i : SE_i >\}$, т.е. во всех последующих вхождениях выбрасываются несовместные предикаты, а предикаты для одинаковых выражений объединяются через дизъюнкцию. Эта

процедура выполняется для всех фактических параметров одновременно, чтобы учесть возможные зависимости между ними.

Детекторы в чувствительном к путям анализе на основе символьных выражений строятся подобно тому, как описано в разделе 2.4.2. Строится предикат ошибочной ситуации напрямую на основе построенного основным анализом предиката пути и символьных выражений. Совместность предиката ошибки проверяется SMT-решателем. Примеры таких детекторов приведены в разделе 4.

2.4. Детекторы в межпроцедурном анализе

В разделах 2.2 и 2.3 были описаны методы построения межпроцедурного анализа, который вычисляет модель памяти программы, классы значений, соответствующие ячейкам памяти, и основные свойства этих значений – интервалы значений для целочисленных ячеек и множества $PtTo$ для указательных ячеек. Для поиска конкретных ошибочных ситуаций строятся детекторы, которые могут вычислять необходимые им свойства классов значений. Такие свойства будем называть *атрибутами*, всегда вычисляемые интервалы значений и множества $PtTo$ – основными атрибутами. После вычисления атрибутов для каждой точки функции или каждого класса значения детектор проверяет истинность *предиката ошибки* – некоторой формулы над атрибутами, которая выражает ошибочную ситуацию в терминах построенной модели программы. Помимо самих атрибутов, детектор может также просматривать сами ячейки памяти, имена функций, другую информацию, связывающую внутреннее представление анализа с исходным кодом программы. Если предикат ошибки истинен, то выдается предупреждение о возможной ошибке, содержащее информацию о ее месте в программе, связанных с ней переменных, ошибочных значениях атрибутах и т.п. Для более детального описания ошибки детектор может вычислять дополнительные атрибуты (см. раздел 3).

Далее в разделе 2.4.1 описываются принципы построения детекторов, использующие анализы раздела 2.2, а в разделе 2.4.2 – чувствительные к путям выполнения детекторы.

2.4.1. Детекторы в анализе с классами значений

Для определения ошибочной ситуации детектор заводит один или несколько атрибутов, отражающих необходимые ему свойства классов значений, совершенно аналогично интервалам значений Val и множествам $PtTo$. Пусть $SL: < As, \sqcup >$ – это полурешетка атрибута $Attr \in As$, As – множество значений атрибута, $\sqcup$ – операция объединения, $AttrName$ – имя атрибута. Тогда в результате выполнения анализа, как было упомянуто в алгоритме 2.6, для каждой точки программы строится отображение $Attrs: VC \times AttrName \to Attr$.

Для этого дополнительно определяются передаточные функции для всех инструкций внутреннего представления $TF: Insn \times Attrs \to Attrs$. Передаточной функции предъявляются значения атрибута для всех классов значений до выполнения данной инструкции, и она строит значение атрибута после выполнения (как правило, это атрибут класса значений результата данной инструкции). Для инструкции присваивания передаточная функция не строится, поскольку достаточно применения функции для классов значений – класс значений ячейки памяти-результата инструкции становится равным классу значения ячейки памяти-правой части.

Далее в алгоритме 2.6 на шаге 3.2 передаточные функции для атрибутов детекторов применяются так же, как и передаточные функции для основных атрибутов. Аналогично, на шаге 3.3 в качестве функции объединения *Unify* для каждого атрибута используется его операция объединения $\sqcup$. Если полурешетка атрибута *SL* имеет бесконечную высоту, то при условии использования расширяющего оператора ∇ для основных атрибутов дополнительно требуется наличие такого же оператора $\nabla: Attr \to Attr$ и для атрибутов детекторов, и этот оператор также применяется на шаге 3.4 алгоритма.

При выполнении межпроцедурного анализа алгоритм 2.7 построения аннотации остается тем же, все атрибуты детекторов для классов значений в точке выхода из функции, попавших в аннотацию, сохраняются аналогично основным атрибутам. Для того, чтобы корректно использовать атрибуты детекторов с внешними значениями, для передаточных функций атрибутов должны выполняться требования теоремы 2.5 об их ассоциативности. Функции объединения будут ассоциативными в силу того, что атрибут является полурешеткой.

В качестве примера рассмотрим детектор ситуаций деления на ноль. Дополним наш модельный язык операциями умножения и деления и введем атрибут $IsZero = \{no, yes, maybe\}$ с операцией объединения $\sqcup(a_1, a_2) = \begin{cases} a_1, a_1 = a_2 \\ maybe, a_1 \neq a_2 \end{cases}$. Передаточные функции атрибута выглядят следующим образом:

- Атрибут устанавливается в *yes* в случае явного присваивания константы 0, умножения на 0 или на операнд с атрибутом, равным *yes*, сравнения с нулем:

$$\frac{a = 0, a = b * 0, a = b * c, IsZero(vc_c) = yes}{IsZero(vc_a) = yes}$$

$$\frac{if(a == 0)}{IsZero_{true}(vc_a) = yes, IsZero_{false}(vc_a) = no}$$

- Атрибут устанавливается в *no* в случае, когда арифметическая операция с нулевыми либо ненулевыми операндами гарантированно имеет ненулевой результат (сложение/вычитание нуля с ненулевым числом, умножение ненулевых чисел и т.д.):

$$\frac{a = b + c, IsZero(vc_b) = yes, IsZero(vc_c) = no}{IsZero(vc_a) = no}$$

$$\frac{a = b - c, IsZero(vc_b) = no, IsZero(vc_c) = yes}{IsZero(vc_a) = no}$$

$$\frac{a = b * c, IsZero(vc_b) = no, IsZero(vc_c) = no}{IsZero(vc_a) = no}$$

- Атрибут устанавливается в *yes* в случае вычитания равных чисел:

$$\frac{a = b - c, vc_b = vc_c}{IsZero(vc_a) = yes}$$

- Атрибут устанавливается в *yes*/*maybe* в случае соответствующего значения интервала значений *Val* (как при присваивании, так и при сравнении):

$$\frac{a = Op(b, c), Val(vc_a) = [0,0]}{IsZero(vc_a) = yes}$$

$$\frac{a = Op(b, c), Val(vc_a) = [d, e], d \leq 0 \leq e}{IsZero(vc_a) = maybe}$$

Выдача предупреждения о возможном делении на ноль производится, если в функции есть операция деления или взятия модуля и атрибут *IsZero* класса значения, соответствующий операнду-знаменателю, в точке операции равен *maybe*. Рассмотрим пример на рисунке 2.8а). На строке 2 атрибут *IsZero* для `k` оказывается равным *no*, а в строке 5 – *yes* (т.к. после обработки присваивания на строке 3 у ячеек памяти, соответствующих переменным `b` и `c`, оказываются одинаковые классы значений). Тогда в строке 6 выдается предупреждение, т.к. $IsZero_6(vc_k) = maybe$.

```
(1)   int foo (int a, int b) {
(2)     int k = 2;
(3)     int c = b;
(4)     if (a > 2)
(5)       k = b - c;
(6)     return a/k;
(7)   }
```

Рисунок 2.8а. Поиск деления на ноль – истинное срабатывание.

Если же рассмотреть пример на рисунке 2.8б), то в строках 5 и 7 будет известно, что $IsZero_5(vc_i) = IsZero_7(vc_i) = no$, т.к. в строке 5 нулевое значение увеличивается на единицу, а в строке 7 значение заведомо не равно нулю. В строку 10 можно попасть только из строки 7,

тем самым в строке 10 $IsZero_{10}(vc_i) = no$, и предупреждение о делении на ноль выдавать не нужно.

```
(1)  void bar() {
(2)    int i = 0;
(3)    while (1) {
(4)      if (i == 0) {
(5)        i++;
(6)      } else {
(7)        break;
(8)      }
(9)    }
(10)   i = 5 / i;
(11) }
```

Рисунок 2.8б. Поиск деления на ноль – отсутствие ошибки.

2.4.2. Чувствительные к путям детекторы

Как уже говорилось в разделе 2.3, в случае создания чувствительных к путям детекторов отслеживается не просто значение атрибута детектора, но также и условия, при которых атрибут принимает это значение. Тем самым в результате анализа в каждой точке программы строится отображение $Attrs: VC \times AttrName \to AttrPred$, где $AttrPred = \{< Pred_i: Attr_i >\}$, при этом $\forall i,j\ Pred_i \wedge Pred_j = false, Attr_i \neq Attr_j$. Если атрибут инициализируется некоторым значением $Attr_i$, то к этому значению в качестве предиката приписывается текущий предикат пути $Curr_{insn}$. Далее, предполагая, что определена передаточная функция для двуместного оператора $TF: Op \times Attr \times Attr \to Attr$, при вычислении передаточной функции вместе с предикатами берутся все попарные комбинации предикатов, записанных в атрибутах операндов, и применяется исходная передаточная функция:

$$\frac{a = b \star c, AttrPred(vc_b) = \{< Pred_i: Attr_i >\}, AttrPred(vc_c) = \{< Pred_j: Attr_j >\}}{AttrPred(vc_a) = \{\forall i,j < Pred_i \wedge Pred_j: Attr_{ij} = TF(\star, Attr_i, Attr_j) >\}}$$

Здесь $\star$ – это произвольная поддерживаемая анализом двуместная операция.

Нужно также отметить, что для сохранения корректности необходимо провести еще два преобразования над множеством *AttrPred*. Во-первых, если какая-либо из формул $Pred_i \wedge Pred_j$ оказывается несовместной, то соответствующее значение атрибута удаляется из множества. Во-вторых, если для некоторых значений i,j,k,l оказывается, что $Attr_{ij} = Attr_{kl}$, то соответствующие предикаты объединяются через дизъюнкцию

$(Pred_i \wedge Pred_j) \vee (Pred_k \wedge Pred_l)$, чтобы в результирующем множестве каждое значение атрибута встречалось только один раз.

При слиянии потока управления выполняются действия, аналогичные описанным ранее для множеств *PtTo*: к предикатам атрибутов классов значений, пришедших в данный базовый блок из некоторого предшественника, дописывается предикат пути соответствующего предшественника, используется операция объединения $\sqcup$ из определения полурешетки атрибута, а далее предикаты, соответствующие одинаковым значениям атрибутов, объединяются через дизъюнкцию, и несовместные предикаты удаляются из результирующего множества:

$$\frac{n \geq 2, b_1, b_2, \dots, b_n \in Preds(b), AttrPred_{b_i}(vc_a) = \{< Pred_j^{b_i} : Attr_j^{b_i} >\}}{AttrPred_b(vc_a) = \sqcup_{b_i}^{i=1,n} (AttrPred_{b_i}(vc_a))}$$

$$\sqcup_{b_i}^{i=1,n} (AttrPred_{b_i}(vc_a)) = \begin{cases} \{\forall i,j < (Pred_i \wedge Curr_{b_1}) \vee (Pred_j \wedge Curr_{b_2}) : Attr_{ij} = \sqcup (Attr_i, Attr_j) >\}, n = 2 \\ \sqcup (\sqcup_{b_i}^{i=1,n-1} \left(AttrPred_{b_i}(vc_a)\right), AttrPred_{b_n}(vc_a)\Big), n > 2 \end{cases}.$$

$$\frac{\exists i,j,k,l : Attr_{ij} \in AttrPred_b(vc_a), Attr_{kl} \in AttrPred_b(vc_a), Attr_{ij} = Attr_{kl}}{Pred_{ij} = (Pred_i \wedge Pred_j) \vee (Pred_k \wedge Pred_l), AttrPred_b(vc_a) \setminus = \{< Pred_{kl} : Attr_{kl} >\}}$$

$$\frac{AttrPred_b(vc_a) = \{< Pred_i : Attr_i >\}, \exists i : Pred_i \vDash false}{AttrPred_b(vc_a) \setminus = \{< Pred_i : Attr_i >\}}$$

Вычисление предикатов для атрибутов детекторов выполняется так же, как описано в разделе 2.3.1 для основных атрибутов. Можно заметить, что корректность вычисления передаточных функций с предикатами сводится к корректности вычисления предиката пути *Curr*, которая уже была установлена, и к корректности исходных передаточных функций атрибута. Аналогично из требования ассоциативности передаточных функций с предикатами следует требование ассоциативности исходных функций (т.к. конъюнкция и дизъюнкция уже ассоциативны) для выполнения требований теоремы 2.5.

Для выдачи предупреждения в некоторой точке программы детектор строит предикат ошибки $Error(insn)$ – формулу над вычисленными атрибутами, истинность которой указывает на наличие ошибки. Далее с помощью SMT-решателя проверяется совместность формулы $Curr(insn) \wedge Error(insn)$, и в случае положительного ответа выдается предупреждение. Детектор может пользоваться и другими определениями ошибочных ситуаций, о которых будет сказано в главе 4 с описаниями детекторов инструмента Svace. Кроме того, стоит задача объяснения причины возникновения ошибки, а именно – описания возможного пути выполнения, вдоль которого произошла ошибка, и возможных значений переменных программы. Для этого, как правило, детекторами отслеживаются дополнительные атрибуты, а

также проверяется совместность предиката ошибки и условий переходов в точках разделения потока управления. Подробнее эти методы также будут описаны в главе 4.

Вернемся к нашему примеру детектора, ищущего ситуации возможного деления на ноль, и модифицируем его для обеспечения чувствительности к путям выполнения. Будем отслеживать предикаты у описанного атрибута *IsZero*. Добавление предиката пути к изначальным значениям атрибута выполняется тривиально. Если же передаточные функции зависят от вычисленных интервалов значений, то соответствующие предикаты берутся оттуда. При слиянии потока управления используются описанный выше общий метод построения операции объединения ⊔ с использованием предикатов.

Рассмотрим пример на рисунке 2.9, который является измененным вариантом кода на рисунке 2.8а). Как и ранее, на строке 2 атрибут будет вычислен как *no*, а в строке 5 – как *yes*, но с соответствующими предикатами: $IsZero_2(vc_k) = \{< true: no >\}$, $IsZero_5(vc_k) = \{< vc_a > 2: yes >\}$. В строке 6 после слияния потока управления обнаруживаем, что $IsZero_6(vc_k) = \{< vc_a > 2: yes >, < vc_a \leq 2: no >\}$, в строке 7 – аналогично, при этом $Error_7 = IsZero_7(vc_k) == yes$, $Curr_7 = vc_a == 1$. Проверка на совместность формулы $vc_a > 2 \wedge vc_a == 1$ дает отрицательный результат, поэтому в строке 7 предупреждение не выдается.

```
(1)   int foo2 (int a, int b) {
(2)     int k = 2;
(3)     int c = b;
(4)     if (a > 2)
(5)       k = b - c;
(6)     if (a == 1)
(7)       return a/k;
(8)     return 0;
(9)   }
```

Рисунок 2.9. Поиск деления на ноль – случай несовместных предикатов.

3. ПРОГРАММНОЕ СРЕДСТВО МНОГОУРОВНЕВОГО СТАТИЧЕСКОГО АНАЛИЗА SVACE

В этой главе описывается архитектура и реализация коллекции статических анализаторов Svace, разрабатываемой в Институте системного программирования РАН, которая реализует набор методов анализа, описанных выше в главе 2, для языков программирования Си, Си++, Java и C#, на всех уровнях анализа – от легковесного анализа абстрактных синтаксических деревьев до чувствительного к путям анализа для поиска наиболее критических дефектов. Устройство архитектуры Svace преследует две цели: во-первых, сделать возможным и удобным реализацию в промышленном окружении методов анализа на всех уровнях, описанных в главе 2; во-вторых, предоставить инфраструктуру анализа, которая делает возможным единообразный запуск множества статических анализаторов и просмотр их результатов в одном интерфейсе, с единым управлением и форматом представления результатов, к которой можно легко подключать новые анализаторы, а также использовать как во время разработки и ночных сборок, так и в системах непрерывной интеграции при рецензировании кода.

В ходе экспериментальной проверки достижения первой цели, описанной в главе 5, которая делалась тестированием на реальных открытых проектах, оказалось, что использование методов главы 2 позволяет анализатору достичь требуемых функциональных (поддержка популярных языков и типов ошибок) и нефункциональных (качество, масштабируемость) характеристик для промышленного использования в жизненном цикле разработки безопасного ПО. То же самое тестирование продемонстрировало, что разработанная инфраструктура может с успехом использоваться для объединения разнородных анализаторов и совместной работы с их результатами – в настоящий момент семейство Svace объединяет пять разных анализаторов.

Настоящая глава содержит описание всех компонент инфраструктуры анализатора, за исключением собственно алгоритмов детекторов конкретных ошибок, которым посвящена глава 4.

Представляется особенно важным подчеркнуть, что для построения качественного статического анализатора и его внедрения в промышленный цикл разработки недостаточно реализовать все теоретически обоснованные методы главы 2. Помимо упомянутых качества и масштабируемости, требуется также по возможности минимизировать время прикладного программиста, которое тратится на взаимодействие с инструментом анализа, а также максимально упростить интеграцию новых статических анализаторов. Первое означает, что необходимо организовывать автоматический анализ без участия пользователя, поддерживать взаимодействие с другими инструментами организации разработки, в частности, с системами

непрерывной интеграции, удобно представлять результаты работы анализатора и т.п. Причем эти требования постоянно меняются по мере развития технологий, упрощающих разработку. Усилия, которые необходимо затратить на реализацию упомянутых возможностей, сравнимы с реализацией собственно алгоритмов анализа, которые часто считают единственными наукоемкими и достойными внимания компонентами анализатора.

Предлагается использовать архитектуру построения набора анализаторов, схематически представленную на рисунке 3.1 вместе с процессом работы анализатора. Данная архитектура реализована в инструменте Svace. Для работы алгоритмов анализа главы 2 прежде всего необходимо построить внутреннее представление анализируемой программы. Требование автоматического анализа означает, что создание этого представления должно происходить прозрачно для пользователя. Предлагаемым решением является контролируемая сборка программы, которая использует собственные компиляторы для генерации необходимых синтаксических деревьев или более низкоуровневого представления, не мешая основному процессу сборки. Необходимые для этого методы и технические решения описаны в разделе 3.1. Устройство компиляторов для генерации внутреннего представления, особенности поддержки диалектов языков, выдача дополнительной информации, необходимой анализам верхних уровней обсуждаются в разделе 3.2.

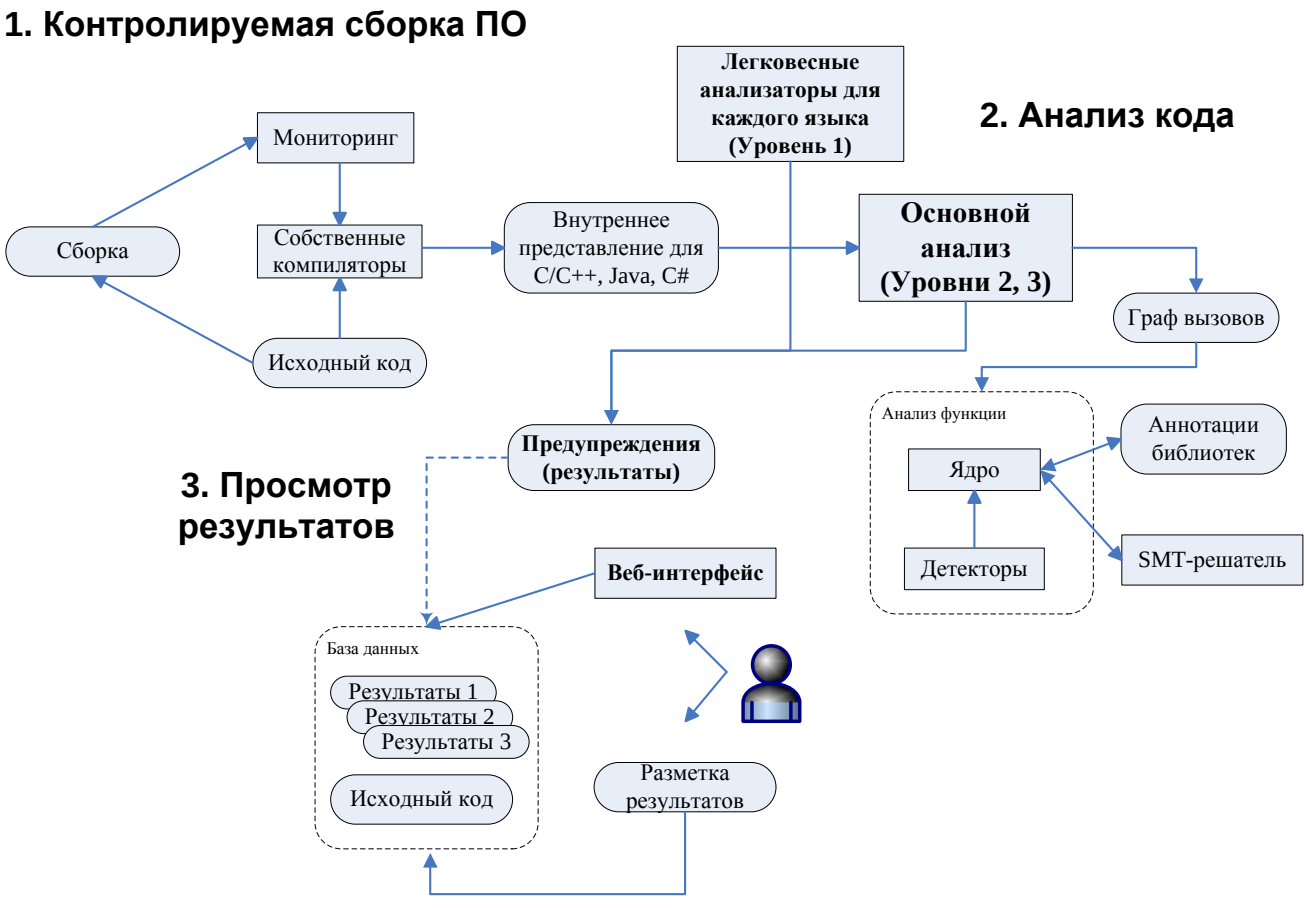

Рисунок 3.1. Организация и схема работы анализатора Svace.

На следующем этапе запускаются собственно алгоритмы анализа для всех уровней анализа, описанных в главе 2. Анализаторы первого уровня (уровня АСД) реализованы отдельно для всех поддерживаемых языков. Анализатор второго и третьего уровней (межпроцедурного анализа, в том числе с чувствительностью к путям) реализован для языков Си, Си++ и Java, и отдельно – анализатор третьего уровня для языка C#. Анализаторы используют представление программы, построенное на предыдущем этапе контролируемой сборки, а также другие собранные данные (например, сведения о компоновке программы). Организация анализов, в том числе параллельный анализ, особенности анализа высокоуровневых языков, а также инкрементальный анализ, требуемый для важных сценариев использования, обсуждаются в разделе 3.3.

После окончания работы анализа наступает время работы пользователя с результатами анализа. Так как предполагается, что работа над анализируемой программой ведется постоянно (анализатор используется на этапах разработки и тестирования), то необходимо хранить набор результатов анализа изменяющихся исходных кодов программы. При этом удобное представление результатов означает, что пользователь должен иметь возможность их просмотра в графическом режиме с поддержкой навигации по исходному коду. Чувствительные к путям детекторы могут вместе с местом предупреждения выдавать его *трассу*, то есть последовательность точек программы, описывающую возможный путь выполнения, приводящий к ошибке. Разобраться в таких предупреждениях без удобных средств просмотра кода непросто. Организация навигации по коду требует сохранения исходных кодов программы и их разметки по типам, переменным, местам их определения и использования и т.д. в базе данных вместе с результатами анализа – ведь просмотр результатов часто выполняется на другой машине. Это в свою очередь требует сборки всех нужных данных на этапе контролируемой сборки.

Кроме навигации, ключевым требованием интерфейса просмотра является скрытие уже известных ошибок (т.е. размеченных пользователем как ложные либо те, что не будут исправлены). Для этого требуется уметь сравнивать разные результаты анализа так, чтобы определение новых ошибок было устойчиво к изменениям исходного кода, сборке программы в разных местах файловой системы и т.п. Наконец, в ходе просмотра предупреждений необходимо поддерживать распределенную работу множества пользователей, хранить большое количество данных в базе. Реализация всех нужных компонент анализатора описывается в разделе 3.4.

Для подключения новых анализаторов в набор инструментов Svace, если они предназначены для уже поддерживаемого языка программирования, достаточно добавить их запуск на соответствующем уровне анализа (АСД или выше) для собранного внутреннего

представления, а также преобразовать формат их вывода в представление Svace. Если же язык программирования новый, то потребуется также расширить систему контролируемой сборки для перехвата компиляторов программ на этом языке или сбора его текстов, если это интерпретируемый язык. Кроме того, необходимо сгенерировать формат DXR для нового языка, чтобы в интерфейсе пользователя можно было удобно просматривать предупреждения. Остальные компоненты инфраструктуры, как и схема работы с новым анализатором, останутся неизменными.

3.1. Контролируемая сборка программы

Для работы анализатора, как собственно алгоритмов анализа, так и для показа его результатов, необходима разнообразная информация о программе. Прежде всего это представление программы, подвергаемое анализу, – АСД или низкоуровневое представление всех функций программы. Это информация, нужная для построения графа вызовов программы, – данные о том, как модули программы были скомпонованы между собой, какие библиотеки использовались. Это исходный код программы и его разметка, необходимые для показа результатов анализа.

Для успешного разбора файлов с исходным кодом потребуется знать, как именно предполагалось скомпилировать данный файл – так, для языков Си/Си++ необходимо правильно задать пути к каталогам с заголовочными файлами, предопределенные макросы компилятора, целевую архитектуру и т.п., для языка Java – пути к библиотекам в JAR-формате, пути к исходным кодам-зависимостям, уровень поддержки языка. Как видим, это информация о том, как пользователь планировал собирать свою программу. Необходим способ получения этой информации, при этом остается требование проведения автоматического анализа с минимальным вмешательством пользователя. Отсюда естественно возникает идея проведения *контролируемой сборки* программы – предполагая, что все нужные данные так или иначе передаются утилитам сборки (компиляторам, ассемблерам, компоновщикам и т.д.), запустить процесс сборки под контролем анализатора, отследить запуски всех интересующих утилит, получить требуемую информацию из их параметров и окружения, организовать запуск собственных компонент анализатора, строящих представление программы для анализа. При этом, чтобы получить верные данные, нельзя влиять на сам исходный процесс сборки.

Схематически устройство контролируемой сборки представлено на рисунке 3.2. Исходная сборка преобразует исходный код программы в *артефакты* сборки – исполняемые файлы, библиотеки, документацию и т.п. Контролируемая сборка выполняет мониторинг исходного процесса сборки, не влияя на него, и отслеживает так называемые *события сборки* – выполнение одного из интересующих нас действий. На каждое такое событие готовится и

выполняется *реакция* по сбору нужной для анализа информации. Примеры часто встречающихся событий и реакций показаны на рисунке.

Подробнее об организации каждого из этапов отслеживания событий и выполнения реакций будет рассказано в нижеследующих подразделах. До этого заметим, что при наступлении события основным способом обработки является *синхронная* обработка – то есть сборка или ее часть приостанавливается до окончания отработки реакции на событие. Сама реакция может наступать как до начала выполнения обнаруженного события, так и сразу после его окончания в зависимости от техники обнаружения события и природы необходимой реакции.

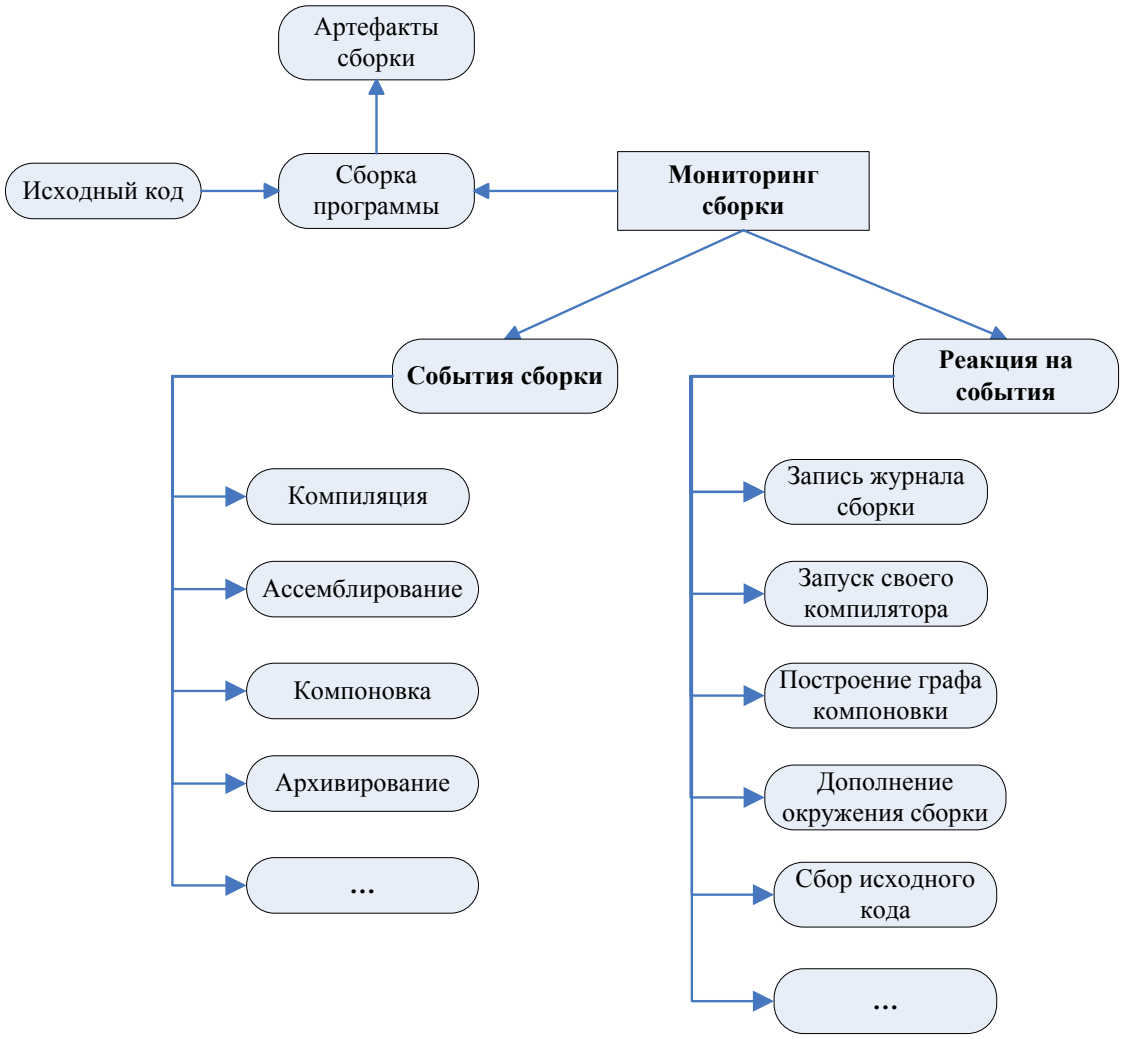

Рисунок 3.2. Контролируемая сборка программы

Причина выбора синхронной обработки в том, что нужная анализу информация может быть недоступна после окончания сборки или даже после выполнения интересующей нас утилиты сборки. Например, используемые библиотеки могут быть перемещены или удалены, сгенерированные исходные коды также удалены и т.п. Асинхронная обработка приведет к

тому, что часть информации может быть потеряна (в случае параллельной сборки – недетерминированно утеряны разные данные, что может привести к различным результатам анализа в зависимости от сборки и крайне нежелательно).

3.1.1. Обнаружение событий сборки

Задача определения интересующих анализатор событий сборки делится на две части: во-первых, определить наступление любого события сборки, и затем выбрать из потока событий собственно нужные события. Выбор решения для первой подзадачи определяется окружением, в котором выполняется сборка, в первую очередь – операционной системой и устройством пакета разработчика для языка, на котором написана собираемая программа. Вторая часть решается, как правило, независимо от ОС по параметрам произошедшего события сопоставлением с параметрами события-образца. Обычно достаточно установить, соответствует ли путь к исполняемому файлу запущенного процесса заданному регулярному выражению.

Классификация методов решения упомянутых задач представлена на рисунке 3.3. Наиболее часто встречаемым случаем события является запуск внешнего процесса (компиляции, компоновки и т.д.). Способы перехвата процессов и получения их параметров и окружения разнятся между операционными системами, но можно выделить несколько основных (рассмотрим на примере ОС семейств Windows и Linux):

- **Подмена функций системных динамических библиотек**, создающих новые процессы, через инъекцию DLL – подгрузку собственной динамической библиотеки, реализующей нужные функции. Для Linux такая возможность предоставляется через механизм `LD_PRELOAD`, управляющий поведением динамического загрузчика, для Windows – например, через функции Windows API (`SetWindowsHookEx`).

  Этот способ в анализаторе Svace является основным способом перехвата для Linux. Причина заключается в низких накладных расходах (по сравнению с другими способами) и простоте доступа к исполняемому файлу, параметрам и окружению процесса (т.к. эти данные напрямую передаются в функции семейства `exec`). Основное ограничение заключается в неприменимости данного способа к статически скомпонованным программам на Linux (для Windows из-за невозможности статической компоновки с системными библиотеками, реализующими Windows API, проблема отсутствует).

Кроме того, возможны ограничения на передачу относящихся к загрузчику переменных окружения при запуске set-user-id/set-group-id процессов (например, переменная `LD_PRELOAD`

будет проигнорирована). В некоторых случаях утилиты предоставляют пользователю способ повлиять на переменные окружения, например, утилита `sudo` позволяет указать переменные окружения, которые необходимо сохранить при запуске дочерних процессов. Для этого требуется сконфигурировать утилиту специальным образом, что применяется, например, системой сборки `gbs`, а также нашим анализатором. Есть и другие сложности, которые будут описаны ниже.

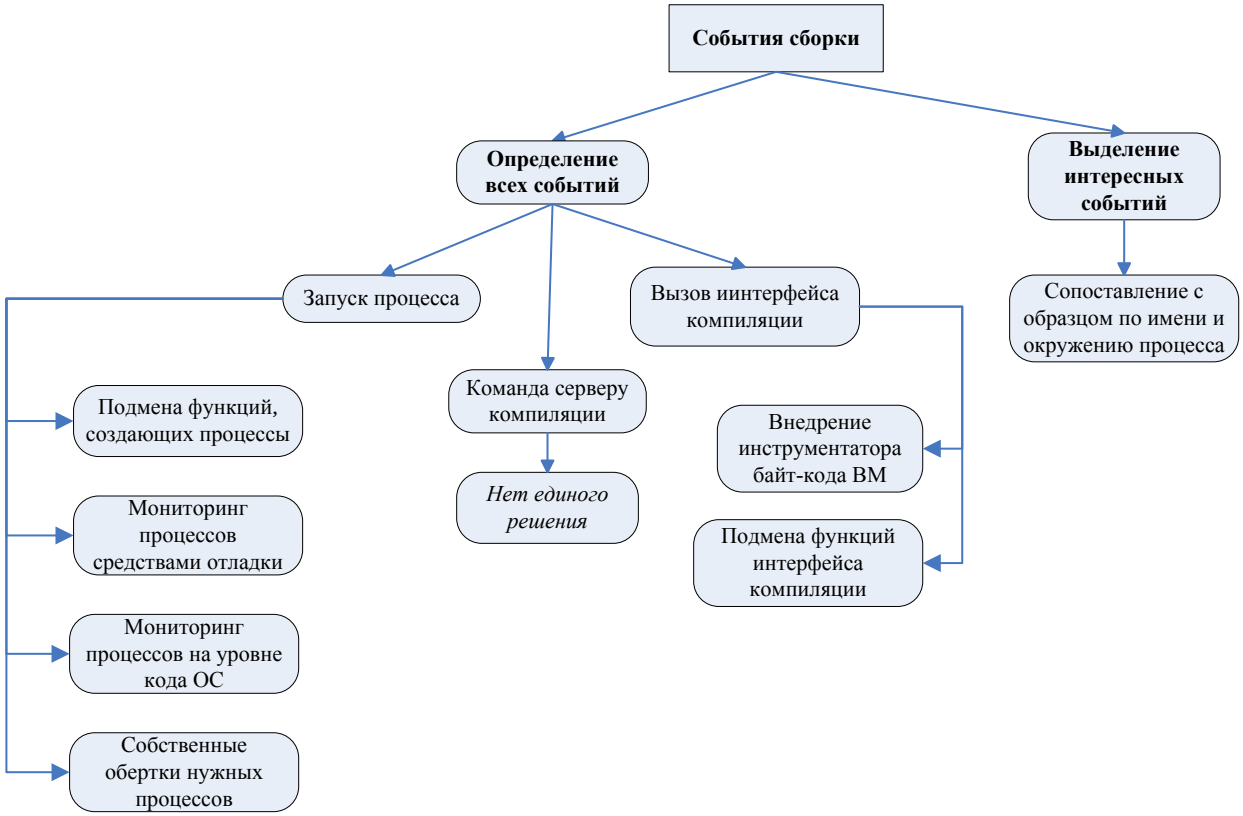

Рисунок 3.3. Определение событий сборки.

- **Мониторинг создания процессов штатными средствами отладки** операционной системы – для ОС Linux через функцию `ptrace`, для ОС Windows – через Windows Debugging API. Этот способ является основным при работе для систем Windows, а для Linux используется при обнаружении статически скомпонованных процессов. Возникающие сложности технического характера также зависят от системы, но можно выделить общую проблему – организацию выполнения нужного кода в контексте отлаживаемого процесса для получения доступа к обрабатываемым им данным – например, файлам с исходным кодом. Так, на Linux запрещена отладка привилегированных процессов непривилегированными, тем самым могут быть случаи, когда необходимо

запускать контролируемую сборку с правами привилегированного пользователя, а это не всегда возможно.

Специфической проблемой на Linux является получение пути к запущенному исполняемому файлу (что необходимо для определения, является ли процесс интересным для анализатора) – для этого требуется трассировка системных вызовов, т.к. путь в исходной форме не попадает ни в какие системные структуры данных, связанные с созданным процессом. На Windows мониторинг через интерфейсы отладки, вообще говоря, меняет поведение системы, например, память через Windows API выделяется другим аллокатором, функция `CloseHandle` может выбросить исключение при обработке некорректных дескрипторов объектов и т.д. Тем самым может измениться поведение исходной сборки программы, что и наблюдалось в очень редких случаях при внедрении Svace.

