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

Автореферат
диссертации на соискание ученой степени
доктора физико-математических наук

Москва – 2017

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской академии наук

Научный консультант: **Аветисян Арутюн Ишханович**,
доктор физико-математических наук, чл.-корр. РАН

Официальные оппоненты: **Галатенко Владимир Антонович**,
доктор физико-математических наук,
заведующий сектором Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Мельник Эдуард Всеволодович,
доктор технических наук, заведующий отделом Федерального государственного бюджетного учреждения науки «Южный научный центр Российской академии наук»

Терехов Андрей Николаевич,
доктор физико-математических наук, профессор,
заведующий кафедрой системного программирования Федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет»

Ведущая организация: Федеральный исследовательский центр «Информатика и управление» Российской академии наук

Защита диссертации состоится 15 февраля 2018 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В.П. Иванникова Российской академии наук» по адресу: 109004, Москва, ул. А. Солженицына, 25.

С диссертацией можно ознакомиться в библиотеке Федерального государственного бюджетного учреждения науки «Институт системного программирования им. В.П. Иванникова Российской академии наук».

Автореферат разослан “___”___________________2017 г.

Ученый секретарь
диссертационного совета Д.002.087.01,
кандидат физико-математических наук Зеленов С.В.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность. 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[1] 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

[1] В дальнейшем будем называть *уровнем анализа* набор его характеристик, таких, как межпроцедурность, чувствительность к контексту или путям выполнения, строгость, анализируемое внутреннее представление и т.п.

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 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 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.

Апробация работы. Основные результаты диссертационной работы обсуждались на конференциях и семинарах различного уровня, в том числе 4 доклада на международных конференциях и 5 на всероссийских. В частности: 6th International Conference on Computer Science and Information Technologies (CSIT'2007), 24-28 September, Yerevan, Armenia; 9th International Conference on Computer Science and Information Technologies (CSIT 2013), 23-27 September, 2013, Yerevan, Armenia; Открытая конференция по компиляторным технологиям 2015, Москва, Россия; Открытая конференция ИСП РАН 2016, Москва, Россия; Tizen Developers' Conference 2017, San Francisco, USA; международная Ершовская конференция по информатике PSI–2017 27-29 июня 2017 года, Москва, Россия; Ломоносовские чтения 2017; конференция OS DAY 2017, 23-24 мая 2017 года, Москва, Россия; X Всероссийская межведомственная конференция "Актуальные направления развития систем охраны, специальной связи и информации для нужд государственной власти Российской Федерации", Академия ФСО России, г. Орел 2017 г.

Гранты и контракты. Работа по теме диссертации проводилась в соответствии с планами исследований по проектам, поддержанными: грантом РФФИ 08-07-00279-а "Исследование и разработка системы автоматического обнаружения дефектов в исходном коде программ"; контрактами в рамках Программы фундаментальных исследований Президиума РАН «Фундаментальные проблемы системного программирования»; контрактами 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.

Публикации. Автором опубликовано более 40 научных печатных трудов по теории компиляции и анализу программного кода. 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. Получено 9 свидетельств [11-19] о регистрации программ для ЭВМ.

Структура и объем диссертационной работы. Диссертация состоит из введения, пяти глав и заключения, изложенных на 229 страницах, списка литературы из 196 наименований, содержит 39 рисунков и 17 таблиц.

СОДЕРЖАНИЕ РАБОТЫ

Во введении обосновывается актуальность исследований, приводятся цель и задачи работы, формулируется научная новизна и практическая значимость полученных результатов.

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. *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 дополнительной информации о том, при каких условиях ошибка может произойти. С точки зрения 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. 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).

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. Для промышленных АСД-анализаторов можно отметить, что в основном все они основаны на открытых компиляторных инфраструктурах для соответствующих языков. Кроме того, детекторы ошибок в описанных инструментах реализованы на универсальных языках программирования. В методах межпроцедурного анализа (**раздел 1.2**) можно отметить подходы к анализу "снизу-вверх", при котором вычисляются аннотации функций, верные для любых контекстов вызова, и анализ "сверху-вниз", начинающий работу от точек входа в программу и переанализирующий вызываемую функцию для каждого нового контекста либо группы контекстов вызовов. Отдельной постановкой задачи межпроцедурного анализа является сведение к задаче

достижимости на расширенном графе (так называемой IFDS-задаче). Промышленные анализаторы в настоящий момент 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. В **разделе 1.3** дается обзор методов анализа, чувствительных к путям выполнения, с упором на методы символьного выполнения, применяющие SMT-решатели для проверки выполнимости предикатов пути и условий искомых ошибочных ситуаций. Основные анализаторы на рынке являются чувствительными к путям, однако детали реализации, как правило, неизвестны.

