МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Э. БАУМАНА

На правах рукописи
УДК 004.4'2

Сидорин Алексей Васильевич

НАЗВАНИЕ ДИССЕРТАЦИОННОЙ РАБОТЫ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин,
комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
к.т.н., доцент
Романова Т. Н.

Москва — 2015

Оглавление

Введение

Для больших и сложных программно-технических комплексов полное покрытие всех путей выполнения программы становится невозможным, поскольку эта задача соотносится с проблемой останова. Ресурсы, выделенные на тестирование сложных программных комплексов, всегда ограничены, что приводит к необходимости рационального их использования. Проблема поиска подходящего компромисса между повышением надежности разрабатываемых программных средств и эффективным использованием ресурсов становится все актуальнее. Для обеспечения надёжности программных средств активно ведётся разработка новых эффективных методов и средств автоматического тестирования, позволяющих за реальное время предупредить и выявить как можно большее количество дефектов в программе. В настоящее время всё большее распространение получают инструменты, предназначенные для поиска дефектов в программном коде.

Обычно различают статический, динамический и смешанный анализ. Под статическим анализом понимают анализ программы, не требующий её непосредственного выполнения. Часть инструментов, таких, как Clang Static Analyzer [1], PVS-Studio [2], Cppcheck [3], Lint [4], исследует непосредственно код программы или структуры данные, строящиеся на его основе, — абстрактное синтаксическое дерево или граф потока управления. Другая часть инструментов статического анализа использует для анализа более низкоуровневое представление программы — скомпилированный объектный или промежуточный код (Coverity Prevent [5], Svace [6], FindBugs [7]). В отличие от статического анализа, для динамического анализа программы требуется её выполнение — на специальных входных данных, в виртуальной машине (Valgrind [8]), с использованием инструментации (AddressSanitizer [9], ThreadSanitizer [10], UndefinedBehaviorSanitizer), с использованием дополнительных библиотек или их подменой. Наконец, смешанный анализ представляет собой комбинацию статического и динамического анализа и используется в таких инструментах как Mayhem [11], KLEE [12], а также других автоматических генераторов контрпримеров.

Перечисленные виды анализа имеют свои достоинства и недостатки, в частности, различные виды анализа наиболее эффективны для поиска различных видов ошибок.

– Динамический анализ наиболее хорошо зарекомендовал себя для поиска ошибок, связанных с многопоточностью и управлением памятью, однако крайне затратен в случае больших проектов. Значительным недостатком динамического анализа является необходимость явного выполнения программы, что влечёт за собой необходимость подготовки входных данных (или их автогенерации), и быстрый рост длительности такого анализа с увеличением объёма проекта. Это также затрудняет интеграцию инструментов, использующих динамический анализ, в процесс разработки, что снижает шансы быстрого обнаружения ошибки.
– Статический анализ позволяет эффективно производить поиск различных видов дефектов: опечаток, некорректного использования типов, проблем безопасности, неопределённого или недокументированного поведения и многих других видов. Инструменты для выполнения статического анализа могут быть легко интегрированы в процесс разработки. При этом они могут быть использованы как индивидуальные вспомогательные инструменты разработки (например, для подсветки кода, содержащего потенциальную ошибку), так и в качестве инструментов, использующихся группой разработчиков (например, для развёртывания и интеграции в систему непрерывной сборки). Сравнительно небольшое время, затрачиваемое на анализ, вкупе с интеграцией в рабочий процесс позволяет быстро находить дефекты в разрабатываемых программах. Недостатком статических анализаторов является возможность выдачи ими некорректных сообщений об ошибках — ложных срабатываний (ошибок первого рода) и возможность пропуска имеющихся дефектов (ошибки второго рода), вероятность которых стараются снизить при разработке анализаторов. Вред от ошибок второго рода очевиден, но и ошибки первого рода играют не меньшую роль при оценке качества анализатора, поскольку их большое количество отвлекает разработчика на длительное время для просмотра ложных срабатываний, поэтому при большом количестве ложных срабатываний инструмент может стать практически непригодным для использования. Од-

нако при небольшом количестве ложных срабатываний польза от применения анализатора в виде снижения времени, затрачиваемого на обнаружение ошибки, быстро перевешивает недостаток в виде времени, затрачиваемого на просмотр ложных срабатываний.