- **Мониторинг создания процессов через интерфейсы ядра ОС** (модули ядра, драйверы), т.е. создания специальных компонент, выполняемых с привилегиями ядра ОС. Такой способ, например, используют антивирусные программы на ОС Windows. В анализаторе Svace этот метод не применяется из-за желания минимизировать влияние на машину, на которой производится сборка. Установка новых компонент на уровне ОС может быть запрещено на промышленно используемых в компании серверах сборки – ошибка в коде мониторинга с большой вероятностью приведет к краху системы.
- **Замена исполняемых файлов интересующих анализатор процессов на собственные обертки,** которые при запуске сообщают анализатору о произошедшем событии запуска, а также выполняют и оригинальный файл. Этот способ исторически был первым, поддерживаемым в Svace, из-за легкости его реализации, после чего от него отказались в пользу остальных перечисленных способов – создание обертки требует вмешательства в окружение сборки или в сам процесс сборки. Сейчас обертка используется в исключительных случаях тогда, когда остальные способы не работают, например, для перехвата 16-битных компиляторов в виртуальной DOS-машине NTVDM.

При перехвате запуска процесса одним из описанных способов читаются его параметры, окружение, путь к исполняемому файлу процесса, после чего полученные данные передаются компоненту, отвечающему за определение «интересности» запущенного процесса. Если на процесс требуется реакция анализатора, то после окончания синхронной обработки события нужно продолжить сборку, вернув управление родительскому процессу. При этом требуется особая тщательность в сохранении контракта перехваченных функций создания процессов, и в

частности – кода возврата дочернего процесса, чтобы не повлиять на исходную систему сборки.

В некоторых контекстах выполнения такие действия также могут доставлять сложности. Например, при использовании функции `vfork` родительский процесс блокируется до вызова дочерним процессом функций `execve` или `_exit`. При этом из-за разделения адресного пространства между родительским и дочерним процессом возможные действия в дочернем процессе сильно ограничены – можно только запустить на выполнение новый процесс или выйти, а до этого момента нельзя изменять никакую память, кроме возвращаемого значения. Это приводит к необходимости специальным образом выделять и освобождать память под структуры данных с информацией о процессе, которые передаются анализатору. Кроме того, нельзя ожидать окончания выполнения процесса, запущенного через `execve`, т.к. нужно немедленно разблокировать родительский процесс. Можно было бы избежать этих проблем, реализовав при перехвате процессов выполнение функции `vfork` через `fork`, однако это приведет к значительному ухудшению производительности контролируемой сборки. Например, популярная утилита `make` использует `vfork` для создания дочерних процессов, и при такой «лобовой» реализации наблюдались ситуации, когда скорость контролируемой сборки по сравнению с обычной падала в два раза.

Следующим возможным типом события является запуск компиляции без создания нового процесса. Это происходит, когда компилятор может использоваться как библиотека через некоторый интерфейс компиляции. Например, в случае языка Java основной компилятор Javac пакета OpenJDK может вызываться через класс com.sun.tools.javac.main.Main. Другие компиляторы Java (Eclipse ECJ, компилятор Jack из ОС Android версий 6-8) также реализованы на Java и поддерживают эту возможность. Наконец, популярные среды сборки Java Ant, Maven, Gradle обычно выполняют компиляцию приложений через такие интерфейсы.

В описанных случаях событие сборки является вызовом некоторого метода в виртуальной машине Java. Для перехвата вызовов в Java, начиная с версии 1.5, можно использовать инструментирование байткода виртуальной машины, которое выполняется через стандартный механизм Java-агентов. *Агентом* является специальная библиотека на языке Java, которой виртуальная машина сообщает о загрузке любого класса и предоставляет возможность изменить этот класс произвольным образом, передавая библиотеке двоичное представление класса (массив байт).

В нашем случае, если переданный класс является одним из известных классов, реализующих интерфейс компилятора, то в этом классе ищутся заданные методы, выполняющие компиляцию, и инструментируются таким образом, что при их вызове дополнительно вызывается метод из нашего Java-агента, которому передаются все

необходимые данные о компиляции. Этот метод запускает специальный процесс, параметрами которого становятся все собранные данные. Процесс, в свою очередь, перехватывается обычными средствами слежения за запуском новых процессов, описанными выше, и распознается анализатором как запуск соответствующего Java-компилятора. По сути этот процесс служит просто контейнером для передачи данных о компиляции из Java-агента в анализатор.

Для внедрения собственного Java-агента в виртуальную машину Java необходимо отслеживать запуск виртуальных машин наряду с остальными процессами, т.е. такой запуск становится событием сборки, на который требуется реакция анализатора – добавление к командной строке запуска виртуальной машины опции с указанием дополнительного Java-агента.

Наконец, третьим вариантом события сборки является запрос на компиляцию, посланный *серверу компиляции*, постоянно работающему в ходе всей сборки (например, для языка C# в Microsoft Visual Studio и утилите сборки MSBuild используется сервер VBCSCompiler.exe; при сборке ОС Android компилятором Jack также используется сервер, принимающий команды компиляции через HTTP-запросы). Для перехвата таких событий нет каких-либо универсальных рецептов. Если сервер функционирует в виртуальной машине, можно попытаться свести задачу к предыдущей, инструментировав код сервера через штатные средства машины. В случае MSBuild можно подключить собственную библиотеку (логгер), которая будет оповещаться утилитой о событиях сборки. Мы реализовали такую библиотеку для перехвата компиляций для языка C#. Можно заметить, что для внедрения библиотеки также нужно перехватывать запуски самой утилиты MSBuild. Наилучшим решением является по возможности уйти от запуска сервера компиляции.

Подводя итоги изложенных вариантов возникновения и перехвата событий сборки, перечислим следующие проблемы организации перехвата общего характера:

- **Завершение контролируемой сборки**. В некоторых случаях системы сборки могут порождать служебные процессы или серверы компиляции, которые не завершаются немедленно после окончания сборки. Тем самым ждать завершения всех дочерних процессов, созданных при сборке, нельзя. Разумным компромиссом является окончание контролируемой сборки сразу после завершения основной команды сборки.
- **Перехват событий, возникающих при выполнении процессов для другой аппаратной архитектуры**. В некоторых системах сборки компиляция программ для встраиваемых систем происходит не с помощью кросс-компилятора для этой системы, а с помощью вызова эмулятора, который выполняет обычный

компилятор для встраиваемой системы, предназначенный для работы непосредственно на этой системе. Например, для компиляции программ для архитектуры ARM на машине семейства X86 запускается эмулятор QEMU, который выполняет компилятор ARM в двоичных кодах архитектуры ARM. В этом случае для поддержки перехвата процессов через `LD_PRELOAD` необходимо собрать библиотеку перехвата для архитектуры ARM. Также нужно отметить, что внутри эмулятора QEMU функция отладки `ptrace` вообще не реализована.

- **Перехват событий сборки в серверах компиляции, уже запущенных на момент начала контролируемой сборки**. Для внедрения в серверы компиляции или виртуальные машины собственного мониторинга необходимо отследить их запуск и изменить параметры этого запуска. Если же сервер уже работает, то перехват событий сборки в нем, как правило, невозможен, и необходимо вмешательство пользователя для организации сборки, которая может быть контролируемой.
- **Поддержка систем распределенной сборки.** Все описанные методы работают на одной вычислительной машине. Если большая программная система собирается распределенно (например, система OBS), то, как правило, организуется контролируемая сборка на каждом узле системы, а потом полученные данные объединяются в один набор, который далее можно анализировать совместно. Например, при сборке операционной системы Tizen через OBS на каждый узел системы инсталлируется компонент мониторинга Svace в виде отдельного RPM-пакета, а после окончания сборки полученные результаты со всех узлов объединяются специальной утилитой так, чтобы впоследствии с точки зрения анализатора казалось, что сборка происходила на одном компьютере.

3.1.2. Определение реакции на события

Возможные реакции на события сборки не сводятся только к построению внутреннего представления. Нужно собрать данные для обеспечения работы всех этапов анализа. Можно выделить следующие виды реакции (упомянуты на рисунке 3.2):

- **Запуск собственного компилятора для построения представления для анализа**. Это основной тип реакции на события сборки. Для обработки события нужно извлечь настройки компиляции из параметров и окружения перехваченного пользовательского компилятора и адаптировать их для настройки запуска собственного компилятора. Сложность заключается в том, что вариантов компиляторов для Си/Си++ – десятки, для Java – три основных. Существенными

для успешной компиляции настройками являются: пути к используемым заголовочным файлам, библиотекам и т.п.; набор заранее определенных пользовательским компилятором макросов (для Си/Си++); диалект языка; целевая архитектура и ее свойства. Выяснить некоторые из этих настроек по команде компиляции не всегда возможно, и приходится организовывать запуск оригинального компилятора со специальными опциями и разбирать его вывод. Для разбора опций компиляции также часто требуется читать файлы с опциями, имена которых находятся в командной строке.

- **Построение графа компоновки программы.** Наблюдая только за компиляциями отдельных исходных файлов, анализатор не сможет понять, как именно устроен граф вызовов программы, так как во многих случаях будет неизвестно, какая именно внешняя функция вызывается. При сборке больших программных систем часто возникают ситуации, когда подходящих функций-кандидатов несколько. Поэтому требуется либо по аналогии с компиляцией организовывать компоновку внутреннего представления для анализа, запуская на каждое перехваченное событие компоновки собственный компоновщик и получая один файл с внутренним представлением на каждую собранную программу, либо записывать информацию о том, какие модули программы были скомпонованы вместе, и использовать ее при построении графа вызовов. Первый метод обеспечивает большую точность результатов, но требует значительных ресурсов для больших программ. Второй метод нетребователен к ресурсам, т.к. собственно компоновки не происходит, но может в крайних случаях давать менее точные результаты.

  В Svace используется второй метод. Для этого перехватываются запуски компиляторов, ассемблеров, компоновщиков, архиваторов, разбирается их командная строка и ведется журнал всех манипуляций с исходными и объектными файлами программы. По окончании контролируемой сборки по этому журналу восстанавливаются данные о том, какие именно модули программы компоновались вместе.

- **Сбор исходных кодов и библиотек.** Некоторые используемые сборкой файлы требуется сохранить как есть для дальнейшего использования анализатором. Примерами таких файлов являются файлы с исходным кодом для показа предупреждений анализа, библиотеки Java в формате JAR, анализ которых проводится перед анализом пользовательского кода, файлы с данными, требуемые для некоторых специфичных детекторов (например, манифесты Android-

приложений). Такие файлы должны быть сохранены немедленно при обработке события, т.к. позже в ходе сборке они могут быть перемещены или удалены.

- **Модификация окружения перехваченного процесса.** Как было описано выше, например, при перехвате Java-компиляций требуется передать запускаемой виртуальной машине Java путь к библиотеке с собственным Java-агентом. Другим интересным применением является возможность автоматической сборки приложения с отладочной информацией, которое можно использовать далее в динамическом анализе при проверке предупреждений, выданных статическим анализатором.
- **Ведение журнала сборки.** Все поступающие события сборки записываются в специальный журнал, который бывает полезен при отладке компонентов мониторинга и при добавлении поддержки новых компиляторов и инструментов.

3.1.3. Реализация контролируемой сборки в анализаторе Svace

Описанные методы организации контролируемой сборки реализованы в анализаторе Svace следующим образом. Для операционной системы Linux поддерживаются методы перехвата событий сборки через `LD_PRELOAD` и отладку процессов, для Windows – только через отладку. Компоненты, ответственные за перехват и выделение интересующих событий, реализованы на языке Си, а сложная часть обработки данных, необходимая для запуска собственных компиляторов,– на языке Питон.

Ядро библиотеки перехвата позволяет регистрировать расширения для обработки события. В общем случае расширение должно уведомить ядро о том, что данное событие ему интересно, после чего ядро передает соответствующему обработчику всю информацию о перехваченном процессе. Под этот интерфейс подпадает большинство не связанных с компиляторами расширений. Для обработки компиляторов, как важного и самого трудоемкого случая, реализован более детальный интерфейс расширений, включающий построение команд с исходным компилятором для запроса путей к заголовочным файлам и встроенных макросов, раскрытия опций компилятора (если требуется чтение файлов с опциями), для компиляторов Си/Си++ – определение конкретного языка компиляции. Расширения делают удобным добавление поддержки нового компилятора – для этого достаточно добавить новое расширение и реализовать соответствующие обработчики интерфейса к ядру перехвата.

```
{
  "version": 1,
  "comp": {
```

```
"cxx": {
  "gcc": {
    "instances": [
      { "path": "/opt/baz/occ-3.1/misc/cc1",
        "enable": true,
        "add": [
          "-D", "MYFUNCTION"
        ]
      }
    ]
  }
}
}
```

Рисунок 3.4. Пример файла пользовательской конфигурации перехвата.

Для языков Си/Си++ поддерживается более 20 компиляторов, включая популярные компиляторы GCC, Clang, MSVC, ARMCC, а также ряд компиляторов для встраиваемых систем. Для языка Java поддерживаются три компилятора – основной компилятор Java из пакета OpenJDK, а также компиляторы Eclipse ECJ и Jack, как для запусков из командной строки, так и через интерфейсы компиляции. Для языка C# поддерживается основной компилятор Microsoft и компилятор Mono через утилиты MSBuild и xbuild.

Кроме того, поддерживается возможность пользовательской конфигурации распознавания компилятора и задания опций компиляции через создание специального файла в формате JSON. Это бывает полезно в случаях, когда стандартное определение типа запущенного компилятора не срабатывает, например, для специфических компиляторов на основе GCC, при конструировании которых была изменена структура каталогов служебных компонент компилятора. Пользователь может указать, что исполняемый файл по заданному пути является компилятором известного типа, либо доопределить или убрать некоторую опцию компиляции, либо выключить перехват сборки определенных файлов. Пример JSON-файла, добавляющую опцию при компиляции конкретным экземпляром совместимого с GCC компилятора, показан на рисунке 3.4.

3.2. Компиляторы для создания внутреннего представления для анализа

Для анализа программ на всех описанных в главе 2 уровнях необходимы инструменты, генерирующее соответствующее внутреннее представление программы – для анализа уровня АСД соответствующее абстрактное синтаксическое дерево и таблицу символов, для межпроцедурного и чувствительного к путям анализов – более низкоуровневое трехадресное представление. При их построении необходимо разрешить ряд проблем, которые в большинстве своем происходят из-за единственного фундаментального противоречия: с одной стороны, разбор программы лучше всего выполняется компилятором промышленного уровня, и в качестве базы удобно брать открытый компилятор соответствующего языка программирования; с другой стороны, цель создания внутреннего представления в компиляторе – последующая оптимизация и генерация исполняемого кода в предположении, что исходная программа корректна – не соответствует цели нашего статического анализатора, состоящей в поиске возможно большего числа ошибок в программе. Нужно особо подчеркнуть, что создание полностью собственного компилятора для генерации представления для анализа не является жизнеспособным вариантом – чтобы постоянно и качественно поддерживать новые стандарты популярных языков, требуются затраты огромного объема ресурсов (десятки исключительно квалифицированных программистов), и даже крупные компании предпочитают не тратить эти ресурсы, используя коммерческие или открытые разборщики программ (фронт-энды), например, EDG, и сосредотачиваясь на оптимизации программ под свои нужды.

Поэтому основным способом построения транслятора во внутреннее представление анализа является следующий: взять за основу существующий открытый промышленный компилятор необходимого языка и модифицировать его по своим потребностям. Решаемые при этом проблемы можно обобщить примерно так:

1. **Поддержка диалектов языка программирования**. Уровень поддержки языка задается взятым за основу компилятором, однако пользовательская программа может собираться другими компиляторами с собственными расширениями и диалектами языка. Конечно, эта проблема наиболее актуальна для Си и Си++ с десятками различных компиляторов для настольных и встраиваемых систем, однако и, например, для Java компилятор языка версии 8 может не компилировать некоторый код, написанный для версии 6. Часто пользовательский компилятор (особенно для встраиваемых систем) допускает большее множество программ, чем компилятор, строго следующий стандарту языка. Необходимо разбирать как можно больше программ собственным компилятором. Требуемые доработки могут разниться от тривиального пропуска новых ключевых слов или атрибутов до

значительных изменений алгоритмов разбора выражений, поиска типов и т.д. В эту же категорию относятся изменения, требуемые для поддержки поведения, задаваемого реализацией (т.н. implementation-defined behavior), когда это важно для компиляции кода.

2. **Максимально надежное восстановление после ошибок.** Количество особенностей в диалектах языка таково, что поддержать их все полностью не представляется возможным. В любом случае, разбор кода с неизвестным компилятору расширением приведет к ошибкам компиляции и пропуску соответствующего файла, а значит, анализатор не сможет найти в нем никакие ошибки. Необходимо поддерживать «нечеткий» режим компиляции (fuzzy parsing), в котором участки кода, при разборе которых произошла ошибка, пропускаются, а все успешно разобранные куски сохраняются. Тем самым даже при наличии ошибок в файле часть его будет проанализирована.

   Может показаться, что поддержка такого режима компиляции делает ненужной поддержку особенностей диалектов языка программирования вообще, однако это не так. Всегда предпочтительней корректно разобрать выражение языка на диалекте и передать его анализатору, чем пропустить его, особенно если ошибка в разборе приведет к каскадным ошибкам далее, например, из-за неполного типа данных.

3. **Максимально полное соответствие генерируемого представления исходному коду**. Уже при создании АСД компилятор может выполнять некоторые оптимизации исходного кода, например, сворачивать константные выражения, не только с целью упростить дальнейшие оптимизации, но и для того, чтобы выполнить требования стандарта языка. Такие оптимизации желательно не делать, чтобы анализатор не делал ложных выводов о программе на основе преобразованного кода либо когда часть нужной информации уже потеряна. Кроме того, нужно генерировать внутреннее представление для всего исходного кода, даже если компилятор считает его неиспользуемым. Наконец, требуется скрупулезно отслеживать связи между сгенерированным и исходным кодом, т.е. создавать как можно более полную отладочную информацию о выражениях, переменных, методах и т.п. В том числе, если некоторый код был сгенерирован компилятором, то этот код должен быть соответствующим образом помечен для того, чтобы анализатор мог отличить его от кода, написанного программистом. Дело в том, что одним из эвристических предположений, часто делаемых детекторами, является то, что программист не пишет неиспользуемого кода (т.е.

любой исходный код предположительно выполняется при некоторых входных данных), что может быть неверно для сгенерированного компилятором кода.

4. **Выдача дополнительной информации для анализатора.** В том случае, если обычной отладочной информации либо данных АСД недостаточно для того, чтобы принять решение о выдаче ошибки, часто бывает нужно сохранять недостающие данные при генерации представления. Для этого, как правило, расширяется выдаваемая компилятором отладочная информация в соответствующем формате (например, .class файлов для Java или метаданных для LLVM-биткода).

Далее мы опишем важнейшие из доработок описанных типов, которыми были подвергнуты компиляторы поддерживаемых Svace языков Си/Си++, Java и C#.

3.2.1. Поддержка Си/Си++

Базовым компилятором для поддержки Си и Си++ является компилятор Clang из пакета LLVM версии 3.4 и 3.8. При построении внутреннего представления генерируется биткод LLVM. В ходе анализа биткод считывается и конвертируется в представление для анализа. В данном разделе описывается только функциональность, относящаяся к генерации биткода.

В компиляторе Clang было выполнено более 1000 изменений представленных выше четырех категорий. В категории поддержки диалектов языка программирования самые существенные изменения требуются для компиляции кода с расширениями компилятора GCC (особенно старых версий семейства 2.x и 3.x), а также для поддержки диалекта Microsoft Visual Studio C++. Например, компилятор GCC поддерживает использование вложенных функций в языке Си, что является сомнительным расширением, однако встречается в реальных библиотеках. Остальные популярные компиляторы Си вполне сознательно не поддерживают вложенные функции. Мы выполнили реализацию подмножества вложенных функций для Си в компиляторе Clang с помощью механизма для генерации кода для лямбда-выражений в Си++.

Важным примером является поддержка разбора шаблонных выражений в случае, если ключевое слово `template` или `typename` опущено, или используется одинаковый идентификатор для именования типа и переменной в одном лексическом блоке. Такие послабления позволяются в компиляторе Microsoft для Си++ (MSVC). Например, на рисунке 3.5 при разборе строки 8 имя T будет по умолчанию считаться переменной, а не именем типа.

Для поддержки этого расширения пришлось реализовывать механизм отката разбора выражения – если произошла ошибка при разборе выражения, связанная с путаницей между типами и именами переменных, то компилятор возвращается к началу выражения и начинает разбор сначала, считая спорный идентификатор переменной, если в предыдущей попытке он

являлся типом, и наоборот. Кроме того, для MSVC выбор вызываемой функции должен быть отложен на этап инстанциации шаблона.

```
(1)   template <typename T>
(2)   struct B {
(3)     int* get();
(4)   };
(5)   template <typename T>
(6)   int* B<T>::get() {
(7)     T t;
(8)     return (T::T*)9;
(9)   }
```

Рисунок 3.5. Пример неоднозначности `template`/`typename`.

Другой областью разбора, часто требующей изменений, являются правила неявного приведения типов. Они определяют совместимость типов между собой и влияют на большое количество решений, принимаемых компилятором. Можно выделить следующие случаи, требующие введения новых правил приведения типов:

- Менее строгая обработка объектов с квалификатором `const` в компиляторе MSVC – иногда квалификатор можно опускать, можно также создавать неконстантные ссылки на временные объекты. Компилятор, строго придерживающийся стандарта, при разборе соответствующих выражений обнаружит несовместимость типов с квалификатором и без квалификатора (они не могут быть приведены друг к другу). Необходимо разрешить такое приведение с минимальным приоритетом по отношению к остальным видам приведения;
- Неявное приведение из указательного типа к типу, указывающему на структуру с одним полем, тип которого совпадает с изначальным указательным типом;
- Неявное приведение между скалярными и векторными типами одинаковых размеров;
- Неявное приведение типов между целочисленными и указательными типами одинакового размера (с низким приоритетом);
- lvalue-приведения типов (т.е. после явного приведения одного типа к другому результирующее выражение может именовать память).

Наконец, достаточно неожиданными доработками, которые необходимо сделать для поддержки компилятора MSVC, являются такие:

- Изменение правил поиска заголовочных файлов. Если при обработке исходного файла был найден в некотором каталоге и включен некоторый заголовочный файл, то на время открытия этого файла (т.е. пока считывается его тело и вставляется на место директивы `#include`) каталог с этим файлом добавляется в список каталогов для поиска заголовочных файлов. Неверная реализация этого правила влечет открытие неверных заголовочных файлов (не их тех каталогов) и последующие ошибки компиляции;
- Поддержка генерирования кода для COM-классов. При использовании COM-атрибутов в написании класса компилятор MSVC автоматически генерирует некоторые дополнительные классы (`CComCoClass<type>`, `CComObjectRoot<type>` и др.), а также дополнительные методы класса (`QueryInterface`, `GetProgID`, `UpdateRegistry` и т.п.). Без знания этих типов и методов дальнейшая компиляция будет проходить с ошибками, при этом знание точной реализации для целей статического анализа не требуется, достаточно генерации некоторых заглушек с минимальным кодом. Для выполнения такой кодогенерации была добавлена возможность в ходе разбора препроцессировать текст из заданной строки и вставить результирующий набор токенов в разбираемый поток с токенами (по сути, «вставить» заданный текст в обрабатываемое в данный момент место в файле). С помощью этого интерфейса вставляются определения необходимых классов и методов.

Доработки остальных категорий (со второй по четвертую) были выполнены описанным образом: разработан режим нечеткой компиляции, пропускающий остаток выражения или лексического блока при ошибках разбора, доработана отладочная информация (в основном для выражений, организующих поток управления – `goto`, `break`, `continue`, меток `case` – сохраняется их положение в исходных файлах, а также для прототипов функций, положения оператора наследования и т.д.), организована генерация неиспользуемого кода.

Для поддержки анализа на уровне АСД после выполнения сборки необходимо обеспечить сохранение (вместе с генерируемым биткодом) некоторого артефакта сборки так, чтобы потом можно было построить АСД в любой момент, в том числе – на другой машине. Можно рассматривать два подхода к решению этой задачи: сериализация АСД на диск при обработке события компиляции в ходе контролируемой сборки и последующее восстановление при анализе; сохранение исходного кода и его последующий разбор. Первый подход заведомо имеет ту сложность, что выполнить поиск ошибок, затрагивающих директивы препроцессора,

будет невозможно, т.к. информация о ходе препроцессирования уже утеряна. Кроме того, эксперименты с сериализацией АСД показали, что занимаемый им размер очень велик. Поэтому было принято решение разрабатывать второй подход.

Однако сохранять исходный код требуется специальным образом – включение заголовочных файлов на другой машине или даже на той же машине после окончания сборки может быть невозможно повторить в точности так, как это происходило во время сборки. Поэтому при сохранении используется специальный режим rewrite-includes препроцессора компилятора Clang, в котором текст всех заголовочных файлов вставляется в основной компилируемый файл, при этом сохраняются пометки о том, откуда был вставлен некоторый кусок кода, а также не раскрываются все остальные директивы препроцессора. В этом случае, используя сохраненный таким образом исходный файл, можно будет повторить в точности то же раскрытие всех макросов, директив условной компиляции и т.п. при условии сохранения значения всех предопределенных макросов компилятора. Аналогичный режим присутствует и в компиляторе GCC (опция `-fdirectives-only`).

Описанный режим компилятора Clang был доработан, исправлены некоторые ошибки работы с именами файлов в `#line`-директивах. Получаемые таким образом файлы имеют достаточно большой размер, однако хорошо сжимаются, делая реальным перенос детекторов уровня АСД с этапа контролируемой сборки на этап выполнения анализа – размер архивов с исходными файлами в несколько раз меньше размера LLVM-биткода для тех же файлов.

3.2.2. Поддержка Java

В качестве базового для компилятора Java был выбран компилятор Javac из пакета OpenJDK для Java 8. Это де-факто стандартный компилятор Java, и таких сложностей с совместимостью, которые были описаны выше для Си++, конечно, преодолевать не приходится. Для выполнения анализа уровня АСД сохраняются исходные Java-файлы, для остальных уровней – сгенерированные class-файлы, по которым далее строится представление для анализа.

Выполненные доработки по улучшению сборки кода на Java предыдущих версий (в частности, Java 6) были незначительны и в основном касались специфичных случаев поиска типов и методов для вызова. Из второй категории доработок был реализован простой вариант восстановления после ошибок компиляции заданного класса пропуском ошибочного класса целиком. Этого достаточно на практике, так как случаи несовместимости редки, и, кроме того, размеры Java-классов обычно невелики, и гранулярность на уровне классов не слишком грубая.

Более интересными являются доработки третьей категории. При компиляции возникают случаи, когда байткод, построенный для некоторого участка исходного кода, дублируется в нескольких местах созданного class-файла. Такое дублирование плохо тем, что теряется однозначное соответствие между байткодом и исходным кодом. Возникают ситуации, в которых генерируется недостижимый код, и перестают работать эвристики анализатора, предполагающие выполнимость написанного пользователем кода хотя бы для некоторых входных данных. Информация о появлении таких копий кода сохраняется компилятором для последующего использования в анализе. Аналогичным образом помечается код, который искусственно сгенерирован компилятором.

Другие популярные компиляторы Java – это компилятор ECJ, использующийся для разбора кода в среде Eclipse, и компилятор Jack, применяемый в ОС Android версий 7 и 8 (в настоящий момент компания Google объявила, что в дальнейшем компилятор Jack использоваться не будет). Поддержка ECJ не доставляет особых сложностей и не требует изменений в компиляторе, а лишь написания расширений для системы контролируемой сборки, описанных в разделе 3.1. Поддержка Jack, напротив, влечет за собой сравнительно большое количество работы. Дело в том, что вместо байткода Jack использует собственный формат для генерируемого кода – Jayce, который не предназначен для стековой машины, а является обычным трехадресным представлением. Аналогом JAR-библиотек являются Jack-библиотеки с набором Jayce-файлов с кодом для скомпилированных классов. Следовательно, любая компиляция, использующая Jack-библиотеки в качестве зависимостей, не может быть повторена с помощью нашего собственного компилятора, т.к. он может разбирать только class-файлы.

Возможным решением этой проблемы является поддержка чтения Jayce-файлов в нашем компиляторе и построение по ним всех необходимых классов (хотя бы заглушек, содержащих список имеющихся методов и полей). Недостаток такого решения в том, что проанализировать код в Jack-библиотеках не получится. Кроме того, сторонние анализаторы также не смогут с ним работать. Более полным решением представляется полная кодогенерация class-файлов по jayce-файлам. Для этого был с помощью интерфейсов компилятора Jack (исходный код которого открыт) был реализован отдельный инструмент Llij, конвертирующий Jack-библиотеки в JAR-библиотеки. Инструмент используется нашим компилятором в том случае, если в списке библиотек, переданных ему через опцию -classpath, встречаются Jack-библиотеки.

При создании инструмента главным является генерация тела метода в байткоде для стековой машины Java по абстрактному синтаксическому дереву компилятора Jack, которое считывается из формата Jayce. Кодогенерация выполняется при обходе АСД метода

непосредственно для каждого типа узла дерева с помощью библиотеки ASM, берущей на себя создание корректного бинарного class-файла и все нужные типы инструкций байткода. Можно перечислить следующие тонкости, которые требуют аккуратности при кодогенерации:

- Для генерации обработчиков исключений нужно отдельно обойти тело метода, построив соответствие между базовыми блоками и обработчиками, в которых они должны оказаться (может быть несколько обработчиков в случае нескольких операторов `catch` или вложенных операторов `try`), после чего сгенерировать обработчики и корректные метки блоков (поток управления) с учетом их вложенности;
- Выдача неиспользуемого кода не приносит сложностей для обычного регистрового представления, но в случае стековой машины такой код оставляет на стеке операнды, которые никогда не будут вытолкнуты. Помимо ухудшения качества кода, такой случай не обрабатывается библиотекой ASM при подсчете максимальной глубины стека операндов для данного метода. Необходимо крайне аккуратно избегать генерации выражений, результат которых далее не будет использован;
- При генерации условных выражений в условном операторе для избегания генерации мертвого кода и поддержки ленивой логики используется простой рекурсивный алгоритм, который генерирует код для заданной части условного выражения по переданным ему меткам, по которым нужно перейти в случае, когда выражение будет истиной или ложью;
- Генерация оператора выбора требует построения массива меток и ключей в порядке увеличения ключей. Нужно поддержать только целочисленные ключи, так как оператор выбора по строкам можно понизить до цепочки условных операторов с помощью встроенной трансформации компилятора Jack.

Для создания class-файлов необходимо сгенерировать т.н. стековые фреймы, которые были введены в Java для облегчения проверок корректности используемых типов во время выполнения. Обычно стековые фреймы заполняются компилятором, который владеет полной информацией об иерархии классов и всегда может вычислить суперкласс, общий для двух заданных классов. В нашем случае требуется поддерживать собственную иерархию классов, которая, вообще говоря, неизвестна в том случае, если изолированно конвертируется одна библиотека. Поэтому основным поддерживаемым случаем является конвертация библиотек из всего передаваемого компилятору списка, в котором необходимо искать неизвестные типы (classpath). Для этого все библиотеки из списка (в том числе JAR-библиотеки наряду с Jack-библиотеками) считываются для того, чтобы узнать для каждого класса его предка. По этой

информации строится иерархия классов и далее при кодогенерации простым алгоритмом поиска наиболее «специального» общего предка реализуется интерфейс для ответов на запросы об общих предках двух классов. Обработка библиотек в порядке их следования в списке classpath существенна, так как гарантируется, что если одна из библиотек из списка зависит от другой (т.е. использует ее типы), то она будет идти в списке раньше.

3.2.3. Поддержка C#

Для языка C# в качестве базового компилятора используется открытый компилятор Roslyn, разрабатываемый компанией Microsoft. Так как в нашем анализаторе C# внутренним представлением для анализа является АСД Roslyn, то для него не требуется отдельная кодогенерация. Кроме того, Roslyn является основным компилятором C# для Microsoft, поэтому с ним нет проблем с диалектами языка и совместимостью между версиями, описанными выше для Си/Си++ и Java. В настоящий момент разбор программ для C# использует Roslyn как есть в качестве библиотеки. Нужно отметить, что сборки C# с кодом в формате MSIL не поддерживаются и соответственно не анализируются (в отличие от JAR-библиотек в программах на Java). Эта поддержка является предметом будущих работ.

3.3. Основная фаза анализа

После построения внутреннего представления для всех программ, которые компилировались на этапе контролируемой сборки, наступает этап собственно анализа этого внутреннего представления (второй этап работы анализатора на рисунке 3.1). Входными данными для анализатора является *объект сборки* – набор артефактов (файлов с внутренним представлением, исходных файлов, файлов разметки и т.п.), которые были собраны в результате обработки событий в ходе контролируемой сборки. Подробнее об устройстве объекта сборки и методах хранения информации в нем будет сказано в разделе 3.4.

Анализ программы выполняется для каждого языка программирования отдельно – межъязыковой анализ в настоящий момент не поддерживается, однако для основных уровней анализатора общий анализ Си/Си++ и Java не представляет особых сложностей, достаточно будет доработать этапы построения графа вызовов программы и диспетчер, отвечающий за выбор очередной функции для анализа (см. далее в разделах 3.3.1 и 3.3.2). Выявление связей по вызовам между программами на Си и Java через механизм JNI сводится к специализированному анализу вызовов по указателю со стороны Си, учитывающему особенности интерфейса JNI-функций, и поддержке способа записи имен JNI-методов (name mangling) со стороны Java. Такая поддержка выполнена в инструменте ИСП РАН по анализу связей между сущностями программ, однако ее описание выходит за рамки настоящей работы.

Итак, для каждого поддерживаемого языка программирования по очереди анализатор проверяет, записаны ли в объекте сборки записаны файлы с внутренним представлением программ в этом языке. Если записаны, то организуется поуровневый анализ программы: сначала для анализа на уровне АСД извлекаются сохраненные исходные файлы и команды компиляции для них, строится АСД текущего файла, запускаются детекторы первого уровня и сохраняются их предупреждения; далее для последующих уровней считываются файлы с низкоуровневым внутренним представлением и выполняется алгоритм 2.3 межпроцедурного анализа на основе аннотаций функции, включающий в себя как детекторы второго уровня, так и чувствительные к путям детекторы. Опишем далее подробнее общее устройство этой части анализа на примере программ на Си и Си++, а в разделах 3.3.4-3.3.6 отметим специфику анализа программ на Си++, Java и C# соответственно. Отметим, что основная часть анализатора (шаги 3-6 нижеописанного алгоритма) реализована на языке Java.

Алгоритм 3.1. Многоуровневый анализ исходного кода для Си/Си++.

1. **Запуск детекторов уровня АСД, которые выполняют обходы АСД и таблицы символов**. Детекторы этого класса для Си и Си++ реализованы в части обходов на основе инфраструктуры компилятора Clang и инструмента Clang Static Analyzer (CSA), не привлекая их модель памяти и внутрипроцедурное символьное выполнение. Для каждого исходного файла, компиляция которого была перехвачена на этапе контролируемой сборки и текст которого сохранен после режима препроцессирования rewrite-includes, запускаются детекторы на основе CSA. Для этого дополнительно сохраняется командная строка для нашего компилятора Clang, из нее убираются более не нужные опции, задающие пути для поиска заголовочных файлов, и добавляются опции, включающие заданные пользователем детекторы. Результаты работы детекторов сохраняются в стандартном XML-формате инструмента CSA (файлы `.plist`).
2. **Сортировка файлов с внутренним представлением по целевой архитектуре**. Этот шаг анализа специфичен для Си/Си++, т.к. для остальных языков внутреннее представление – это либо АСД, либо байткод для виртуальной машины, и оно не зависит от архитектуры. В случае Си анализ происходит для каждой целевой архитектуры отдельно (например, в случае анализа ОС Android отдельно анализируются инструменты, собранные для инструментальной машины, как правило, для архитектуры x86-64, и системный уровень ОС, собранный для устройства архитектуры ARM или AARCH64). Так же, как и для случая многоязыковой программы, в будущем планируется поддержать совместный анализ программы, написанной для нескольких архитектур, например, программу

на CUDA, которая выполняется частично на основном процессоре и частично на акселераторе. Граф вызовов такой программы будет общим, но различные функции будут предназначены для выполнения на разных архитектурах.

3. **Построение графа вызовов программы** (шаг 1 алгоритма 2.3). Подробнее этот этап описан в следующем подразделе 3.3.1. Здесь достаточно отметить, что для достижения производительности для программ на языках Си/Си++ реализован специальный режим чтения биткод-файлов – читается только глобальная информация (список глобальных переменных, типов, функций) и тела функций, а отладочная информация, как правило, составляющая большую часть файла, не читается.
4. **Выделение компонент связности в графе вызовов, разрыв циклов** (шаг 2 алгоритма 2.3), этап, необходимый для дальнейшего однократного обхода графа вызовов.
5. **Параллельный межпроцедурный анализ графа вызовов** (шаг 3 алгоритма 2.3). Организации параллельного анализа посвящен раздел 3.3.2. Диспетчер параллельного анализа выдает на выполнение анализа очередную функцию. Для этой функции выполняется внутрипроцедурный анализ, состоящий из следующих шагов:
    a. Построение графа потока управления функции, выделение в нем компонент сильной связности, топологическая сортировка вершин (с учетом выделенных компонент).
    б. Консервативный анализ потока данных функции. Этот этап по сути служит для реализации детекторов уровня АСД, которым требуется сведения о внутрипроцедурном потоке данных (раздел 2.1.2). Однако собранные на этом этапе данные о недостижимости участков графа потока управления и интервалы значений потребуются на дальнейших этапах анализа. Поэтому в инструменте Svace детекторы первого уровня анализа разделены – обход АСД выполняется на шаге 1 в рамках инфраструктуры CSA, а детекторы, требующие базового внутрипроцедурного анализа потока данных, – в основном анализаторе. Разделение оказалось оправданным на практике, т.к. детекторам этой группы, как правило, не требуется сведения об исходном коде, которые нельзя получить из сохраненной отладочной информации. Запускать их на первом шаге анализа и потом сохранять вычисленные сведения о потоке данных и недостижимом коде и считывать эти сведения на основном этапе анализа менее удобно.