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. 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. При статическом анализе в том или ином виде всегда имеется неизвестное анализатору окружение, являющееся источником неопределенных значений переменных программы при вызове одной из анализируемых функций: программа на входе анализатора может быть неполной, либо, при достаточно больших размерах программы в ходе межпроцедурного анализа неизбежно произойдет потеря точности вычисляемых данных. 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. *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.

Разделы 1.5 и **1.7** посвящены развиваемым в настоящее время методам анализа, а также методам ранжирования предупреждений по вероятности их истинности и методам автоматизированного исправления найденных ошибок. **Раздел 1.6** посвящен опыту применения промышленных анализаторов. Можно отметить наиболее перспективные направления исследований:

- 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 системы "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.

Вторая глава посвящена разработанной методологии многоуровневого статического анализа исходного кода, состоящей из выполнения анализа на трех уровнях – уровне абстрактного синтаксического дерева (АСД), уровне межпроцедурного контекстно-чувствительного анализа и уровне чувствительного к путям анализа, разработанных моделей программы и алгоритмов анализа соответствующих уровней. Математическое обоснование предложенных алгоритмов состоит в формулировке и доказательстве теорем об оценках сложности и о корректности алгоритмов.

При разработке методов анализа необходимо удовлетворить следующим требованиям: поддержка распространенных статически типизированных языков программирования (Си, Си++, Java, C#); поддержка актуальных классов дефектов (в заданных языках) различной степени критичности; возможность относительно несложно добавить поддержку новых классов дефектов; масштабируемость до проведения анализа миллионов строк кода за несколько часов; не менее 60% истинных срабатываний; пропуск незначительного количества дефектов.

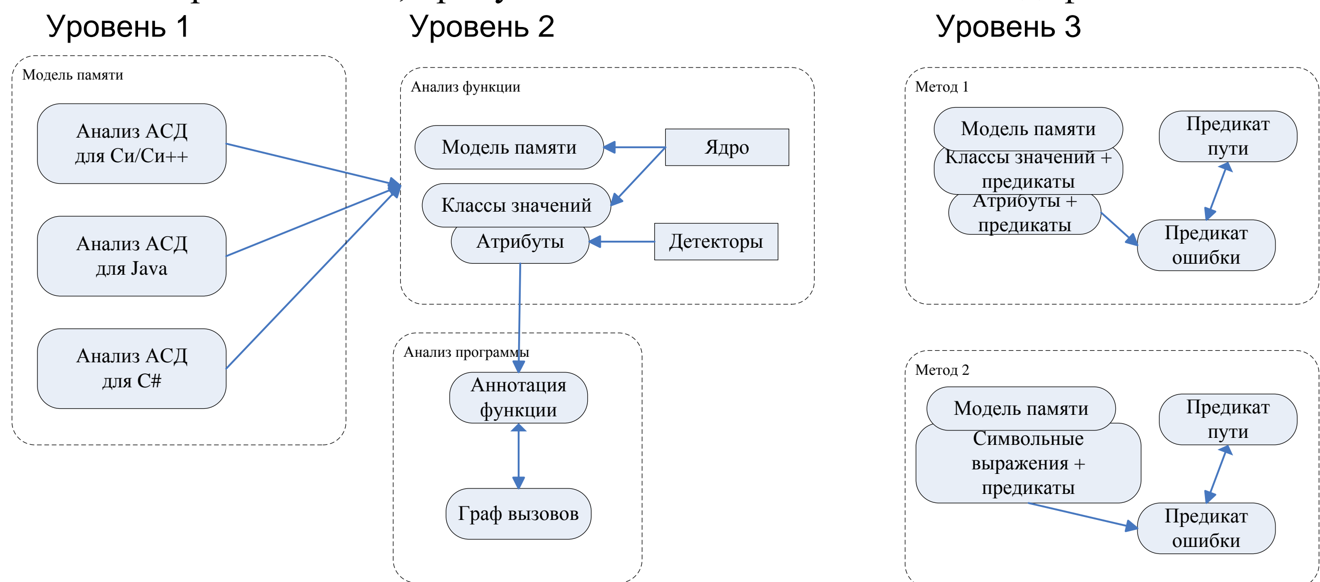

Рисунок 1. Организация методов многоуровневого статического анализа.

Анализ известных классов дефектов показывает, что обойтись единственным уровнем анализа для поиска всех 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 АСД и максимально детальной информации об исходном коде. Наконец, существуют дефекты, занимающие промежуточное положение между этими двумя категориями – для них требуется межпроцедурный анализ, чувствительный к контексту выполнения, но, как правило, не требуется чувствительность к путям. Примером таких дефектов являются специфичные ситуации разыменования нулевого указателя, консистентное использование операторов new/new[] – delete/delete[] в языке Си++ и некоторые другие.

Предлагаемая организация набора методов анализа на различных уровнях представлена на рисунке 1.

В разделе 2.1 описывается первый уровень статического анализа для 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 (нарушения правил кодирования, неверное использование возможностей языка программирования или программных интерфейсов);
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).