Первоначально распространение у разработчиков получили инструменты, использующие методы на основе анализа синтаксического дерева программы и её графа потока управления. Преимуществами данных методов анализа программного кода являются:

- высокая скорость работы,
- незначительное потребление памяти,
- возможность его реализации в компиляторе для выполнения дополнительных проверок и предупреждения программиста о потенциально некорректном поведении компилируемого кода. Это становится возможным благодаря малому потреблению системных ресурсов, позволяющему лишь незначительно снижать производительность компилятора,
- возможность интеграции в среды разработки для осуществления анализа «на лету», непосредственно в процессе набора кода программистом, или в качестве дополнительного инструмента для быстрого обнаружения дефекта.

Аналогичные методы применяются в компиляторах для предупреждения программиста о потенциально некорректном поведении программы, поскольку и синтаксическое дерево, и граф потока управления являются основными структурами данных, с которыми работает компилятор. Однако проверка, включаемая в состав компилятора, должна исключать возможность ложных срабатываний, т. е. являться консервативной. Инструменты же статического анализа могут включать также и неконсервативные проверки, с возможностью выдачи ложных срабатываний.

Данные методы могут обнаруживать лишь очень узкие классы дефектов в программном коде: простые ошибки, затрагивающие лишь несколько операторов, расположенных в пределах одной функции. Это может быть простейший поиск использования неинициализированных переменных, ошибок при преобразовании типов, потенциально лишние операции, а также другие дефекты, для поиска которых не требуется анализировать циклы и условные переходы. При наличии циклов и переходов в анализируемой функции эффективность ви-

дов анализа, нечувствительных к путям выполнения, резко падает, поскольку данные методы позволяют корректно определить достижимость одних операторов из других операторов при выполнении программы лишь в тривиальных случаях.

Значительно более ресурсоёмким, но и более подробным является анализ на основе обхода путей выполнения программы. Основы этих методов были заложены ещё в 70-х годах. Метод символьного выполнения был предложен Джеймсом Кингом в 1976 году [13]. В основе метода лежит идея разбиения входных данных на классы эквивалентности в зависимости от встречаемых по пути выполнения условий. Метод абстрактной интерпретации, предложенный в 1977 году супругами Кузо [14], предполагает использовать абстрагирование данных и их анализ на основе алгебры решёток. Однако данные походы стали получать распространение только в последнее время. Это связано с увеличившейся мощностью компьютеров: время анализа растёт пропорционально количеству путей выполнения, что означает экспоненциальный рост времени анализа с увеличением размера программы. (Вообще говоря, абсолютно полный и точный анализ программы невозможен в связи с проблемой останова, независимо от применяемого подхода.) В отличие от базового анализа графа потока управления, анализ путей выполнения способен учитывать условия выполнения тех или иных ветвей программы, следствием чего являются преимущества данного вида анализа — его более высокая точность и способность покрыть намного больший класс дефектов. Такие методы, как абстрактная интерпретация и символьное выполнение, нашли применения в известных инструментах для поиска дефектов, например, Coverity SAVE, Clang Static Analyzer и многих других.

Одними из наиболее актуальных целевых языков для статического анализа традиционно являются языки C и C++. Причин для этого несколько. Во-первых, это связано с большим количеством видов потенциальных ошибок, которые может допустить программист, ведущий разработку с использованием этих языков. Наиболее специфичными среди таких ошибок являются ошибки, связанные с неправильной работой с указателями — переполнение буфера, обращение к неинициализированной памяти или к памяти по некорректному адресу. Во-вторых, стандарты языков трактуют достаточно большое количество ситуаций как не имеющих определённого поведения (например, порядок вычисления

аргументов функций может быть произвольным), что, с одной стороны, позволяет компилятору проводить более глубокие оптимизации и получить наибольшую скорость выполнения результирующего кода, но, с другой стороны, требует от программиста повышенного внимания в процессе написания кода программы для учёта этих особенностей. В-третьих, эти языки являются одними из самых распространённых и известных, с их использованием было разработано и продолжает создаваться большое количество как системного, так и прикладного программного обеспечения. Кроме того, язык C является практически единственным выбором при разработке низкоуровневых компонентов, например, компонентов операционных систем и драйверов, что также предъявляет повышенные требования к качеству программного кода.