в. Основной внутрипроцедурный анализ, вычисляющий множества *PtTo,* классы значений, предикат пути, атрибуты всех детекторов, включая межпроцедурные, согласно алгоритмам 2.6 и 2.6'. Вся совокупность ячеек памяти, классов значений, их атрибутов называется *контекстом* некоторой точки программы. Анализ заключается в продвижении контекста по графу потока управления согласно упомянутым алгоритмам. При обработке текущей точки программы сначала выполняются действия по обновлению модели памяти и классов значений, а далее о посещении точки оповещаются все детекторы, которые могут обновить свои атрибуты. Так как весь контекст функции при обработке одной инструкции меняется незначительно, то в начале базового блока всегда хранится разница между текущим контекстом и контекстом-предшественником, а в пределах базового блока единственный контекст продвигается через инструкции, вообще не храня промежуточных состояний. Это существенная особенность хранения данных при внутрипроцедурном анализе, предложенная еще в первых версиях инструмента Svace [22].

г. Обратный внутрипроцедурный анализ. Для ряда детекторов оказалось удобным выполнять распространение интересующих их атрибутов в обратном направлении по графу потока управления функции (от выходного блока ко входному). Для такого анализа требуется, чтобы для атрибутов были сформулированы соответствующие передаточные функции и функции *Unify*, вызываемые в точках разделения потока управления (точки слияния потока в обратном направлении становятся точками разделения и наоборот). Для каждой точки программы атрибуты вычисляются алгоритмом, похожим на алгоритм 2.6, с той разницей, что граф потока управления обходится в обратном направлении, а передаточные функции вычисляются только для таких специальных атрибутов (классы и интервалы значений, множества *PtTo,* другие атрибуты остаются в неприкосновенности).

д. Построение аннотации функции (алгоритм 2.7).

6. **Сортировка и сохранение предупреждений**. На последнем этапе анализа происходит постобработка выданных на предыдущем шаге анализа предупреждений. Во-первых, из списка предупреждений удаляются дубликаты предупреждений: в том случае, если несколько детекторов из одного класса, например, разыменования нулевого указателя, независимо пришли к выводу о

наличии ошибки в некотором месте программы, то только одно предупреждение будет выдано. Для таких детекторов анализатор поддерживает список приоритетов, т.е. для каждой пары детекторов из единой группы известно, выдача предупреждений каким детектором предпочтительна. Обычно выбирается детектор с более высоким уровнем критичности и с лучшим процентом истинных срабатываний. Во-вторых, удаляются предупреждения для файлов, исходный код которых по каким-либо причинам недоступен – такие предупреждения не удастся показать пользователю.

3.3.1. Построение графа вызовов программы

Граф вызовов строится анализатором Svace в два этапа. Первый этап заключается в сборке информации о том, как именно файлы, составляющие программу, компоновались друг с другом (данные сохраняются при обработке реакции на события сборки, как упоминалось в разделе 3.1). На втором этапе собственно строится граф вызовов с разрешением связей по вызову внешних функций с учетом данных первого этапа, а также эвристических алгоритмов.

Итак, пусть собираемая программа состоит из основной программы и двух библиотек. Основная программа `prog` написана в файле `main.c` и использует внешнюю функцию `foo`. Функция `foo` реализована в файле `foo1.c`, который составляет библиотеку `libfoo1`, а также в файле `foo2.c`, составляющую библиотеку `libfoo2`, но программа `prog` компонуется только с библиотекой `libfoo1`. Обе библиотеки используют функции из файла `common.c`.

Анализатору на вход предъявляются четыре файла с внутренним представлением – `main.bc`, `common.bc`, `foo1.bc`, `foo2.bc`. Необходимо правильно разрешить внешние связи при построении графа вызовов. Несложно заметить, что для этого нужно знать, как соотносятся между собой файлы с исходным кодом и файлы с внутренним представлением; файлы с исходным кодом и объектные файлы; какие объектные файлы были скомпонованы в данный исполняемый файл, файл с архивом или библиотечный файл. Тогда станет понятно, что для искомой функции есть два кандидата из разных файлов, но выбрать нужно только один, так как именно с этим файлом компоновалась основная программа.

Для вычисления всех упомянутых соотношений интересующими анализатор событиями сборки, помимо компиляций, становится ассемблирование файла, запуск компоновщика, запуск архиватора объектных файлов (ar). Перехват компилятора позволяет построить связь между исходным файлом и сгенерированным из него объектным, а также между исходным файлом и файлом внутреннего представления. Часто компилятор генерирует ассемблерный файл, а уже по нему ассемблер создает объектный файл. В этом случае требуется сначала сопоставить исходный файл с ассемблерным файлом, а уже потом этот ассемблерный файл – с

объектным файлом. Перехват компоновщика показывает, как компоновались вместе объектные файлы, причем нужно учитывать, что некоторые из них – ранее собранные библиотеки или архивы, а не единичные объектные файлы.

Так как все события компиляции, ассемблирования, компоновки обычно происходят параллельно, то построение всех нужных соответствий между файлами сложно выполнять в ходе сборки. Удобнее дождаться окончания сборки и обработать совместно все собранные данные. Для этого ведется подробный журнал сборки, куда вносятся записи при наступлении одного из указанных событий. Журнал является сериализованным отражением хода параллельной сборки (подобно журналу утилиты `make`). Записи в журнале обрабатываются последовательно и строятся все необходимые соответствия. Записи связываются между собой через полные пути к исходным, объектным и ассемблерным файлам, которые восстанавливаются, исходя из знаний системы контролируемой сборки о формате команд компилятора и остальных утилит сборки, а также о текущем каталоге запуска утилиты.

Возникают следующие сложности, с которыми необходимо бороться. Во-первых, записи в журнале о запуске ассемблера и компилятора при параллельной сборке могут быть перепутаны местами, то есть не всегда можно предполагать, что при обработке записи о запуске ассемблера запись о запуске соответствующего ему компилятора уже была обработана. Во-вторых, если драйвер компилятора запускает параллельно собственно компилятор и ассемблер и связывает их через каналы (случай `gcc -pipe`), то промежуточного файла с ассемблерным текстом не создается, и связать две записи в журнале через путь к файлу не представляется возможным. Для разрешения этой ситуации на Linux в журнал сборки дополнительно записывается идентификатор родительского процесса, создавшего данный процесс. У компилятора и ассемблера, соединенных каналом, идентификатор будет совпадать. Эта же информация оказывается полезной в случаях, когда компилятор создает временный ассемблерный файл, имя которого в дальнейшем переиспользуется (данную ситуацию можно наблюдать на параллельных сборках большого количества файлов с определенными версиями компилятора `gcc`).

Второй этап построения графа вызовов заключается в построении локального графа вызовов для каждого файла с внутренним представлением, а потом объединении этих графов в один глобальный граф вызовов. Это делается для возможности параллельной обработки входных файлов. В случае вызова внешней функции используется информация о компоновке с первого этапа построения. Если в скомпонованных с данным файлом других файлов можно найти единственного кандидата для разрешения вызова (внешнюю функцию с совпадающей сигнатурой), то выбирается эта функция. Если информация о компоновке неполна либо отсутствует, но во всех входных файлах по-прежнему имеется единственная функция-

кандидат, то выбирается она. Если кандидатов несколько, и один из них находится в файле, путь к которому достаточно похож на путь к данному исходному файлу, то выбирается этот кандидат. Если же ни одного кандидата предпочесть не получается, то создается функция-заглушка (неизвестная функция с заданным прототипом), и вызов помечается как неизвестный.

Далее из графа вызовов удаляются функции-дубликаты. Этот этап специфичен для программ на Си и Си++, в которых включение заголовочных файлов часто приводит к тому, что одни и те же функции оказываются в разных файлах с заголовочным представлением. Желательно оставлять только одну из таких копий функций для ускорения анализа. Для этого достаточно посчитать хэш от тела функции (ее инструкций) с учетом вызываемых ей функций: одна и та же функция, скомпонованная в разных контекстах (например, с разными библиотеками, реализующими одну и ту же функцию), может иметь разную семантику, и удаление ненужных копий можно надежно выполнить только после построения графа вызовов.

Наконец, в построенном графе вызовов перед началом анализа ищутся сильно связанные компоненты, в которых произвольным образом разрывается одно из ребер. Полученный граф является ациклическим, и к нему можно применять межпроцедурный анализ, как описано в алгоритме 2.3.

3.3.2. Организация параллельного детерминированного межпроцедурного анализа

Порядок анализа отдельных функций программы при межпроцедурном анализе задается обходом графа вызовов. Единственным ограничением корректности анализа является обход в обратном топологическом порядке, чтобы гарантировать посещение всех вызываемых функций до вызывающих. При однократном посещении всех функций в ходе анализа сильно связной компоненты графа приходится считать вызов вдоль разорванной в цикле дуги вызовом неизвестной функции. Потеря точности анализа на практике из-за этого, как правило, незначительна. Частично ее можно избежать, дважды проанализировав сильно связные компоненты или хотя бы запустив на первом проходе легковесный анализ потока данных для поиска недостижимого кода. Эти улучшения являются предметом будущих работ в Svace.

Внутрипроцедурный анализ независимых функций (между которыми нет пути в графе вызовов) можно выполнять параллельно. Для этого анализ конфигурируется объемом доступной памяти и количеством выполняющихся параллельно потоков. Важнейшим требованием к внутрипроцедурному анализу функции является *детерминированность* результатов между запусками анализатора и в разных ограничениях по ресурсам, иначе при регулярном анализе исходного кода нельзя полагаться на стабильность разметки результатов. Детерминированность означает, что все ограничения на максимальный размер структур данных при анализе, а также на части аннотации функции не должны зависеть от имеющихся

ресурсов, деталей процесса сборки, конфигурации анализатора и т.п., а могут только опираться на базовые элементы модели памяти и программы (количество ячеек памяти, классов значений, длины логических формул). Построив детерминированную аннотацию функцию, можно обеспечить детерминированность и всего анализа при корректном построении графа вызовов и организации параллельной работы при его обходе.

Диспетчер межпроцедурного анализа определяет порядок обхода графа вызовов и выдает очередную функцию на анализ одному из имеющихся потоков так, чтобы потребляемая потоками память не превысила заданного ограничения. Топологически возможные варианты обхода графа вызовов не являются равноценными по производительности. При этом построить заранее оптимальный порядок обхода представляется затруднительным. Во-первых, сложно оценить время анализа одной функции – оно зависит не только от количества инструкций, но также от количества циклов в ней и их вида, от высоты этой функции в графе вызовов (чем выше функция, тем более объемную аннотацию[5] она генерирует, т.к. ее влияние на память программы включает и влияние вызываемых ей функций). Во-вторых, затруднительно оценить объем необходимой для анализа памяти, включающий в себя внутреннее представление функции, структуры данных для хранения ячеек памяти и их атрибутов, аннотацию функции, т.к. опять же эти параметры сильно зависят от вида графа потока управления функции, ее размера и высоты в графе вызовов. Поэтому задачей диспетчера анализа является определять порядок обхода динамически при работе анализатора.

При выборе функции диспетчер оценивает следующие параметры:

- Объем занятой анализатором памяти и максимально доступный объем памяти;
- Количество свободных потоков;
- Положение границы между уже анализированными и еще не анализировавшимися функциями в графе вызовов;
- Распределение функций по файлам с внутренним представлением. Этот параметр особо существенен для Си и Си++, поскольку каждый файл для анализа содержит множество функций. Производительность анализа в значительной мере определяется тем, как читаются файлы с внутренним представлением, т.к. функции из одного файла могут быть далеко разнесены по графу вызовов (например, вспомогательная статическая функция, которая использует только функции стандартной библиотеки для вывода сообщения об ошибке, и функция `main`), и файл придется читать с диска несколько раз.

[5] До некоторого предела, по достижении которого количество ячеек памяти, классов значений и их атрибутов, а также длины хранящихся логических формул более не растут.

Количество чтений файлов с диска оптимизируется за счет *кэша функций*: при запросе анализа определенной функции читается весь содержащий ее файл, и остальные функции из файла также попадают в кэш. Последующие запросы функций из того же файла не приведут к его повторной загрузке, за исключением ситуации недостатка памяти – тогда давно используемые файлы могут быть удалены из кэша. Размер кэша является компромиссом между скоростью анализа и количеством необходимой памяти – при достаточном объеме памяти все файлы будут прочитаны лишь единожды независимо от вида графа вызовов.

Диспетчер анализа распределяет функции по потокам, исходя из структуры файлов с внутренним представлением. Кандидаты на анализ выбираются группами так, чтобы по возможности анализировать доступные функции из уже загруженных файлов, которые находятся в кэше функций. Кроме того, используется естественная «локальность» функций внутри файла – часто функция вызывает функции в том же файле, а внешними является небольшая часть функций, значит, велики шансы, что после анализа функции другие функции из того же файла станут доступными для анализа.

Для этой цели очередь файлов с представлением для анализа формируется, исходя из *высоты* файла, которая определяется как максимальная высота функции из этого файла в графе вызовов (файлы с минимальной высотой попадают в начало очереди). Далее, проходя по очереди, диспетчер набирает группу функций, доступных для анализа, которые попадают в список кандидатов на анализ. По этому списку диспетчер сначала отдает команды на загрузку файлов, которые еще не были прочитаны, чтобы заполнить кэш функций. Потом выбирается функция для анализа из тех, файлы которых уже загружены, при этом предпочитается функция из файла с большим количеством функций (для использования локальности).

Выданная на анализ функция анализируется одним из свободных потоков. При внутрипроцедурном анализе большую часть памяти занимают структуры данных, описывающих состояние памяти и значений программы в каждой точке программы. Поэтому при запуске на анализ очередной функции потребление памяти может вырасти скачкообразно, и для окончания анализа и формирования аннотации функции (после которых эти структуры данных очищаются) памяти может не хватить. В этой ситуации диспетчер анализа может принять решение о приостановке данного потока анализа до момента, пока не освободится достаточное количество памяти для повторной попытки анализа функции.

Диспетчер отслеживает объем свободной памяти, вычисляя скользящее среднее за последние 100 наблюдений, а также следит за количеством анализируемых процедур, количеством спящих и потоков, получает уведомления о событиях сборки мусора виртуальной машины Java. Далее на основе этих параметров применяется набор эвристических оценок, полученных из многочисленных экспериментов, которые пытаются предсказать, достаточно ли

в данный момент памяти для анализа еще одной функции. В случае положительного решения диспетчер будит поток и предпринимает следующую попытку анализа функции.

После окончания анализа функции она удаляется из кэша функций и освобождаются все структуры данных, связанные с ее анализом, за исключением аннотации функции. Аннотация функции хранится в памяти, но может также сохраняться на диск при нехватке памяти и восстанавливаться при необходимости (обработке вызова функции, для которой сохранена аннотация). Аннотация освобождается тогда, когда все функции, вызывающие данную, проанализированы – в ней более нет необходимости.

Наконец, параллельные потоки анализа обслуживаются внешними утилитами, не реализованными на Java, к которым анализатор по мере необходимости делает запросы. В настоящий момент таких утилит две – это `c++filt`, программа, восстанавливающая сигнатуры функций по их декорированным именам (demangling), и SMT-решатель Z3. Для обеих утилит поддерживается пул параллельно работающих процессов, в котором выбирается один из процессов для выполнения запроса. В случае `c++filt` достаточно нескольких процессов на анализ, в случае Z3 по умолчанию создается по одному процессу на каждый поток анализа. Если какой-либо из процессов заканчивается аварийно, то в пул добавляется новый процесс. Это повышает надежность, т.к. иначе крах нативного процесса может привести к аварийному завершению всей виртуальной машины Java.

Организация параллельной работы анализа, используемая в инструменте Coverity [196], также содержит схожие соображения о диспетчере параллельного анализа, детерминизме результатов и аннотаций, построении расписания по высоте критического пути в графе вызовов. Кроме того, в [196] приведено важное соображение о целесообразности разделения этапов вычисления необходимой информации в ходе абстрактной интерпретации и ее использовании для выдачи ошибок детекторами – в этом случае можно достичь лучшей масштабируемости. К сожалению, в текущей архитектуре анализа Svace сложно выполнить такое разделение, однако работа над ним планируется в будущем.

3.3.3. Спецификации внешних функций

В ходе межпроцедурного анализа очередной функцией, выданной на анализ, может оказаться неизвестная анализатору функция (для нее нет тела, а известен только прототип функции). Однако функция может находиться в списке тех, для которых имеется *спецификация* – описание ее поведения, содержащее необходимые для анализа сведения, что позволит более точно обработать вызовы этой функции.

Спецификации функции пишутся на том же языке программирования, программы на котором анализируются. Это позволяет единообразно работать со спецификациями и

обычными пользовательскими функциями. А именно, спецификации компилируются собственными компиляторами в файлы с внутренним представлением, и при построении графа вызовов эти файлы загружаются перед всеми пользовательскими файлами. Функция, для которой написана спецификация, анализируется так же, как и пользовательская функция, с одним отличием. Текст спецификации пишется с использованием специальных встроенных функций Svace, которые называются *спецфункциями*, и при обнаружении вызова спецфункций в ходе межпроцедурного анализа модель программы, вычисляемая анализом, напрямую модифицируется нужным образом. Это, как правило, удобнее, чем добиваться того же эффекта с помощью эквивалентного кода на Си. Кроме того, даже для упрощенного Си-кода по результатам анализа в аннотации будут отражены изменения модели программы, которые не интересны анализатору. Наконец, заводя спецфункции с понятным неспециалисту в области статического анализа смыслом, можно дать возможность пользователю создавать свои спецификации для применяемых им библиотек. Эта возможность является существенной для удобного использования анализатора, т.к. распространять спецификации на все мыслимые библиотеки вместе с инструментом нереалистично.

```
(1)   void free (void *ptr) {
(2)     sf_set_must_be_not_null (ptr, FREE_OF_NULL);
(3)     sf_delete (ptr, MALLOC_CATEGORY);
(4)   }
(5)   void exit (int code) {
(6)     sf_terminate_path ();
(7)   }
```

Рисунок 3.6. Пример спецификаций функций.

На рисунке 3.6 приведен пример текста спецификаций для двух библиотечных функций. В первом случае функции `free` вызов спецфункции `sf_set_must_be_not_null` на строке 2 означает, что переданный ей параметр должен быть ненулевым, и в случае нарушения этого правила будет выдано предупреждение типа, соответствующего второму параметру спецфункции. Вызов спецфункции `sf_delete` в третьей строке отражает, что функция освобождает ресурс, переданный через первый аргумент, и этот ресурс – динамическая память. В случае функции `exit` важно знать, что ее вызов завершает программу, а значит, код после этой функции недостижим. Для этого служит вызов спецфункции `sf_terminate_path` на

строке 6, помечающий, что на данном пути выполнения после этого вызова программа завершает свою работу.

В таблице 3.1 приведен список наиболее важных спецфункций с кратким описанием и примером библиотечной функции, в спецификации которой применяется данная спецфункция.

Таблица 3.1. Спецфункции для программ на языках Си и Си++.

Спецфункция	Описание	Пример библиотечной функции
`void sf_handle_acquire (void *ptr, int reserved);`	Ассоциирует указательную переменную `ptr` с выданным дескриптором по управлению ресурсом.	`FILE *popen (const char *command, const char *mode);`
`void sf_not_acquire_if_eq (const void* pval, int var, int code);`	Обозначает, что дескриптор управления ресурсом не выдается, если `var == code`. Обычно комбинируется с предыдущей спецфункцией `sf_handle_acquire`.	`FILE *fopen (const char *filename, const char *mode)`, если функция возвращает `NULL`.
`void sf_not_acquire_if_less (const void* ptr, int var, int code);`	Обозначает, что дескриптор управления ресурсом не выдается, если `var < code`. Обычно комбинируется со спецфункцией `sf_handle_acquire`.	`int asprintf (char **ret, const char *format, ...)`, если функция возвращает ноль или отрицательное значение.
`void sf_handle_release (void *ptr, int reserved);`	Маркирует освобождения дескриптора управления ресурсом, ассоциированного с переменной `ptr`.	`int fclose (FILE *stream);`
`void sf_must_not_be_release (void *ptr);`	Обозначает, что дескриптор управления ресурсом не освобождается до конца работы заданной функции.	`char *fgets (char *s, int num, FILE *stream);`
`void`	Помечает, что по указателю	`void *malloc`

`sf_new (void* pval, int category);`	`pval` выделена динамическая память, требующая освобождения. Параметр category обозначает тип функции (работающей по аналогии с `malloc`, `kalloc`, `new` и некоторыми другими).	`(size_t size);`
void `sf_delete (void* pval, int category);`	Помечает, что динамическая память, выделенная по указателю `pval`, освобождена.	**void** `free (void *ptr);`
void `sf_escape (void *ptr);`	Помечает, что память по указателю `ptr` остается живой за пределами функции (англоязычный термин – escapes).	`ssize_t getline (char **lineptr, size_t *n, FILE *stream)` – для указателя `*lineptr`
void `sf_buf_size_limit (const void* ptr, int bytes_size);`	Обозначает, что по указателю `ptr` доступно `size` байт памяти, и доступ по этому указателю за пределами индексов `[0, bytes_size-1]` является ошибкой.	**char** `*fgets (char *s, int num, FILE *stream);`
void `sf_buf_size_limit_read (const void* ptr, int bytes_size);`	Аналогично предыдущей спецфункции `sf_buf_size_limit`, но доступ запрещен только по чтению.	**char** `*strncat (char *s, const char *append, size_t count);`
void `sf_set_possible_null (const void* ptr);`	Обозначает, что указатель `ptr` может быть равен `NULL`.	**char** `*getenv (const char *key);`
void `sf_not_null (const void* ptr);`	Обозначает, что указатель `ptr` не может быть равен `NULL`.	**void** `*operator new (size_t size) throw (std::bad_alloc);`
void `sf_set_must_be_not_null`	Обозначает, что указатель `ptr` разыменовывается функцией, и	**void** `free (void *ptr);`

(**const void*** ptr, **const char*** **const** warning_type);	поэтому не должен быть равен NULL.	
void sf_fread (**int** res, **int** file);	Обозначают соответствующие функции стандартной библиотеки Си для проверки семантики функции fread.	size_t fread (**void** *ptr, size_t size, size_t nitems, FILE *stream);
void sf_ferror (**int** var);		**int** ferror (FILE *stream);
void sf_feof (**int** var);		**int** feof (FILE *stream);
void sf_must_be_checked (**int** var);	Помечает, что результат работы данной функции должен быть проверен в вызывающей функцией.	**int** fclose (FILE *stream);
void sf_set_possible_negative (**int** int_val);	Помечает, что значение int_val может быть отрицательным.	size_t find (**const** basic_string& str, size_t pos = 0);
void sf_set_must_be_positive (int int_val);	Помечает, что значение int_val не должно быть отрицательным. Обычно используется совместно с предыдущей спецфункцией.	**int** connect (**int** sockfd, **const** struct sockaddr *addr, socklen_t len) – возвращаемое значение должно быть проверено на неотрицательность перед использованием.
void sf_set_tainted (**const void*** str); **void** sf_set_tainted_int (**int** i);	Помечает, что значение по указателю str либо целочисленное значение i получено от ввода пользователя.	ssize_t recv (**int** s, **void** *buf, size_t len, **int** flags);
void sf_set_trusted_sink_int	Помечает, что значение по указателю str либо	**int** chdir (**const char** *fname);

(**int** i); **void** sf_set_trusted_sink_ptr (**const void*** str);	целочисленное значение i должно быть доверенным.	
void sf_bitcopy (**void*** dst, **const void*** src);	Память по указателю src в точности копируется в память по указателю dst.	**void** *memcpy (**void** *dst, **const void** *src, size_t num);
void sf_sanitize (**const char*** str);	Память по указателю str проверяется на корректность.	size_t strlen (**const char** *s);
void sf_strlen (**int** len, **const char*** str);	В аргументе len вычисляется длина строки str.	size_t strlen (**const char** *s);
void sf_lock (**const void*** lock); **void** sf_unlock (**const void*** lock); **void** sf_trylock (**const void*** lock);	Функции, моделирующие работу синхронизационных примитивов через интерфейсы захвата и освобождения абстрактного мьютекса.	**int** pthread_mutex_lock (pthread_mutex_t *mutex); **void** _raw_spin_lock (raw_spinlock_t *mutex);
void sf_thread_shared (**const void** *data);	Помечает, что данные по указателю data являются разделяемыми между потоками выполнения.	**int** pthread_create (pthread_t *thread, **const** pthread_attr_t *attr, **void** *(*start_routine) (**void** *), **void** *arg);
void sf_use_format (**const void*** str);	Помечает, что строка str используется как форматная строка.	**int** fprintf (FILE *stream, **const char** *format, ...);
void sf_fun_printf_like (**int** format_string_index);	Помечает, что данная функция имеет интерфейс, схожий с интерфейсами функций	**int** snprintf (**char** *str, size_t size, **const char**

`void sf_fun_scanf_like (int format_string_index);`	`printf/scanf`, где `format_string_index` задает позицию аргумента форматной строки.	`*format, ...);`
`void sf_could_throw (const char*);`	Показывает, что на данном пути выполнения функция, возможно, выбрасывает исключение.	`void *operator new (size_t size) throw (std::bad_alloc);`

3.3.4. Особенности анализа программ на Си++

Изложенное выше в разделах 3.1 и 3.2 устройство анализатора применимо и к программам на Си++ за счет того, что используется единое внутреннее представление для анализа, единый компилятор для Си/Си++ и формат биткода, применимый к обоим языкам. Отметим некоторые существенные особенности. Во-первых, доработки диалектов для Си++ обычно существеннее по размерам и сложности, чем для Си, во многом за счет необходимости поддержки экосистемы Windows (компилятор MSVC). Во-вторых, размер файлов с исходным кодом и биткодом в среднем значительно больше за счет объемных заголовочных файлов, дополнительных уровней абстракции, для биткода – за счет объема отладочной информации. В этих условиях особенно важным становится удаление лишних копий функций из графа вызовов для более быстрого анализа. В-третьих, растет роль спецификаций функций стандартной библиотеки, поскольку восстановить необходимую анализатору семантику функций через весь объем вспомогательного кода, доступного в заголовочных файлах, затруднительно.

Важным изменением внутреннего представления по сравнению с Си является наличие исключений. С точки зрения биткода LLVM вызов функции, которая может выбросить исключение, осуществляется с помощью инструкции `invoke`, которая показывает направление передачи потока управления в случае появления исключения. При конвертации биткода во внутреннее представление Svace в базовом блоке, содержащем код, который выполняется при возникновении исключения, вставляется специальная инструкция семейства `assume` (подробнее об инструкциях внутреннего представления см. главу 4), которая отмечает это предположение в тексте функции. Дальнейший анализ исключений, выполненный в виде анализа потока данных, отслеживает вызовы специальных функций (`__cxa_throw`, `__cxa_begin_catch`) и распространяет атрибут, означающий, что в данной точке

программы будет выброшено исключение, а также использует его для определения недостижимого кода из-за исключения.

Вторым структурным изменением анализа, связанным с Си++, является фаза анализа конструкторов и деструкторов. Основной целью является проверить конструкторы и деструкторы на согласованность между собой – ресурсы, выделяемые конструктором, должны освобождаться деструктором. Для этого выполняется так называемый *парный* анализ: для двух функций `f1`, `f2` генерируется и анализируется обычным образом код, вызывающий сначала `f1`, а затем `f2`. Тем самым анализатор и детекторы ошибок наблюдают ситуацию консистентного вызова функций непосредственно, и после анализа функции `f2` могут обнаружить любые несогласованности.

Парный анализ запускается для конструкторов и деструкторов, а также для операторов присваивания и деструкторов. Так как деструктор не имеет параметров и не возвращает результата, то в данном случае необходимый код генерируется вставкой вызова конструктора в начало деструктора. После окончания анализа деструктора запускается детектор поиска утечек памяти, и если им найдены ячейки памяти, которые не были освобождены, то это указывает на утечку ресурсов, выделенных в конструкторе (подробнее см. раздел 4.2.1).

3.3.5. Особенности анализа программ на Java

Многоуровневый анализ Java-программ выполняется следующим образом. На первом этапе описанным в разделе 3.1 образом организуется контролируемая сборка программы, сохраняются исходные коды на Java, а также генерируются class-файлы для последующего анализа собственным компилятором, описанным в разделе 3.2. На этапе анализа детекторы уровня АСД реализованы внутри нашего компилятора, а детекторы уровня внутрипроцедурного потока данных – внутри открытого анализатора FindBugs версии 3.0, на вход которому на каждую исходную компиляцию Java-файлов подается набор class-файлов, соответствующих результатам этой компиляции.

Для межпроцедурных и чувствительных к путям детекторов используется основная инфраструктура анализа Svace. Сначала читаются все JAR-библиотеки, которые требовались при компиляции, и сгенерированные class-файлы, и байткод этих классов преобразуется во внутреннее представление Svace. Байткод представляет из себя код для стек-машины Java, при этом для каждого базового блока некоторого метода известна входная глубина и максимально используемая глубина стека операндов. Тем самым становится возможным непосредственная трансляция байткода во внутреннее представление, когда для каждой глубины стека заводится локальная ячейка памяти, и в ходе трансляции полностью моделируется состояние стека.

Загрузка значения из стека транслируется как загрузка значения из ячейки памяти, соответствующей текущей глубине стека, аналогично и запись в стек.

Таблица 3.2. Пример трансляции байткода Java во внутреннее представление.

Байткод	Тривиальная трансляция	Трансляция с оптимизацией
`iload_1`	`tmp1 = deref addr_x` `pmove tmp1 to stack0`	`tmp1 = deref addr_x`
`iload_2`	`tmp2 = deref addr_y` `pmove tmp2 to stack1`	`tmp2 = deref addr_y`

`iadd`	`tmp3 = deref stack1`
	`tmp4 = deref stack0`
	`tmp5 = tmp3 + tmp4`
	`pmove tmp5 to stack0`
		`tmp3 = tmp1 + tmp2`

`istore_3`	`tmp6 = deref stack0` `pmove tmp6 to addr_z`	`pmove tmp3 to addr_z`

Однако такая трансляция приводит к коду с большим количеством лишних пересылок, и дополнительно при обработке базового блока промежуточные результаты вычислений оставляются на псевдорегистрах, которые переиспользуются при условии, что соответствующие ячейки стека не были перезаписаны (см. таблицу 3.2). Окончательно значения с псевдорегистров сохраняются обратно в соответствующие ячейки памяти при условии, что базовый блок заканчивается с непустым стеком (то есть они будут использоваться в дальнейшем). Эти значения можно брать с псевдорегистров и в следующих базовых блоках при условии, что вдоль разных путей выполнения ячейки стека заполнялись одинаково – это условие нарушается сравнительно редко, например, при трансляции тернарных операторов.

Можно также провести после окончания трансляции дополнительный анализ, который выгрузит на псевдорегистры все ячейки памяти, которые не остаются живыми после окончания работы функции. Подобного рода преобразование реализовано в трансформации `mem2reg` в рамках инфраструктуры LLVM. Однако при этом необходимо сохранять соответствие с исходным кодом программы, чтобы не потерять информацию о том, какие псевдорегистры находились в памяти, и сейчас данная оптимизация является предметом будущих работ.

Отдельно отметим, что для построения связи между переменными внутреннего представления и переменными программы необходимо учитывать, что в байткоде ячейки стека могут переиспользоваться для локальных переменных, области видимости которых не пересекаются. Для точного сопоставления необходимо пользоваться отладочной информацией, сохраняемой в байткоде.

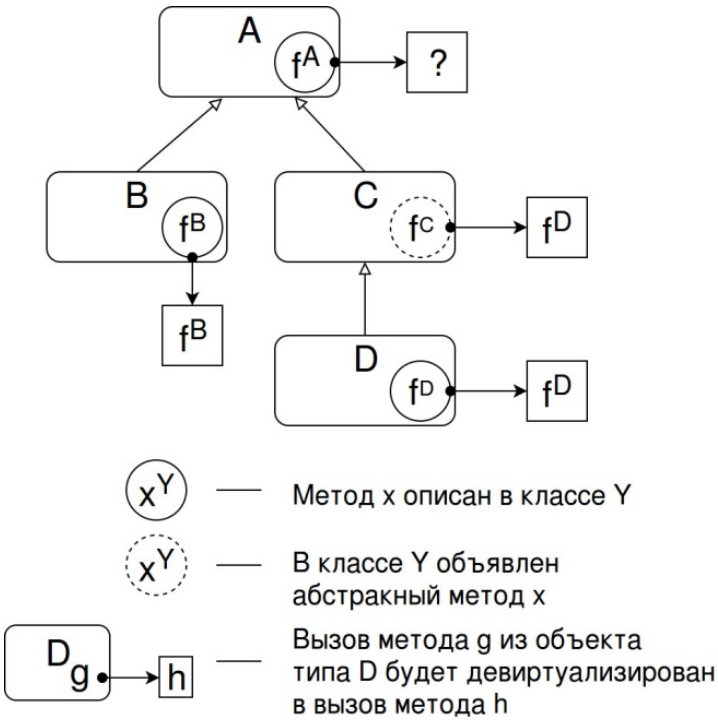

Рисунок 3.7. Пример девиртуализации.

После чтения байткода выполняется построение графа вызовов. Для этого необходимо выполнять *девиртуализацию*, то есть при вызове виртуального метода, в случае наличия нескольких кандидатов, по возможности определить единственного кандидата либо хотя бы сузить множество возможных кандидатов. Для Java используется набор эвристик, которые исходят из доступной анализу иерархии классов и девиртуализуют только те случаи, для которых можно отсеять всех кандидатов, кроме одного. Например, на рисунке 3.7 для объектов типа B, C и D можно однозначно определить, какая именно функция будет вызвана. В классе C метод `f()` объявлен как абстрактный, однако поскольку единственной реализацией в примере является реализация из класса D, то можно точно сказать, что эта реализация и будет вызвана. Если же переменная имеет тип A, то её реальный тип может быть A, B или D, что не позволяет девиртуализовать вызовы метода `f()` у переменных с типом A.

Диспетчер параллельного анализа программы на языке Java использует более простой алгоритм выбора следующей функции для анализа: так как один файл с байткодом в Java содержит тело единственного класса, и их размер, как правило, невелик, то для заполнения кэша функций не требуется учитывать распределение функций по файлам. Поэтому при заполнении списка кандидатов на анализ выбирается следующая функция, исходя из топологического порядка в графе вызовов.

При создании спецификаций для функций стандартной библиотеки Java нужно учитывать особенности ее устройства. Например, при вызове метода `close` у объекта через интерфейс

базового класса будет очищаться выделенный ресурс. Следовательно, описанной выше девиртуализации становится недостаточно, чтобы отслеживать утечки ресурсов, так как вызов остался бы виртуальным. Задача решается написанием спецификаций интерфейсов и абстрактных функций, чтобы превратить виртуальный вызов функции из стандартной библиотеки в подстановку спецификации.

Другой особенностью является наличие большого количества классов-оберток, таких, как `BufferedInputStream`. В результате надо учитывать, что вызов метода `close` из базового интерфейса будет, как правило, закрывать все дочерние ресурсы. Это предположение применяется при обработке спецификаций по умолчанию, а также предоставляется спецфункция `sf_handle_recursive_release (Object ptr)`.

Большинство спецфункций для Java аналогичны таковым для языков Си и Си++ в случае, если они отражают изменения модели программы общего характера, не связанные с особенностями стандартных библиотек. Только для Java заведены функции `sf_monitorenter (Object ptr)` и `sf_monitorexit (Object ptr)`, которые отражают свойства соответствующих инструкций байткода, использующихся для реализации конструкции `synchronize (ptr)` эксклюзивного доступа.

3.3.6. Особенности анализа программ на C#

Анализ C# в рамках инструмента Svace разработан и реализован как отдельная инфраструктура, интегрированная в общую архитектуру анализатора. Основой для компиляции файлов C# является проект Roslyn, открытый компанией Microsoft в 2015 году как единая платформа компиляции для языков C# и Visual Basic. Все уровни анализа программ на C# используют АСД компилятора Roslyn как внутреннее представление.

Для построения АСД на этапе контролируемой сборки используется два подхода. Изначальный подход использовал в качестве входных данных файл решения (solution file) среды Visual Studio. Этот файл описывает проекты, входящие в решение, конфигурации и способ их сборки, и аналогичен файлам для утилит сборки `make` и `cmake`. Среда Roslyn умеет разбирать такие файлы и компилировать их нужным способом, тем самым в первом приближении необходимость выполнения исходной контролируемой сборки отпадает. Однако в существующей реализации поддержки файлов решений в Roslyn есть недоработки, которые приводят к ошибкам обработки некоторых файлов, а также к неверному выбору конфигураций проектов для сборки, которые не применялись пользователем. Кроме того, не поддерживалась компиляция в системе Mono.

Текущий подход использует инфраструктуру контролируемой сборки, описанную в разделе 3.1, для перехвата запуска утилит MSBuild и xbuild, которые осуществляют сборку

программ на C# для Visual Studio и Mono соответственно. При обнаружении запуска командная строка утилит модифицируется для указания пути к собственной библиотеке отслеживания событий сборки (логгеру). Обе утилиты реализуют интерфейсы для мониторинга событий сборки, через которые можно получать сведения о запуске компиляторов. Далее, если запущен компилятор C#, то логгер получает оповещение об этом напрямую из утилит сборки, минуя общие механизмы нашей системы контролируемой сборки. Логгером собираются все исходные коды и файлы-сборки C# (assemblies) для последующей компиляции собственным компилятором на основе Roslyn на этапе анализа.

Собственно при начале анализа данные о выполненных компиляциях извлекаются, и для каждой компиляции запускается Roslyn для построения представления программы в виде набора абстрактных синтаксических деревьев для методов и таблиц символов для модулей компиляции. Далее для каждого АСД метода запускаются детекторы первого уровня анализа.