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 $M = \langle S, F \rangle$, где S – таблица символов модуля, а F – множество функций модуля, каждая из которых представлена своим АСД. Природа выполняемых детекторами проверок может быть совершенно различной, и для масштабируемости нужно потребовать, чтобы сложность алгоритмов детекторов была линейной от размера представления программы, т.е. от количества узлов АСД и элементов таблиц символов. 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.
- **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. Данный тип детекторов наиболее распространен.
- **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.
- **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.

Детекторы выделенных типов либо работают за линейное время, либо для достижения линейной сложности необходимо дополнительно ограничить количество просматриваемых ими узлов АСД, как сформулировано ниже.

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.

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,
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.

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. Также для экономии памяти

возможно хранить ссылки на следующие узлы нужного типа только в некоторых узлах (назовем их базовыми), а в остальных узлах хранить единственную ссылку на ближайший родительский базовый узел (сформулировано соответствующее улучшение алгоритма 2.1).

В разделе 2.1.2 строится модель памяти программы для внутрипроцедурного анализа. Представление функции в виде АСД дополняется графом потока управления. 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.

Для моделирования памяти мы используем понятие *ячеек памяти*, которые принадлежат некоторым *областям памяти*. 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, весь массив по отношению к его элементам); S и O задают размер ячейки памяти (в битах) и смещение относительно базового адреса. Родительская ячейка также задает для данной ячейки *базовый адрес*, т.е. либо виртуальный адрес начала стека функции, глобальный указатель, либо начало динамически выделенного участка памяти.

Ячейки памяти создаются «лениво» в момент первого доступа к ним. 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*: R = Local, S – вычисленный компилятором размер типа переменной, O –смещение данной локальной переменной относительно начала фрейма функции, P = *MLFrameFoo*.
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 –размер типа переменной, O = 0, P – эта же ячейка памяти.
5. Выделение динамической памяти (через вызовы функции malloc или подобных ей). Считается, что каждый вызов функции динамической памяти возвращает 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 –размер типа переменной; O – смещение относительно родительской ячейки, 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, чтобы ограничить точность модели 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.

Рассматривается модельный язык, состоящий из целочисленных переменных, массивов, указателей и структур, в котором имеются операторы присваивания, арифметические и логические выражения, вызовы функций и вызовы по указателю, условный оператор и оператор цикла. Для выражений языка записываются *передаточные функции*, описывающие, как изменяются интервалы значений и множества *PtTo* при интерпретации этого выражения. Предполагая, что соответствующие данные вычислены, изменяются правила создания ячеек памяти в случае неизвестных размеров ячеек или их адресов. Приводится **алгоритм 2.2,** вычисляющий для всей функции ячейки памяти, абстрактные значения *Val* и множества *PtTo*. 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. <u>Инициализация</u>: положить $M_{Entry} = \{ML_{static}\}$, где ML_{static} – ячейки памяти для глобальных и статических переменных.
2. <u>Структурный анализ</u>: построить дерево управления, состоящее из выделенных в графе потока управления шаблонов вида *Block*, *IfThen*, *IfThenElse*, *While*.
3. <u>Основной цикл</u>: 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, 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*.
- *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; далее накладываются ограничения из обработки условий тела цикла, проходит анализ тела цикла, и i увеличивается на 1;
    - 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 ∇ . 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 1;
    - 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 с дуги, минующей цикл. 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.

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.

В разделе 2.2 вводится второй уровень анализа – межпроцедурный анализ на основе аннотаций функции. Внутреннее представление анализа понижается до четверок <*Op*, *VarDest*, *VarLeft*, *VarRight*>. Анализ получает на вход множество

функций $F = \{ f{:}\langle S, G\rangle \}$, 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 $Glob = \{ g{:}\langle S, I\rangle \}$, где аналогично S – определение переменной, т.е. ее имя и тип, I – начальное значение переменной.

Алгоритм 2.3. Межпроцедурный анализ на основе аннотаций. 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 переменных для всех точек программы.

1. Построение графа вызовов программы. В графе вызовов вершинами являются анализируемые функции, а дуги показывают связи по вызовам между ними. Требуется выполнить анализ указателей на функцию для построения достаточно точного графа вызовов.
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.
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 раздела 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 таких функций мы будем называть *спецификацией функции*.

При построении графа вызовов сложности доставляют вызовы по указателю. Целесообразно провести возможно быстрый анализ для уточнения потенциальных целей косвенных вызовов. Для этого (**алгоритм 2.4**) 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, которые содержат указатели на функцию – и вычислить множества $PtTo$ только для таких указателей. Далее для вызовов функций по указателю проводятся косвенные дуги до всех функций, попавших в соответствующее множество $PtTo$. **Алгоритм 2.5** уточняет результаты алгоритма 2.4, выполняя межпроцедурный анализ снизу вверх в обратном топологическом порядке по статическому графу вызовов аналогично шагу 3 алгоритма 2.3.