Целью данной работы является разработка метода межпроцедурного межмодульного анализа крупных программных комплексов, разработанных с использованием языков C и C++, способного осуществлять анализ проектов масштаба ОС Android и ОС Tizen за приемлемое время и обеспечивающего достаточное покрытие путей выполнения программы.

Для достижения поставленной цели необходимо было решить следующие задачи:

1. Разработать метод межпроцедурного анализа программ с высокой масштабируемостью, пригодный для анализа крупных программных проектов, а также расширяемый на различные классы проверок
2. Разработать метод межмодульного анализа программ, разработанных с использованием языков C и C++
3. Разработать метод отображения результатов анализа при использовании разработанного метода межпроцедурного анализа
4. Реализовать разработанные методы с использованием инфраструктуры статического анализатора Clang Static Analyzer
5. Осуществить проверку разработанных методов на реальных программных проектах
6. Провести анализ разработанного метода на предмет масштабируемости и качества анализа с учётом результатов, полученных при проверке реальных программных проектов.

Основные положения, выносимые на защиту:

1. Метод межпроцедурного анализа программ на основе резюме для метода символьного выполнения для программ, разработанных с использованием языков C и C++
2. Метод межмодульного анализа программ, разработанных с использованием языков C и C++, архитектура анализатора, эвристики, связанные с объединением синтаксических деревьев различных модулей трансляции
3. Метод построения отчёта о дефекте при использовании метода резюме для метода символьного выполнения

Научная новизна:

1. Разработан и подробно описан метод межпроцедурного анализа программ на основе резюме для метода символьного выполнения для программ, разработанных с использованием языков C и C++
2. Разработан и подробно описан метод межмодульного анализа программ, разработанных с использованием языков C и C++ для статического анализатора, использующего в качестве входных данных непосредственно исходный код программы
3. Разработан метод построения отчёта о дефекте при использовании метода резюме для метода символьного выполнения
4. Вышеперечисленные методы реализованы с использованием инфраструктуры статического анализатора Clang Static Analyzer и апробированы на реальных проектах (ОС Android)

Научная и практическая значимость. Предложена новая модификация метода межпроцедурного анализа на основе резюме для метода символьного выполнения, а также метод межмодульного анализа и архитектура анализатора, использующего в качестве входных данных непосредственно исходный код программы. Разработанные методы применимы для проектов масштаба операционных систем и их наборов пользовательских приложений. Программное обеспечение, реализующее разработанные методы, внедрено в Samsung Electronics и используется для анализа исходного кода ПО различного назначения, в частности, мобильных приложений и операционных систем, телевизионное ПО, ПО медицинских систем и т. д.

Степень достоверности полученных результатов обеспечивается . . . Результаты находятся в соответствии с результатами, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на: перечисление основных конференций, симпозиумов и т. п.

Личный вклад. Автор принимал активное участие . . .

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях [15–20], 4 из которых изданы в журналах, рекомендованных ВАК [15–18], 2 — в тезисах докладов [19; 20].

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и двух приложений. Полный объём диссертации составляет 123 страницы с 14 рисунками и 10 таблицами. Список литературы содержит 71 наименование.