Для последующих уровней сначала строится граф вызовов программы. Это происходит в два этапа. Сначала обходятся таблицы символов для построения иерархии классов программы. Далее обходятся АСД методов, в которых ищутся вершины с вызовами методов, геттеров/сеттеров, с созданием лямбда-выражений и анонимных функций, с неявными вызовами операторов преобразования, переопределенных операторов, финализаторов, и проводятся соответствующие дуги к вызываемым функциям. Наконец, запускается алгоритм девиртуализации на основе адаптированного для C# подхода к девиртуализации через анализ иерархии классов (Class Hierarchy Analysis).

Алгоритм девиртуализации обходит все скомпилированные файлы и для каждого выражения вызова упомянутых выше типов, встречающихся в C#, проводит дуги в графе вызовов до методов классов, совпадающих по имени и сигнатуре с указанными в выражении вызова. При этом возможный класс должен соответствовать статическому типу объекта – быть его наследником, самим этим классом или ближайшим по иерархии базовым классом с нужным методом. Потенциальные динамические классы определяются с учетом виртуальности методов (ключевые слова `virtual`, `base` и `override`). Отдельно обрабатываются делегаты – типы-контейнеры для наборов методов с определенной сигнатурой, связь которых с конкретными методами можно устанавливать динамически операциями `+=`, `-=` и методами `Combine`/`Remove`. Анализируются указанные операции и методы, чтобы получить список методов-кандидатов для вызова, и считается, что могут быть вызваны все методы из этого списка.

После построения графа вызовов сильно связанные компоненты в нем разрываются, аналогично основному движку Svace, и выполняется алгоритм 2.3 межпроцедурного анализа на основе аннотаций – функции посещаются в обратном топологическом порядке, выполняется

внутрипроцедурный анализ каждой функции, после чего создается ее аннотация и используется в дальнейшем при обработке вызовов данной функции. Поддерживается параллельный анализ независимых функций, аналогично описанному в разделе 3.3.2, с более простым выбором следующей функции для анализа – как и в Java, для достижения хорошей производительности нет нужды учитывать распределение функций по файлам с внутренним представлением. При необходимости (нехватке памяти) АСД функций и аннотации функций сериализуются на диск и считываются в момент возникновения нужды в них.

Опишем подробнее ход внутрипроцедурного анализа. В отличие от инфраструктуры межпроцедурного анализа Svace для Си, Си++ и Java, в случае C# анализ выполняется не над трехадресным представлением, а над вершинами АСД и графом потока управления непосредственно. С одной стороны, это позволяет удобно пользоваться всей информаций об исходном коде программы, не заботясь о ее возможных потерях при трансляции в представление более низкого уровня, не восстанавливая высокоуровневые конструкции по низкоуровневым и т.п. С другой стороны, возникает некоторая избыточность при реализации анализатора из-за большого количества типов узлов АСД. Более существенным недостатком является невозможность автоматического анализа библиотек C# в формате байткода CIL, аналогично тому, как анализируются JAR-библиотеки для программ на языке Java. В будущем планируется выполнять легковесный анализ CIL-библиотек, однако, к сожалению, переиспользовать для этого существенную часть реализованной инфраструктуры анализа C# не представляется возможным.

Граф потока управления для внутрипроцедурного анализа строится непосредственно по АСД метода. Ребра графа размечены для облегчения последующего анализа. В частности, хранятся пометки об обратных ребрах циклов, ветвях условных операторов, о передаче управления через исключения (пользовательские либо системные) и т.п. Дело в том, что в C# практически любая инструкция или вызов функции может вызвать исключение, и необходимо различать явно брошенные пользователем исключения и те исключения, которые, например, генерируются библиотечными функциями. Явное разделение базового блока из-за возникновения исключений выполняется только для выброса пользовательских исключений, а остальные ребра добавляются как возможные выходы в конце базового блока.

Такое решение имеет свои недостатки, как демонстрируется рисунком 3.8. В примере на рисунке для того, чтобы найти ошибку возможного разыменования нулевого указателя в строке 11, нужно добавить ребро в граф потока управления, отображающее, что на строке 9 после вызова управление может перейти сразу на строку 11 без выполнения присваивания. То есть базовый блок с `try` необходимо разделить, даже если исключение происходит из-за вызова функции, а не из-за явного использования оператора `throw`.

```
(1)  public StreamReader GetStreamReaderOrThrow (bool b) {
(2)    if (b)
(3)      return new StreamReader ("finestream");
(4)    throw new NotImplementedException ();
(5)  }
(6)  public void foo (bool b) {
(7)    StreamReader reader = null;
(8)    try {
(9)      reader = GetStreamReaderOrThrow (b);
(10)   } finally {
(11)     reader.Dispose ();
(12)   }
(13) }
```

Рисунок 3.8. Пример кода с исключениями для C#.

Другую сложность представляют лямбда-выражения и анонимные функции. Точное место их вызова затруднительно определить без чувствительного к путям межпроцедурного анализа. Поэтому в качестве компромисса при работе детекторов уровня внутрипроцедурного потока данных выполняется непосредственное «встраивание» тела лямбды в граф потока управления – добавляются ребра, ведущие в лямбду и из лямбды, а также обратное ребро из выхода лямбды во вход и ребро, ведущее в обход лямбды. Так отражается заранее неизвестное количество выполнения лямбды. При работе детекторов межпроцедурного и чувствительного к путям уровня добавленные ребра снова разрываются.

Первым этапом внутрипроцедурного анализа является классический анализ потока данных, который проводится для только внутрипроцедурных детекторов, например, для поиска константных и неиспользуемых выражений. Далее выполняется построение модели памяти и программы на основе ячеек памяти и вычислении символьных выражений, как описано в разделе 2.3.2 (символьное выполнение организуется без участия классов значений, как в основном движке Svace, а напрямую над символьными выражениями), строятся логические формулы для чувствительных к путям детекторов, создается аннотация функции.

Дополнительно выполняются детекторы на основе анализа помеченных данных, который организуется как на основе символьного выполнения, так и на основе IDFS-подсистемы анализа. Так как последняя реализована только в рамках анализа программ на C#, рассмотрим

ее подробнее. Алгоритм IDFS-анализа заключается в межпроцедурном распространении информации о помеченных данных от источников данных (прямой анализ) до стоков данных (обратный анализ) независимо для каждого источника или стока. Направление анализа выбирается для каждого детектора индивидуально в зависимости от соотношения количества источников и стоков.

Единицей пометки данных является *путь доступа*, определяемый как некоторая последовательность обращений к полям или элементам массива, начиная от локальных переменных, параметров, литералов, статических полей или ссылки `this`. *Точкой анализа* является информация о том, что заданный путь доступа помечен в заданной точке программы. Помеченный путь доступа вместе с предыдущим местом пометки называется *фактом*. Сведения о каждом пути доступа распространяются независимо от других путей.

Внутрипроцедурный анализ начинается от точек источников или входных точек в методы, продвигает пометки вдоль графа потока управления и создает аннотации функции. Если для заданной точки программы есть аннотация (вызов функции), то результатом продвижения являются точки анализа из аннотации, у которых входная точка заменена на текущую точку анализа. Если же нужно выполнить обычное продвижение, то поддерживается очередь из точек на обработку, для каждой из которых выполняется распространение вдоль следующей инструкции межпроцедурного графа потока управления, и полученные точки записываются в очередь. По окончании анализа точки, которые достигли конца метода, записываются в его аннотацию.

Распространение вдоль инструкции заключается либо в применении обычных передаточных функций, которые достраивают путь доступа согласно виду инструкции, либо – при обработке вызова – в сопоставлении путей доступа из фактических параметров в формальные и в рекурсивном обращении к алгоритму внутрипроцедурного распространения, после чего пути доступа, возвращенные алгоритмом, переписываются в точку возврата в вызывавшей функции. При этом сохраняются только те пути, которые начинаются от параметров метода, статических полей, либо от `this`, т.к. другие пути вызывающему методу недоступны.

Так как анализ помеченных данных начинается не с метода `main`, а с источников или стоков, в ходе анализа может возникнуть ситуация несбалансированных вызовов и возвращений из методов, при которой анализ возвратился в метод, с которого он начинался, и выход из этого метода происходит в неизвестное место. Для обработки этой ситуации требуется найти все места вызова заданного метода и продолжить анализ «возвращением» в точки после этих мест (см. рисунок 3.9, на котором зелеными стрелками помечены

сбалансированные возвраты, а красными – несбалансированные; анализ начинается с первого метода).

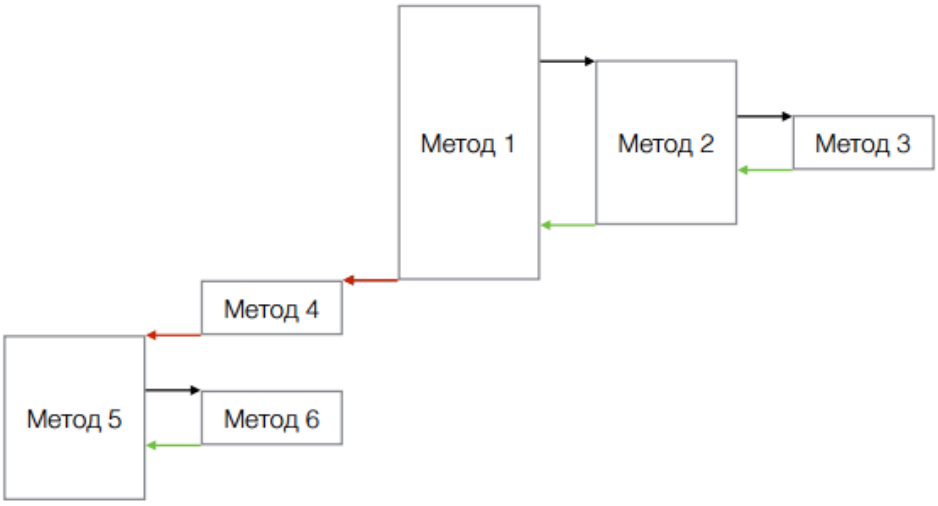

Рисунок 3.9. Несбалансированный возврат из метода.

Остановимся подробнее на способах записи спецификаций неизвестных функций, используемых при анализе программ на C#. Первый способ в рамках инфраструктуры Svace реализован только для C# и основан на том, что для библиотеки C# доступна подробная официальная документация MSDN, которая в описании конкретного метода содержит информацию о его особенностях (выбрасываемых исключениях, ограничениях на параметры и т.п.). Эта информация разбирается автоматически и сохраняется в базу данных, содержащую в настоящий момент информацию о свойствах более 60 тыс. стандартных функций. При этом требуется небольшое количество ручных правок результатов автоматического разбора. Тем самым такой способ позволяет быстро построить спецификации для большого количества методов, которые удовлетворительны для большинства детекторов.

Однако не все побочные эффекты методов могут быть смоделированы таким образом. Для характерного примера кода на рисунке 3.10 на строке 8 выдается ложное предупреждение о том, что возможно разыменование нулевой ссылки `check`. Дело в том, что после вызова метода `List<T>.Add(T)` в строке 3 известно, что список `list` не пуст, однако в автоматически построенной по документации спецификации этого метода невозможно представить эту информацию, и анализатор считает, что цикл на строках 5-7 может ни разу не выполниться. Поэтому дополнительно к автоматической спецификации используется способ записи спецификации в виде упрощенной реализации метода на языке C# с вызовом специальных функций анализатора, которые напрямую модифицируют модель программы и памяти нужным образом. Этот способ является основным для всех других языков, поддерживаемых Svace.

```
(1)  public string Bar () {
(2)    var list = new List<int>();
(3)    list.Add(5);
(4)    string check = null;
(5)    foreach (var elem in list) {
(6)      check = "Not null";
(7)    }
(8)    return check.ToString ();
(9) }
```

Рисунок 3.10. Ложное срабатывание из-за неточности спецификации.

3.3.7. Удаленный и инкрементальный анализ

Для применения статического анализатора в промышленных условиях постоянной разработки кода важной является поддержка двух сценариев использования. Первый сценарий заключается в том, что все три этапа применения анализатора (контролируемая сборка, собственно анализ и просмотр результатов) могут выполняться на разных машинах. Как правило, в компаниях на момент внедрения статического анализа уже развернуты системы регулярной сборки своих приложений на отдельных серверах, и для обеспечения надежности администраторы IT-инфраструктуры неохотно идут на выполнение каких-либо других задач на этих машинах, либо категорически это запрещают. Кроме того, требования к машине запуска анализатора на миллионах строк кода другие, чем для сборочных серверов – важен значительный объем памяти и количество ядер. Наконец, сервер, аккумулирующий результаты периодических запусков анализатора, должен обладать большими дисками для хранения информации, а также давать возможность многим разработчикам просматривать результаты анализа независимо друг от друга.

Этот сценарий реализуется в Svace с помощью хранения всей необходимой информации для анализа в объекте сборки, а для просмотра результатов – в объекте анализа (см. раздел 3.4). Этап сборки должен сохранить файлы с внутренним представлением и исходные файлы для всех поддерживаемых языков, команды компиляции, трассы сборки, необходимые библиотеки, данные по разметке исходного кода и т.п. Этап анализа должен дополнительно сохранить результаты работы всех уровней анализа, которые выполняются отдельными анализаторами. Между этапами хранилище с объектами сборки и анализа может быть свободно перенесено

между машинами. Кроме того, команды на выполнение анализа можно выдавать анализатору удаленно по сети. При этом задается объект сборки и настройки для анализа.

Вторым важнейшим сценарием использования является инкрементальный анализ. Его необходимость возникает в следующих случаях. Во-первых, время этапа сборки обычно составляет значительную долю времени анализа (30-100%). При постоянных запусках анализатора (ежедневных, раз в несколько дней) значительная часть исходного кода не меняется. Поддержка инкрементальной сборки позволила бы выполнять полную пересборку кода реже, например, раз в две недели, а ежедневно делать инкрементальную сборку изменившегося кода и последующий полный переанализ всего кода. Для этого необходимо обновить в объекте сборки все изменившиеся файлы с исходным кодом, перестроить файлы с внутренним представлением.

Во-вторых, популярным случаем является совместная работа многих программистов над единым большим проектом. Например, программисты занимаются развитием разных частей ОС Android, включая как инфраструктурную часть на Си/Си++, так и приложения на Java. В ходе дневной разработки программист хотел бы как можно быстрее проверять только что написанный код статическим анализатором, не дожидаясь результатов периодического анализа всего проекта. Программиста интересуют только предупреждения на том коде, который он менял, но при этом при анализе этого кода нужно учитывать все окружение проекта, то есть весь остальной код Android.

Наконец, популярным современным сценарием является применение статического анализатора на этапе CI (Continuous Integration). При подаче изменения в коде (патча) на рецензию экспертами-разработчиками одним из условий просмотра патча рецензент хочет ставить отсутствие предупреждений статического анализатора. Для этого анализ измененного участка должен происходить в течение нескольких минут, а его результаты должны быть просмотрены программистом и либо исправлены, либо помечены как ложные или нерелевантные для последующей проверки рецензентом. Такая система публично развернута, например, для проверки изменений в коде ОС Tizen [189].

Упомянутые сценарии поддерживаются анализатором Svace в специальном режиме *быстрого* или *инкрементального* анализа. Предполагается, что была выполнена полная контролируемая сборка и полный анализ на некотором сервере, при этом анализатору было дано указание сохранить граф вызовов программы и все построенные аннотации функций. Такой сервер мы будем называть *сервером аннотаций*. Программистскую машину, на которой выполняется сборка и анализ небольшого приложения (некоторой части программы), будем называть *клиентом*. Клиент может выступать одновременно и как сервер.

Решение задачи быстрого анализа состоит из двух частей:

- Каким образом на этапе контролируемой сборки генерируется список файлов, которые требуется переанализировать, а также внутреннее представление для них и все остальные данные для повторного анализа;
- Каким образом, имея эти данные, анализатор организует анализ файлов из этого списка для всех уровней анализа с учетом данных обо всей программе (о полном окружении этих файлов).

Первая часть решается подсистемой сборки в зависимости от необходимого сценария использования тремя способами:

1. **Определение изменившихся файлов за счет инкрементальных свойств исходной сборки.** В этом случае выдается и перехватывается обычная команда исходной сборки, все то, что пересобирается этой командой, и будет изменившимися файлами. Очевидно, требуется, чтобы на клиентской машине была ранее проделана полная сборка.
2. **Явное указание изменившихся файлов.** Этот режим полезен для случая работы программиста в интегрированной среде разработки, тогда такое указание может взять на себя эта среда. Исходной инкрементальной сборки не происходит, вместо этого система контролируемой сборки извлекает записанные на этапе полной сборки команды компиляции для указанных файлов и «проигрывает» их повторно. Если были указаны заголовочные файлы Си или Си++, то повторяются все компиляции для файлов, которые зависят от этих заголовочных файлов, для чего помимо команды компиляции на этапе контролируемой полной сборки сохраняются также и зависимости между исходными файлами. В случае компиляции программ на Java пересобираются только заданные файлы, а дополнительно собираются те файлы, которые компилятор Java установил как измененные зависимости для данных файлов согласно собственным алгоритмам отслеживания зависимостей [190]. Обязательным предположением является то, что окружение сборки не изменилось (необходимые исходные файлы, библиотечные зависимости и т.п. остались в тех же местах), чтобы повторение записанной ранее команды компиляции было успешным.
3. **Автоматическое определение изменившихся файлов**. В этом режиме при полной сборке дополнительно сохраняются времена последних изменений (timestamps) всех исходных файлов. При выдаче команды на пересборку текущие времена изменений сравниваются с сохраненными, и в случае их отличия измененные файлы включаются в список пересборки. Особенности при

пересборке файлов Си или Java, а также предположение о неизменности окружения сборки те же, что и в предыдущем случае.

Кроме этого, поддерживается два способа формирования окончательного объекта сборки для анализа. В первом случае в объект сборки попадут только те файлы, которые были определены как изменившиеся одним из описанных способов (*режим обновления*). Во втором случае в родительском объекте сборки, соответствующем предыдущей выполненной полной контролируемой сборке, устаревшие файлы будут заменены на новые версии из списка изменившихся файлов, а неизмененные файлы останутся прежними (*режим слияния*). Этот режим нужен для поддержки первого из описанных выше сценариев использования (инкрементальная сборка и полный последующий анализ).

На этапе межпроцедурного анализа случай получения файлов для анализа в режиме слияния не представляет сложностей – всю работу берет на себя этап контролируемой сборки, и анализ выполняется в обычном режиме. В остальных случаях анализ получает на вход набор файлов, которые надо переанализировать, и граф вызовов предшествующего полного анализа. На первом этапе строится обновленный граф вызовов, в котором функции из изменившихся файлов заменяются на функции из новых файлов, удаленные функции (те, которых нет в измененных файлах) выбрасываются, все функции из удаленных файлов также выбрасываются, а из созданных файлов – добавляются. Сопоставление функций из старого и нового графа вызовов происходит с учетом того, что компиляции, создавшие соответствующие файлы, выполнялись на разных машинах (клиенте и сервере). Для Си и Си++ замена функций происходит на этапе построения локальных графов вызовов для каждого файла (см. раздел 3.3.1) – локальный граф старого файла полностью заменяется на локальный граф нового, после чего объединение локальный графов в полный граф вызовов производится обычным образом. Для Java делается предположение, что добавление новых файлов не изменит иерархию классов в программе (т.е. могут быть добавлены новые классы, но существующие отношения родитель-потомок не изменятся), поэтому на стадии полного анализа сразу сохраняется полный девиртуализованный граф вызовов, и уже в нем обновляются измененные функции. Для C# инкрементальный анализ в настоящий момент не поддерживается.

После этого этапа дальнейшему анализу передается подграф графа вызовов, соответствующий новым и измененным файлам, и анализ выполняется обычным образом с единственным отличием: если встречается вызов неизвестной функции, то ее аннотация запрашивается у сервера аннотаций, и в случае, если эта функция была известна полному анализу, то используется сохраненная аннотация. Наконец, если на этапе сборки пользователь явно указал список файлов, которые надо переанализировать, то окончательное множество выданных анализом предупреждений фильтруется согласно полученному списку файлов. Дело

в том, что если какие-то файлы были переанализированы потому, что они были зависимостями указанных файлов и были перекомпилированы на этапе сборки, то в этих файлах выдавать предупреждения не нужно. Во всех остальных случаях выдаются только те предупреждения, которые были найдены в новых и измененных файлах.

Для анализа уровня АСД и внутрипроцедурного потока данных, которые обрабатывают исходные файлы по одному, просто повторяются запомненные в ходе полного анализа команды запусков соответствующих анализаторов с учетом изменений в путях к исходным файлам и зависимостям. В режиме слияния, в отличие от межпроцедурного анализа, по возможности не переанализируются неизменившиеся файлы, для которых сохраненные результаты полного анализа остаются корректными.

```
(1)  class A {
(2)    static B getB() {
(3)      return B.createB();
(4)    }
(5)  }

(1)  class B {
(2)    static B createB() {
(3)      return new B();    //return null;
(4)    }
(5)    void foo() {
(6)      B b = A.getB();
(7)      b.toString();
(8)    }
(9)  }

(1)  class C {
(2)    void foo() {
(3)      B b = A.getB();
(4)      b.toString();
(5)    }
(6)  }
```

Рисунок 3.11. Пример выполнения инкрементального анализа.

Для пояснения работы различных режимов инкрементального анализа рассмотрим пример исходного кода на языке Java на рисунке 3.11. Пример содержит три класса A, B и C, записанные в соответствующих Java-файлах. В исходной ситуации, представленной на рисунке, выполнение анализа не приводит к выдаче предупреждений. Пусть метод `createB` класса B был изменен, как показано в комментарии на строке 3 – теперь метод возвращает нулевую ссылку вместо создания нового объекта. Тогда в первом случае инкрементальной сборки предупреждение выдано не будет – изменился только файл с классом B, поэтому при анализе функции `foo` класса B будет взята старая аннотация метода `getB` класса A, в которой не отражено, что теперь метод `createB` возвращает нулевую ссылку. То же самое будет происходить и во втором случае (явное указание изменившихся файлов), если указан только файл класса B, и в третьем случае (автоматическое определение изменившихся файлов).

Все вышеперечисленное произойдет, если анализ выполняется в режиме обновления. Если же анализу предписан режим слияния, то будет пересобираться единственный файл с классом B, но переанализироваться все файлы, и в этом случае мы получим два предупреждения в методах `foo` классов B (строка 7) и C (строка 4).

Дополнительно рассмотрим случай, когда был изменен класс A (способом, не влияющим на приведенный код). Если изменение было выполнено до обсуждавшегося изменения класса B, и для анализа явно указан файл с классом A, то будут пересобраны и переанализированы оба класса A и B, анализом будет найдено указанное предупреждение в строке 7 метода `foo`, но оно не будет выдано из-за фильтрации предупреждений по списку явно указанных пользователем файлов. Если же пользователем также указан файл с классом A, либо список файлов получен автоматически (третьим способом), либо класс A изменен позже класса B, то оба класса будут пересобраны и переанализированы, и предупреждение в строке 7 будет выдано.

Как видно, функциональность инкрементального анализа содержит достаточно тонкостей, связанных с получением списка файлов для анализа и с проведением межпроцедурного анализа. Для каждого случая использования такого анализа требуется аккуратная настройка этапов контролируемой сборки и анализа с учетом сборочного окружения, состава анализируемого проекта, процесса разработки пользователя. Кроме того, для получения быстрой реакции на запрос инкрементального анализа в описанной архитектуре сознательно делается выбор анализа только изменившихся и новых исходных файлов, хотя легко построить пример, когда изменения кода вызываемых функций приводят к появлению новых ошибок в не изменившихся вызывающих функциях. Залогом успешного внедрения быстрого анализа является донесение до конечных пользователей этого компромисса между

скоростью и точностью анализа с ожиданием того, что новые ошибки в не менявшемся исходном коде могут быть получены после очередного выполнения полного анализа. В будущем планируется реализовать режим быстрого анализа с контролируемой глубиной анализа вызывающих функций, зависимости которых были изменены (сейчас эта глубина равна нулю).

3.4. Хранение и просмотр результатов анализа

Хранилище данных, используемое в инфраструктуре Svace, решает две задачи. Во-первых, требуется обеспечивать сохранение всей информации, которая необходима для выполнения одного запуска анализа (контролируемая сборка приложения и запуск анализаторов всех уровней, что соответствует этапам 1 и 2 рисунке 3.1). Результатом этих этапов должен стать набор данных (будем называть его *результатом анализа* или *снимком*), который достаточен для показа пользователю выданных предупреждений на исходном коде программы с поддержкой разметки предупреждений и навигации по коду.

Во-вторых, требуется хранить множество результатов анализа для выполняемых регулярно запусков анализатора и обеспечивать возможность многим пользователям работать с результатами – просматривать предупреждения, сравнивать результаты запусков между собой, предоставлять некоторое резюме по набору результатов (например, метрики качества кода, скорость исправления программистами ошибок и т.п.), управлять ролями пользователей, их правами, различными анализируемыми проектами и т.д. Эта вторая задача непосредственно не связана с самим анализом, однако может потребовать сборки дополнительных данных о программе в ходе анализа. Еще раз отметим, что хранение и просмотр результатов может выполняться (и обычно выполняется) на другой машине, отличной от той, на которой выполнена сборка или анализ. Часто один сервер результатов просмотра аккумулирует множество снимков анализов с разных машин.

Организация хранения данных в анализаторе схематически представлена на рисунке 3.12. Совокупность данных, являющихся результатом сборки, называется *объектом сборки*, а данных анализа – *объектом анализа*. В ходе сборки все инструменты (компиляторы, обработчики событий, утилита разрешения компоновочных связей), которые генерируют необходимые данные, передают их *сборщику объектов* для сохранения в объекте сборки. Эти данные включают в себя как исходные файлы и файлы с внутренним представлением для анализа, так и разнообразные метаданные, описывающие сборку. На этапе анализа входными данными для анализатора является данные конкретного объекта сборки. В результате работы анализаторов всех уровней создается объект анализа, который включает в себя данные об исходном коде, получаемые из объекта сборки, общий набор предупреждений, созданный

всеми анализаторами, а также отдельные результаты работы анализаторов первого уровня для каждой компиляции. Эта последняя часть необходима для организации инкрементального анализа и позволяет удобно заменять устаревшие результаты работы таких анализаторов новыми (как описано в разделе 3.3.7).

Рисунок 3.12. Хранение данных в ходе анализа и просмотра результатов.

Для просмотра результатов анализа конкретный объект анализа должен быть импортирован в базу данных. Результатом импорта является появление в базе очередного снимка анализа. Для каждого снимка анализа хранится набор предупреждений вместе с их разметкой, набор исходных файлов, на которых был проведен анализ, и разметка исходных файлов по токенам, необходимая для организации навигации по коду при просмотре предупреждений.

Организовано хранилище данных следующим образом. Содержащиеся объекты реализуют общий интерфейс SvaceObject, который дает возможность получить данные объекта в виде потока байт, их размер, тип объекта и хеш содержимого (SHA-1). Хеш объекта является его идентификатором, то есть хранилище организовано контентно-адресуемым способом (по аналогии с системой контроля версий Git). Такая схема хорошо подходит в случае хранения «исторических» данных, которые не меняются, будучи однажды сгенерированными, что соответствуют большинству объектов Svace. Она дает следующие преимущества:

- Легко ссылаться на объект по его хэшу с гарантией того, что объект не изменится;
- Одинаковые по содержанию объекты (не изменившиеся между снимками анализа исходные файлы, биткод и т.д.) хранятся в одном экземпляре, т.к. имеют один и тот же хеш;
- Можно просто организовать синхронизацию объектов в двух хранилищах, затребовав недостающие, т.к. объекты с одинаковыми хэшами не нужно передавать. Это используется при работе инкрементального анализа в момент запроса у сервера анализа недостающих аннотаций, которые хранятся в сериализованном виде с тем же интерфейсом.

Основными типами объектов являются: файл (исходный или двоичный, его содержимое никак не интерпретируется хранилищем), отображение «ключ»-«список объектов», где ключ является строкой, дерево объектов, объекты сборки и анализа, результаты анализа некоторой компиляции заданным анализатором. В зависимости от типа выбирается *пул*, задающий способ хранения объекта. Файловый пул хранит объекты, соответствующие бинарным файлам, но не имеет средств поиска объектов, кроме как по хэшу. Поэтому он используется в случаях, когда нужно посетить все объекты (например, обойти все файлы с внутренним представлением для анализа). Пул объектов Git хранит исходные файлы в базе результатов анализа для максимального уменьшения избыточности хранения даже слегка изменившихся текстов программ. Наконец, пул базы данных хранит объекты в SQL-базе, для которых нужен поиск и изменение. В основном это пользовательские результаты оценки предупреждений.

Само по себе хранилище результатов анализа еще не обеспечивает работы с ним. Для этого сервером результатов Svace предоставляется ряд программных интерфейсов, реализованных как на языке Java, так и в виде REST API, который обрабатывается встроенным веб-сервером. Вместе со Svace поставляется реализация пользовательского веб-интерфейса, который также служит как пример работы с программными интерфейсами сервера Svace.

Сервер результатов работает с пользователями и проектами. Пользователь может иметь доступ к разметке результатов анализа проекта или только к просмотру этих результатов. Администратор может создавать пользователей и регулировать их права. Проекты для

удобства администрирования объединяются в группы. Результаты анализа, как они представлены на рисунке 3.12, принадлежат некоторому проекту и дополнительно хранят момент времени, в который был произведен анализ. Список пользователей, проектов и списки результатов анализа хранятся в обычной базе данных.

В таблице 3.3 представлен список некоторых REST-запросов, реализованных сервером Svace, относящихся к чтению имеющихся данных. Аналогичные запросы на изменение и удаление данных не представлены для краткости.

Таблица 3.3. Примеры запросов к серверу результатов.

Запрос	Описание
`/history/api?act=get_projects`	Получить список групп проектов, конкретных проектов, снимков анализа для проекта
`/history/api?act=get_project_group_info`	Получить информацию о группе проекта (имя, описание, владелец, время создания)
`/history/api?act=get_project_info`	Получить данные о проекте (имя, описание, владелец, путь к базе данных с результатами)
`/history/api?act=get_value_filter&field=warn_types`	Получить список типов предупреждений, содержащихся в заданном снимке анализа
`/history/api?act=get_warnings`	Получить список предупреждений из заданного снимка с заданными типом, критичностью, статусом разметки. Поля, описывающие предупреждение, включают идентификатор, сообщение, критичность, разметку (вердикт пользователя, комментарии), место предупреждения (файл и строка)
`/history/api?act=get_warning_trace`	Получить трассу предупреждения – набор дополнительных сообщений , привязанных к точкам в файле, организованных в виде иерархической структуры
`/history/api?act=get_all_reviews`	Получить все результаты разметки данного предупреждения, включая историю смены статуса предупреждения, комментарии и т.д.

Реализация описанных интерфейсов и прочей функциональности сервера результатов не доставляет значительных сложностей, за исключением вопроса о перенесении разметки

предупреждений между снимками анализа. Одним из основных требований к интерфейсу пользователя просмотра предупреждений является автоматическое скрытие предупреждений, на которые по каким-либо причинам программисты не хотят тратить времени. Обычно таких причин две: либо предупреждение было классифицировано как ложное, и в последующих анализах его не нужно еще раз показывать для разметки, либо предупреждение формально истинное, но в данном проекте ошибка, описанная предупреждением, не может принести никакого вреда, либо ее исправление нецелесообразно. Для такого скрытия нужно уметь переносить результаты разметки предупреждений старого снимка анализа на новый.

Центральной частью переноса разметки является решение следующей задачи: по предупреждению, которое было выдано в заданном месте файла с заданными свойствами, оценить, было ли это предупреждение выдано в старом снимке анализа, который выполнялся для несколько иного исходного кода (разумеется, в предположении, что изменения с тех пор были невелики – наилучшим образом перенос работает при регулярном анализе). Нами предложен следующий алгоритм решения.

Алгоритм 3.2. Сопоставление предупреждений между результатами анализа.

1. По заданному предупреждению и имени файла получается список предупреждений того же типа, содержащийся в том файле, путь к которому имеет наибольший общий суффикс с путем заданного файла из всех файлов с одинаковыми именами. Этот подход позволяет учесть разницу в путях к файлам, которая могла появиться из-за выполнения разных анализов на различных машинах или в разных местах файловой системы одной машины.
2. Просматривается весь список предупреждений, и для двух файлов, в которых были найдены предупреждения, генерируется список различий между ними (diff), т.е. строки, которые были изменены, удалены, или вставлены.
3. Для строки предупреждения и строки кандидата на сопоставление по списку различий оценивается соответствие между строками по следующей шкале: различные строки; одинаковые строки; одинаковы строки и одна из соседних строк; одинаковы строки и обе соседних строки; одинаков весь контекст (т.е. заданное число строк снизу и сверху данной, обычно по три).
4. Для предупреждений, у которых наивысшее соответствие с данным, оценивается, сколько частей строки со свойством предупреждения совпадают или не совпадают. Предпочитаются предупреждения с наибольшим числом совпадений и наименьшим числом промахов, а при прочих равных условиях – предупреждения с уже имеющейся разметкой.

4. ДЕТЕКТОРЫ ОШИБОК ВСЕХ УРОВНЕЙ АНАЛИЗА В ПРОГРАММНОЙ СИСТЕМЕ SVACE

Настоящая глава посвящена описанию детекторов ошибок, реализованных в анализаторе Svace на всех уровнях анализа, от уровня АСД до чувствительных к путям детекторов. Представленный материал сконцентрирован на описании детекторов, находящих реальные ошибки с приемлемым уровнем истинных срабатываний. Такие детекторы включены по умолчанию, и по поводу них у разработчиков и пользователей инструмента есть общее мнение актуальности соответствующих ошибок. Опущены детекторы, касающихся стилевых сторон кодирования, которые могут требоваться в зависимости от проверяемого проекта, детекторы, представляющие из себя задел на будущее, которые на выбранном уровне анализа не были реализованы с должным качеством, а также используемые детекторы первого уровня анализа из сторонних инструментов (в основном FindBugs и некоторые детекторы Clang Static Analyzer).

Всего в анализаторе Svace около 450 детекторов, кроме того, еще около 340 детекторов – это детекторы первого уровня из сторонних инструментов. Около 300 детекторов реализованы в основном движке Svace (это детекторы второго и третьего уровней анализа), подавляющее большинство из них работают для Си, Си++, Java. Примерно 20 детекторов первого уровня реализованы нами в Clang Static Analyzer, 11 детекторов – в компиляторе Javac. 35 детекторов реализованы для языка C#.

Из упомянутых 450 детекторов в дистрибутиве Svace включены по умолчанию 232 детектора всех уровней. Ниже будут описаны наиболее интересные из этих детекторов, как с точки зрения алгоритмов поиска, так и с точки зрения находимых ими ошибок. Для описываемого детектора будет приводиться пример ошибки в виде кода либо текстового описания, алгоритм поиска и (при наличии) тонкости, которые возникали при проверке детектора на реальном коде.

В качестве предварительных замечаний представляется важным отметить, что объяснение феномена большого количества детекторов, помимо широты спектра ошибок, которые необходимо найти, лежит в принципиальных ограничениях на точность статического анализа. Для детекторов любого из представленных уровней сложности желание добиться масштабируемости с одной стороны и качества анализа с другой ведет к ограничениям на ситуации, которые детектор считает ошибочными. Пользование очень общими определениями ошибки (например, предупреждать о потенциальном переполнении буфера во всех случаях, когда индекс буфера может превышать его длину) даст много ложных срабатываний из-за неточности выполняемого анализа указателей и моделирования целочисленных вычислений даже с учетом чувствительности к путям – нельзя забывать ограничения на размер аннотации

функции, из-за которых точность моделирования снижается, трудности с обработкой пользовательских данных и т.п. С другой стороны, закрыть критическую ошибку общего характера большим количеством очень узкоспециализированных «шаблонов» ошибочных ситуаций также невозможно – это ведет к пропуску реальных ошибок и является уделом анализаторов первого уровня. Требуется сводить общую ошибочную ситуацию к нескольким вариантам, каждый из которых отрабатывается отдельным детектором, и которые можно находить чувствительными к путям межпроцедурными детекторами с приемлемой точностью. Эти варианты так или иначе будут представлять из себя некоторые эвристические компромиссы.

При рассмотрении конкретного типа ошибочной ситуации может являться спорным, считать ли такую ситуацию за ошибку или нет. Обычно это также компромисс между количеством предупреждений, выдаваемых детектором, и шансами на их исправление прикладными разработчиками. Частым решением является выделение спорной «шумной» ситуации в отдельный подтип детектора, выдача которого может конфигурироваться отдельно. Такой детектор будет включаться только для тех разработчиков, которые действительно хотят исправлять соответствующие предупреждения в своем проекте, не мешая остальным. Эти ситуации также ведут к увеличению числа детекторов.

4.1 Детекторы ошибок уровня АСД и внутрипроцедурного потока данных

Детекторы уровня АСД реализованы отдельно для каждого поддерживаемого языка в инфраструктуре Clang Static Analyzer (CSA), Javac и Roslyn для Си/Си++, Java и C# соответственно. Детекторы внутрипроцедурного потока данных при этом реализованы совместно для Си/Си++ и Java в основном движке Svace и в Roslyn для C#. Причина в том, что помимо выдачи предупреждений, внутрипроцедурный анализ поставляет важные сведения для дальнейших уровней межпроцедурного и чувствительного к путям анализа, уточняя целочисленные значения для модели памяти анализатора и осуществляя консервативный поиск недостижимого кода.

Промах в оценки недостижимости кода влияет на все детекторы, так как анализатор учитывает влияние потока данных с не выполняющихся путей и наоборот, не учитывает с путей, ошибочно принятых за недостижимые. Например, типичным случаем ложных срабатываний при анализе низкоуровневого системного кода является непонимание анализом функций типа `exit`, которые призваны завершить работу текущего потока выполнения. Отсюда важность «близости» этого детектора к основному движку анализа.