Раздел 2.2.2 предлагает внутрипроцедурный алгоритм для шага 3.1 алгоритма 2.3. Значения отслеживаются с использованием *классов значений* (КЗ): $VCs{:}\, ML \times I \to VC$, которые играют роль абстрактного значения. Правила присвоения нового класса значений схожи с правилами заведения номера значений в нумерации значений, основанной на хэшировании – 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. Такие свойства мы будем называть *атрибутами*.

Основными атрибутами, вычисляемыми ядром, являются уже упомянутые интервалы значений Val и множества потенциальных целей указателей $PtTo$. Приводятся передаточные функции для построения КЗ в модельном языке. Передаточные функции для интервалов и множеств $PtTo$ остаются прежними с той лишь разницей, что теперь для их получения по ячейке памяти необходимо сначала узнать КЗ, хранящийся в ячейке, а потом по этому классу получить искомый атрибут. Формулы для передаточных функций получают соответствующие механические изменения. Приводится **алгоритм 2.6** внутрипроцедурного анализа функции с применением классов значений, являющийся вариантом алгоритма 2.2 с описанными изменениями передаточных функций: при обработке инструкций и слиянии потока управления сначала обрабатываются КЗ, а потом их атрибуты.

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.

Раздел 2.2.3 содержит алгоритм построения аннотации функции. При обработке вызова *функции* используется созданная аннотация, повторного анализа функции не происходит, что позволяет достичь масштабируемости. Контекстная чувствительность анализа задается *параметризацией* аннотации, т.е. выражением вычисленных значений от входных значений функции. Введем понятие *внешних ячеек памяти* EML для данной функции – ячеек, соответствующих 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.

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. Тогда при сохранении аннотации функции свойства соответствующих КЗ будут вычислены не до конца, а выражены символьно через внешние абстрактные значения. Далее, при обработке аннотации в конкретном контексте вызова символьное значение заменяется на фактическое и производится довычисление значений, чтобы как можно более точно охарактеризовать КЗ и ячейки памяти в точке вызова.

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 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.

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 КЗ 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 КЗ 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 КЗ 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*. После

окончания процесса сопоставления происходит перевычисление атрибутов, которые зависели от внешних значений, подставлением их фактических значений и повторным применением передаточных функций и функций объединения.

В разделе 2.3 предлагается третий уровень анализа – межпроцедурный анализ с чувствительностью к путям. Для адаптации анализа раздела 2.2 предлагаются следующие изменения:

- Построение предиката пути. Для каждой точки программы внутрипроцедурный анализ строит 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.
- 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. Приводятся соответствующие изменения в передаточных функциях. Для интервалов значений сохраняется консервативное отслеживание через функцию *Unify* для вычисления неконстантных смещений в ячейках памяти.
- Отслеживание предикатов для классов значений. При объединении КЗ 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 КЗ. Приводятся соответствующие изменения в передаточных функциях.
- 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.

Для внутрипроцедурного анализа, создающего модель памяти, классы значений и их атрибуты, можно применять алгоритмы 2.2 и 2.6 соответственно. Действительно, передаточные функции, создающие множества *PtTo* и КЗ, 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 КЗ 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 КЗ алгоритмом 2.6 гарантируется теоремой 2.4, а мы не меняем этот алгоритм – введение предикатов состава данного класса значений не влияет на выбор. Поэтому построение множеств *PtTo* и КЗ является корректным, а корректность созданных предикатов следует из корректности построения предиката пути. Тем самым д**оказана**

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.

Во время межпроцедурного анализа алгоритм 2.7 выделения ячеек памяти и КЗ, 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. При этом, т.к. исходные передаточные функции являются ассоциативными, и конъюнкция/дизъюнкция предикатов также ассоциативны, требования теоремы 2.5 выполняются. Алгоритм сопоставления ячеек памяти и классов значений также остается неизменным.

В разделе 2.3.2 излагается организация чувствительного к путям анализа без применения КЗ. 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 заданному значению. Задачей анализа является определение того, какие символьные выражения хранятся в ячейках памяти программы в каждой ее точке.

Каждый детектор строит собственную формулу выполнимости ошибочной ситуации с использованием предикатов над символьными выражениями. Совместность формулы проверяется с помощью SMT-решателей.

Для эффективного использования такого подхода необходимо пересмотреть способ построения модели памяти. Если анализируется язык, допускающий адресную арифметику и произвольные смещения дочерних ячеек памяти относительно родительских, то полноценная поддержка символьной памяти будет требовательна к ресурсам. Поэтому целесообразно ограничить применение подхода 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. Будем 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]) ...). Ограничим

модельный язык, запретив арифметические операции с указателями и взятие адреса произвольной ячейки памяти. Будем вычислять множества $PtTo$, отслеживая предикаты, при выполнении которых ячейка памяти указывает на ту или иную другую ячейку.

Передаточные функции, обновляющие символьные выражения для ячеек памяти, схожи с функциями для КЗ 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.