1 Методы статического межпроцедурного анализа программ

1.1 Методы обеспечения качества ПО. Статический анализ

Одной из основных задач, возникающих при создании программного продукта любой сложности, является обеспечение его качества. Для определения соответствия разрабатываемого программного продукта критериям качества в состав жизненного цикла программного обеспечения включают стадии или процессы верификации и валидации программных средств [21; 22]. Верификация, согласно [23] — это процесс для определения, выполняет ли программный комплекс и его компоненты требования, наложенные на них в последовательных этапах жизненного цикла комплекса программ. В водопадной модели жизненного цикла программного обеспечения (ПО) стадия верификации и тестирования непосредственно следует за стадией реализации программного продукта [24] (рис. 1.1); в итеративной модели стадия тестирования также следует за реализацией, однако циклически повторяется по мере разработки системы. Ряд методологий, например, разработка через тестирование [25], подразумевает интеграцию тестирования с процессом разработки.

Рисунок 1.1 — Жизненный цикл программы в каскадной модели разработки и место в нём тестирования

Целью верификации является подтверждение соответствия конечного программного продукта предопределённым эталонным требованиям, для чего осуществляется обнаружение, регистрация и устранение дефектов, внесённых при разработке или модификации программных комплексов и компонентов или требований к ним. Для повышения результативности издержек верификация должна быть объединена как можно раньше с процессами проектирования, производства и сопровождения [23].

Вместе с тем, отдельной стадии тестирования, как правило, недостаточно для построения безопасного комплекса с низким количеством дефектов. В

связи с эти для построения безопасных программных продуктов используются специальные методологии разработки, в которых требования к безопасности и обеспечению высокого качества программного продукта предъявляются на каждой из стадий разработки. Одной из наиболее известных методологий безопасной разработки ПО является Microsoft Security Development Lifecycle (Microsoft SDL, жизненный цикл защищённой разработки Microsoft) [26]. Диаграмма жизненного цикла ПО согласно данной методологии представлена на рисунке 1.2.

Рисунок 1.2 — Жизненный цикл защищённой разработки Microsoft (Microsoft SDL)

Данная методология предполагает использование 17 различных практик, применяемых на различных стадиях создания программного продукта, для повышения качества и надёжности разрабатываемого ПО:

1. *Обучение участников разработки* основам информационной безопасности и поддержка их знаний в актуальном состоянии. Необходимые навыки включают безопасное проектирование, моделирование угроз, безопасную разработку, тестирование безопасности и обработку конфиденциальных данных.
2. *Разработка требований безопасности* и их анализ. Определение требований безопасности на ранних стадиях позволяет осуществлять их реализацию неотъемлемо от разработки программного продукта.
3. *Создание стандартов качества*, устанавливающих минимально допустимый уровень безопасности и конфиденциальности. Установка требований качества на начальной стадии создания ПО улучшает восприятие рисков безопасности и стимулирует устранение дефектов безопасности в процессе разработки. Кроме того, стандарты качества задаются для

каждой стадии создания ПО, что позволяет контролировать процесс появления и устранения дефектов.

4. *Оценка рисков безопасности и конфиденциальности* определяет, какие элементы системы должны подвергаться дополнительным видам тестирования и верификации, и какому риску они могут подвергнуться в процессе эксплуатации.
5. *Определение требований проектирования* включает создание проектных спецификаций возможностей безопасности и конфиденциальности, с которыми непосредственно взаимодействует пользователь, а также описаний методик разработки функциональных возможностей с учётом требований безопасности.
6. *Анализ атак* позволяет уменьшить риск предоставления атакующему возможностей для эксплуатации потенциально слабого места или уязвимости и включает ограничение доступа к системным сервисам, применение принципа наименьших привилегий и использование многослойной защиты.
7. *Моделирование угроз* позволяет структурированно учитывать, документировать и обсуждать влияние требований безопасности на разработки, что, в свою очередь, позволяет рассмотрение проблем безопасности на уровне компонента или приложения.
8. *Использование одобренных инструментов* подразумевает определение и использование списка инструментов разработки, одобряемого консультантом по вопросам безопасности, и их возможностей проверок безопасности, таких как опции и предупреждения компилятора или компоновщика.
9. *Избежание небезопасных функций* означает запрет на использование потенциально небезопасных и устаревших функций и интерфейсов с их заменой на безопасные альтернативы и проверки кода на их использование.
10. ***Статический анализ*** предоставляет возможность масштабируемого аудита безопасности программного кода и соблюдение политик безопасной разработки.