Дополнительно к собственным реализованным детекторам для уровня АСД используются некоторые открытые детекторы инфраструктуры CSA и детекторы инструмента FindBugs, для

уровня внутрипроцедурного анализа – также детекторы FindBugs. Этот инструмент обрабатывает class-файлы, что позволяет запускать его на том же представлении, что собирается и для основного движка Svace. Он является популярным для проверки кодирования среди разработчиков Java и также входит в состав коммерческого пакета Coverity Prevent.

4.1.1. Детекторы Си/Си++

4.1.1.1. Детекторы ALLOC_SIZE_MISMATCH / MEMSET_SIZE_MISMATCH

Детектор анализирует параметры функций, выделяющих динамическую память, и ищет несоответствия между базовым типом указателя, по которому сохраняется выделенная память, и объемом этой памяти. Как правило, эти несоответствия указывают на описки в коде либо на непонимание программистом операции `sizeof`. Примеры подозрительных ситуаций включают:

- Использование `sizeof (pointer)` вместо `sizeof (*pointer)`;
- Передача функции `memset` указателя на структуру и одновременно количества байт, меньшего размера структуры (разрешаются ситуации, когда передается адрес структуры и размер первого поля структуры);
- Неверное умножение, вычисляющее размер выделяемой памяти в функции `malloc` (размер типа, передаваемый в операцию `sizeof`, меньше размера типа элемента предполагаемого массива);
- Использование `new(constant)` вместо `new[constant]` при попытке выделить память под массив;
- Выделение памяти под массив шаблонного типа, размер которого зависит от размера типа инстанцирования, при этом использованная операция `sizeof` не зависит от инстанцирования.

Детектор является детектором третьего типа – ищутся все вызовы функций из заданного списка, после чего для принятия решения о выдаче одного из указанных предупреждений посещаются все подузлы этого вызова.

4.1.1.2. Детектор INFINITE_LOOP

Детектор пытается найти циклы, которые из-за описки в теле цикла или условии оказываются бесконечными. Для этого анализируется условие в заголовке цикла и проверяется, изменяются ли переменные, упомянутые в условии, в теле цикла, либо их адреса передаются извне функции, либо функция содержит безусловный оператор `break` или `return`. Если ни одно из этих условий не выполняется, то выдается предупреждение. Также предупреждение выдается в том случае, если условием продолжения цикла for является сравнение индуктивной

переменной с другой переменной или константой, однако на каждом шаге цикла индуктивная переменная изменяется в сторону, противоположную направлению сравнения (то есть переменная сравнивается с некоторой верхней границей, но при этом уменьшается). Детектор является детектором третьего типа, так как обходит тела всех найденных циклов в поисках побочных эффектов, переходов управления или утекающих адресов переменных.

Рассмотрим примеры ситуаций на рисунке 4.1. В функциях `foo1` и `foo2` содержится бесконечный цикл на строках 2 и 8, соответственно. В первом случае переменная `n` только увеличивается, и в случае, когда параметр `i` не меньше единицы, цикл будет бесконечным. Во втором случае увеличивается верхняя граница сравнения, а нижняя не меняется, и если сравнение было истинным на первой итерации (что произойдет при положительном `n` на второй итерации внешнего цикла), то оно и останется таковым.

В функции `foo3` переменная в условии цикла на строке 12 не меняется, однако она представляет из себя память, которая доступна извне. Если, например, существует другой поток выполнения, который может модифицировать эту память во время выполнения функции `io_wait`, то цикл завершится, и предупреждение выдавать нельзя. А вот в функции `foo4` параметр сначала присваивается локальной переменной, и уже эта переменная тестируется в условии цикла. Так как локальная переменная не может быть изменена извне функции, то цикл на строке 17 является бесконечным.

```
(1)  void foo1 (int i) {
(2)    for (int n = i - 1; n >= 0; n++)
(3)      bar (n);
(4)  }
(5)  void foo2 (int n) {
(6)    int j;
(7)    for (int i = 0; i < n; ++i)
(8)      for (j = 0; j < i; ++i)
(9)        ;
(10)  }
(11)  int foo3 (struct A &a) {
(12)    while (!a.status)
(13)      io_wait ();
(14)  }
(15)  int foo4 (struct B &b) {
(16)    Status x = b.status;
```

```
(17)    while (x != OK)
(18)      io_wait ();
(19)  }
```

Рисунок 4.1. Пример обнаружения ошибки INFINITE_LOOP.

4.1.1.3. Детектор BAD_SIZEOF

Детектор определяет потенциально некорректные использования операции `sizeof`, одним из примеров которых было описанное для детектора ALLOC_SIZE_MISMATCH ее применение к указателю вместо применения к указываемой памяти. Детектор является детектором второго типа, обходя АСД для поиска операций `sizeof` и последующему выполнению константного количества тестов, а именно: применения к указательному параметру функции (в таком случае всегда получится размер указателя вместо ожидаемого размера массива), к массиву, который неявно приводится к указателю (более широкий вариант предыдущего случая, т.н. array to pointer decay в терминах стандарта языка Си), к указателю `this`, к операции взятия адреса либо к операциям адресной арифметики.

4.1.1.4. Детектор BAD_OVERRIDE

Детектор предназначен для поиска ошибочных ситуаций с наследованием методов в Си++, когда методы базового класса и наследника похожи настолько, что можно предположить, что один метод должен был переопределять другой, однако из-за ошибки в коде этого не происходит. Первым вариантом ошибочной ситуации является то, что базовый метод не является виртуальным, однако методы совпадают по именам, количеству и типу параметров, типу возвращаемого значения. Вторым вариантом является допущение разницы в квалификаторах типов параметров или возвращаемого значения, тогда не играет роли, объявлен ли метод в базовом классе как виртуальный. Это пример детектора первого типа, который не обходит АСД метода, а имеет дело только с таблицей символов – иерархией классов и методами классов.

4.1.1.5. Детекторы NO_EFFECT/INVARIANT_RESULT

Детекторы ищут выражения, которые являются либо ненужными (результаты вычислений ни на что не влияют), либо константными для всех входных данных, хоть и не выглядят таковыми. Из-за константности некоторых выражений может следовать наличие мертвого кода в программе, поэтому оба детектора рассматриваются вместе. В первом случае выражения, которые проверяются детектором, включают сравнение массивов с нулем (вместо сравнения элемента или указателя), присваивания переменной самой себе, выражения и циклы без побочных эффектов, сравнения беззнаковых чисел с нулем, неверное использование

операции `delete` (например, код вида `delete p, q;` делает не то, что хотелось бы его автору).

Для второго детектора обрабатываются сравнения, арифметические, логические и битовые операции, и если оказывается, что результат операции всегда предопределен, то выдается предупреждение. Оба детектора являются детекторами третьего типа. Примеры находимых детектором ситуаций представлены на рисунке 4.2. Всегда константными будут выражения на строках 3, 7, 12, 17 и 21.

```
(1)    void foo1 () {
(2)      unsigned short var;
(3)      (var << 32) & 0xff;
(4)    }
(5)    void foo2 () {
(6)      unsigned short var;
(7)      if (var < 100000) {}
(8)    }
(9)    struct A { int x; int y; };
(10)   void foo3 (A a, A *pa, int n, int x, int y) {
(11)     for (int i = 0; i < n; ++i)
(12)       if ( pa[i].x != x || pa[i].y != pa[i].y)
(13)         ;
(14)   }
(15)   struct bar { int f; };
(16)   int foo4 (struct bar *b) {
(17)     return (int) ((b->f < 1) == 2);
(18)   }
(19)   #define MAGIC_VAL 0x05
(20)   int foo5 (unsigned char protocol) {
(21)     if (!protocol == MAGIC_VAL)
(22)       return 0;
(23)     return 1;
(24)   }
```

Рисунок 4.2. Примеры ошибок INVARIANT_RESULT.

4.1.1.6. Детектор BAD_COPY_PASTE

Детектор предназначен для поиска *ошибок копирования* – групп выражений, которые похожи друг на друга так, что можно думать, что одна группа выражений является копией второй, которая потом была изменена, и при изменении сделана ошибка. Такого рода ошибки в англоязычной литературе называют *copy-paste errors* – программист скопировал кусок текста, чтобы написать похожий текст, однако изменил не все необходимые места. Этот детектор очень актуален и реализован для всех поддерживаемых языков, при этом он является ярким примером того, как можно было бы реализовать единый детектор для нескольких входных языков, если бы инфраструктурно такая возможность была бы. Детектор принадлежит четвертому типу – в настоящий момент поддерживается обработка только условных операторов, однако после поиска таких операторов и посещения всех подузлов оператора выполняется дополнительная проверка. Нужно отметить, что известны алгоритмы решения более общей задачи поиска клонов кода в программе, которые можно приспособить для нахождения ошибок копирования, в том числе межпроцедурного, однако эти алгоритмы достаточно тяжеловесны (наиболее точные алгоритмы на основе поиска изоморфных подграфов в графе зависимостей программы даже с набором ускоряющих их эвристик займут время, сравнимое с временем работы всего остального анализатора), поэтому на уровне АСД реализованы легковесные эвристические подходы.

Детектор для Си и Си++ работает следующим образом. Для всех условных операторов и коротких циклов в пределах функции строятся векторы из токенов, составляющих тело оператора, их типов и строковых представлений. Для каждой пары векторов ряд эвристических проверок пытается определить степень их похожести – по длине векторов, типам составляющих их токенов, вложенности, расположению на строках исходного кода. Веса разных проверок подбирались экспериментально на большом объеме реального кода. Если два оператора признаны похожими, то далее они сравниваются, чтобы установить, везде ли, где в теле первого оператора используются одни и те же имена (переменные, функции, аргументы функций), во втором операторе на тех же позициях также используются одинаковые имена. Места найденной асимметрии в использовании имен выдаются как ошибочные.

```
(1)   int maxVal (int sVal, int sHgt, int k) {
(2)     int sMaxIdx=0; int sMaxVal=0; sMaxVal = 100;
(3)     if (sVal > sMaxVal) {
(4)       sMaxVal = sVal;
(5)       sMaxIdx = k;
(6)     }
```

```
(7)     if (sHgt > sMaxVal) {
(8)       sMaxVal = sVal; // 'sVal' might be 'sHgt'
(9)       sMaxIdx = k;
(10)    }
(11)    return 0;
(12)  }
(1)   void foo2 (CardArea* pT) {
(2)     int i,j,k,l;
(3)     for (i=1,k=1; i<pT->iColNum; i++) {
(4)       for (j=k+1; j<pT->pNum[i]-1; j++) { 'i' might be 'l'
(5)         pT->pEndPos[l][j] = pT->pEndPos[l][j+1];
(6)         pT->pStartPos[l][j] = pT->pStartPos[l][j+1];
(7)       }
(8)       for (j=k+1; j<pT->pNum[i]-1; j++) {
(9)         pT->pEndPos[i][j] = pT->pEndPos[i][j+1];
(10)        pT->pStartPos[i][j] = pT->pStartPos[i][j+1];
(11)      }
(12)      k++;
(13)      l++;
(14)   }
(15)  }
```

Рисунок 4.3. Примеры ошибки BAD_COPY_PASTE.

Рассмотрим примеры кода на рисунке 4.3. Ошибочные места выделены полужирным курсивом и отмечены комментариями для простоты восприятия. В функции `foo1` похожи условные операторы на строках 3-6 и 7-10. Если подробней посмотреть на тела этих операторов, то видно, что первый оператор обрабатывает значение `sVal`, а второй – `sHgt`, однако на строке 8 сделана ошибка – вместо переменной `sHgt` используется неверное значение `sVal`. Аналогично, в функции `foo2` похожи тела двух циклов, однако один использует переменную `i` как индекс, а другой – переменную `l`, и в строке 4 сделана ошибка (что неудивительно – написание переменных похоже).

4.1.1.7. Детектор CONFUSING_IDENTATION

Детектор ищет ситуации, в которых вертикальное выравнивание строк заставляет пользователя предположить структуру потока управления, отличную от той, которая

присутствует на самом деле. Это может быть признаком не только небрежности в коде, но и реальной ошибки, самым знаменитым примером которой является уязвимость безопасности в коде обработки SSL компании Apple [191], известная как Apple goto fail. Она представлена на рисунке 4.4 в строках 7-8 – неверное выравнивание оператора `goto fail;` не позволило программисту заметить ошибку, в результате которой всегда происходит возврат успешного кода проверки SSL-сертификата. Другой пример ошибочного кода для цикла `for` показан на строках 4-5.

```
(1)  static OSStatus SSLVerifySignedServerKeyExchange(...) {
(2)    OSStatus err;
(3)    ...
(4)    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
(5)      goto fail;
(6)    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
(7)      goto fail;
(8)      goto fail;
(9)    ...
(1)  void foo (void) {
(2)    int i;
(3)    for (i = 0; i < 10; ++i)
(4)      ++i;
(5)      --i;
```

Рисунок 4.4. Примеры ошибки CONFUSING_INDENTATION.

Детектор является детектором второго типа и находит все тела циклов и условных операторов, после чего производит проверку номеров строк и колонок следующих операторов после тела найденного оператора.

4.1.1.7. Детекторы BAD_ITERATOR.INVALID и BAD_ITERATOR.MISMATCH

Детекторы предназначены для поиска ситуаций неверного использования итераторов в Си++, а именно, использование итератора после того, как он стал некорректным (первый детектор), либо использование итератора неверного базового типа (второй детектор). Детекторы реализованы совместно для экономии обходов АСД и являются детекторами четвертого типа. Сначала обходится таблица символов в поисках классов, называющихся так

же, как известные типы контейнеров. Далее в их методах ищутся итераторы, т.е. методы с именами `begin/end` без параметров, возвращающие указатель на некоторый класс. После этого обходятся тела всех функций, ищутся переменные итераторного типа и соответствующие им контейнеры. Для первого детектора определяются ситуации разыменования итераторной переменной после того, как произошел выход из цикла, в котором итерация шла до достижения итератором значения `container.end ()`, либо после выполнения метода `erase (iter)`. Для второго детектора определяются ситуации сравнения итераторов по двум разным контейнерам либо вызова метода `erase` или `insert` у контейнера с передачей итератора неверного типа.

```
(1)   class A {
(2)     public:
(3)     int a;
(4)   };
(5)   void foo1 (std::vector<A> &v) {
(6)     int b;
(7)     std::vector<A>::iterator it = v.begin();
(8)     for (std::vector<A>::iterator end = v.end();
(9)          it != end || it->a > 5; ++it) {
(10)      b += it->a;
(11)    }
(12)  }
(13)  void foo2 (vector<int>& v1, vector<int>& v2) {
(14)    vector<int>::iterator it = v1.begin();
(15)    vector<int>::iterator it2 = it;
(16)    ++it2;
(17)    v2.insert (it2, 7);
(18)  }
```

Рисунок 4.5. Примеры ошибок BAD_ITERATOR.INVALID, BAD_ITERATOR.MISMATCHED.

На рисунке 4.5 приведены примеры ситуаций, находимых описанными детекторами. В первом случае в цикле на строке 9 итератор `it` может быть разыменован тогда, когда итерация по контейнеру уже достигла его конца, и переменная итератора указывает на один элемент за

границу вектора. Во втором случае в строке 17 в вектор `v2` производится попытка вставить элемент, используя итератор от другого вектора, `v1`.

4.1.1.8. Детектор WRONG_ARGUMENTS_ORDER

Детектор ищет достаточно любопытную ситуацию вызова некоторой функции, в котором предположительно перепутан необходимый порядок аргументов. Так как речь идет об анализе скомпилировавшегося кода, то нужно заранее предполагать, что типы аргументов, несмотря на ошибку, совпали. Конечно, речь может идти лишь об эвристическом алгоритме поиска. Признаком ошибочной ситуации в данном случае является несоответствие имен формальных и фактических параметров. Ищутся следующие ситуации:

- Формальные параметры называются именами (частями имен), похожими на source и destination, а фактические параметры (части имен параметров) также похожи, но перепутаны местами;
- Функция имеет два формальных параметра, а два фактических параметра называются похоже и перепутаны местами;
- Список формальных параметров получается из списка фактических параметров с похожими именами сдвигом на +/-1 позицию.

Ключевым для детектора является определение степени похожести имен. Формальные и фактические параметры разбираются на кусочки, разделителем для которых является знак подчеркивания или смена регистра букв. На похожесть сравниваются два вектора из найденных частей имен, и она вычисляется как суммарная длина одинаковых частей, если эти части не состоят только лишь из цифр. Перепутанность местами определяется как похожесть формальных параметров на фактические в большей степени в неверном порядке.

Примеры некоторых ошибок показаны на рисунке 4.6 (строки 16 и 18).

```
(1)   class ClassWithConstr {
(2)     int m_width;
(3)     int m_height;
(4)   public:
(5)     ClassWithConstr (int width, int height) :
(6)                       m_width(width), m_height(height) { }
(7)   };
(8)   class WHProvider {
(9)    public:
(10)     int getWidth() { ... }
```

```
(11)     int getHeight() { ... }
(12)  };
(13)  void bar (int source_var, int destination_var, bool f =
false);
(14)  void foo {
(15)    int my_src = 4, my_dst = 2;
(16)    bar (my_dst, my_src, true);
(17)    WHProvider wh;
(18)    ClassWithConstr cc (wh.getHeight(), wh.getWidth());
(19)  }
```

Рисунок 4.6. Примеры ошибки WRONG_ARGUMENTS_ORDER.

4.1.1.9. Детектор UNREACHABLE_CODE

Детектор занимается поиском недостижимого кода, то есть такого кода, которого не достигнет ни один путь выполнения. Как упоминалось выше, помимо самого поиска такой ситуации, данные этого детектора требуются дальнейшему анализу (межпроцедурному уровню), т.к. частой причиной ложных срабатываний является учет потока данных от недостижимого кода. По этой причине детектор реализован как в инфраструктуре CSA, так и как внутрипроцедурный детектор, выполняющийся основным движком анализа на первой стадии (перед основным межпроцедурным алгоритмом, см. раздел 3.3). Детектор CSA был выполнен на основе уже существующего открытого детектора, к которому добавлены подтипы о выдаче недостижимого кода в операторе `return`, вызовах встроенной функции `va_end`, подавлении предупреждений из макроподстановок и др. Далее этот детектор рассматриваться не будет.

Детектор основного движка Svace ищет ситуации, связанные с тремя причинами возникновения недостижимого кода – из-за недостижимости по графу потока управления, из-за вызова функций, которые завершают выполнение, и из-за вычислений, в результате которых некоторые ветви условных операторов или операторов выбора становятся недостижимыми. Для того, чтобы не только выдать предупреждения об ошибочных ситуациях, но и предоставить данные о недостижимом коде дальнейшему анализу, требуется консервативность получаемых результатов. Применяются методы, подобные описанным в главе 2, однако при вычислениях делается упор на корректность получаемых результатов (тогда как основной анализ может получать неконсервативные результаты из-за предположений об отсутствии

псевдонимов среди параметров функций, обработке конечного числа итераций цикла и других, описанных ранее).

Анализ недостижимого кода комбинирует результаты четырех анализов: оценку целочисленных значений через интервальный анализ, анализ не равных нулю переменных (анализ выколотой точки), анализ достижимости точек программы и поиск функций, завершающих работу программы. Первые два анализа выполняются как анализы над номерами значений, которые консервативно вычисляются по классической схеме, основанной на хэш-таблицах и SSA-представлении программы (классы значений, представленные в главе 2, являются неконсервативным аналогом для предложенной нами модели памяти). Возможные интервалы значений для номеров значений вычисляются обычными передаточными функциями [192]. Для ускорения сходимости оператор расширения ∇ применяется, как только в сильно связанной компоненте графа потока управления перевычисляется интервал (сужение границ запрещено, а при расширении границ интервала они немедленно расширяются до $\pm\infty$), а далее передаточные функции работают в обычном режиме (применяется оператор сужения).

```
(1)   if (a != 0) {
(2)     ...
(3)     if (a == 0) {
(4)       ...
(5)     }
(6)   }
```

Рисунок 4.7. Выколотая точка и недостижимый код.

Анализ ненулевых значений применяется для отработки часто встречающихся сравнений с нулем, которые образуют в интервале выколотую точку, не поддерживаемую единственным интервалом (как упоминалось в главе 2, нужно применять анти-интервалы либо объединение нескольких интервалов для учета выколотой точки в интервальном анализе). Например, на рисунке 4.7 код на строке 4 будет недостижим, однако интервальный анализ не поможет это установить, т.к. обработка сравнения на неравенство нулю на первой строке не приведет к осмысленному интервалу для `a`. Анализ отслеживает атрибут номера значений, который показывает, когда некоторое значение отлично от нуля. Передаточная функция этого атрибута устанавливает его в значение «отлично от нуля» при сравнениях с нулем на неравенство, присваивании ненулевой переменной либо вычитаниях двух разных переменных (про которых

известно, что они не равны). Если же обрабатывается сравнение с нулем на равенство номера значений с атрибутом «отлично от нуля», то код помечается как недостижимый.

Анализ достижимости точек программы строит конъюнкции, состоящие из предикатов сравнения (дизъюнкции не учитываются и соответствующие условные операторы не обрабатываются для легковесности анализа, но ее можно добавить). Если при проходе очередного сравнения результирующая формула содержит одновременно некоторый атомарный предикат и его отрицание, то далее код внутри условного оператора помечается как недостижимый.

Наконец, последним компонентом является обратный межпроцедурный анализ, распространяющий атрибут, который означает, что на данном пути выполнения функция завершает работу программы. Если такой атрибут при внутрипроцедурном анализе встретился на всех путях, то функция помечается как завершающая программу, и атрибут распространяется далее (в данном случае аннотация функции для этого анализа содержит только этот единственный атрибут). Такой анализ позволяет обрабатывать пользовательские функции-обертки над известными функциями типа `exit/abort`.

4.1.2. Детекторы Java

Реализованные для Java детекторы включают в себя аналоги уже описанных для Си/Си++ детекторов – это детекторы BAD_COPY_PASTE, CONFUSING_IDENTATION, INVARIANT_RESULT, WRONG_ARGUMENTS_ORDER. Далее они описываться не будут. Некоторые другие детекторы (WRONG_SEMICOLON, SIMILAR_BRANCHES, FALL_THROUGH) указывают на просто находимые в АСД ситуации и подробно тоже описываться не будут, мы упоминаем их для полноты описания.

Остановимся подробнее на более интересном для Java детекторе WRONG_LOCK_OBJECT. Это детектор второго типа, который ищет операторы `synchronize(object)` для случаев, когда типом переменной `Object` является один из типов-оберток над примитивными типами – `Boolean`, `Integer` и т.д. Дело в том, что для оптимизации экземпляры объектов для популярных значений таких типов создаются единожды во время работы программы (это объекты `Boolean(true)`, `Boolean(false)`, `Integer(0)`, `Integer(1)` и еще несколько десятков примеров). Так как эти экземпляры могут быть переиспользованы, то синхронизация по объекту данного типа может привести к взаимной блокировке, которая не была предвидена пользователем.

Рассмотрим пример кода из операционной системы Android на рисунке 4.8. Функция `resetAudioPortGeneration` на строке 5 использует оператор `synchronized` для объекта `mAudioPortGeneration` типа `Integer`, однако этот объект равен `Integer(0)`,

который создается в единственном экземпляре. Тем самым может возникнуть нежелательная взаимная блокировка, так как в разных объектах поле `mAudioPortGeneration` будет одинаковым.

```
(1)     static final int AUDIOPORT_GENERATION_INIT = 0;
(2)     Integer mAudioPortGeneration = new Integer
(AUDIOPORT_GENERATION_INIT);
(3)     int resetAudioPortGeneration() {
(4)       int generation;
(5)       synchronized (mAudioPortGeneration) {
(6)         generation = mAudioPortGeneration;
(7)         mAudioPortGeneration = AUDIOPORT_GENERATION_INIT;
(8)       }
(9)       return generation;
(10)    }
```

Рисунок 4.8. Пример ошибки WRONG_LOCK_OBJECT.

Нужно отметить, что в случае Java многие нужные детекторы первого уровня уже реализованы в открытом инструменте FindBugs (таких детекторов более 200), поэтому реализация собственных детекторов не так актуальна, как для других языков.

4.1.3. Детекторы C#

Как и для Java, часть детекторов для C# повторяет уже описанные детекторы для Си/Си++. Это детекторы BAD_COPY_PASTE, INVARIANT_RESULT, INFINITE_LOOP, CONFUSING_INDENTATION, WRONG_ARGUMENTS_ORDER. Остановимся на более интересных детекторах, специфичных для языка C#.

Детектор WRONG_LOCK.STATIC ищет ситуации, в которых внутри блоков синхронизации по некоторому не статическому объекту происходит доступ к статическому полю класса. Такой доступ может создать ситуацию гонки, т.к. даже для разных объектов синхронизации обращение к статическому полю означает доступ к одному и тому же участку памяти из разных потоков выполнения. Похожий детектор реализован и для языка Java.

Детектор RETURN_USING реагирует на возвращение из метода объекта внутри блока `using`, который был объявлен в заголовке этого блока. Такой объект будет закрыт до его возвращения, и работа с ним извне функции будет невозможна. Эта ошибка в чем-то

аналогична ситуации возврата из функции адреса локальной переменной в языке Си. Пример кода с ошибкой из проекта Lucene.NET приведен на рисунке 4.9.

```
(1)  public static MemoryStream SerializeToStream (object o) {
(2)    using (var stream = new MemoryStream()) {
(3)      Formatter.Serialize(stream, o);
(4)      return stream;
(5)    }
(6)  }
```

Рисунок 4.9. Пример ошибки RETURN_USING.

Детектор THREAD_STATIC_FIELD обнаруживает неверные использования атрибута [ThreadStatic]. Атрибут заводит статические поля класса отдельно для каждого метода, однако нельзя выполнять инициализацию этого поля (т.к. она будет работать только из одного потока) либо применять его для нестатических полей.

Детектор VIRTUAL_CALL_IN_CONSTRUCTOR ищет виртуальные вызовы методов класса, которые выполнены изнутри тела конструктора этого же класса. Такая ситуация может привести к неконсистентному состоянию объекта, т.к. часть полей может быть еще не проинициализирована.

Детектор UNUSED_VALUE ищет выражения (правые части присваиваний), которые после присваивания в дальнейшем не используются. Это детектор реализован как классическая задача потока данных, которая продвигает флаг об использовании выражения через присваивания переменных. Использование сбрасывает этот флаг, а присваивания снова ставят. Обнаруживаются ситуации, когда выражение остается неиспользованным до конца метода, либо переменная, его содержащая, получает другое значение, либо кончается область ее видимости.

Детектор недостижимого кода UNREACHABLE_CODE работает подобно соответствующему детектору в Си (раздел 4.1.1.9) – отслеживаются предикаты, по которым происходят переходы, и совместность конъюнкций предикатов, составляющих необходимое условие попадания в некоторую точку программы, проверяется SMT-решателем.

4.2 Межпроцедурные детекторы ошибок

Детекторы ошибок второго уровня (межпроцедурный контекстно-чувствительный анализ) реализованы в основном движке анализатора Svace для языков Си, Си++ и Java, а

также в движке Roslyn для языка C#. Перед описанием конкретных детекторов представим список инструкций внутреннего представления, список событий, на которые реагируют детекторы. Далее опишем атрибуты, используемые детекторами для выдачи предупреждений – многие детекторы используют одни и те же атрибуты, означающие наличие некоторых свойств программы, поэтому список атрибутов помогает лучше ориентироваться в логике работы детекторов. Наконец, опишем основные детекторы по классам находимых ими ошибок.

Инструкции внутреннего представления Svace представлены в таблице 4.1. Дается краткое описание каждой инструкции. Некоторые из инструкций появились для удобства чтения биткода LLVM, некоторые – для обработки языка Java. Однако таких инструкций меньшинство, подавляющая часть инструкций общая для всех языков основного движка.

Таблица 4.1. Инструкции внутреннего представления Svace.

Инструкция	**Описание**
`a = b` *op* `c`, где – *op* одна из `+`, `-`, `*`, `/`, `%`, `&`, `<<`, `>>`, `^`	Арифметические и побитовые бинарные операции.
`a = b`	Присваивание.

`a = *p`
`*p = a`
`p->f = a`
`a = p->f`
	Разыменование указателя при чтении или записи.

`s.f = a` `a = s.f`	Запись или чтение по полю без разыменования.
`p = alloca t`	Выделение локальной памяти для объекта типа `t` и запись указателя на нее в `p`.
`p = new t` / `new[num] t`	Выделение памяти для объекта либо массива объектов типа `t` и запись указателя на нее в `p`. Используется для моделирования соответствующей инструкции байткода Java.
`a = (cast) b` `a = inttofloat b`	Приведение переменной `b` к типу переменной `a` и запись результата в `a`. Поддерживает различные виды приведений (обрезание, расширение со знаком / без знака и т.п.).

`p = gep q, accesspath`	Получение адреса элемента сложного типа, аналогичное инструкции `getelementptr` LLVM-представления.
`landingpad, resume, annotateException`	Инструкции, которые используются при обработке исключений (указание места обработки, повторный выброс исключения, то же без явного пользовательского кода). Первые две инструкции также присутствуют в LLVM.
`call, methodcall, pcall`	Инструкции для обычного вызова, виртуального вызова метода и вызова по указателю соответственно.
`a = b == c` `a = b > c`	Инструкции сравнения.
`a = b ? c : d`	Тернарная инструкция. Используется также для моделирования φ-функций.
`nop`	Инструкция без какого-либо действия.
`unreachable`	Инструкция означает, что далее код недостижим.
`annotate`	Инструкция помечает конец функции для создания аннотации.
`ret`	Инструкция помечает конец ветви в графе потока управления.
`assume cond`	Помечает, что на данной ветви управления условие cond выполняется (либо не выполняется).

События вводятся для удобства написания детекторов. Частой является ситуация, при которой детектор интересуют не все инструкции, а только некоторая их часть, например, работающая с памятью, или с арифметическими операциями. При этом может статься, что интересующая детектор порция информации может появиться при выполнении различных инструкций. Например, разыменование указателя происходит в нескольких инструкциях (записи или загрузки по указателю). Другой пример – переменная может измениться некоторым неявным образом (например, волатильная переменная либо общая для нескольких потоков, либо ее меняет стандартная функция способом, детали которого неизвестны анализу), и требуется сообщить этот факт детекторам. Для этого заводятся события анализа, разбитые по группам. Детектор выбирает, о каких событиях требуется его оповещать вызовом

соответствующих функций. Ядро анализа, помимо вычисления модели памяти, классов значений, при обработке некоторой инструкции генерирует все необходимые события. При наступлении события детектор меняет нужным образом вычисляемые им атрибуты. По сути, события становятся внутренним представлением для детекторов, которые должны создать для них передаточные функции.

Список событий анализатора Svace представлен в таблице 4.2. Можно заметить, что некоторые инструкции (например, арифметические) транслируются в события как есть. Другие инструкции генерируют множество событий, например, событие разыменования, копирования, вычисления адреса и т.п. Наконец, некоторые события не соответствуют напрямую конкретной инструкции, а возвещают детектору, например, что анализатор находится на истинной или ложной ветви ветвления, либо что изменился вычисленный интервал значений.

Таблица 4.2. События, генерируемые ядром анализа.

Группа событий / события	**Описание**
`AddrOperationGroup / ptrElemsShift, ptrConstantBytesShift`	Группа, соответствующая операциям над адресами. Обозначает сдвиг базового указателя на некоторое количество элементов (отдельно обрабатываются константные сдвиги).
`BooleanOperationGroup / equals, notEquals, notGreater, greater, notSmaller, smaller`	Операции над булевыми значениями. Возникают, как правило, при соответствующих операциях сравнения.
`ConstantOperationGroup / newInt, newLong, newFloat, newString, undef`	Операции с константами. Возникают при заведении новой константы либо некоторого неизвестного значения.
`CopyOperationGroup / allocNotify, copyNotify, castNotify, addrOfNotify, deref, pmove bufferAssignment, compositeAssignment, bufferAccess, compositeAccess apply, annotate`	Самая многочисленная группа, события которой соответствуют вариантам пересылки данных из одной ячейки памяти в другую. Выделяются события выделения памяти, взятия адреса, копирования по указателю, доступа к массиву и структуре, передаче памяти в неизвестную функцию, возникновения исключения. Также сюда относятся основные служебные события межпроцедурного анализа: `annotate` – запрос к детектору передать свои

passedToUnknownFunc, mayBeChanged exception	структуры данных для сохранения в аннотацию – и apply, изменить атрибуты согласно обрабатываемому вызову и переданной аннотации.
IntegerOperationGroup / add, sub, mul, div, mod, binaryAnd, binaryOr, binaryShl, binaryShr, binaryXor	Группа арифметических и битовых операций, практически точно соответствующая инструкциям.
PathOperationGroup / in, out, assumeCondition, retBranch	События, обозначающие достижение определенной точки на обрабатываемом пути выполнения. Включают ситуации до либо после применения передаточных функций, а также инструкцию assume с наступлением некоторого условия.
ProjectionOperationGroup / notGreater, greater, equals, notEquals, notSmaller, smaller, trueCond, falseCond	События, означающие, что наступление заданного условия нужно отразить в атрибутах детектора.

Теперь рассмотрим, какие атрибуты чаще всего используются детекторами при выдаче предупреждений. Многие атрибуты принадлежат одному из имеющихся классов:

- **AndBooleanFlag**. Это двоичный атрибут (т.е. имеющий только два значения – истина или ложь), который соответствует некоторому факту, выполняющемуся на всех путях через программу (must-отношение). Поэтому в точках слияния потока выполнения функция объединения заключается в конъюнкции значений атрибута. Таких атрибутов в Svace насчитывается около пятидесяти.
- **OrBooleanFlag**. Этот атрибут отражает возможность – некоторый факт выполняется хотя бы на одном пути, соответственно его значения объединяются через дизъюнкцию (may-отношение). Имеется примерно двадцать атрибутов этого класса.
- **TernaryFlag**. Данный атрибут является комбинацией предыдущих двух и необходим для случаев, когда интересует как отношение «может», так и отношение «должен». Он содержит три значения, истинное соответствует

выполнению интересующего свойства на всех путях выполнения, а значение «может быть» – выполнению только на некоторых.

- **AndBoolenFlagWithTrace**, **OrBooleanFlagWithTrace**, **TernaryFlagWithTrace**. Эти классы атрибутов соответствуют упомянутым выше, но дополнительно имеют возможность отслеживать *трассы* – последовательности точек изменения атрибута, про которые необходимо выдать информацию в предупреждении для лучшей информативности. Трасса хранится вместе с атрибутом, и в нее добавляется новая точка при изменении значения на истину либо в другой момент по желанию детектора. Таких атрибутов еще около пятидесяти (на все четыре класса).
- **EnumAttrWithTrace.** Это более точный атрибут, чем троичный атрибут TernaryFlag, с двумя промежуточными значениями «уверенности» в описываемом событии. Значение maybe показывает, что событие произошло, но оно зависит от истинности условия, которая на может быть оценена анализатором (например, условие зависит от внешних параметров или от вызова функции по указателю, цель которого не удалось установить). Значение likely показывает, что событие произошло, но условие его наступления никак не контролируется функцией и вычисленными данными. Например, некоторый указатель разыменуется в том случае, если функция `malloc` вернула `NULL` – наступление этого события никак не зависит от поведения кода, вызывающего `malloc`. Остальные значения false и true традиционны (ничего не известно либо событие точно происходит). Как и атрибуты выше, этот атрибут сохраняет трассу своего изменения.
- **AbstractIntervalType**, **AbstractIntervalValueWithTraceType**. Это атрибуты, содержащие целочисленный интервал (с возможностью отслеживать трассу изменений границ интервала). Таких атрибутов около десяти.
- **AbstractValueIdParameterType**, **ValueIdParameterWithTraceType**. Такие атрибуты содержат класс значений, с которым у данного класса значений, хранящего этот атрибут, есть некоторая связь. Например, атрибут SizeLimitParam устанавливается у класса значений, хранящегося в указательной переменной, и содержит другой класс значений, ограничивающий сверху размер буфера, к которому обращаются через этот указатель. Таких атрибутов (вместе с вариантом, хранящим трассу его распространения) более двадцати.
- **ValueIdSetType.** Атрибут аналогичен предыдущему, но хранит сразу некоторое множество классов значений.

- **ConditionAttr.** Этот атрибут требуется для чувствительных к путям детекторов и отвечает за хранение некоторого условия (логической формулы), при котором наступает интересующее детектор событие. Атрибут поддерживает возможность запросить совместность формулы у SMT-решателя. Модель (набор значений), при которой указанная формула является совместной, также генерируется решателем, после чего трассу выполнения можно восстановить, зная условия переходов для ветвлений потока управления. Поэтому явно хранить трассу не требуется.

Каждый детектор самостоятельно принимает решение о выдаче предупреждения, как правило, при обработке одного из событий таблицы 4.2, на основе текущих значений отслеживаемых им атрибутов. При этом часто используется *гипотеза о существовании контекста*: для некоторой точки программы существует хотя бы один контекст выполнения (набор входных данных программы), при котором выполнение пройдет через эту точку, и программа будет работать без ошибок. Следовательно, если найдена некоторая точка программы, любое прохождение через которую заканчивается ошибкой, то возможно два варианта: либо гипотеза верна, и тогда необходимо выдать предупреждение, либо найден код, который никогда не выполняется. В этом случае также стоит сообщить об ошибке, предполагая, что пользователь не хотел бы видеть в программе ненужного кода. Похожий подход используется и в чувствительных к путям детекторах, но в этом случае точка программы превращается в некоторый путь выполнения.

4.2.1. Разыменование нулевого указателя

Детектор DEREF_OF_NULL ищет ситуации, в которых разыменуется указатель, который был сравнен с `NULL` (с положительным результатом) на всех путях выполнения, приходящих в точку разыменования. Это самая базовая из ситуаций разыменования, обнаруживаемых инструментом Svace. Разыменование может происходить как непосредственно в коде текущей функции, так и в вызываемой функции, в которую передается указатель в качестве параметра. Если такая вызываемая функция неизвестна анализу, то потенциальные предупреждения выделяются в отдельный детектор DEREF_OF_NULL.UNCLEAR.