Алгоритм выполнения внутрипроцедурного и межпроцедурного анализа строится аналогично алгоритму 2.6'. 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, но без применения оператора ∇.

При межпроцедурном анализе 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.

В разделе 2.4 описываются принципы построения детекторов для межпроцедурного анализа и для чувствительного к путям анализа. Детекторы в анализе с классами значений заводят один или несколько атрибутов, отражающих необходимые им свойства КЗ. 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 для каждой точки программы строится отображение $Attrs: VC \times AttrName \rightarrow Attr$. Для

этого дополнительно определяются передаточные функции для всех инструкций внутреннего представления $TF: Insn \times Attrs \rightarrow Attrs$. Передаточной функции предъявляются значения атрибута для всех КЗ до выполнения данной инструкции, и она строит значение атрибута после выполнения. Для инструкции присваивания передаточная функция не строится, поскольку достаточно применения функции для КЗ. 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 $\nabla: Attr \rightarrow Attr$ 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 КЗ 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 случае создания чувствительных к путям детекторов (**раздел 2.4.2**) в каждой точке программы строится отображение $Attrs: VC \times AttrName \rightarrow AttrPred$, 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 $TF: Op \times Attr \times Attr \rightarrow Attr$, при вычислении передаточной функции вместе с предикатами берутся все попарные комбинации предикатов, записанных в атрибутах операндов, и применяется исходная передаточная функция. При слиянии потока управления к предикатам атрибутов КЗ, 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. 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 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. Кроме того, стоит задача– описания возможного пути выполнения, вдоль которого произошла ошибка, и возможных значений переменных программы. Для этого детекторами отслеживаются дополнительные атрибуты, а также проверяется совместность предиката ошибки и условий переходов в точках разделения потока управления.

В третьей главе описана архитектура предлагаемой системы анализа, реализованная в семействе инструментов Svace, которая обеспечивает все этапы работы анализатора – построение внутреннего представления программы собственными компиляторами с помощью организации контролируемой сборки программы, устройство основной фазы анализа с возможностью параллельной работы, а также инкрементального анализа, хранение и просмотр результатов анализа.

Рисунок 2. Организация и схема работы анализатора Svace.

Устройство архитектуры Svace преследует две цели: во-первых, сделать возможным 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.

Предлагается использовать архитектуру построения набора анализаторов, схематически представленную на рисунке 2 вместе с процессом работы анализатора. Для работы алгоритмов анализа главы 2 прежде всего необходимо построить внутреннее представление анализируемой программы 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. На следующем этапе запускаются собственно алгоритмы анализа для всех уровней, 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.

После окончания работы анализа наступает время работы пользователя с результатами анализа. Так как предполагается, что работа над анализируемой программой ведется постоянно, 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, что 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. 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.

В разделе 3.1 разбирается устройство контролируемой сборки программы. 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.

Обнаружение событий сборки для самого распространенного случая запуска нового процесса выполняется с помощью подмены функций системных динамических библиотек на ОС Linux, а также мониторингом создания процессов средствами отладки на ОС Windows и Linux (для статических процессов). В особых случаях используется замена исполняемых файлов интересующих

анализатор процессов на собственные обертки. Если событие происходит без запуска процесса (например, при компиляции через интерфейсы библиотеки компилятора), то используются дополнительные методы, например, инструментирование байткода виртуальной машины Java для перехвата таких вызовов.

Реакцией на событие обычно является запуск собственного инструмента (например, компилятора), ведение журнала событий компоновки и т.п., сбор исходных кодов, модификация окружения перехваченного процесса.

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 поддерживается 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.

В разделе 3.2 описывается создание собственных компиляторов для генерации внутреннего представления анализаторов. Решаются проблемы поддержки многочисленных диалектов языков программирования, нестандартного кода, надежного восстановления после ошибок, максимально полного соответствия исходному коду, а также собирается необходимая дополнительная информация для анализатора. Так, в компиляторе Clang было выполнено более 1000 изменений представленных категорий. Компилятор Javac потребовал сравнительно небольшого количества изменений.

В разделе 3.3 представлено устройство основного этапа анализа. Его задает **алгоритм 3.1**: выполняется запуск детекторов уровня АСД (для языков Си и Си++ – на основе анализатора CSA компилятора Clang), сортировка файлов внутреннего представления по целевой архитектуре, построение графа вызовов программы, выделение в нем компонент связности и разрыв циклов, параллельный межпроцедурный анализ графа вызовов, сортировка и сохранение выданных предупреждений. При построении графа вызовов происходит «виртуальная» компоновка программы (без построения физически единого файла) на основе информации о событиях ассемблирования, архивирования и компоновки из журнала событий, удаляются функции-дубликаты.