11. ***Динамический анализ*** подразумевает верификацию времени выполнения. Он необходим для проверки соответствия функциональности программы проектной.
12. ***Нечёткое тестирование (fuzz testing)*** проверяет программу на устойчивость к повреждённым, случайным или злонамеренно искажённым данным.
13. *Обзор векторов атак* представляет собой пересмотр модели угроз и векторов атак по окончании разработки кода и необходим для проверки отсутствия появления или устранения обнаруженных новых векторов атак в результате изменений системы в процессе разработки.
14. *План реагирования на инциденты* перечисляет персонал, ответственный за оперативное устранение обнаруживаемых уязвимостей, и содержит планы обслуживания кода, внешнего по отношению к команде разработчиков.
15. *Окончательная проверка безопасности* является последовательной проверкой всех аспектов безопасности продукта перед его выпуском и проводится с целью идентификации и устранения дефектов и уязвимостей, внесённых на различных стадиях разработки ПО.
16. *Архив выпусков продукта* используется для обслуживания ПО после его выпуска и включает в себя, помимо непосредственно программного кода, все спецификации, двоичные файлы, модели угроз, документацию, лицензии, планы оперативного реагирования и прочую информацию, которая может понадобиться для обслуживания выпущенного ПО.
17. *Инспекции кода* специалистами в области безопасности подвергаются критические с точки зрения безопасности компоненты разрабатываемой системы. Инспекция кода проводится с целью проверки корректности работы с конфиденциальными данными и поиска дефектов при реализации криптографических алгоритмов.

Как можно заметить, три практики предусматривают непосредственное использование специализированных инструментальных средств тестированя и верификации. Другие существующие практики безопасной разработки, такие как TSP-Secure (Team Software Process for Secure Development) [27], также вклю-

чают использование инструментальных анализаторов исходного или бинарного кода.

Целью инструментального анализа кода программы является определение наличия в нём *ошибок*. В широком смысле под ошибкой (или *дефектом*) понимается неправильность, погрешность или искажение объекта или процесса [28]. Анализаторы обычно оперируют более узким понятием дефекта, понимая его как закодированную в программе ошибку или упущение человека [29]. Данное определение подразумевает описание свойства программы, которое необходимо анализатору для поиска дефекта. Для многих видов дефектов, встречаемых при разработке программ, существуют описания и классификация. Наиболее известными среди них являются каталоги Common Weaknesses Enumeration (CWE) [30], Computer Emergency Readiness Team (CERT) [31], Motor Industry Software Reliability Association (MISRA) [32]. Существуют также отраслевые и проектные стандарты, включающие список возможных нарушений, такие как JSF C++ Coding Standards [33].

Поскольку автоматизированный поиск дефектов производится с использованием формальных методов, для поиска дефектов обычно недостаточно описания. На основе описания класса дефектов строится *критерий дефектности* — формальное описание класса дефекта или его подмножества, позволяющее производить автоматический поиск. Для автоматизации поиска используется программа, называемая *анализатором*. Критерий дефектности обычно формулируется в терминах *модели анализатора* — описания анализатора и его принципа работы, учитывающего особенности его реализации. Задачей анализатора обычно является построение внутренних структур данных на основе кода программы, проверка полученных структур на соответствие заданному критерию дефектности и выдача диагностических сообщений в случае обнаружения соответствия.

К анализаторам программ предъявляется следующий ряд требований, зачастую противоречащих друг другу [34]:

- *корректность* — основное свойство анализа, означающее возможность обнаружения дефекта, если он присутствующет в коде программы
- *точность* означает отсутствие ложных срабатываний: любое срабатывание точного анализатора должно означать наличие дефекта в программе

– *масштабируемость* означает возможность анализа реальных, возможно, крупных программ, разработанных с использованием промышленных языков программирования
– *практичность* означает простоту в использовании, в т. ч., отсутствие необходимости большого количества дополнительных аннотаций и понятность итогового отчёта анализатора.