Простейший пример кода, находимого детектором, представлен на рисунке 10. Тем не менее, такая ситуация часто встречается в реальных проектах, когда ошибка допущена при обработке нештатной ситуации равенства нулю указателя.

```
(1)  void foo (struct boo *pb) {
(2)    if (!pb) {
(3)      if (debug) { // некоторая глобальная переменная
```

```
(4)        fprintf (stderr, "fatal: %s\n", pb->message);
(5)      }
(6)    } else ...
(7)  }
```

Рисунок 4.10. Пример ошибки DEREF_OF_NULL.

Предупреждения выдаются детектором как внутрипроцедурно, так и межпроцедурно. Основным действием детектора является отслеживание равенства нулю указателя через атрибуты PtrIsNull и PtrIsNotNull. Первый атрибут имеет три возможных значения своей полурешетки – истинно, ложно и передано в неизвестную функцию. Функция объединения атрибута выставляет его в истину, только если оба сливаемых атрибута были истинны. Кроме того, отслеживается трасса атрибута для истинного значения. Второй атрибут – это двоичный must-атрибут, который обозначает, что указатель точно не равен нулю. Он устанавливается в истину при создании новой памяти, а также при обработке сравнения на неравенство, если известно, что аргумент равен нулю. Дополнительно атрибут PtrDereferenced отслеживает ситуации, когда разыменование указателя приведет к ошибке. Смысл атрибута в том, чтобы предотвратить повторные выдачи предупреждения на последующих разыменованиях.

Итого, внутрипроцедурные срабатывания детектора происходят, когда выполняется условие `PtrIsNull(p) != false && PtrDereference(p) != true && PtrIsNotNull(p) = false`. Межпроцедурные предупреждения выдаются, когда при вызове функции в ее аннотации записано, что она разыменовывает свой параметр. Тогда дополнительно к этом условию проверяется, что для этого параметра `PtrDereference = true`.

Кроме того, обрабатываются ситуации, когда разыменование в вызываемой функции происходит условно в зависимости от некоторой переменной, которая также не контролируется в вызывающей функции. В таком случае считается, что предупреждение необходимо выдавать. Связь разыменованного указателя и переменной, которая определяет условие разыменования, задается атрибутом DerefIfVarIsUnknown класса ValueIdParameterWithTraceType. Он устанавливается, если в условии находятся только внешние классы значений, и локально для функции не удалось установить, что разыменование приведет к ошибке (т.е. `PtrDereference=false`).

Пример исходного кода для такой ситуации показан на рисунке 4.11. При анализе функции `bar` для переменной `p` в строке 7 устанавливается атрибут DerefIfVarIsUnknown, т.к. `x` является внешним значением. Далее при анализе функции `foo` при вызове `bar` в строке 3

оказывается, что указатель p, соответствующая одноименной переменной из аннотации функции bar, разыменовывается под неизвестным условием, в функции foo этот указатель заведомо равен нулю в строке 3, и значение переменной x также не известно. «Неизвестность» определяется тем, что не определен ни интервал значений, ни выколотая точка для x (т.е. нет значения c, про которое известно, что $x \neq c$). В этом случае выдается предупреждение для строки 3, в котором сохраняется трасса – строка, в которой атрибут, отслеживающий значение указателя, стал истинным (строка 2), и строка разыменования из функции bar (строка 7).

```
(1)  void foo (int *p, int x) {
(2)    if (!p)
(3)      bar (p, x);
(4)  }
(5)  void bar (int *p, int x) {
(6)     if (x == 27)
(7)       *p = 0;
(8)  }
```

Рисунок 4.11. Пример ошибки DEREF_OF_NULL с неизвестными внешними значениями.

При обработке циклов детектор проверяет с помощью SMT-решателя, что заданная точка программы достижима с предикатом пути, который собирает анализатор. Это требуется, чтобы исключить ложные срабатывания для кода, который работает только для некоторых итераций. Например, на рисунке 4.12 разыменование на строке 5 выполняется, начиная со второй итерации цикла, благодаря булевой переменной first. Без проверки достижимости описанный выше внутрипроцедурный детектор выдаст предупреждение.

Нужно отметить, что это достаточно показательный пример того, как детекторы второго уровня могут быть улучшены простым отсечением неисполнимых путей в графе потока управления. Тем самым можно получить детекторы «2.5-го» уровня с частичной чувствительностью к путям.

```
...
(1)  if (!p) {
(2)    bool first = true;
(3)    for (int i = 0;i < N; i++) {
(4)      if (!first) {
```

```
(5)         *p++;  //код недостижим для первой итерации
(6)         first = true;
(7)       }
(8)       p = &q;
(9)     }
(10) }
```

Рисунок 4.12. Пример ложного срабатывания для детектора DEREF_OF_NULL.

Схожим детектором является детектор DEREF_OF_NULL.CONST, который выдает предупреждения, если указателю была явно присвоена константа `NULL` (вместо сравнения). Эти ситуации редко встречаются в реальном коде, но само предупреждение позволяет отсеивать возможные опечатки в коде, возникающие при разработке. Организация детектора аналогична DEREF_OF_NULL, только используется атрибут PtrIsSetToNull вместо PtrIsNull, отслеживающий присваивания вместо сравнений. В циклах предупреждения выдаются только на последней итерации. Если переменная разыменовывается не на всех путях, проходящих через точку присваивания, то выдается предупреждение отдельным подтипом детектора – DEREF_OF_NULL.ASSIGN. Такая ситуация может возникнуть при использовании операции `dyn_cast` в Си++, и в этом случае генерируется предупреждение меньшей критичности.

Следующим типом детектора является более сложный детектор DEREF_AFTER_NULL, для которого важны ситуации сравнения указателя с нулем и последующего разыменования, причем при разыменовании не для всех путей указатель имеет нулевое значение. Из-за того, что, в отличие от DEREF_OF_NULL, нет полной уверенности в том, что указатель нулевой, приходится отсеивать менее вероятные ошибочные ситуации в отдельные подтипы детектора. Такими подтипами являются:

- DEREF_AFTER_NULL.MIGHT для ситуаций, когда между сравнением и разыменованием встречается вызов функции, условно завершающей программу, и анализатор не может понять, всегда ли верно это условие.
- DEREF_AFTER_NULL.COND означает, что разыменование указателя происходит при некоторых условиях, которые в точности не могут быть оценены анализатором (т.е. соответствующее значение атрибута отлично от истины и отражает только возможность).
- DEREF_AFTER_NULL.UNCLEAR соответствует передаче указателя в неизвестную функцию. Как видим, в отличие от DEREF_OF_NULL, в данном

детекторе такие предупреждения имеют меньшую критичность, чем основные ситуации для детектора.

Кроме того, для увеличения точности детектора применяется обратный анализ. В точке сравнения указателя с нулем еще неизвестно, существуют ли пути, проходящие через сравнение, но не через разыменование. Для определения этого факта заводится дополнительный атрибут BackwardPtrDereferenced, который будет распространяться в обратную сторону по графу потока управления после окончания прямого анализа. Тогда предупреждение будет выдано, если для некоторого класса значений указателя окажется, что этот атрибут не является ложным, а указатель может быть нулевым. Дополнительно отсеиваются предупреждения, в которых между сравнением и разыменованием встречается вызов функции по указателю, если не удалось точно установить, какая функция вызывается.

Наконец, последним детектором разыменования нулевого указателя без чувствительности к путям является детектор NULL_AFTER_DEREF, который ищет «перевернутую» относительно DEREF_AFTER_NULL ситуацию – указатель, который сначала был разыменован, впоследствии сравнивается с нулем. Нужно отметить, что в корректной программе код после положительного сравнения будет недостижим, т.к. предшествующее разыменование нулевого указателя повлечет за собой неопределенное поведение. В частности, компилятор вправе предполагать, что разыменование означает предположение об отличии указателя от нуля, и может выполнять оптимизации на основе этого предположения.

Если значения атрибута PtrDereferenced отличны от истины и лжи (обозначают лишь возможность), то выдается отдельный подтип NULL_AFTER_DEREF.MIGHT. Такая ситуация возникает, например, когда разыменование зависит от внешнего неконтролируемого условия (скажем, открытия файла). Отсеиваются предупреждения, когда разыменование выполняется внутри вызываемой функции, а сравнение с нулем ее возвращаемого значения – внутри вызывающей. Это может быть защитными проверками результата библиотечных функций. Кроме того, отдельно обрабатываются сравнения результата вызова `new` – вообще говоря, в пользовательском коде они лишние, и нужно отбрасывать сгенерированные компилятором сравнения (например, для вызова `delete ptr`).

Детекторы DEREF_AFTER_NULL.RET отслеживают ситуации проверки результата некоторых функций, возвращающих указатель, на нулевое значение. Основным является поддетектор DEREF_AFTER_NULL.RET.LIB, который проверяет библиотечные функции. Детектор использует троичный атрибут PossibleNull, устанавливаемый из спецификаций. Предупреждение выдается только для истинного значения атрибута, в том числе и межпроцедурно.

4.2.2. Использование памяти после освобождения

Детекторы этой группы ищут ситуации, в которых некоторый указатель используется после того, как память, на которую он указывает, была освобождена. В зависимости от типа использования выделены детекторы DEREF_AFTER_FREE – указатель разыменовывается, USE_AFTER_FREE – указатель записывается в глобальную переменную либо возвращается из функции, PASSED_TO_PROC_AFTER_FREE.EX – передается в некоторую функцию, DOUBLE_FREE.EX – повторно освобождается. Детекторы используют как обычные, так и чувствительные к путям атрибуты, поэтому являются промежуточными между детекторами второго и третьего типа.

Центральной частью группы детекторов является межпроцедурное отслеживание освобожденной памяти. Для этого заводятся три атрибута. Атрибут FreedValueIdsAttr содержит множество классов значений, соответствующих указателям, память которых была освобождена в текущем базовом блоке. По окончании текущей ветви выполнения информация об условии, при котором освобождается память, для каждого такого класса значений вносится в атрибут ConditionalFreeFlagValue (подробнее о нем ниже). Дополнительно must-атрибут MustFreeFlag помечает указатели, память по которым точно была освобождена. Последние два атрибута распространяются межпроцедурно.

Комбинация указанных атрибутов призвана отсеивать ложные срабатывания в часто встречающихся случаях, в которых при обработке некоторой исключительной ситуации память по указателю, являющемуся параметром функции, освобождается и возвращается некоторый код ошибки. Далее в вызывающей функции этот код проверяется, и работа с указателем далее ведется только при отсутствии ошибки. На примере кода с рисунка 4.13 функция `foo` освобождает память по указателю `p` в строке 4 в случае возникновения ошибки, контролируемой внешней функцией `is_error` (строка 3), и возвращает ошибочное значение -1, иначе в обычном случае возвращает 0. Функция `bar` при возврате функцией `foo` отрицательного значения немедленно прекращает работу (строка 12). Для того, чтобы не выдавать ложное предупреждение об использовании освобожденной памяти в строке 13, в атрибуте ConditionalFreeFlagValue отмечается условие освобождения памяти по классу значения указателя `p`, и дополнительно сохраняется возвращаемое при этом значение. Далее в функции `bar` после обработки аннотации функции `foo` на строке 11 окажется, что память по указателю `p` освобождена в базовом блоке строки 12, но не строки 13, т.к. соответствующее условие будет несовместно с отрицанием условия строки 11.

```
(1)   int is_error();
(2)   int foo (int *p) {
```

```
(3)     if (is_error ()) {
(4)       free (p);
(5)       return -1;
(6)     }
(7)     ...
(8)     return 0;
(9)   }
(10)  void bar (int *p) {
(11)    if (foo (p) < 0)
(12)      return;
(13)    p[0] = 1; // нельзя выдавать DEREF_AFTER_FREE.
(14)  }
```

Рисунок 4.13. Пример отсеивания предупреждения для детектора DEREF_AFTER_FREE.

Более детально, атрибут FreedValueIdsAttr имеет пустую функцию слияния (всегда возвращает значение атрибута по умолчанию), так как он отслеживает только освобождение памяти в текущем базовом блоке. При обработке аннотации во множество классов значений, хранящихся в атрибуте, добавляется класс значения из аннотации в случае, если для него атрибуты MustFreeFlag установлен в истину или выполняется условие из атрибута ConditionalFreeFlagValue. Сам этот атрибут отслеживает два условия, будем обозначать их FullCond **и** RetCond: условие FullCond является необходимым для того, чтобы выполнение достигло точки программы, в которой освобождается память, а условие RetCond определяет возвращаемое функцией значение после освобождения памяти и имеет вид `ret == value`.

Функция слияния атрибута ConditionalFreeFlagValue для условия RetCond просто формирует дизъюнкцию условий, пришедших по веткам слияния, т.к. обычно возвраты из функций немногочисленны, и дополнительное упрощение условия не требуется. Условие же FullCond упрощается при слиянии, как упоминалось в разделе 2.4.2, в данном случае – через удаление общей части условия, относящегося к блоку, доминирующему оба блока, пути из которых сливаются. Формально, пусть блок D непосредственно доминирует блоки A и B, управление из которых сливается, condD, condA, condB – необходимые условия попадания в блок D, A, B соответственно. Тогда $join(FullCond(A), FullCond(B)) = FullCond(a) \wedge (condD / condA) \vee FullCond(b) \wedge (condD / condB)$.

При освобождении памяти атрибут MustFreeFlag устанавливается в истину, как и оба условия атрибута ConditionalFreeFlagValue. При обработке аннотации, если в ней MustFreeFlag для некоторого класса значений равен истине, то в вызывающей функции для этого класса условие FullCond устанавливается в условие RetCond из аннотации, а условие RetCond устанавливается в истину. В противном случае условие FullCond переходит также в FullCond. При окончании ветви выполнения для всех классов значения VC из FreedValueIdsAttr соответствующие условия FullCond и RetCond дополняются через конъюнкцию вида `retVal`VC == VC. Атрибуты также переносятся через указатели, полученные адресной арифметикой и взятием элементов сложных типов, а при передаче в неизвестную функцию условия сбрасываются в ложь.

При выдаче предупреждений проверяется совместность текущего условия и условия FullCond для класса значения указателя, а также дополнительные ограничения, зависящие от детектора. Для случая двойного освобождения проверяется равенство истине атрибута MustFreeFlag, для разыменования – атрибута PtrDereferenced (в самом событии разыменования проверяется только условие). При передаче в неизвестную функцию тип предупреждения зависит от того, что это за функция – если это сгенерированный компилятором вызов memcpy, то в исходном коде было обычное разыменование, и выдается предупреждение DEREF_AFTER_FREE, а для обычной функции выдается PASSED_TO_PROC_AFTER_FREE.EX. Это пример сложностей, связанных с анализом внутреннего представления низкого уровня. Наконец, при записи по указателю в глобальную переменную без дополнительных проверок выдается предупреждение USE_AFTER_FREE.

4.2.3. Утечки памяти и ресурсов

За потенциальную утечку ресурсов отвечает группа детекторов MEMORY_LEAK и HANDLE_LEAK. Устройство детекторов отличается незначительно (основные алгоритмы устроены одинаково, но может быть некоторая разница в поведении атрибутов, отслеживающих состояния ресурсов), поэтому далее наше описание будет сконцентрировано на MEMORY_LEAK, который определяет утечки памяти вдоль некоторых путей выполнения.

Подтипы детектора с меньшей вероятностью истинного срабатывания включают в себя MEMORY_LEAK.STRUCT (утечка памяти, выделенной под сложную структуру данных; трудности связаны с обработкой ссылочных структур типа деревьев или списков), MEMORY_LEAK.GLOBAL (память зависит от глобальной переменной). Также отдельно выделяется подтип MEMORY_LEAK.STRDUP для утечек памяти строк (частый случай) и чувствительный к путям подтип MEMORY_LEAK.EX.

Утечки памяти являются примером ошибок, поиск которых плохо укладывается в схему создания детекторов, предложенную в разделе 2.4. Детекторы утечек, по сути, представляют из себя компонент, состоящий из нескольких отдельных анализов. Основной частью является алгоритм поиска достижимых объектов, начиная с некоторого стартового множества. Если находятся объекты, на которые нет ссылок (указателей либо дескрипторов ресурса), то выдается предупреждение об утечке. Сложность в том, что для такого поиска необходимо отслеживать не только вновь созданные объекты, но и их освобождение, в том числе в вызываемых функциях, либо сохранение ссылок на них во внешних структурах данных (англоязычный термин – escaping). Также нужно учитывать не только указатели, покидающие область видимости, но и на указываемую через них память (см. пример на рисунке 4.14, в котором в строке 7 переписывается указатель на элемент списка, что приводит к утечке другого указателя, хранящегося в этом элементе).

```
(1)   struct list { struct list *p; struct val *val };
(2)   extern int error();
(3)   void create (struct list **pplst) {
(4)     struct list *plst = *pplst;
(5)     plst->val = malloc (sizeof (struct val));
(6)       if (error ()) {
(7)         *plst = 0; //здесь потеряна ссылка на plst->val
(8)     }
(9)   }
```

Рисунок 4.14. Непрямая утечка памяти.

Алгоритму поиска достижимых объектов передается множество «видимых» классов значений visibleValues, тип поиска (для вызова функции, возврата из функции, присваивания указательных переменных), конкретный детектор, задающий атрибуты, содержащие интересующие нас ресурсы и условия их создания и освобождения. На первом этапе в зависимости от типа поиска в стартовое множество visibleValues добавляются глобальные переменные, переменная, которой было присвоено значение, параметры функции, возвращаемое значение, возможно, локальные переменные. Далее во множество добавляются элементы, для которых родители (в терминах ячеек памяти) уже находятся во множестве, и escape-элементы (т.е. сохраняющиеся вовне функции). Наконец, итеративно добавляются элементы, которые могут быть получены либо разыменованием, либо адресной арифметикой

по имеющимся элементам (аналогично тому, как строится множество сохраняемых ячеек памяти при построении аннотации функции).

После окончания работы алгоритма достижимости просматриваются значения, не попавшие во множество видимых и потенциально являющиеся выделенными ресурсами (конкретные значения определяются детектором), но при этом не освобождены. Для них обычным детектором проверяется, что если место возникновения подозрения об утечке постдоминирует точку выделения ресурса, то выдается предупреждение об утечке. В случае же детектора, чувствительного к путям, проверяется совместность условия выделения и освобождения ресурса совокупно с необходимыми условиями попадания в данную точку программы (предикатом пути).

Алгоритм распространения escape-пометок (атрибут EscapeValue) обрабатывает ячейки памяти следующим образом:

- Атрибут распространяется через разыменования и родительские ячейки памяти;
- Если функция освобождает элементы рекурсивных структур данных, то помечается указатель на головную структуру;
- Помечается указатель, записанный в память, на которую ссылаются глобальные переменные;
- Помечаются указатели, переданные в спецфункции, в неизвестные анализу деструкторы или функции, вызванные по указателю;
- Помечаются указатели, переданные в известные анализу шаблонные функции, относящиеся, например, к `shared_ptr`, `unique_ptr` и т.п.;
- Помечаются указатели, переданные в неизвестные функции, принимающие другие внешние параметры (т.е. также помеченные как EscapeValue).

Для выполнения описанного анализа детекторы предоставляют следующие атрибуты. Атрибут HeapAmlFlag помечает ячейки памяти, выделенные на куче, а его аналог ResourceAmlFlag – ячейки, соответствующие другим ресурсам. Атрибут FreeFlag также устанавливается на ячейки памяти. Имеет значения NotAllocated, соответствующее состоянию «память не была выделена» (это происходит, если указатель равен нулю либо после проверки результата функции выделения памяти), FreeCalled – память была освобождена (устанавливается в спецификациях функций освобождения ресурсов) и may-варианты этих значений (память не была выделена либо была освобождена только вдоль некоторого пути выполнения).

Для отслеживания ошибок выделения ресурсов дополнительно используется атрибут BadAllocResult, связывающий ячейку памяти указателя на ресурс и интервал значений для

возвращаемого значения, при котором выделение не произошло (к примеру, так бывает в функции `asprintf`). Если при сравнении на равенство используется значение, попадающее в отслеживаемый интервал, то атрибут FreeFlag для ячеек памяти, на которые указывает связанный с этим интервалом в атрибуте BadAllocResult указатель, устанавливается в NotAllocated.

Атрибуты AmlCreatedIfCond и AmlResourceCreatedIfCond отслеживают условия, при которых память (или, соответственно, другой ресурс) были выделены. Если ячейка памяти стала помечена атрибутом EscapeValue, то отслеживаемые условия сбрасываются. Кроме того, условия сбрасываются для указателей на локальные переменные (они не могли быть выделены в динамической памяти), а также для нулевых указателей (для других ресурсов ноль может быть корректным значением дескриптора, поэтому такое действие выполняется только для AmlCreatedIfCond). Первоначальные условия берутся из спецификаций стандартных функций выделения.

Нужно заметить, что почти все атрибуты детекторов поисков утечек связаны не с классами значений, а с самими ячейками памяти. Эта ситуация встречается редко, но может быть оправдана, если требуется отследить именно свойства ячейки, а не хранящегося в ней значения, как в нашем случае. Например, если несколько указателей указывают на одну и ту же ячейку памяти, то атрибут FreeFlag можно ассоциировать с этой ячейкой вместо того, чтобы связывать его с несколькими указателями.

4.2.4. Отслеживание помеченных данных

Группа детекторов с именами, начинающимися на TAINTED, связана с проверкой корректного использования данных, полученных из ненадежных источников (в частности, от пользовательского ввода). Как правило, для такого вида ошибок определяют набор функций, поставляющих небезопасные данные, функции, в которых должны использоваться проверенные данные (параметры), а также способы «очистить» (sanitize) данные, т.е. превратить их из ненадежных в проверенные (обычно это также некоторые функции). Детекторы должны убедиться, что нет путей, по которым данные могут течь от небезопасных источников к проверенным функциям, не покрываясь предварительно «очистительными» вызовами функций.

Многие ошибки неправильного использования API, в том числе ошибки безопасности, могут быть найдены с помощью отслеживания помеченных данных. В основном движке Svace в настоящий момент не реализована отдельная инфраструктура для написания таких детекторов, это дело будущих работ. Пока такая инфраструктура представлена только для языка C#, о чем будет подробно сказано в своем месте. А текущие taint-детекторы в Svace

связаны с отслеживанием небезопасных данных самых базовых типов. Это детекторы TAINTED_INT, обрабатывающие целые числа как опасное данное, TAINTED_PTR, отслеживающий строки, и TAINTED_ARRAY_INDEX, направленный на небезопасные индексы массивов.

```
(1)   char * increase (char *p, int n) {
(2)     int x;
(3)     if (fill())
(4)       scanf ("%d", &x);
(5)     else
(6)       x = n + 1;
(7)     char * buf = realloc (ptr, x); // опасное использование
(8)     return buf;
(9)   }
```

Рисунок 4.15. Пример детектора TAINTED_INT.

Детектор TAINTED_INT имеет подтипы TAINTED_INT.LOOP (для случаев использования небезопасного целого числа в условии выхода из цикла), TAINTED_INT.CTYPE (для функций, объявленных в заголовочном файле `ctype.h`, т.е. функций `islower`, `isdigit`, `isascii` и т.п.), а также might-версии этих типов, которые выдаются, если значение помечено как небезопасное не для всех путей выполнения. Проверка выполнимости пути, для которого выдается предупреждения, эти детекторы не выполняют (чувствительная к путям версия есть только у детектора TAINTED_ARRAY_INDEX). Например, на рисунке 4.15 будет выдано предупреждение в строке 7, так как размер выделенной памяти для функции `realloc` мог быть получен из внешнего источника в строке 4.

Детектор отслеживает атрибуты MustTaintedInterval и MightTaintedInterval, описывающие интервалы значений, которые получены из внешнего источника (на всех либо некоторых путях). Атрибуты ведут себя подобно основному атрибуту ValueInterval в смысле реакции на арифметические операции, сравнения и т.п., но получают изначальные небезопасные значения из спецификаций, а кроме того, сохраняют в себе трассу изменения интервала. За описание, наоборот, проверенных значений отвечают атрибуты TrustedIntSinkFlag и TrustedCharSinkFlag (для символов), принадлежащие классу атрибутов EnumAttrWithTrace. Наличие атрибута на формальных параметрах функции означает, что фактический параметр не должен принимать небезопасные значения. Изначально эти атрибуты также устанавливаются в спецификациях.

Предупреждение об использовании небезопасного значения выдается, если trusted-атрибуты установлены в истину, а tainted-атрибуты указывают на полный небезопасный интервал (т.е. $[-\infty, +\infty]$), либо его левая граница меньше нуля, либо правая равна $+\infty$. Дополнительно нежелательные срабатывания отсекаются атрибутом MainArgcPar, который отслеживает значения, полученные из параметра `argc` функции `main` – считается, что обычно они слишком малы для принесения вреда при передаче в доверенные функции. Кроме того, пара атрибутов UpperBoundIsChecked и LowBoundIsChecked отмечают проверки верхней и нижней границы небезопасного значения сравнением с неизвестным анализу числом. В этом случае также разумнее не выдавать предупреждение.

Схожим образом детектор TAINTED_PTR отслеживает небезопасные строки через троичный атрибут TaintedPtr, указывающий строки, полученные из внешнего источника, и двоичный TrustedPtrSinkFlag, помечающий указательные параметры, которые должны быть безопасны для корректного использования функции. Первый атрибут устанавливается в спецификациях функций чтения из файла, сети и окружения, второй – для функций копирования памяти и строк, работы с файлами, запуска процессов и т.п. Для выдачи предупреждения нужно, чтобы tainted-атрибут был отличен от лжи, а trusted-атрибут равен истине. Помимо этого поддерживается атрибут SanitizationInvoked, помечающий функции, которые «очищают» данные (такими функциями эвристически считаются функции сравнения строк и вычисления их длины). Если этот атрибут отличен от лжи, то предупреждение не выдается.

Детектор TAINTED_ARRAY_INDEX похож на TAINTED_INT, но в его случае использованием небезопасных данных, о которых нужно предупреждать, является применение небезопасного целого в качестве индекса массива. Как следствие, он сигнализирует о ситуациях возможного переполнения буфера. Поэтому у него есть вариант с чувствительностью к путям, в котором предупреждение выдается детектором переполнения буфера, который дополнительно проверяет помеченность индекса.

4.2.5. Другие детекторы для Си, Си++ и Java

В этом подразделе мы опишем несколько детекторов, которые не принадлежат какой-либо большой группе – это детекторы простого переполнения буфера, несоответствия функций выделения и освобождения памяти, нетерминированной строки. Остальные критические детекторы являются чувствительными к путям и будут описаны в следующем разделе.

```
(1)   int buf10[10]; int buf20[20]; int *p;
(2)   ...
```

```
(3)   if (cond1)
(4)     p = buf10;
(5)   else
(6)     p = buf20;
(7)   if (cond2)
(8)     p++;
(9)   else
(10)    p += 10;
(11)  ...
```

Рисунок 4.16. Пример кода для атрибута BufferSizeAttr.

Детектор STATIC_OVERFLOW предназначен для поиска простых переполнений буфера, которые происходят всегда с константным индексом и размером массива. Для отслеживания индекса используется обычный атрибут интервала значений, а для оценки размера массива – атрибут BufferSizeAttr, который содержит интервал для размера и для смещения указателя относительно начала массива. Например, для кода на рисунке 4.16 в строке 11 интервал размера будет равен [10, 20], а интервал смещения – [1, 10]. Атрибут инициализируются в функциях выделения памяти и появления новых ячеек памяти и изменяется при адресной арифметике и вычислениях адреса элементов массивов и структур (getelementptr).

Предупреждение выдается, когда вычисленный интервал индекса (с учетом смещения) превосходит интервал размера или меньше нуля. Дополнительно предупреждение подавляется, если выставлен в истину один из двоичных must-атрибутов, отслеживающих, что указатель был преобразован через тип объединения (union) либо его длина проверена функцией `strlen`. Кроме того, смещение может быть не известно точно, а массив представлять из себя VLA-массив, тогда предупреждение также подавляется. В циклах также проверяется достижимость места выдачи предупреждения, как описывалось ранее в разделе 4.2.1 для рисунка 4.12.

Детекторы HEAP_INCOMPATIBLE.FREE и HEAP_INCOMPATIBLE.ARRAY ищут случаи, в которых память выделяется и освобождается с помощью не соответствующих друг другу функций: в первом случае перепутаны интерфейсы `malloc/free` и `new/delete`, во втором – `new/delete` и `new[] / delete[]`. Для этого заводятся троичные атрибуты CreatePtrFlag и DeletePtrFlag, которые отслеживают тип вызванной функции выделения или освобождения памяти. Если второй атрибут установлен в истину, а первый – в истину или возможную истину, то из них извлекаются типы вызванных функций и сравниваются на совместимость. Атрибуты распространяются с учетом того, что некоторые указатели могут

быть нулевыми – в этом случае значение атрибута такого указателя не принимает участия в функции слияния атрибутов.

Детектор NONTERMINATED_STRING предназначении для ситуаций, когда после копирования строки (функции `strncpy/memcpy`) в строке-результате не будет последнего нулевого символа. Для этого атрибут StringLength обычного интервального типа отслеживает длину строки, созданной через константные строки, функции копирования строк, обрабатывая явное присваивание символу строки нуля, вызовы функции `memset` и т.п. При нахождении вызова функций копирования выполняется проверка длины функции и размера массива под нее, в который происходит копирование.

Детектор UNINIT.CTOR специфичен для Си++ и выдает предупреждения для полей класса, которые не инициализированы в конструкторе класса. Для каждого поля заводится троичный атрибут MemberIsInit, который устанавливается в истину в операциях присваивания этому полю, передачи адреса поля внешним функциям и копирования его в другие переменные. Если для нескольких полей инициализация не выполнена, то выдается отдельный подтип UNINIT.CTOR.MANY (такая ситуация с меньшей вероятностью является ошибкой).

Детекторы поиска ошибок синхронизации реализованы для Си и Java, однако в Java-варианте они более эффективны из-за того, что оператор `synchronized` всегда синтаксически сбалансирован. Поэтому сконцентрируемся на работе Java-детекторов DEADLOCK (взаимные блокировки) и NO_LOCK.STAT (потенциальные состояния гонки). Они основаны на анализе дополненного графа вызовов — модели параллельной программы, описывающей выполнение программы в несколько потоков.

Локальным дополненным графом вызовов будем называть двунаправленный граф, имеющий вершины трех типов: начало метода, захват ресурса, вызов метода. Вершина любого типа содержит в себе ссылку на место в исходном коде программы, где произошло то или иное событие. Вершины начала или вызова метода, кроме того, содержат уникальный идентификатор метода. Вершина захвата ресурса содержит абстрактный ресурс, идентифицирующий ресурс, который необходимо захватить потоку для входа в критическую секцию. Локальный граф строится, соединяя вершины захвата ресурсов согласно путям в графе потока управления, проходящим через них и начало метода, и аналогично соединяя вершины с вызовами методов. Целью является представить информацию о том, какие ресурсы необходимо захватить, чтобы добраться до вызова метода. *Дополненным графом вызовов* будем называть граф, полученный из локальных, путем соединения соответствующих вершин первого и третьего типов. Так как граф вызовов обходится анализом от вызываемых функций к вызывающим, то дополненный граф можно строить инкрементально по ходу анализа – локальные графы для вызываемых функций в точке вызова уже будут известны.

Пусть t и t′ – два абстрактных потока, а l_1, l_2, l_1', l_2' – инструкции блокировки в исходном коде программы. Взаимная блокировка потоков t и t′ будет иметь место, если t и t′ будут ждать освобождения ресурсов r_1 и r_2 при выполнении l_2 и l_2' , при этом владея ресурсами r_2 и r_1 после выполнения l_1 и l_1' соответственно. Детектор DEADLOCK ищет определенный таким образом дефект обходом в глубину дополненного графа вызовов. При этом предупреждения подавляются для случая, когда выполняется захват общего для потоков t и t' ресурса до выполнения инструкций l_1 и l_1'. Для вершин дополненного графа вызовов, соответствующим l_1 и l_1', формируется множество доминирующих вершин захвата ресурса. Если пересечение полученных множеств не пусто, то предупреждение не выдается.

Детектор NO_LOCK.STAT собирает статистику обращения к полям классов во время обхода графа потока управления, отдельно считая обращения внутри и вовне критической секции, а также сохраняя ресурс, который захватывается при входе в секцию. Если защищенные обращения случаются значительно чаще незащищенных, то для последних выдается предупреждение о потенциальном состоянии гонки. Дополнительно проверяется достижимость точек обращения для случая чувствительного к путям подтипа детектора.

Кроме того, поскольку детектор не учитывает возможные контексты вызова метода, внутри которого произошло небезопасное обращение к полю, возможна выдача ложных предупреждений. Такая ситуация встречается, например, когда контракт метода предусматривает вызов данного метода только при захвате потоком необходимого ресурса. Для их определения исследуются все пути в дополненном графе вызовов, ведущие в вершину начала метода с небезопасным обращением, и если на каждом пути встречается вершина захвата необходимого ресурса, то предупреждение не выдается.

4.2.6. Детекторы C#

Межпроцедурные детекторы в движке анализа для языка C# все являются чувствительными к путям и, следовательно, будут описаны ниже в разделе 4.3. Здесь мы кратко остановимся на детекторах, отслеживающих небезопасные значения, аналогичных детекторам раздела 4.2.4.

Детекторы используют общую инфраструктуру поиска на основе IFDS-задачи, описанную в разделе 3.3.6. Реализовано около 20 детекторов, относящихся в основном к уязвимостям безопасности, возникающих вследствие неверного использования популярных программных интерфейсов. Детекторы делятся на два типа – в первом случае информация продвигается от множества источников вперед к доверенным стокам, во втором, наоборот, обратный анализ прослеживает переменные заданных типов от доверенных стоков к местам их определения, чтобы проверить их на небезопасность. Примером детектора первого типа

является детектор SQL_INJECTION, который прослеживает массивы и строки от функций ввода из файлов или сети до заданного списка функций работы с базами данных. Примером детектора второго типа является детектор CONSTANT_CREDENTIALS, который прослеживает связь между функциями установки паролей или криптографии и некоторыми константными строками.

4.3 Детекторы, различающие пути выполнения

В этом разделе описываются основные чувствительные к путям детекторы, сначала для основной инфраструктуры Svace (с применением классов значений и атрибутов), а потом для языка C# (на основе только лишь символьного выполнения и символьной модели памяти). Некоторые из детекторов, использующие чувствительные к путям атрибуты, для удобства были описаны в предыдущем разделе в рамках одной группы детекторов (например, MEMORY_LEAK.EX). Кроме того, мы уже отмечали, что часть детекторов, атрибуты которых нечувствительны к путям, использует SMT-решатель для отсечения нереализуемых путей выполнения.

4.3.1. Переполнение буфера

Основным детектором ситуаций переполнения буфера является детектор BUFFER_OVERFLOW.EX. Он делится на две части, первая из которых посвящена глобальным буферам и локально выделенным буферам с известным размером, а вторая – динамически выделенным буферам (DYNAMIC_OVERFLOW.EX). Каждая из частей ставит себе целью поиск ошибки следующего вида: если заданы символьные переменные $\vec{x}$, а условие достижимости точки программы q вдоль пути p обозначается как $RC_q^p(\vec{x})$, то требуется проверить, что $\exists p \in P: (\exists \vec{x} : \left(RC_q^p(\vec{x})\right) \wedge \ \forall \vec{x}: (RC_q^p(\vec{x}) \rightarrow i(\vec{x}) \notin [0, S(\vec{x}) - 1]))$, где S и i – размер и индекс буфера соответственно. В общем виде построение этой формулы и ее проверка решателем неудобны, поэтому детектор использует инфраструктуру Svace для создания более удобной для проверки формулы, но и более строгой (то есть из построенной формулы следует представленное выше определение, но не наоборот).

Детектор заводит атрибут ValueDefinitionAttr, целью которого является сохранять историю операций, выполняемых с индексами массива, так, что в заданной точке программы можно построить достаточные условия того, что значение индекса не меньше либо не больше некоторого числа. Тогда достаточным условием ошибки будет конъюнкция условия достижимости точки обращения к массиву по этому индексу и условий того, что индекс не превосходит размера массива и не меньше нуля. Для этого ход выполненных с классом значения индекса операций сохраняется в виде ориентированного ациклического графа, в

котором листья соответствуют константам, а узлы – операциям над индексом, которые могут быть арифметическими операциями, сравнениями, или слиянием потока управления. Можно заметить, что нечто подобное происходит при сохранении истории вычислений над внешними классами значений, как это описано в разделе 2.2.3. Также важно, что сохраняются не сами условия, которые могут быть достаточно громоздкими, а данные об операциях с классом значений, которые необходимы для последующего построения нужных условий по требованию.

Оказывается, что хранение дополнительных условий (помимо выполненных операций) требуется только для функции слияния, а в остальных случаях условия могут быть построены непосредственно по виду операции и таким же условиям для ее операндов. Действительно, для констант формула является тривиальной: если v – значение, равное константе c, и нужно построить условие того, что это значение не меньше некоторого числа x, то это условие есть $(v = c) \wedge (c \geq x)$. Для вычитания $a - b$, например, если известны условия верхней и нижней границы $HB(a, x)$ и $LB(a, x)$, то условие верхней границы есть $(v = a - b) \wedge \left(\exists \tilde{a} \exists \tilde{b}\, LB(a, \tilde{a}) \wedge HB\left(b, \tilde{b}\right) \wedge \left(\tilde{a} - \tilde{b} \geq x\right)\right)$, так как верно, что $a \geq \tilde{a} \wedge b \leq \tilde{b} \wedge \tilde{a} - \tilde{b} \geq x \Rightarrow a - b \geq x$. Аналогично, для сравнения $v > expr$ условие записывается как $(v > expr) \wedge HB(expr, x - 1)$. Наконец, для слияния запись формулы полностью аналогична формулам отслеживания предикатов для классов значений, рассмотренным в разделе 2.3.1, и представляет из себя дизъюнкцию условий на верхнюю границу, пришедших по левой и правой ветви слияния, взятых с конъюнкцией условия выполнения соответствующей ветви: $\bigvee \begin{matrix} (v = l) \wedge HB(l, x) \wedge c_l \\ (v = r) \wedge HB(r, x) \wedge c_r \end{matrix}$, где l и r – это классы значений, пришедшие из левой и правой ветви, а c_l и c_r – соответствующие условия выполнения. Тем самым видна необходимость дополнительного сохранения этих условий ветвей при слиянии.