При межпроцедурном анализе диспетчер определяет порядок обхода графа вызовов и выдает очередную функцию на анализ одному из имеющихся параллельных потоков так, чтобы потребляемая потоками память не превысила заданного ограничения. При выборе функции диспетчер отслеживает объем занятой памяти, прогресс анализа, распределение функций по файлам для обеспечения локальности чтений с диска. Тщательно обеспечивается детерминированность внутрипроцедурного анализа и построения аннотаций, что позволяет добиться детерминированности всего межпроцедурного анализа.

Спецификации внешних функций, неизвестных анализатору, пишутся на анализируемом языке программирования с использованием специальных встроенных функций Svace, которые называются *спецфункциями*. Спецификации компилируются собственными компиляторами, и при построении графа вызовов эти файлы загружаются перед всеми пользовательскими файлами. При обнаружении вызова спецфункций в ходе анализа модель программы, вычисляемая анализом, напрямую модифицируется нужным образом. Это, как правило, удобнее, чем добиваться того же эффекта с помощью эквивалентного кода на Си или другом языке.

Особенностью анализа языка Си++ по сравнению с изложенным выше является учет исключений в потоке управления функции, а также отдельная фаза парного анализа конструкторов и деструкторов, проверяющая их согласованность. Для программ на языке Java анализ АСД проводится в компиляторе Javac, а также в инструменте FindBugs. Для дальнейшего анализа байткод Java переводится во внутреннее представление Svace, что также позволяет анализировать JAR-библиотеки. Важную роль играет девиртуализация с использованием иерархии классов при построении графа вызовов и спецификации функций стандартной библиотеки, написанные с учетом ее особенностей (например, рекурсивного закрытия ресурсов по классам-оберткам).

Анализ программ на языке C# реализован в виде отдельного движка на основе проекта Roslyn, который поддерживает все уровни анализа. Контролируемая сборка выполняется с помощью логгера утилит MSBuild и xbuild. Далее запускаются детекторы уровня АСД. Строится граф вызовов с учетом геттеров/сеттеров, лямбда-выражений, анонимных функций и прочих особенностей языка C#. Проводится девиртуализация на основе адаптированного для C# подхода к девиртуализации через анализ иерархии классов. Выполняется алгоритм 2.3 межпроцедурного анализа на основе аннотаций – функции посещаются в обратном топологическом порядке, выполняется внутрипроцедурный анализ каждой функции непосредственно по АСД функции, после чего создается ее аннотация и используется в дальнейшем при обработке вызовов данной функции.

Первым этапом внутрипроцедурного анализа является классический анализ потока данных, который проводится для только внутрипроцедурных детекторов, далее выполняется построение модели памяти и программы на основе ячеек памяти и вычислении символьных выражений, 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. Поддерживается параллельный анализ независимых функций. При необходимости АСД и аннотации функций сериализуются на диск и считываются в момент возникновения нужды в них.

В разделе 3.3.7 описывается функциональность удаленного и инкрементального анализа. Удаленный анализ поддерживается за счет хранения всех необходимых данных для анализа в *объекте сборки*, а для просмотра результатов – в *объекте анализа*, которые могут свободно перемещаться между

машинами. В инкрементальном анализе сначала выполняется полная сборка и анализ программы на некотором сервере, при этом сохраняется граф вызовов программы и все построенные аннотации функций. На клиентской машине в режиме инкрементального анализа сначала создается список файлов для повторного анализа на основе инкрементальной сборки, явного указания файлов пользователем или автоматического определения измененных файлов. Для этих файлов проигрываются записанные на этапе полной сборки команды компиляции для построения внутреннего представления и команды анализаторов АСД. Далее строится обновленный граф вызовов, в котором функции из изменившихся файлов заменяются на функции из новых файлов, удаленные функции выбрасываются, все функции из удаленных файлов также выбрасываются, а из созданных файлов – добавляются. После этого 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.

В разделе 3.4 описывается инфраструктура хранения и просмотра результатов анализа. Совокупность данных, являющихся результатом сборки, называется *объектом сборки*, а данных анализа – *объектом анализа*. В ходе сборки все инструменты, 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 уровня АСД для каждой компиляции. 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.

Содержащиеся в хранилище объекты реализуют общий интерфейс, который дает возможность получить данные объекта в виде потока байт, их размер, тип объекта и хэш содержимого (SHA-1). Хэш объекта является его идентификатором, то есть хранилище организовано контентно-адресуемым способом. Основными типами объектов являются: файл (исходный или двоичный), 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.

Для работы с хранилищем 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.

Основным вопросом при реализации сервера результатов является перенос разметки предупреждений между снимками анализа, требуемое для скрытия однажды размеченных ложных или не требующих реакции предупреждений. Для сопоставления предупреждений между разными результатами анализа (**алгоритм 3.2**) для файлов, где были выданы предупреждения, генерируется список различий между ними (diff), и для строки предупреждения и строки кандидата на сопоставление по списку различий оценивается соответствие между строками. 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.