Срабатывания анализатора и отсутствие срабатывания можно разделить на корректные и ложные. Срабатывание анализатора является *корректным (true positive, TP)*, если отчёт анализатора указывает на действительно присутствующий в коде программы дефект. Срабатывание анализатора является *ложным (false positive, FP)*, или *ошибкой второго рода*, если программный код на самом деле не содержит дефекта, указанного в отчёте анализатора. Отсутствие срабатывание анализатора является *корректным (true negative, TN)*, если программный код, на котором анализатор не выдал диагностическое сообщение, действительно является корректным. Отсутствие срабатывания анализатора является *некорректным (false negative, FN)*, или *пропуском*, если анализатор не выдал диагностическое сообщение при анализе программного кода, содержащего дефект.

Ряд срабатываний анализатора являются положительными с точки зрения соответствия правилу поиска, однако по-каким либо причинам код невозможно изменить так, чтобы он соответствовал правилу кодирования. Такая ситуация может возникнуть, например, при необходимости поддерживать стандартизированный программный интерфейс (например, JNI — Java Native Interface, Posix-функции). Такие срабатывания принято классифицировать отдельной категорией — *вынужденные (Intentional)*.

Основных источников ложных срабатываний несколько. Во-первых, в силу различных ограничений анализатору недоступна вся возможная информация о программе и её окружении. Многие анализаторы производят анализ программы по частям, что дополнительно уменьшает объём доступной анализатору информации. Во-вторых, структуры данных анализатора зачастую строятся с упрощением, поскольку хранение и использование всей возможной информации о программе требует большого количества ресурсов, а время анализа должно быть ограничено. В-третьих, источником ложных срабатываний может являться недостаточно однозначная спецификация дефекта, которая может включить

образец поиска в список дефектных, хотя он таковым не является. В-четвёртых, некорректная реализация проверки (ошибка при реализации или неучёт каких-либо условий) также может приводить к ложным срабатываниям.

Из-за проблемы разрешимости невозможно объединить в анализаторе абсолютнубю точность и абсолютную корректность анализа. Как следствие, многие алгоритмы анализа включают *консервативные* аппроксимации: в зависимости от выбираемой разработчиком анализатора стратегии, анализатор или выдаёт ложные срабатывания, не допуская пропусков (при условии, что ложных срабатываний не слишком много), или допускает пропуски, не допуская, по возможности, ложных срабатываний. Точность также влияет на масштабируемость: обычно лучшая точность означает меньшую масштабируемость, а быстрые алгоритмы обычно не являются точными. В сфере анализа программ был разработан большой набор техник, комбинируя которые разработчик может выбирать между вычислительно дешёвыми и неточными и точными, но ресурсоёмкими алгоритмами, в зависимости от цели анализа. В частности, ряд свойств анализаторов выделяют в зависимости от их мощности.

- *Чувствительность к потоку (flow-sensitivity)* подразумевает свойство анализатора учитывать порядок операторов в программе и возможность вычислять различную информацию для различных операторов программы; анализ, нечувствительный к потоку, вычисляет только одно глобальное решение для всей программы.
- *Чувствительность к пути (path-sensitivity)* подразумевает свойство анализатора учитывать возможные пути выполнения программы и их достижимость, получая для этого информацию из условий операторов ветвления; контекстно-нечувствительный анализ не различает пути выполнения программы и не учитывает достижимость операторов.
- *Чувствительность к объекту (object-sensitivity* означает возможность различать информацию для одного и того же поля, относящегося к различным объектам класса; объектно-нечувствительный анализ относит информацию о поле ко всем объектам класса, содержащего это поле.
- *Чувствительность к контексту (context-sensitivity)* (применительно к межпроцедурному анализу) означает учёт анализатором контекста вызова функции, т. е. каждый из различных вызовов функции моде-

лируется отдельно; контекстно-нечувствительный анализ не различает вызовы функций между собой.