При выдаче предупреждения о потенциально ошибочном доступе к буферу извлекается информация об операциях с индексом доступа, по ней рекурсивно строятся условия верхней и нижней границы, где числом-параметром является размер буфера, и берется конъюнкция с условием достижимости. Результирующая формула проверяется SMT-решателем, и при обнаружении совместности формулы по модели, выданной решателем, строится трасса ошибки, описывающая ход изменения индекса.

Дополнительно в теле цикла при построении условий учитывается информация об индуктивных переменных, построенная на этапе консервативного анализа потока данных (шаг 5б алгоритма 3.1). Например, если известно, что индуктивная переменная $i = init + k * step$, и цикл повторяется, пока $i \leq n$, то $HB(i, x) = \exists k\colon (init + k * step \geq x) \wedge \big((MAXINT - init) \div$

$step \geq k \geq 0) \wedge HB(n, init + k * step)$, то есть нужно, чтобы нашлась достаточно большая итерация k, чтобы значение i перешло заданную верхнюю границу.

Рассмотрим пример срабатывания детектора на рисунке 4.17, взятый из кода операционной системы Android. Из сравнения переменной `ci` с константой 2 на строке 7 следует вывод, что это значение возможно реализовать в цикле (с учетом гипотезы существования контекстов и того, что в условном операторе строке 3 `ci < 4`), а значит, на следующей итерации цикла переменная `ci` станет равна трем, и в строке 5 произойдет переполнение буфера.

```
(1)   for (ci = 0; ci < folder->NumCoders; ci++) {
(2)     ...
(3)     if (folder->NumCoders == 4) {
(4)       UInt32 indices[] = { 3, 2, 0 };
(5)       si = indices[ci];
(6)       ...
(7)       if (ci == 2) {
(8)         ...
(9)       }
(10)    }
(11)  }
```

Рисунок 4.17. Пример ошибки BUFFER_OVERFLOW.EX.

Атрибут ValueDefinitionAttr распространяется и межпроцедурно, что позволяет искать ошибки, в которых вычисление индекса происходит в вызываемой функции, а доступ к массиву – в вызывающей. Сохраненные в атрибуте условия и операции переписываются при обработке аннотации в термины вызывающей функции. Если же доступ происходил в вызываемой функции с участием индекса, вычислявшегося в вызывающей, то выдать ошибку в вызываемой функции не представляется возможным, т.к. про переданные в нее значения еще ничего не известно. Для таких случаев при обработке инструкций доступа в аннотации функции сохраняется та часть графа операций с индексом, которая зависит от внешних значений, и (в отдельном атрибуте) размер массива. Собственно проверка будет проведена при обработке аннотаций в вызывающей функции.

Отдельным подтипом детектора является TAINTED_ARRAY_INDEX.EX, который вводит в историю операций с классами значений, помимо констант, пометку о том, что данное

значение получено из небезопасного источника. При этом значение становится свободной переменной в условиях для SMT-решателя. Интервалы значений, вычисленные обычным детектором, в этом случае игнорируются, так как все необходимые условия и вычисления и так будут переданы решателю. Пример найденного таким образом срабатывания показан на рисунке 4.18. Массив `rtnl_rtprot_tab` размера 256 элементов, определенный в строке 1, передается в функцию `rtnl_tab_initialize` на строке 8. Эта функция считывает из файла строку, а из нее число в переменную `id`, не контролируя его размер, и проверяет на соответствие размеру массива, переданному через параметр `size`, в строке 23. Однако в проверке допущена опечатка, допускающая индекс, равный 256, и происходит переполнение буфера на один элемент в строке 25.

```
(1)   static char * rtnl_rtprot_tab[256] = {
(2)           [RTPROT_UNSPEC] = "none",
(3)           [RTPROT_REDIRECT] ="redirect",
(4)           ...
(5)   }
(6)   static void rtnl_rtprot_initialize (void) {
(7)     rtnl_rtprot_init = 1;
(8)     rtnl_tab_initialize (CONFDIR "/rt_protos",
(9)     rtnl_rtprot_tab, 256);
(10)  }
(11)   static void rtnl_tab_initialize (char *file, char **tab,
int size) {
(12)    char buf[512];
(13)    ...
(14)    while (fgets (buf, sizeof (buf), fp)) {
(15)      char *p = buf; int id;
(16)      char namebuf[512];
(17)      ...
(18)       if (sscanf(p, "0x%x %s\n", &id, namebuf) != 2 &&
(19)       ...
(20)           sscanf(p, "%d %s #", &id, namebuf) != 2) {
(21)         return;
(22)       }
(23)       if (id < 0 || id > size)
```

```
(24)         continue;
(25)       tab[id] = strdup (namebuf);
(26)     }
(27)     ...
(28)  }
```

Рисунок 4.18. Пример ошибки TAINTED_ARRAY_INDEX.EX.

Детектор DYNAMIC_OVERFLOW.EX обнаруживает ошибки доступа к буферу, выделенному в динамической памяти, максимально близко к приведенной формуле-определению ошибки. В условии ошибки индекс и размер буфера участвуют непосредственно как символьные переменные, и дополнительно в условие ошибки записываются все предикаты, описывающие историю операций с соответствующими классами значений в атрибуте ValueDefinitionAttr. Условия достижимости отдельных путей не используются, вместо этого все элементарные условия переходов в функции, формирующие путь выполнения, считаются независимыми и обозначаются свободными булевыми переменными, и в формулу ошибки записывается условие достижимости текущей точки, пересеченное с определениями этих переменных. Тем самым SMT-решатель может выбрать любые значения этих переменных, совместимые с текущим условием. При этом за счет задания значений переменных, не связанных с найденным путем выполнения, могут быть получены как ложные, так и истинные срабатывания, что зависит от совместности пути, на котором была вычислена привнесенная информация об индексе, с найденным путем. Работа над детектором будет продолжаться в ближайшем будущем.

4.3.2. Разыменование нулевого указателя

Группа детекторов DEREF_AFTER_NULL.EX и DEREF_OF_NULL.EX являются версиями соответствующих детекторов раздела 4.2.1, использующих дополнительный атрибут, который отслеживает условия равенства нулю указателя (для одного детектора – сравнения с нулем, для другого – присваивания нулю). Детектор запускается, если атрибут PtrIsNull установлен в ложь (потому что иначе будет выдано предупреждение обычным вариантом детектора), проверяет на совместность собранное условие равенства нулю и условие текущей точки, а также дополнительно проверяет, что для данного указателя предупреждение еще не было выдано (это полезно для сокращения вызова SMT-решателя, особенно в операторах выбора). Для выдачи межпроцедурных предупреждений условия разыменования указателя в функции распространяются в вызывающую функцию, кроме указателей типа `this`, чтобы также сэкономить на вызовах решателя.

Детектор DEREF_OF_NULL.RET.STAT также предназначен для ситуаций проверки возвращаемого значения функций на ноль, как и другие DEREF_OF_NULL.RET детекторы, однако он использует статистический подход. Если в заданном количестве случаев (обычно 80%) результат функции проверяется, то для остальных случаев детектором также будет предложено проверить результат. Используется атрибут FuncCallAttr, хранящий условие того, что указатель был получен из вызова функции, при этом отсекаются случаи мертвого кода и функции выделения памяти (для них существует отдельный подтип детектора). В точке разыменования проверяется совместность текущего условия точки с условием, хранящимся в атрибуте, и в случае успеха разыменование засчитывается как непроверенное. Аналогично, проверка указателя засчитывает обращение к нему как проверенное в случае совместности условий. После обработки условий атрибут сбрасывается.

4.3.3. Другие детекторы Си/Си++ и Java

Детектор DIVISION_BY_ZERO выдает предупреждения о возможных делениях на ноль в случаях, когда переменной был присвоен ноль и найдется реализуемый путь выполнения, проходящий через деление на эту переменную, либо когда деление выполняется в вызываемой процедуре, а нулевое значение образуется в ходе вычислений, связанных с параметрами функции. Используется чувствительный к путям атрибут ZeroIfCond, который хранит условия равенства нулю класса значений, к которому он привязан. Атрибут распространяется через сравнения и арифметические операции способами, схожими с описанными в разделе 2.4. Дополнительно используется обычный двоичный атрибут DenominatorFlag, который равен истине в случае, когда класс значений выступал знаменателем, и его интервал значений содержал ноль, а также атрибуты, наличие которых позволяет предположить, что значение может быть равно нулю – это атрибут небезопасного значения MustTaintedValueInterval и атрибут нулевого указателя ptrIsSetToNull. Атрибуты проверяются при операциях деления и взятия остатка, а также в точке вызова функции при обработке аннотации.

Детектор FREE_NONHEAP_MEMORY.EX отслеживает ситуации, когда выполняется освобождение памяти, а переданный указатель не указывает на динамическую память. Интересно, что такая ошибка на первый взгляд не кажется сложной, однако для хорошего качества поиска необходим анализ третьего, чувствительного к путям уровня. Основными атрибутами детектора является атрибут IsNonHeap, хранящий условия, при которых указательный класс значения не указывает на кучу, и атрибут DeletePtrIf, который хранит условия освобождения памяти. Первый атрибут определяется в условие текущей точки программы в местах создания памяти на стеке и взятия адреса глобальных переменных, а также при адресной арифметике с ячейками памяти, у которых класс памяти не является динамическим. Второй атрибут инициализируется при вызове функций освобождения памяти.

Предупреждение выдается при обработке вызова функции в случае, если конъюнкция условий обоих атрибутов и условия достижимости текущей точки программы совместна. Кроме того, в аннотации функции дополнительно проверяется уже описанный MustFreeFlag (переменная обязательно освобождалась), и если вызываемой функции в качестве соответствующего параметра передается указатель на глобальную переменную, тогда также выдается предупреждение.

Наконец, группа детекторов UNINIT ищет использования неинициализированных переменных. По разным причинам эта ошибка является одной из самых интересных и сложных для поиска. Они в том, что необходимо очень точно оценивать влияние всех путей выполнения на определения переменной – единственная неточность в учете определений и их условий приведет к ложному срабатыванию. Кроме того, последствия такой ошибки могут быть самыми разными, вплоть до возможности эксплуатации программной системы. Даже при динамическом анализе, где, казалось бы, имеется полный контроль над поведением программы, можно легко получить ложные срабатывания или неправильно понять суть предупреждения анализатора.

Выделяется подтип детектора UNINIT.LOCAL_VAR, отслеживающий неинициализированные использования локальных переменных примитивных типов, UNINIT.STRUCT для полей структур и UNINIT.ARRAY для элементов массива. Каждый из них содержит отдельный атрибут, хранящий условия инициализации соответствующих объектов. Любая запись в переменную сбрасывает условия атрибутов, поскольку при этом меняется класс значений переменной. Аналогично происходит в случае передачи адреса отслеживаемого объекта в другую функцию. Предупреждение выдается в случае, если при использовании класса значений в атрибутах записано отличное от лжи условие инициализации, и отрицание этого условие совместно с текущим предикатом достижимости.

4.3.4. Детекторы C#

Основными чувствительными к путям детекторами движка C# являются детекторы разыменования нулевого указателя и утечек ресурсов. Как было сказано в разделе 2.4.2, идея построения детекторов схожа с чувствительными к путям детекторами на основе атрибутов классов значений, однако память моделируется напрямую (без КЗ) с помощью символьных выражений, содержащихся в ячейках памяти, а далее детекторы сохраняют условия интересующих их событий и проверяют на совместность, предварительно пересекая их с условием достижимости текущей точки.

Детектор разыменования нулевого указателя устроен подобно соответствующим детекторам основной части Svace, описанным в разделе 4.3.2. В ходе анализа отслеживаются

условия разыменования указателя и условия присваивания ему нуля либо сравнения с нулем. При обнаружении присваивания либо сравнения (или вызова функции) анализируются возможные условия разыменования для символьного выражения, соответствующего этому указателю, и проверяется совместность конъюнкции из этих условий и условия текущей точки. Аналогично, при встрече разыменования анализируются условия присваивания или сравнения с нулем. Все условия по окончанию работы функции сохраняются в аннотации, поэтому находятся межпроцедурные срабатывания, в которых получение указателем нуля и разыменования произвольно разнесены по графу вызовов (с учетом огрубления аннотаций).

Детектор утечек ресурсов ищет ситуации, когда теряется ссылка на некоторый объект класса, который реализует интерфейс IDisposable, при этом у объекта до потери ссылки не был вызван метод Dispose. Для символьных выражений, соответствующих ресурсам, вычисляется условие того, что ресурс не освобожден. Условие инициализируется текущим условием достижимости при создании ресурса, а вызов Dispose или сохранение ресурса в глобальной памяти считается признаком его освобождения (сейчас или в вызывающих методах). При обработке вызовов дополнительно сохраняется информация о том, освобождает ли вызывающая функция свои параметры. В точке выхода из функции проверяется, совместно ли условие наличия утечки для ресурсных переменных, и в положительном случае выдается предупреждение. Также отслеживается момент выхода переменной из области видимости или потери всех ссылок на ресурс в модели памяти, и условие ошибки проверяется и при наступлении этих ошибок. Это не расширяет множество выданных предупреждений, но делает их более понятными пользователю, потому что ошибка выдается так рано, как только возможно после утечки.

5. РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ КОЛЛЕКЦИИ АНАЛИЗАТОРОВ SVACE К ПРОМЫШЛЕННОМУ ИСХОДНОМУ КОДУ

В настоящей главе приведены экспериментальные результаты запусков анализатора Svace на больших объемах открытого исходного кода. Эксперименты делались для оценки масштабируемости и качества анализатора, а также его отдельных компонент (хранилища и подсистемы контролируемой сборки). Использовался релиз Svace 2.3.5 середины 2017 года. Объем исходного кода инструмента составляет около 440 тыс. строк исходного кода, из них примерно 220 тыс. написано на языке Java (основной движок анализа, хранилище данных, интерфейсы просмотра результатов), 100 тыс. – на языке C# (анализатор C#), остальной код написан на языках Си, Си++, Python, Bash.

Помимо результатов самого инструмента Svace, интерес представляют также результаты сравнения с другими промышленными анализаторами сопоставимого уровня (Coverity Prevent, Klocwork K11 и подобными). К сожалению, такие инструменты являются закрытыми, результаты их работы недоступны. Кроме того, насколько известно, лицензия на промышленные анализаторы кода прямо запрещает публикацию их результатов и обсуждение найденных ошибок, даже в том случае, когда ошибки были найдены в открытом исходном коде. В некоторых случаях в репозиториях открытого ПО можно найти сообщения об исправлениях кода, сделанных в связи с найденными статическим анализатором ошибками, однако речь идет о единичных случаях, из анализа которых затруднительно сделать выводы. Известные открытые инструменты не предоставляют все необходимые уровни анализа, как правило, ограничиваясь первым уровнем – алгоритмами обхода узлов АСД.

В работе [96] анализатор Svace сравнивался с доступной версией анализатора Saturn, реализующего межпроцедурные алгоритмы с чувствительностью к путям, на нескольких открытых программах размером в 100-200 тыс. строк кода. Даже для этих программ Saturn был в 5-10 раз медленнее Svace, при этом использовались только два детектора ошибок (разыменования нулевого указателя и ошибок многопоточности), тогда как в Svace использовались все детекторы (более 100), что указывает на недостаточную масштабируемость анализатора Saturn. В той же работе Svace также сравнивался со своей предыдущей версией (только в части языков Си/Си++ и кода ОС Anroid 4.4 – таблица Б.3), и было показано улучшение характеристик качества анализа (больше срабатываний при том же или лучшем уровне истинности).

5.1 Подсистема контролируемой сборки

Для оценки реализации контролируемой сборки были выполнены замеры производительности перехвата на операционных системах Windows и Linux, результаты

которых представлены в таблице 5.1. В ходе замеров были выключены все расширения, выполняющие обработку реакции на события, кроме ведения журналов сборки, так, чтобы можно было оценить затраты собственно на организацию перехвата.

Таблица 5.1. Производительность перехвата сборки.

Приложение	ОС / Вид перехвата	Время сборки (сек), последовательно		Время сборки (сек), параллельно	
		Без перехвата	С перехватом	Без перехвата	С перехватом
Php	Windows / debug	95	96 (+1%)	57	60 (+5.2%)
Z3	Windows / debug	473	488 (+3.1%)	209	224 (+6.2%)
Php	Linux / `LD_PRELOAD`	570	589 (+3.3%)	115	118 (+2.6%)
Z3	Linux / `LD_PRELOAD`	1358	1369 (+0.8%)	175	174 (-0.5%)
Php	Linux / `ptrace`	570	604 (+5.9%)	115	122 (+6.0%)
Z3	Linux / `ptrace`	1358	1379 (+1.5%)	175	175 (0.0%)

Были выбраны приложения среднего размера – дистрибутив языка Php и SMT-решатель Z3, которые собирались дважды – с последовательной и с параллельной сборкой (в 8 потоков на Windows и в 16 на Linux). В каждом случае проводилось по 5 запусков исходной сборки и сборки с включенным мониторингом, из которых выбрасывались два крайних по времени результата, а остальные усреднялись.

Можно заметить, что наименее затратным методом перехвата является использование механизма `LD_PRELOAD` на ОС Linux – он дает 1-3% замедления. Отладочные механизмы ОС медленнее – на Windows последовательная сборка дает те же 1-3% замедления, однако сборка в 8 потоков уже демонстрирует 5-6% замедления, на Linux диапазон замедления составляет от 1% до тех же 6%. Эти замедления являются приемлемыми и в ходе реального перехвата скрываются за счет затрат на запуск собственных компиляторов для построения внутреннего представления.

5.2 Время сборки/анализа проекта и объем потребляемой памяти

Производительность сборки анализируемого кода и последующего анализа замерялась отдельно на ОС Linux и Windows для программ на языках Си/Си++, Java, C#. В таблицах 5.2-5.4 представлены данные, полученные при тестировании сборки на ОС Linux. Таблица 5.2

содержит основные сведения о проектах (название, версия, количество строк кода и функций), таблица 5.3 – время, затраченное на сборку проектов и соответствующее замедление, таблица 5.4 – размер собранных артефактов сборки. Все запуски выполнялись на сервере с 32-ядерным процессором Intel Xeon E5-2650@ 2.00GHz и 264Гб ОЗУ под управлением ОС Ubuntu 14.04.

Таблица 5.2. Тестируемые проекты для ОС Linux.

Проект / Версия	Количество строк (Си), тыс.	Количество строк (Си++), тыс.	Количество строк (Java), тыс.	Количество строк (всего), тыс.	Количество функций
Android 5.0.2	3193	5363	4381	12938	1709515
Tizen 4.0	17473	9574	0	27047	3151343
Ant 1.9.3	0	0	130	130	17241
Azureus 4.3.0.6	0	0	466	466	41560
Linux 4.4.0	9898	0	0	9898	320042
Openssl 1.0.1	264	0	0	264	6920
Pulseaudio 4.0	216	0	0	216	5072
Qtiplot 0.9.8.9	60	233	0	293	15578
Xpdf 3.03	0	63	0	63	536

Как видно из таблицы 5.2, выбран набор средних проектов (300-500 тыс. строк исходного кода), как на Java, так и на Си/Си++. Из больших проектов взяты ядро ОС Linux (примерно 10 млн. строк кода на Си, 320 тыс. функций), исходный код ОС Android 5.0.2 (13 млн. строк кода, из которых 3.2 млн. строк на Си, 5.4 млн. строк на Си++ и 4.4 млн. строк на Java, 1.7 млн. функций), а также исходный код ОС Tizen 4.0 – самый большой проект, содержащий 27 млн. строк кода, из них 17.5 млн. строк на Си и 9.5 млн. на Си++, а также 3.1 млн. функций. Все данные по исходному коду получены утилитой SLOCCount [195], которая была применена к исходным файлам, компиляция которых была перехвачена системой контролируемой сборки (т.е. это только тот код, который компилировался).

Таблица 5.3. Время сборки проектов для ОС Linux.

Проект / Версия	Количество строк (всего), тыс.	Время сборки (без перехвата), сек	Время сборки (с перехватом), сек	Замедление, раз
Android 5.0.2	12938	47313	102447	2.17

Tizen 4.0	27047	N/A	N/A	N/A
Ant 1.9.3	130	215	312	1.45
Azureus 4.3.0.6	466	29	185	6.38
Linux 4.4.0	9898	952	2935	3.08
Openssl 1.0.1	264	348	844	2.43
Pulseaudio 4.0	216	226	352	1.56
Qtiplot 0.9.8.9	293	730	1529	2.09
Xpdf 3.03	63	10	18	1.80
Среднее замедление				2.62

Таблица 5.3 показывает, что в среднем при построении внутреннего представления замедление исходной сборки составляет 2.6 раза. Дополнительное время тратится на запуск собственного компилятора, который генерирует биткод LLVM или байткод Java, создает разметку в формате DXR, сохраняет все данные в объекте сборки. Для ОС Tizen замедление сборки не измерялось, т.к. система собиралась с помощью утилиты GBS, которая последовательно компилировала в chroot-окружении каждый из примерно 900 пакетов Tizen, инсталлируя необходимые для сборки RPM-пакеты и создавая локальный репозиторий из собранных пакетов. Основная часть времени сборки при этом тратится на работу с диском, и в этом окружении затраты на создание представления для анализа становятся актуальны только при сборке большого пакета (например, браузера efl-chromium).

Таблица 5.4. Объем артефактов сборки для ОС Linux.

Проект / Версия	Количество строк (всего), тыс.	Размер артефактов сборки, Мб / Доля от объекта сборки, %					
		Полный объект сборки	Биткод / байткод		DXR		Исходный код и пр.	
Android 5.0.2	12938	19457	16587	85%	1301	7%	1569	8%
Tizen 4.0	27047	45148	37167	82%	2645	6%	5336	12%
Ant 1.9.3	130	109	16	15%	15	14%	78	72%
Azureus 4.3.0.6	466	292	41	14%	39	13%	212	73%
Linux 4.4.0	9898	6352	2904	46%	839	13%	2609	41%
Openssl 1.0.1	264	106	46	43%	17	16%	43	41%
Pulseaudio 4.0	216	56	20	36%	17	30%	19	34%

Qtiplot 0.9.8.9	293	459	373	81%	32	7%	54	12%
Xpdf 3.03	63	10	2	20%	4	40%	4	40%
		Среднее:	50%		13%		36%	

Таблица 5.4 демонстрирует размер полученного объекта сборки по проектам, который включает в себя внутреннее представление (биткод или байткод), разметку DXR, сохраненный исходный код и прочие служебные данные (например, список используемых библиотек для языка Java, команды компиляции и т.п.). Можно заметить, что для проектов на Java байткод занимает всего около 15% объекта сборки, как из-за его компактности, так и из-за того, что основная доля хранилища тратится на библиотеки (особенно стандартную rt.jar) и сохранение исходного кода. Для проектов на Си биткод составляет 40-50% всех собранных данных, тогда как для больших проектов со значительной долей Си++ доля биткода вырастает выше 80%, что является показателем большого объема отладочной информации, которую необходимо сохранять для Си++-кода для дальнейшего использования в анализаторе. Эта проблема знакома всем программистам на Си++.

В будущем планируется перейти к собственному формату представления отладочной информации, который позволит в рамках анализируемого проекта сохранять только одну копию данных для каждого типа, включая инстанциации шаблонов. Для этого в систему контролируемой сборки необходимо ввести некоторый диспетчер, обладающий знанием обо всем проекте, который может принять решение о необходимости сохранения данных или их избыточности. В настоящий момент все перехваченные компиляции обрабатываются отдельно, и модуль записи данных в объект сборки сохраняет все переданные ему артефакты, не занимаясь никаким интеллектуальным удалением избыточности.

Таблица 5.5. Время анализа проектов на ОС Linux.

Проект / Версия	Коли-чество строк, тыс.	Время анализа, сек				Скорость основного анализа, строк в сек.	Пиковая память, Мб	Выдано преду-преждений
		Основной анализ	Анализ в CSA	Анализ FindBugs	Анализ в JavaC			
Android 5.0.2	12938	20347	4459	1193	486	636	214095	71453
Tizen 4.0	27047	60157	11163	0	0	450	225739	86309
Ant 1.9.3	130	117	0	74	13	1112	23888	1492

Azureus 4.3.0.6	466	281	0	209	50	1659	49958	5382
Linux 4.4.0	9898	5468	2503	0	0	1810	108279	47380
Openssl 1.0.1	264	219	77	0	0	1206	46020	484
Pulseaudio 4.0	216	86	30	0	0	2507	26563	520
Qtiplot 0.9.8.9	293	317	102	0	0	925	53917	572
Xpdf 3.03	63	18	6	0	0	3583	4426	34

Таблица 5.5 показывает время и скорость основной части анализа на ОС Linux. Наиболее интересны данные по скорости анализа для больших проектов (Android и Tizen) , в которых она составляет около 500 строк кода в секунду. Такая скорость позволяет проанализировать 1.8 млн. строк кода в час или 9 млн. за 5 часов. Анализ ОС Android занимает около 5.5 часов, анализ ОС Tizen 4 – около 16 часов. Нужно отметить, что анализировались все пакеты ОС Tizen из профиля Tizen Unified, т.е. в реальной конфигурации системы (для мобильного, носимого устройства или для интернета вещей) будет использована только часть этих пакетов.

В таблицах 5.6-5.7 приведены аналогичные данные времени сборки и анализа для нескольких проектов на ОС Windows. Выбраны уже известные проекты Php и Z3, представляющие из себя проекты средних размеров на Си и Си++ соответственно, а также два проекта на языке Java, которые также тестировались на ОС Linux. Эксперименты выполнялись на машине с Intel Core i7-6700 3.4 GHz и 32 Гб памяти под управлением Windows 10 Professional. Как видим, полученные данные в целом подтверждают собранную на Linux статистику. Замедление сборки является более значительным (4-5 раз), на проекте Azureus оно достигает 9 раз – этот же проект демонстрировал самое большое замедление и на Linux. Распределение объема артефактов сборки также аналогично – доля внутреннего представения в общем объеме объекта сборки растет от Java (10-15%) к Си++ (90%). Скорость анализа существенно падает на Z3, в котором, по-видимому, широко используются возможности современного языка Си++. Кроме того, Z3 является единственным случаем, в котором АСД-анализы, реализованные в инструменте CSA, работают медленнее основного анализатора. Большое количество предупреждений для Ant и Azureus связано с инструментом FindBugs.

Таблица 5.6. Время сборки проектов и объем артефактов для ОС Windows.

Проект / Версия	Коли-чество	Время сборки	Время сборки	Замед-ление,	Размер артефактов сборки, Мб / Доля от объекта сборки, %

	строк, тыс.	(без перехвата), сек	(с перехватом), сек	раз	Полный объект сборки	Биткод / байткод		DXR		Исходный код и пр.	
PHP 7.1.6	244	56	314	5.61	66.6	25.9	39%	29.7	45%	11.0	16%
Z3 4.4.2	336	203	962	4.74	880.5	794.8	90%	49.9	6%	35.8	4%
ant 1.9.3	129	30	136	4.53	89.4	9.2	10%	14.7	16%	65.6	73%
azureus 4.3.0.6	379	31	297	9.58	117.7	16.2	14%	30.5	26%	71.0	60%
Среднее				6.1		38%		23%		39%	

Таблица 5.7. Время анализа проектов на ОС Windows.

Проект / Версия	Количество строк, тыс.	Время анализа, сек				Скорость основного анализа, строк в сек.	Пиковая память, Мб	Выдано предупреждений
		Основной анализ	Анализ в CSA	Анализ FindBugs	Анализ в JavaC			
PHP 7.1.6	244	248	107	0	0	984	10426	586
Z3 4.4.2	336	1530	1983	0	0	220	17275	1211
Ant 1.9.3	129	111	0	71	18	1162	8938	1423
Azureus 4.3.0.6	379	271	0	38	103	1399	9674	4108

Таблица 5.8 посвящена результатам проверки времени сборки и анализа для языка C#. Эти данные приводятся отдельно, поскольку основной движок анализа для C# отличен от остальных языков и проводит анализы всех уровней. Поэтому скорость анализа здесь меньше, так как учитываются все уровни анализа (не только самые глубокие, но и уровень АСД). Замедление при сборке здесь также включает подсчет метрик по исходному коду. В целом характеристики анализатора C# аналогичны анализаторам остальных языков из таблиц 5.2-5.5.

Таблица 5.8. Время сборки и анализа проектов на языке C#.

Проект	Количество строк,	Время сборки (без пере-	Время сборки (с пере-	Замедление, раз	Время анализа, сек	Скорость анализа, строк в	Пиковая память, Мб	Предупреждения

	тыс.	хвата), сек	хватом), сек			сек		
Roslyn	2500	270	816	3.02	8424	297	12285	1230
Corefx	3200	180	556	3.08	6363	503	10636	1723
Lucenet	530	65	129	1.98	2257	235	7450	884
CSParser	60	1	12	12	45	1333	551	20

Кроме того, было проведено тестирование инкрементального анализа на ОС Linux для проекта ОС Android 5.0.2. Был выполнен полный анализ исходного кода ОС Android с сохранением аннотаций на диск, чтобы можно было организовать сервер инкрементального анализа. Сохраненные аннотации составили 42 Гб (для Си, Си++ и Java, т.к. анализировалась вся система). Далее для инкрементального анализа было выбрано приложение Camera (каталог *packages/apps/Gallery/src/com/android/camera* в исходном коде Android) размером 8728 строк кода, анализ которого в инкрементальном режиме занял 3 минуты 40 секунд (из этого времени 1 минуту 50 секунд занимает слияние статистических данных полного и инкрементального анализа для корректной работы статистических детекторов). Получены 62 предупреждения, которые совпали с теми, что были выданы при полном анализе.

5.3 Качество анализа

Была проведена оценка выданных предупреждений на исходном коде ОС Android и Tizen отдельно для языков Java и Си/Си++, а также для приложений на языке C#. Среди предупреждений делалась случайная выборка, которая просматривалась разработчиками на предмет истинности или ложности. Истинные с точки зрения анализатора предупреждения, которые могут являться некритическими, также засчитывались. Результаты представлены в таблицах 5.9-5.12.

Можно видеть, что процент истинных срабатываний для критических детекторов колеблется между 50% и 80%. В среднем по всем детекторам для Android 5 истинные срабатывания составляют около 70% для Си и Си++ и около 80% для Java. Эта же картина обычно наблюдается и в остальных проектах – для языка Java из-за отсутствия указателей и адресной арифметики, как правило, проще строить модель памяти (на первый план скорее выходит девиртуализация и учет исключений в потоке управления, как описано в разделе 3.3).

Интересно также, что для ОС Tizen доля истинных срабатываний ниже, чем для ОС Android (таблица 5.11). Частично это объясняется тем, что анализатор Svace используется в

компании Samsung для проверки исходного кода Tizen уже в течение почти двух лет, и многие истинные ошибки уже исправлены (для кода, контролируемого компанией; ошибки во внешнем коде не всегда исправляются, что связано со сложностями поддержки изменений для сотен пакетов дистрибутива). Так, можно заметить, что для детектора STATIC_OVERFLOW вообще не было найдено истинных срабатываний из 17 просмотренных, что не является типичной картиной.

Таблица 5.9. Результаты анализа для ОС Android 5.0.2 (языки Си и Си++).

Детектор	Всего	Просмотренных	Истинных	Доля
CONFUSING_INDENTATION	17	17	17	100.00%
MEMSET_SIZE_MISMATCH	14	14	14	100.00%
USE_AFTER_FREE.REALLOC	4	4	4	100.00%
NO_EFFECT	55	48	44	91.67%
HEAP_INCOMPATIBLE.ARRAY	27	21	21	100.00%
HEAP_INCOMPATIBLE.FREE	7	5	5	100.00%
PASSED_TO_PROC_AFTER_FREE.EX	25	22	17	77.27%
DEREF_OF_NULL	19	17	12	70.59%
ALLOC_SIZE_MISMATCH	9	9	5	55.56%
MEMORY_LEAK.STRDUP	17	9	9	100.00%
FREE_NONHEAP_MEMORY.EX	4	4	2	50.00%
DOUBLE_FREE.EX	39	35	18	51.43%
OVERFLOW_UNDER_CHECK	145	98	59	60.20%
USE_AFTER_FREE	8	6	3	50.00%
DIVISION_BY_ZERO	54	27	20	74.07%
UNINIT.LOCAL_VAR	144	61	47	77.05%
NONTERMINATED_STRING.READ	37	20	11	55.00%
BUFFER_OVERFLOW.EX	213	100	62	62.00%
OVERFLOW_AFTER_CHECK.EX	37	27	10	37.04%
TAINTED_PTR	52	15	14	93.33%
DIVISION_BY_ZERO.EX	198	62	52	83.87%
STATIC_OVERFLOW	21	21	5	23.81%
DEREF_AFTER_FREE	43	18	10	55.56%
DEREF_AFTER_NULL	104	35	24	68.57%
MEMORY_LEAK	84	28	19	67.86%
OVERFLOW_AFTER_CHECK	46	30	10	33.33%
TAINTED_INT	77	20	15	75.00%
DEREF_OF_NULL.CONST	99	29	19	65.52%
HANDLE_LEAK.EX	89	22	17	77.27%
UNREACHABLE_CODE.SWITCH	166	20	20	100.00%

BAD_COPY_PASTE	142	33	17	51.52%
DEREF_AFTER_NULL.EX	695	128	78	60.94%
NONTERMINATED_STRING	223	35	25	71.43%
TAINTED_INT.LOOP	47	6	5	83.33%
DEREF_OF_NULL.EX	268	46	28	60.87%
NULL_AFTER_DEREF	184	23	17	73.91%
BUFFER_OVERFLOW.LIB.EX	45	7	4	57.14%
HANDLE_LEAK	152	19	12	63.16%
CHECK_AFTER_OVERFLOW	27	3	2	66.67%
MEMORY_LEAK.EX	284	46	21	45.65%
UNREACHABLE_CODE.NO_PATH	135	10	9	90.00%
UNREACHABLE_CODE	1153	85	75	88.24%
UNINIT.ARRAY	159	26	9	34.62%
DEREF_OF_NULL.RET.STAT	486	30	25	83.33%
UNINIT.CTOR	1710	20	13	65.00%
UNINIT.STRUCT	141	2	0	0.00%
Всего	**7705**	**1363**	**925**	**68.52%**

Таблица 5.10. Результаты анализа для ОС Android 5.0.2 (язык Java).

Детектор	Всего	Просмотренных	Истинных	Доля
CONFUSING_INDENTATION	8	8	8	100.00%
BAD_COPY_PASTE	41	41	37	90.24%
DEREF_OF_NULL	9	9	8	88.89%
DYNAMIC_OVERFLOW	8	8	5	62.50%
INVARIANT_RESULT	86	36	36	100.00%
DIVISION_BY_ZERO.EX	33	23	12	52.17%
DEREF_AFTER_NULL	138	51	46	90.20%
UNREACHABLE_CODE.SWITCH	11	2	2	100.00%
NULL_AFTER_DEREF	278	42	36	85.71%
DEREF_OF_NULL.RET.STAT	209	23	17	73.91%
UNREACHABLE_CODE	281	32	21	65.63%
HANDLE_LEAK.EX	361	36	25	69.44%
DEREF_OF_NULL.EX	259	25	17	68.00%
DEREF_AFTER_NULL.EX	310	25	18	72.00%
USE_AFTER_RELEASE	294	23	17	73.91%
HANDLE_LEAK	1321	32	26	81.25%
HANDLE_LEAK.EXCEPTION	1080	15	13	86.67%
Всего	**4729**	**433**	**344**	**79.45%**

Таблица 5.11. Результаты анализа для ОС Tizen 4 (языки Си и Си++).

Детектор	Всего	Просмотренных	Истинных	Доля
FREE_NONHEAP_MEMORY.EX	18	18	11	61.11%
ALLOC_SIZE_MISMATCH	20	20	11	55.00%
CONFUSING_INDENTATION	77	23	21	91.30%
MEMSET_SIZE_MISMATCH	57	16	9	56.25%
DEREF_OF_NULL	57	6	5	83.33%
USE_AFTER_FREE	50	4	4	100.00%
HEAP_INCOMPATIBLE.ARRAY	14	3	1	33.33%
OVERFLOW_AFTER_CHECK.EX	138	23	9	39.13%
HEAP_INCOMPATIBLE.FREE	78	20	5	25.00%
USE_AFTER_FREE.REALLOC	18	4	1	25.00%
NO_EFFECT	401	23	20	86.96%
OVERFLOW_AFTER_CHECK	67	5	3	60.00%
TAINTED_INT	275	11	11	100.00%
PASSED_TO_PROC_AFTER_FREE.EX	135	9	5	55.56%
DIVISION_BY_ZERO.EX	380	21	13	61.90%
NONTERMINATED_STRING	628	23	19	82.61%
OVERFLOW_UNDER_CHECK	374	15	10	66.67%
DEREF_AFTER_FREE	198	15	5	33.33%
DEREF_OF_NULL.RET.STAT	932	23	22	95.65%
BUFFER_OVERFLOW.EX	351	23	8	34.78%
TAINTED_INT.LOOP	221	7	5	71.43%
DOUBLE_FREE.EX	258	14	5	35.71%
TAINTED_PTR	339	5	5	100.00%
DEREF_OF_NULL.EX	861	23	12	52.17%
NULL_AFTER_DEREF	1093	20	15	75.00%
DEREF_AFTER_NULL.EX	1373	23	18	78.26%
HANDLE_LEAK	462	10	6	60.00%
BAD_COPY_PASTE	440	23	5	21.74%
MEMORY_LEAK	354	7	4	57.14%
DEREF_OF_NULL.CONST	406	17	4	23.53%
DEREF_AFTER_NULL	328	4	3	75.00%
UNINIT.LOCAL_VAR	481	4	4	100.00%
DIVISION_BY_ZERO	141	2	1	50.00%
MEMORY_LEAK.STRDUP	162	6	1	16.67%
UNREACHABLE_CODE	3181	21	18	85.71%
UNREACHABLE_CODE.SWITCH	363	2	2	100.00%
UNINIT.CTOR	2710	19	12	63.16%
HANDLE_LEAK.EX	246	2	1	50.00%
MEMORY_LEAK.EX	1913	21	5	23.81%

CHECK_AFTER_OVERFLOW	71	3	0	0.00%
STATIC_OVERFLOW	107	23	0	0.00%
Всего	**19778**	**561**	**319**	**56.86%**

Таблица 5.12 содержит экспериментальные результаты, полученные при анализе исходного кода на языке C#. Детекторы уровня АСД обычно имеют уровень истинных срабатываний, близкий к 100%, за исключением детекторов, построенных на эвристиках (примерами являются INFINITE_LOOP и BAD_COPY_PASTE). Чувствительные к путям детекторы демонстрируют точность в 50-80% (несколько хуже для утечек ресурсов и лучше для разыменования нулевого указателя из-за более простой модели памяти в C#).