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 и предлагают методы поиска ошибок для Си/Си++, Java и C#. Выполняется поиск всех популярных типов критических ошибок (разыменование нулевого указателя, переполнение буфера, ошибки управления памятью и ресурсами, использование неинициализированных переменных, ошибки многопоточных примитивов, недостижимый код), а также многих видов ошибок кодирования, некорректного использования программных интерфейсов и т.п.

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.

Раздел 4.1 посвящен детекторам ошибок уровня АСД и внутрипроцедурного потока данных. Для языков Си/Си++ рассмотрены детекторы

ALLOC_SIZE_MISMATCH / MEMSET_SIZE_MISMATCH (ищет несоответствия между базовым типом указателя, по которому сохраняется выделенная память, и объемом этой памяти); INFINITE_LOOP (пытается найти циклы, которые из-за описки в теле цикла или условии оказываются бесконечными); BAD_SIZEOF (определяет потенциально некорректные использования операции `sizeof`); BAD_OVERRIDE (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); NO_EFFECT/INVARIANT_RESULT (ищут выражения, которые являются либо ненужными, либо константными для всех входных данных, хоть и не выглядят таковыми); BAD_COPY_PASTE (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); CONFUSING_IDENTATION (ищет ситуации, в которых вертикальное выравнивание строк заставляет пользователя предположить структуру потока управления, отличную от той, которая присутствует на самом деле); BAD_ITERATOR.INVALID и BAD_ITERATOR.MISMATCH (предназначены для поиска ситуаций неверного использования итераторов в Си++, а именно, использование итератора после того, как он стал некорректным, либо использование итератора неверного базового типа); WRONG_ARGUMENTS_ORDER (ищет ситуацию вызова некоторой функции, в котором предположительно перепутан необходимый порядок аргументов); UNREACHABLE_CODE (занимается поиском недостижимого кода). Приведены примеры кода, краткие алгоритмы детекторов и тип детектора по предложенной классификации. Также описываются детекторы уровня АСД для языков Java и C#, которые частично пересекаются с детекторами Си/Си++. Рассмотрены важные специфичные для этих языков детекторы (например, детектор WRONG_LOCK_OBJECT для языка Java, который ищет операторы `synchronize(object)` для случаев, когда типом переменной `Object` является один из типов-оберток над примитивными типами; WRONG_LOCK.STATIC для языка C#, для которого внутри блоков синхронизации по некоторому не статическому объекту происходит доступ к статическому полю класса; RETURN_USING для языка C#, который находит возвращение из метода объекта внутри блока using, объявленного в заголовке этого блока и некоторые другие).

В разделе 4.2 описываются межпроцедурные детекторы ошибок второго уровня основного движка Svace. Приводятся инструкции внутреннего представления и список *событий анализа*, которые генерируются ядром при выполнении интерпретации функции. Детектор выбирает, о каких событиях требуется его оповещать вызовом соответствующих функций. 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.

Рассматриваются часто используемые типы атрибутов: двоичный атрибут, выражающий must-отношение (верно на всех путях) или may-отношение (верно на некоторых путях); троичный атрибут со значениями "нет-может быть-да"; атрибут с двумя промежуточными значениями "уверенности" в описываемом событии (событие зависит от истинности условия, которое на может быть оценено анализатором, или событие произошло, но условие его наступления никак не контролируется функцией); атрибуты, содержащие целочисленный интервал; атрибуты, содержащие класс значений, с которым у данного КЗ есть некоторая связь; атрибуты, хранящие условие, при котором наступает событие; варианты описанных атрибутов, также отслеживающие трассу своих изменений (в трассу добавляется новая точка программы при заданном детектором изменении значения атрибута).

Детекторы разыменования нулевого указателя содержат следующие варианты: детектор DEREF_OF_NULL ищет ситуации, в которых разыменовывается указатель, сравненный с NULL (с положительным результатом) на всех путях выполнения, приходящих в точку разыменования; детектор DEREF_OF_NULL.CONST, который выдает предупреждения, если указателю была явно присвоена константа NULL (вместо сравнения); 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 (в отличие от DEREF_OF_NULL, здесь нет полной уверенности в том, что указатель нулевой, и приходится отсеивать менее вероятные ошибочные ситуации в отдельные подтипы детектора); детектор NULL_AFTER_DEREF, который ищет "перевернутую" относительно DEREF_AFTER_NULL ситуацию – указатель, который сначала был разыменован, впоследствии сравнивается с нулем.

Группа детекторов использования памяти после освобождения включает в себя 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.

Детекторы MEMORY_LEAK и HANDLE_LEAK отвечают за потенциальную утечку ресурсов. 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 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. Они состоят из нескольких отдельных анализов. Основной частью является алгоритм поиска достижимых объектов, начиная с некоторого стартового множества. Если

находятся объекты, на которые нет ссылок, 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. Также нужно учитывать не только указатели, покидающие область видимости, но и на указываемую через них память.