Таблица 5.12. Результаты анализа программ на языке C#.

Детектор	Всего	Просмотренных	Истинных	Доля
UNREACHABLE_CODE	455	404	240	59.41%
CONFUSING_INDENTATION	3	2	2	100.00%
NRE	77	42	35	83.33%
INFINITE_LOOP	7	7	3	42.86%
INVARIANT_RESULT	762	45	44	97.78%
HANDLE_LEAK	534	41	20	48.78%
NRE.ARGUMENT	304	31	20	64.52%
BAD_COPY_PASTE	110	21	7	33.33%
NRE.NULL_AFTER_DEREF	65	28	23	82.14%
NRE.DEREF_AFTER_NULL	172	42	36	85.71%
INCOMPLETE_SWITCH	98	48	36	75.00%
NRE.DEREF_AFTER_AS	172	30	29	96.67%
ITERATED_ONCE	7	7	7	100.00%
			Среднее	**74.58%**

ЗАКЛЮЧЕНИЕ

Данная работа посвящена разработке методологии выполнения статического анализа для поиска дефектов, состоящей из разработки и реализации методов статического анализа исходного кода программ для поиска ошибок в программах, детекторов ошибок на основе разработанных методов. Кроме того, целью работы являлась разработка и реализация соответствующих программных средств, обеспечивающих совместную работу этих методов для сверхбольших программ на популярных языках программирования. В результате проведенных исследований получены следующие результаты:

1. Разработана методология проведения статического анализа исходного кода программ для поиска ошибок в программах, заключающаяся в проведении многоуровневого статического анализа с помощью набора моделей программы и методов анализа с общей моделью памяти на уровнях анализа АСД, внутрипроцедурного анализа, межпроцедурного контекстно-чувствительного анализа, чувствительного к путям выполнения анализа. Предложенные модели и алгоритмы математически обоснованы, имеют линейную масштабируемость и пригодны для популярных императивных языков программирования, а также позволяют переиспользовать вычисленную информацию с предыдущих уровней анализа на следующих уровнях;
2. Разработаны алгоритмы поиска конкретных ошибок в программе (детекторы) на основе предложенных методов, которые выполняют поиск популярных классов ошибок: ошибок кодирования, неверного использования стандартных интерфейсов, критических ошибок (разыменование нулевого указателя, переполнение буфера, ошибки управления памятью и ресурсами, использование неинициализированных переменных, ошибки многопоточных примитивов, недостижимый код и др.). Детекторы позволяют искать заданный тип ошибки на разных уровнях анализа и не выдавать ошибку на последующих уровнях, если она уже была найдена на предыдущих;
3. Разработана архитектура программной системы, обеспечивающая автоматическую работу всех предложенных методов на протяжении всего процесса анализа, а также управление набором анализаторов для различных языков и единообразный показ их результатов. Разработанные компоненты системы анализа включают: автоматическое построение внутренних представлений для анализа на основе прозрачной для пользователя контролируемой сборки; единое переносимое хранилище собранной для анализа информации и результатов анализа, обеспечивающее запуск анализа на любой машине; подсистема просмотра и разметки результатов анализа, которая обеспечивает перенос выполненной пользователем разметки между результатами анализа программы; инкрементальный анализ только лишь изменившейся части программы;

4. Выполнена реализация разработанных моделей программы, алгоритмов анализа, детекторов и инфраструктуры анализа в коллекции анализаторов Svace для языков Си, Си++, Java и C#, демонстрирующая масштабируемость анализа до десятков миллионов строк кода и приемлемое количество истинных срабатываний (60-80% и выше).

В дальнейших работах по совершенствованию предложенных методов анализа и разработанного инструмента анализа можно выделить следующие направления:

- Улучшение предложенных моделей и алгоритмов анализа: уточнение модели программы и модели памяти привлечением исследований по сепарационной логике, использование работ по верификации, в частности, приемов "ангельской" верификации для фильтрации ложных срабатываний в чувствительных к путям детекторах, разработка методов анализа программ на нескольких языках;
- Сокращение ручных затрат программиста на работу с анализатором: привлечение методов ранжирования выдачи предупреждений на основе машинного обучения, использование алгоритмов генерации исправлений ошибок, выданных анализом;
- Разработка или интеграция анализаторов других языков (Go, Python, Rust).

БЛАГОДАРНОСТИ

Я хочу выразить самую горячую благодарность Сергею Суреновичу Гайсаряну, научившему меня всему, что я умею в компиляторах, Арутюну Ишхановичу Аветисяну, показавшему мне, что со всем этим делать, и Виктору Петровичу Иванникову, который сделал институт местом, где радостно работать и всегда можно найти единомышленников. Я глубоко благодарю всех своих коллег по отделу компиляторных технологий и группе статического анализа, потому что наш общий успех в статическом анализе был достигнут только благодаря им всем. И самое главное, я благодарю свою семью, без которой ни я, ни эта работа не смогли бы состояться.

ЛИТЕРАТУРА

Статьи автора в журналах, рекомендованных ВАК РФ

1. 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.
2. V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin, A. I. Avetisyan. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, 2014, vol. 40, issue 5, pp. 265-275.
3. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды ИСП РАН. 2014. Т. 26, выпуск 1. Стр. 231-250.
4. А.Аветисян, А.Белеванцев, Алексей Бородин, В.Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды ИСП РАН, т.21, 2011. Стр. 23-38.
5. V. K. Koshelev, V. N. Ignat'ev, A. I. Borzilov, and A. A. Belevantsev. SharpChecker Static Analysis Tool for C# Programs. Programming and Computer Software, 2017, Vol. 43, No. 4, pp. 268–276.
6. И.А. Дудина, А.А. Белеванцев. Применение статического символьного выполнения для поиска ошибок доступа к буферу. Программирование, 2017, № 5, стр. 3-17.
7. А.А. Белеванцев, А.О. Избышев, Д.М. Журихин. Организация контролируемой сборки в статическом анализаторе Svace. Системный администратор, выпуск 6-7 (176-177), 2017, стр. 135-139.
8. А.П. Меркулов, С.А. Поляков, А.А. Белеванцев. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74. DOI: 10.15514/ISPRAS-2017-29(3)-5
9. А. А. Белеванцев. Многоуровневый статический анализ исходного кода программ для обеспечения качества программ. Программирование, 2017, Том 43, №6, стр. 3-26.
10. Беляев М.В., Шимчик Н.В., Игнатьев В.Н., Белеванцев А.А. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99-116. DOI: 10.15514/ISPRAS-2017-29(3)-7

Другие публикации автора по теме диссертации (статьи, материалы конференций), свидетельства о регистрации программ

11. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Велесевич Е.А. «Инфраструктура анализа потоков данных инструмента статического анализа «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017610519 от 12.01.2017.
12. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Велесевич Е.А. «Набор детекторов ошибок в программах на языке C# инструмента статического анализа «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017610524 от 12.01.2017.
13. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Велесевич Е.А. «Инфраструктура чувствительного к контексту вызова, потоку и путям исполнения анализа инструмента «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017610526 от 12.01.2017.
14. Игнатьев В.Н., Чукляев И.И., Белеванцев А.А. «Инструмент статического анализа «RuleChecker» для языков C и C++». Свидетельство о государственной регистрации программы для ЭВМ № 2016611555 от 04.02.2016.
15. Игнатьев В.Н., Чукляев И.И., Белеванцев А.А. «Проверочные модули инструмента статического анализа «RuleChecker» для языков C и C++». Свидетельство о государственной регистрации программы для ЭВМ № 2016611504 от 03.02.2016.
16. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Шимчик Н.В., Беляев М.В. «Расширение Microsoft Visual Studio 2015 для интеграции с инструментом статического анализа «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017660039 от 13.09.2017.
17. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Шимчик Н.В., Беляев М.В. «Детектор недостижимого кода в программах на языке C# инструмента статического анализа «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017660156 от 18.09.2017.
18. Игнатьев В.Н., Кошелев В.К., Борзилов А.И., Белеванцев А.А., Шимчик Н.В., Беляев М.В. «Инфраструктура анализа помеченных данных инструмента статического анализа «SharpChecker». Свидетельство о государственной регистрации программы для ЭВМ № 2017660157 от 18.09.2017.
19. Белеванцев А.А. «Инструмент преобразования Java-библиотек ОС Android формата Jack в формат JAR «Llij». Свидетельство о государственной регистрации программы для ЭВМ № 2017660048 от 13.09.2017.

20. Аветисян А.И., Белеванцев А.А., Чукляев И.И. Технологии статического и динамического анализа уязвимостей программного обеспечения. Вопросы кибербезопасности. №3 (4) июль-сентябрь 2014 г., стр. 20-28.
21. А. А. Белеванцев, И.А. Дудина. К вопросу о преодолении ограничений статического анализа при поиске дефектов переполнения буфера. Ломоносовские чтения, 2017.

Цитируемая литература

22. Маликов Олег Рустэмович. Исследование и разработка методики автоматического обнаружения уязвимостей в исходном коде программ на языке Си: дис. ... канд. физ.-мат. наук: 05.13.11, Москва, 2006.
23. Аветисян А. И. Современные методы статического и динамического анализа программ для автоматизации процессов повышения качества программногообеспечения: Диссертация на соискание звания доктора физико-математических наук // ИСП РАН. 2012.
24. S C Johnson. 1978. Lint, a C Program Checker. *Comp. Sci. Tech. Rep* (1978), 78–1273.
25. Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, and Anders Møller. 2015. In defense of soundiness. *Commun. ACM* 58, 2 (2015), 44–46. DOI:https://doi.org/10.1145/2644805
26. Matthias Endler's Awesome Static Analysis List. https://github.com/mre/awesome-static-analysis. Дата обращения 29.07.2017
27. NIST Source Code Security Analyzers. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html. Дата обращения 29.07.2017
28. List of tools for static analysis. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis. Дата обращения 29.07.2017
29. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. Static analyzer for finding dynamic programming errors. *Softw. - Pract. Exp.* 30, 7 (2000), 775–802. DOI:https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H
30. Yichen Xie and Alex Aiken. 2005. Saturn: A Scalable error detection using boolean satisfiability. *ACM SIGPLAN Not.* 40, 1 (2005), 351–363. DOI:https://doi.org/10.1145/1047659.1040334
31. Статический анализатор CheckMarx CxSAST. https://www.checkmarx.com. Дата обращения 29.07.2017

32. Анализатор RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/downloads. Дата обращения 29.07.2017
33. Анализатор Splint. http://splint.org. Дата обращения 29.07.2017
34. Анализатор Flexelint. http://www.gimpel.com/html/flex.htm. Дата обращения 29.07.2017
35. 30 лет анализатору PC-lint. http://blog.gimpel.com/2015/05/celebrating-30-years-of-pc-lint.html. Дата обращения 29.07.2017
36. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
37. Компилятор Clang. http://clang.llvm.org. Дата обращения 29.07.2017
38. Анализатор Clang Static Analyzer. http://clang-analyzer.llvm.org. Дата обращения 29.07.2017
39. Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A memory model for static analysis of C programs. *4th Int. Symp. Leveraging Appl. (ISoLA 2010)* (2010), 535–548. DOI:https://doi.org/10.1007/978-3-642-16558-0_44
40. Инструмент FindBugs. http://findbugs.sourceforge.net. Дата обращения 29.07.2017
41. N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (2008), 22–29. DOI:https://doi.org/10.1109/MS.2008.130
42. Инструмент SonarLint. http://www.sonarlint.org/visualstudio/rules/index.html. Дата обращения 29.07.2017
43. Платформа Roslyn. https://github.com/dotnet/roslyn. Дата обращения 29.07.2017
44. Анализ кода в инструменте Resharper. https://www.jetbrains.com/resharper/features/code_analysis.html. Дата обращения 29.07.2017
45. Инструмент PVS-Studio. https://www.viva64.com/ru/pvs-studio. Дата обращения 29.07.2017
46. Андрей Фадин, Сергей Борзых, Павел Гусев. Appchecker – инструмент статического анализа. Control Engineering Россия, номер 2 (68), стр. 26-29, 2017.
47. Библиотека ASM чтения и манипуляции байт-кода Java. http://asm.ow2.org. Дата обращения 29.07.2017
48. Анализатор CppCheck. http://cppcheck.sourceforge.net. Дата обращения 29.07.2017
49. Дергачёв А.В., Сидорин А.В. Основанный на резюме метод реализации произвольных контекстно-чувствительных проверок при анализе исходного кода посредством символьного выполнения. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 41-62. DOI: 10.15514/ISPRAS-2016-28(1)-3
50. Игнатьев В.Н. Использование статического анализа для проверки настраиваемых ограничений языков программирования C и C++. Диссертация на соискание звания кандидата физико-математических наук // ИСП РАН. 2015.

51. Aoun Raza, Gunther Vogel, and Erhard Plödereder. 2006. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. *Reliab. Softw. Technol. Ada Eur. 2006* (2006), 71. DOI:https://doi.org/10.1.1.83.9583
52. Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In Proceedings of the 26th International Conference on Software Engineering (ICSE '04). IEEE Computer Society, Washington, DC, USA, 625-634.
53. М.В. Зубов, А.Н. Пустыгин, Е. В. Старцев. Применение универсальных промежуточных представлений для статического анализа исходного программного кода. Доклады ТУСУРа, № 1 (27), март 2013, 64–68.
54. Philip Mayer and Andreas Schroeder. 2014. Automated multi-language artifact binding and rename refactoring between Java and DSLs used by Java frameworks. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 8586 LNCS, (2014), 437–462. DOI:https://doi.org/10.1007/978-3-662-44202-9_18
55. Philip Mayer and Andreas Schroeder. 2013. Patterns of cross-language linking in java frameworks. *IEEE Int. Conf. Progr. Compr.* (2013), 113–122. DOI:https://doi.org/10.1109/ICPC.2013.6613839
56. Philip Mayer and Andreas Schroeder. 2012. Cross-language code analysis and refactoring. *Proc. - 2012 IEEE 12th Int. Work. Conf. Source Code Anal. Manip. SCAM 2012* (2012), 94–103. DOI:https://doi.org/10.1109/SCAM.2012.11
57. Rolf-Helge Pfeiffer and Andrzej Wasowski. 2012. TexMo: A Multi-language Development Environment. In *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle and Dimitris Kolovos (eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 178–193. DOI:https://doi.org/10.1007/978-3-642-31491-9_15
58. Dennis Strein, Hans Kratz, and Weif Lowe. 2006. Cross-language program analysis and refactoring. *Proc. - Sixth IEEE Int. Work. Source Code Anal. Manip. SCAM 2006* (2006), 207–216. DOI:https://doi.org/10.1109/SCAM.2006.10
59. Daniel Quinlan and Thomas Panas. 2009. Source code and binary analysis of software defects. *Proc. 5th Annu. Work. Cyber Secur. Inf. Intell. Res. Cyber Secur. Inf. Intell. Challenges Strateg. - CSIIRW '09* (2009), 1. DOI:https://doi.org/10.1145/1558607.1558653
60. Компиляторная инфраструктура Rose. http://rosecompiler.org. Дата обращения 29.07.2017
61. Н.Л. Луговской, С.В. Сыромятников. Применение языка KAST для преобразования исходного кода и автоматического исправления дефектов. Труды Института системного программирования РАН, том 25, 2013, стр. 51-66.

62. С.В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST. Труды Института системного программирования РАН, том 20, 2011, стр. 51-68.
63. Gerard J. Holzmann. 2017. Cobra: a light-weight tool for static and dynamic program analysis. *Innov. Syst. Softw. Eng.* 13, 1 (2017), 35–49. DOI:https://doi.org/10.1007/s11334-016-0282-x
64. Stan Jarzabek. 1998. Design of flexible static program analyzers with PQL. *IEEE Trans. Softw. Eng.* 24, 3 (1998), 197–215. DOI:https://doi.org/10.1109/32.667879
65. Инструмент PMD. https://pmd.github.io. Дата обращения 29.07.2017
66. Список языков запросов свойств программ. http://cs.nyu.edu/~lharris/content/programquerylangs.html. Дата обращения 29.07.2017
67. Язык запросов к АСД системы PuppetDB. https://docs.puppet.com/puppetdb/latest/api/query/v4/ast.html. Дата обращения 29.07.2017
68. Язык запросов ASTq. https://github.com/rse/astq. Дата обращения 29.07.2017
69. Обход АСД Babylon. https://github.com/pugjs/babylon-walk. Дата обращения 29.07.2017
70. Обходы АСД в Acorn. https://github.com/ternjs/acorn#distwalkjs. Дата обращения 29.07.2017
71. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. // Program Flow Analysis: Theory and Applications, chapter 7. Prentice-Hall, 1981.
72. Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '95). ACM, New York, NY, USA, 49-61. DOI=http://dx.doi.org/10.1145/199448.199462
73. Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A system and language for building system-specific, static analyses. *ACM SIGPLAN Not.* 37, 5 (2002), 69. DOI:https://doi.org/10.1145/543552.512539
74. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. Static analyzer for finding dynamic programming errors. *Softw. - Pract. Exp.* 30, 7 (2000), 775–802. DOI:https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H
75. Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. *PLDI '11 Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implement.* (2011), 567–577. DOI:https://doi.org/10.1145/2345156.1993565
76. C. Calcagno, D. Distefano, Pw. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. Acm* 58, 6 (2011), 26:1–26:66. DOI:https://doi.org/10.1145/2049697.2049700

77. Sumit Gulwani and Ashish Tiwari. 2007. Computing procedure summaries for interprocedural analysis. In *Proceedings of the 16th European Symposium on Programming* (ESOP'07), Rocco De Nicola (Ed.). Springer-Verlag, Berlin, Heidelberg, 253-267.

78. Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14). ACM, New York, NY, USA, 249-258. DOI=http://dx.doi.org/10.1145/2594291.2594328

79. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global Sparse Analysis Framework. *ACM Trans. Program. Lang. Syst.* 36, 3, Article 8 (September 2014), 44 pages. DOI: http://dx.doi.org/10.1145/2590811

80. Кошелев В.К., Избышев А.О, Дудина И.А. Межпроцедурный анализ помеченных данных на базе инфраструктуры LLVM. Труды Института системного программирования РАН, том 26, вып. 2, 2014, стр. 97-118.

81. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259-269. DOI=http://dx.doi.org/10.1145/2666356.2594299

82. Alan Mycroft. 1993. Completeness and predicate-based abstract interpretation. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (PEPM '93). ACM, New York, NY, USA, 179-185. DOI=http://dx.doi.org/10.1145/154630.154648

83. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (PLDI '01). ACM, New York, NY, USA, 203-213. DOI=http://dx.doi.org/10.1145/378795.378846

84. Cristian Cadar and Koushik Sen. 2013. Symbolic execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82. DOI:https://doi.org/10.1145/2408776.2408795

85. Библиография исследований по символьному выполнению. https://github.com/saswatanand/symexbib. Дата обращения 29.07.2017

86. Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. A Survey of Symbolic Execution Techniques. *arXiv* i (2016), 1–39. Retrieved from http://arxiv.org/abs/1610.00502

87. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proc. 8th USENIX Conf. Oper. Syst. Des. Implement.* (2008), 209–224. DOI:https://doi.org/10.1.1.142.9494

88. Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. *Jt. Meet. Found. Softw. Eng.* (2015), 842–853. DOI:https://doi.org/10.1145/2786805.2786830
89. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 265-278. DOI=http://dx.doi.org/10.1145/1961295.1950396
90. Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using Boolean satisfiability. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 16 (May 2007). DOI=http://dx.doi.org/10.1145/1232420.1232423
91. Domagoj Babic and Alan J. Hu. 2008. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th international conference on Software engineering* (ICSE '08). ACM, New York, NY, USA, 211-220. DOI=http://dx.doi.org/10.1145/1368088.1368118
92. Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER : Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. *Access* 28, 5 (2003), 327–336. DOI:https://doi.org/10.1145/940071.940115
93. Zhenbo Xu, Jian Zhang, and Zhongxing Xu. 2015. Melton: a practical and precise memory leak detection tool for C programs. *Front. Comput. Sci.* 9, 1 (2015), 34–54. DOI:https://doi.org/10.1007/s11704-014-3460-8
94. David A Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution : Correctness Checking for Real Code. *USENIX Secur. Symp.* (2015), 49–64. Retrieved from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos
95. Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2007. Program Analysis Using Symbolic Ranges. *Static Anal.* (2007), 366–383. DOI:https://doi.org/10.1007/978-3-540-74061-2_23
96. А.Е. Бородин. Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++. Диссертация на соискание ученой степени кандидата физ.-мат.наук, ИСП РАН, Москва 2016.
97. Erika Abraham, Gereon Kremer. *Satisfiability Checking: Theory and Applications*. Proceedings of the 14th International Conference on Software Engineering and Formal Methods (SEFM'16), Volume 9763 of LNCS, pages 9–23, Springer International Publishing, 2016.
98. Список SMT-решателей. http://smtlib.cs.uiowa.edu/solvers.shtml. Дата обращения 29.07.2017
99. Библиотека SMTLib. SMTLib http://smtlib.cs.uiowa.edu. Дата обращения 29.07.2017
100. Соревнование SMT-решателей SMT-comp 2016. http://smtcomp.sourceforge.net/2016. Дата обращения 29.07.2017
101. SMT-решатель Z3. https://github.com/Z3Prover/z3. Дата обращения 29.07.2017

102. SMT-решатель CVC4. http://cvc4.cs.stanford.edu/web. Дата обращения 29.07.2017
103. SMT-решатель Yices2. http://yices.csl.sri.com. Дата обращения 29.07.2017
104. Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and Effective Symbolic Analysis for Buffer Overflow Detection. *Proc. Eighteenth ACM SIGSOFT Int. Symp. Found. Softw. Eng.* (2010), 317–326. DOI:https://doi.org/10.1145/1882291.1882338
105. М.У. Мандрыкин, В.С. Мутилин, А.В. Хорошилов. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды Института системного программирования РАН, том 24, 2013, стр. 219-292.
106. Franjo Ivančić, G.a Gogul Balakrishnan, A.a Aarti Gupta, Sriram S.b Sankaranarayanan, N.c Naoto Maeda, Takashi T.c Imoto, R.d Rakesh Pothengil, and M.d Mustafa Hussain. 2014. Scalable and scope-bounded software verification in Varvel. *Autom. Softw. Eng.* 22, 4 (2014), 517–559. DOI:https://doi.org/10.1007/s10515-014-0164-0
107. Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Naoto Maeda, Hiroki Tokuoka, Takashi Imoto, and Yoshiaki Miyazaki. 2011. DC2: A framework for scalable, scope-bounded software verification. *2011 26th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2011, Proc.* (2011), 133–142. DOI:https://doi.org/10.1109/ASE.2011.6100046
108. Dirk Beyer and M. Erkan Keremoglu. 2009. CPAchecker: A Tool for Configurable Software Verification. (2009). Retrieved from http://arxiv.org/abs/0902.0019
109. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. *Integr. Form. Methods* (2004), 1–20. DOI:https://doi.org/10.1007/978-3-540-24756-2_1
110. Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. *Proc. 4th Int. Conf. Verif. Softw. Theor. Tools, Exp.* (2012), 146–161. DOI:https://doi.org/10.1007/978-3-642-27705-4_12
111. Швед П. Е., Мутилин В. С., Мандрыкин М. У. Опыт развития инструмента статической верификации BLAST // Программирование. — 2012. — Т. 3. — С. 24–35.
112. Alain Deutsch. 2003. Static verification of dynamic properties. *Compute* (2003), 1–8.
113. Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. 2016. Inferring annotations for device drivers from verification histories. *Ase'16* 1 (2016), 450–460. DOI:https://doi.org/10.1145/2970276.2970305
114. Dawson Engler and Madanlal Musuvathi. 2004. Static Analysis versus Software Model Checking for Bug Finding. In *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004 Venice, Italy, January 11-13, 2004 Proceedings*, Bernhard Steffen and Giorgio Levi (eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–210. DOI:https://doi.org/10.1007/978-3-540-24622-0_17
115. Анализатор Coverity. http://www.coverity.com. Дата обращения 29.07.2017

116. Анализатор Klocwork. http://www.klocwork.com. Дата обращения 29.07.2017
117. Анализатор HP Fortify. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast. Дата обращения 29.07.2017
118. Анализатор IBM AppScan. http://www-03.ibm.com/software/products/en/appscan. Дата обращения 29.07.2017
119. *MISRA C 2012 : Guidelines for the Use of the C Language in Critical Systems*. Misra, S.l.
120. Scott Meyers. 2005. *Effective C++ : 55 specific ways to improve your programs and designs*. Addison-Wesley, Upper Saddle River, NJ.
121. *MISRA-C++2008 : guidelines for the use of the C++ language in critical systems*. MISRA Limited, Nuneaton.
122. SEI CERT C Coding Standard. Carnegie Mellon University. (2016), 528.
123. Тестовые пакеты SAMATE NIST. https://samate.nist.gov/SRD/testsuite.php. Дата обращения 29.07.2017
124. Классификация ошибок CWE. https://cwe.mitre.org. Дата обращения 29.07.2017
125. 10 популярных уязвимостей для веб-приложений – OWASP Top Ten. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Дата обращения 29.07.2017
126. Isil Dillig, Thomas Dillig, and Alex Aiken. 2007. Static Error Detection using Semantic Inconsistency Inference. *ACM SIGPLAN Not.* 42, 6 (2007), 435. DOI:https://doi.org/10.1145/1273442.1250784
127. Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Oper. Syst. Rev.* (2001), 57–72. DOI:https://doi.org/10.1145/502034.502041
128. Кошелев В. К. Формализация определения ошибок при статическом символьном выполнении // Труды Института системного программирования РАН. — 2016. — Т. 28, № 5. — С. 105–118.
129. Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. 2010. Doomed program points. *Form. Methods Syst. Des.* 37, 2–3 (2010), 171–199. DOI:https://doi.org/10.1007/s10703-010-0102-0
130. Arindam Chakrabarti and Patrice Godefroid. 2006. Software partitioning for effective automated unit testing. *Proc. 6th ACM IEEE Int. Conf. Embed. Softw. - EMSOFT '06* (2006), 262. DOI:https://doi.org/10.1145/1176887.1176925
131. Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization using Maximum Satisfiability. *Pldi'11* (2011), 437–446. DOI:https://doi.org/10.1145/1993498.1993550

132. Shuvendu Lahiri, Akash Lal, Yi Li, and Ankush Das. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *Computer Aided Verification (CAV)*. Retrieved from https://www.microsoft.com/en-us/research/publication/angelic-verification-precise-verification-modulo-unknowns/
133. Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. 2015. Test suites for benchmarks of static analysis tools. *2015 IEEE Int. Symp. Softw. Reliab. Eng. Work.* (2015), 12–15. DOI:https://doi.org/10.1109/ISSREW.2015.7392027
134. Patrice Godefroid. 2005. The Soundness of Bugs is What Matters (Position Statement). *BUGS* (2005), 46574986.
135. Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, and Anders Møller. 2015. In defense of soundiness. *Commun. ACM* 58, 2 (2015), 44–46. DOI:https://doi.org/10.1145/2644805
136. K. Tsipenyuk, B. Chess and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," in *IEEE Security & Privacy*, vol. 3, no. 6, pp. 81-84, Nov.-Dec. 2005. doi: 10.1109/MSP.2005.159
137. A.Y. Gerasimov. 2017. Survey on static program analysis results refinement approaches. *Proc. Inst. Syst. Program. RAS* 29, 3 (2017), 75–98. DOI:https://doi.org/10.15514/ISPRAS-2017-29(3)-6
138. Tukaram Muske and Alexander Serebrenik. 2016. Survey of approaches for handling static analysis alarms. *Proc. - 2016 IEEE 16th Int. Work. Conf. Source Code Anal. Manip. SCAM 2016* (2016), 157–166. DOI:https://doi.org/10.1109/SCAM.2016.25
139. Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.* 53, 4 (2011), 363–387. DOI:https://doi.org/10.1016/j.infsof.2010.12.007
140. Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11--13, 2003 Proceedings*, Radhia Cousot (ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 295–315. DOI:https://doi.org/10.1007/3-540-44898-5_16
141. Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound non-statistical clustering of static analysis alarms. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 7148 LNCS, (2012), 299–314. DOI:https://doi.org/10.1007/978-3-642-27940-9_20
142. Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th Working Conference on*

Mining Software Repositories (MSR 2014). ACM, New York, NY, USA, 152-161. DOI: http://dx.doi.org/10.1145/2597073.2597100

143. J.R. Ruthruff, J. Penix, J.D. Morgenthaler, S. Elbaum, and G. Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings. *Proc. 30th Int. Conf. Softw. Eng.* (2008), 341–350. DOI:https://doi.org/10.1145/1368088.1368135
144. Sunghun Kim and Michael D. Ernst. 2007. Prioritizing warning categories by analyzing software history. *Proc. - ICSE 2007 Work. Fourth Int. Work. Min. Softw. Repos. MSR 2007* (2007), 0–3. DOI:https://doi.org/10.1109/MSR.2007.26
145. Cathal Boogerd and Leon Moonen. 2006. Prioritizing software inspection results using static profiling. *Proc. - Sixth IEEE Int. Work. Source Code Anal. Manip. SCAM 2006* (2006), 149–158. DOI:https://doi.org/10.1109/SCAM.2006.22
146. Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. 2008. An Approach to Merge Results of Multiple Static Analysis Tools (Short Paper). *2008 Eighth Int. Conf. Qual. Softw.* (2008). DOI:https://doi.org/10.1109/QSIC.2008.30
147. Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. *Icse* (2017), 519–529. DOI:https://doi.org/10.1109/ICSE.2017.54
148. Francesco Logozzo and Matthieu Martel. 2013. Automatic Repair of Overflowing Expressions with Abstract Interpretation. *Electron. Proc. Theor. Comput. Sci.* 129, (2013), 341–357. DOI:https://doi.org/10.4204/EPTCS.129.21
149. Martin Monperrus. 2015. Automatic Software Repair : a Bibliography. *Others* June (2015), 1–34.
150. Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically fixing C buffer overflows using program transformations. *Proc. - 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks, DSN 2014* (2014), 124–135. DOI:https://doi.org/10.1109/DSN.2014.25
151. Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. *ACM SIGPLAN Not.* 47, 10 (2012), 133. DOI:https://doi.org/10.1145/2398857.2384626
152. Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for C programs. *Proc. - Int. Conf. Softw. Eng.* 1, 1 (2015), 459–470. DOI:https://doi.org/10.1109/ICSE.2015.64
153. Al Bessey, Dawson Engler, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, and Scott McPeak. 2010. A Few Billion Lines of Code Later. *Commun. ACM* 53, 2 (2010), 66–75. DOI:https://doi.org/10.1145/1646353.1646374
154. Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. *Proc. - Int. Conf. Softw. Eng.* 1, (2015), 598–608. DOI:https://doi.org/10.1109/ICSE.2015.76

155. Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Autom. Softw. Eng.* 22, 4 (2015), 561–602. DOI:https://doi.org/10.1007/s10515-014-0169-8
156. Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion* (ICSE '16). ACM, New York, NY, USA, 11-20. DOI: https://doi.org/10.1145/2889160.2889222
157. Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis : An Empirical Study. *Ace'16* (2016), 332–343. DOI:https://doi.org/10.1145/2970276.2970347
158. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings - International Conference on Software Engineering*, 672–681. DOI:https://doi.org/10.1109/ICSE.2013.6606613
159. Страница Доусона Энглера на сайте Стенфордского университета. https://web.stanford.edu/~engler. Дата обращения 29.07.2017
160. Edison Design Group. https://www.edg.com. Дата обращения 29.07.2017
161. Инструменты динамического анализа Google Sanitizers. https://github.com/google/sanitizers. Дата обращения 29.07.2017
162. Sarah Heckman and Laurie Williams. 2008. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. *Proc. Second ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.* (2008), 41–50. DOI:https://doi.org/10.1145/1414004.1414013
163. Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.* 217, C (2008), 5–21. DOI:https://doi.org/10.1016/j.entcs.2008.06.039
164. Gabriel Díaz and Juan Ramón Bermejo. 2013. Static analysis of source code security: Assessment of tools against SAMATE tests. *Inf. Softw. Technol.* 55, 8 (2013), 1462–1476. DOI:https://doi.org/10.1016/j.infsof.2013.02.005
165. Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of C programs. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 6617 LNCS, (2011), 459–465. DOI:https://doi.org/10.1007/978-3-642-20398-5_33
166. Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Martino Luca, Peter O Hearn, and Irene Papakonstantinou. Moving Fast with Software Verification.

https://research.facebook.com/publications/422671501231772/moving-fast-with-software-verification/ Дата обращения 29.07.2017

167. Fraser Brown, Andres Nötzli, and Dawson Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. *ASPLOS '16 Proc. Twenty-First Int. Conf. Archit. Support Program. Lang. Oper. Syst.* (2016), 143–157. DOI:https://doi.org/10.1145/2872362.2872364
168. Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. *Proc. Twenty-Second Int. Conf. Archit. Support Program. Lang. Oper. Syst.* (2017), 389–404. DOI:https://doi.org/10.1145/3037697.3037744
169. Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. *Ecoop 2016* (2016), 1–25. DOI:https://doi.org/10.4230/LIPIcs.ECOOP.2016.2
170. Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. *Proc. - IEEE Symp. Secur. Priv.* (2014), 590–604. DOI:https://doi.org/10.1109/SP.2014.44
171. Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. *Proc. - IEEE Symp. Secur. Priv.* 2015–July, (2015), 797–812. DOI:https://doi.org/10.1109/SP.2015.54
172. Fabian Yamaguchi. 2015. Pattern-Based Vulnerability Discovery. Ph.D. thesis, Georg-August University School of Science, Gottingen 2015.
173. Инструмент Google ErrorProne. https://github.com/google/error-prone. Дата обращения 29.07.2017
174. Инструмент Microsoft SDV. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier. Дата обращения 29.07.2017
175. Ralf Huuck, Ansgar Fehnker, Sean Seefried, and Jörg Brauer. 2008. Goanna: Syntactic software model checking. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 5311 LNCS, (2008), 216–221. DOI:https://doi.org/10.1007/978-3-540-88387-6-17
176. Инструмент InferBo. https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer. Дата обращения 29.07.2017
177. Д. Кнут. Искусство программирования. Том 3. Сортировка и поиск. М., Вильямс, 2014.
178. Samuel Livingston Kelly. 2014. AST Indexing : A Near-Constant Time Solution to the Get-Descendants-by-Type Problem. Honor's Thesis. Dickinson College. (2014).

179. D.J. Lilja. 1999. Exploiting basic block value locality with block reuse. *Proc. Fifth Int. Symp. High-Performance Comput. Archit.* August (1999), 106–114. DOI:https://doi.org/10.1109/HPCA.1999.744342
180. Diego Novillo. A Propagation Engine for GCC. In Proceedings of the 2005 GCC Developers Summit, Ottawa, Canada.
181. M Sharir. 1980. Structural analysis: A new approach to flow analysis in optimizing compilers. *Comput. Lang.* (1980). DOI:https://doi.org/10.1016/0096-0551(80)90007-7
182. Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. *International Symposium on Programming*, 106–130. DOI:https://doi.org/10.1145/390019.808314
183. Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified latice model. *POPL '77 Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.* (1977), 238–252. DOI:https://doi.org/10.1145/512950.512973
184. Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming* (PLILP '92), Maurice Bruynooghe and Martin Wirsing (Eds.). Springer-Verlag, London, UK, UK, 269-295.
185. Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula, Antonia Zhai, and Pen Chung Yew. 2006. Loop selection for thread-level speculation. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 4339 LNCS, (2006), 289–303. DOI:https://doi.org/10.1007/978-3-540-69330-7_20
186. Darren C. Atkinson. 2004. Accurate call graph extraction of programs with function pointers using type signatures. *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC* (2004), 326–335. DOI:https://doi.org/10.1109/APSEC.2004.16
187. Tucker Taft. The use of value numbers in static analysis. Adacore. http://www.adacore.com/knowledge/technical-papers/the-use-of-value-numbers-in-static-analysis. Дата обращения 29.07.2017
188. Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1995. Value Numbering. *Softw. Pract. Exp.* 27, 0 (1995), 701–724. DOI:https://doi.org/10.1002/(SICI)1097-024X(199706)27:6<701::AID-SPE104>3.0.CO;2-0
189. Использование статического анализатора в системе рецензирования для ОС Tizen. https://developer.tizen.org/community/tip-tech/how-process-potential-defects-static-analysis-tool-using-public-jira-system. Дата обращения 29.07.2017

190. Поиск классов при компиляции в Javac. http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html#BHCCHDGH. Дата обращения 29.07.2017
191. Уязвимость Apple CVE-2014-1266. https://nvd.nist.gov/vuln/detail/CVE-2014-1266. Дата обращения 29.07.2017
192. Р.Р. Мулюков, А.Е. Бородин. Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 145-158. DOI: 10.15514/ISPRAS-2016-28(5)-9
193. Цикл безопасной разработки Microsoft. https://www.microsoft.com/en-us/sdl. Дата обращения 29.07.2017
194. ГОСТ Р 56939-2016. Защита информации. Разработка безопасного программного обеспечения. Общие требования. http://protect.gost.ru/document.aspx?control=7&id=203548. Дата обращения 29.07.2017
195. Инструмент SLOCCount. https://www.dwheeler.com/sloccount. Дата обращения 29.07.2017
196. Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. *2013 9th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. ESEC/FSE 2013 - Proc.* (2013), 554–564. DOI:https://doi.org/10.1145/2491411.2501854