Группа детекторов с именами, начинающимися на TAINTED, связана с проверкой корректного использования данных, полученных из ненадежных источников (в частности, от пользовательского ввода). 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.

Детектор STATIC_OVERFLOW предназначен для поиска простых переполнений буфера, которые происходят всегда с константным индексом и размером массива. Предупреждение выдается, когда вычисленный интервал индекса (с учетом смещения) превосходит интервал размера или меньше нуля. 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[]`. Детектор NONTERMINATED_STRING предназначении для ситуаций, когда после копирования строки (функции `strncpy/memcpy`) в строке-результате не будет последнего нулевого символа. Детектор UNINIT.CTOR специфичен для Си++ и выдает предупреждения для полей класса, которые не инициализированы в конструкторе класса. Для поиска ошибок синхронизации в Си++ и Java реализованы детекторы DEADLOCK (взаимные блокировки) и NO_LOCK.STAT (потенциальные состояния гонки). Для C# реализован набор детекторов отслеживания небезопасных значений на основе IFDS-инфраструктуры.

Раздел 4.3 посвящен чувствительным к путям детекторам. Детектор BUFFER_OVERFLOW.EX для поиска ситуаций переполнения буфера 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 $\exists p \in P\colon (\exists \vec{x} : \left(RC_q^p(\vec{x})\right) \wedge \forall \vec{x}\colon (RC_q^p(\vec{x}) \to i(\vec{x}) \notin [0, S(\vec{x}) - 1])))$, 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, но и более строгой формулы (то есть из нее 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.

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 $(a - b) \wedge \left(\exists\tilde{a}\exists\tilde{b}\; LB(a,\tilde{a}) \wedge HB\big(b,\tilde{b}\big) \wedge \big(\tilde{a} - \tilde{b} \geq x\big)\right)$, так как верно, что $a \geq \tilde{a}\ \wedge\ b \leq \tilde{b}\ \wedge\ \tilde{a} - \tilde{b} \geq x \ \Rightarrow\ a - b \geq x$. Для сравнения $v > expr$ условие записывается как $(v > expr) \wedge HB(expr, x - 1)$. Для слияния формула полностью аналогична формулам отслеживания предикатов КЗ, 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.

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 (отдельно) 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.

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.

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 его более консервативный вариант не сработал, проверяет на совместность собранное условие равенства нулю и условие текущей точки, а также дополнительно проверяет, что для данного указателя предупреждение еще не было выдано (полезно для сокращения вызова SMT-решателя). 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, 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.

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, который хранит условия равенства нулю класса значений, к которому он привязан, а также обычный двоичный атрибут, который равен истине в случае, когда КЗ выступал знаменателем и его интервал значений содержал ноль, и атрибуты, наличие которых позволяет предположить, что значение может быть равно нулю. 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. 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 атрибут, означающий, что переменная обязательно освобождалась, и если вызываемой функции передается указатель на глобальную переменную, то также выдается предупреждение.

Группа детекторов UNINIT ищет использования неинициализированных переменных. Эта ошибка является одной из самых интересных и сложных для поиска. Причины 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. 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.

Для анализатора C# основными чувствительными к путям детекторами являются детекторы разыменования нулевого указателя и утечек ресурсов. Детектор разыменования нулевого указателя устроен подобно соответствующим детекторам основной части Svace. 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.

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.

Пятая глава содержит экспериментальные результаты применения инструмента Svace к промышленному исходному коду, которые показывают необходимую масштабируемость инструмента и качество анализа. 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 диссертационной работе А.Е. Бородина (раздел 1.5.5) 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.

Раздел 5.1 посвящен тестированию контролируемой сборки. 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.

В разделе 5.2 оценивалась время сборки и анализа проекта, а также объем потребляемой анализатором памяти. Был 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.

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 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. 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 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.

Скорость анализа для больших проектов (Android и Tizen) 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. Для ОС Windows скорость анализа в целом аналогична замеренной на Linux.

Для анализатора C# скорость несколько меньше, т.к. основной движок анализа выполняет анализы всех уровней (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.

Раздел 5.3 содержит данные о качестве 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.

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++ (таблица 1) 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.

Интересно также, что для ОС Tizen доля истинных срабатываний ниже, чем для ОС Android. 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).

Таблица 1. Результаты анализа для ОС Android 5.0.2 (языки Си и Си++).

Колонка "детектор" задает название конкретного детектора ошибки, "всего" – количество найденных этим детектором ошибок на всем исходном коде, "просмотренных" – количество предупреждений, проанализированных вручную, "истинных" – количество истинных предупреждений из просмотренных, "доля" – процент истинных срабатываний по отношению к просмотренным.

Детектор	**Всего**	**Просмот-ренных**	**Истин-ных**	**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%**

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

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

В результате проведения теоретических и практических 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).

ПУБЛИКАЦИИ ПО ТЕМЕ ДИССЕРТАЦИИ

Статьи 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 статьи, материалы конференций, свидетельства о регистрации

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.