Обычно с целью повысить независимость проверок друг от друга и отделить их от реализации моделирования языка программирования анализатор архитектурно разделяют на:

1. *фронтэнд* — часть, отвечающую за разбор исходного файла и построение базовых структур данных (синтаксическое дерево, таблица идентификаторов и др.). Зачастую в качестве фронтэнда используется уже существующий компилятор, который строит все структуры данных сам.
2. *ядро анализатора* отвечает за реализацию алгоритма анализа и моделирование операторов языка
3. *проверяющие модули* непосредственно реализуют проверки, используя данные, предоставляемые ядром анализатора, а также предоставляют ядру анализатора информацию, влияющую на моделирование.

1.2 Метод анализа программ с помощью символьного выполнения

В данной работе исследуется метод символьного выполнения [13], применяемый для анализа путей выполнения программ. Этот метод подразумевает абстрактное движение по путям программы, имитирующее её выполнение в зависимости от входных данных, сопровождающееся изменением состояния программы в различных точках. Суть метода символьного выполнения заключается в разбиении множества входных данных на классы эквивалентности, что позволяет оперировать при анализе не отдельными входными значениями (число которых может быть очень большим и экспоненциально растёт в зависимости от количества входных аргументов) и их перебором, а целыми классами эквивалентности, число которых может оказаться и не конечным, но не превышает общее количество комбинаций отдельных входных значений. Однако, как правило, количество классов эквивалентности комбинаций входных данных оказывается значительно ниже числа всех возможных комбинаций входных данных, что резко увеличивает возможности анализатора по обработке путей выполнения.

Основной алгоритм символического выполнения [13] заключается в следующем.

1. При старте анализа функции входные значения её аргументов и внешних по отношению к ней переменных неизвестны, т. е. они потенциально могут принимать любые значения. Начальное состояние является корневым узлом специального графа — дерева выполнения программы.
2. Каждой неизвестной величине назначается абстрактное значение, называемое символьным значением.
3. Обработка операторов языка изменяет значения переменных программы, т. е. их символьные значения. Считается, что над символьными значениями уже определён необходимый набор операций для вычисления новых символьных значений на основе уже имеющихся. Выполнение каждого оператора добавляет узел к дереву выполнения программы с входящим ребром от предыдущего оператора.
4. При обработке условных операторов в дерево выполнения программы добавляется не один, а два узла: в первом узле условие выполняется, во втором — нет. Совокупность условий, при которых достижим данный узел графа выполнения программы, определяет класс эквивалентности входных данных программы или функции. Таким образом, каждый лист дерева выполнения программы соответствует классу эквивалентности входных данных, которая приводит программу в конечное состояние, обозначаемое данным листом.
5. Обычно каждое из условий, определяющих класс эквивалентности входных данных, можно представить в виде уравнения или неравенства. В результате наложения на путь выполнения программы нескольких условий в соответствие каждому классу эквивалентности входных данных ставится система уравнений или неравенств. Решениями этих систем уравнений являются множества реальных значений входных величин, при которых будут выполнены ветви выполнения программы. Если система является несовместной, т. е. не имеет ни одного решения, то путь выполнения, соответствующий этой системе, недостижим.
6. Анализатор производит обход всех путей получившегося дерева выполнения с целью поиска ситуаций, которые могли бы трактоваться как некорректное поведение. В случае обнаружения такой ситуации

анализатор сообщает о дефекте и при этом указывает набор условий, при выполнении которых программа проявит некорректное поведение.

Данный алгоритм можно рассмотреть на следующем примере. Пусть имеется следующая программа чтения из файла на языке C:

```
1  int test(int a, int b) {
2    FILE *f = fopen("file.txt");
3    int result;
4    fscanf(f, "%d", &result);
5    fclose(f);
6    if (a == 0 && b > 2) {
7      fclose(f);
8      return 0;
9    }
10   return result;
11 }
```

В результате выполнения этой программы будут достижимы три конечных состояния с уравнениями $a \neq 0, b \in [INT\_MIN, INT\_MAX]$; $a = 0, b \leqslant 2$; $a = 0, b > 2$. Соответствующий граф представлен на рис. 1.3.

В результате получены три пути с соответствующими ограничениями. Теперь анализатор просматривает каждый из этих путей и обнаруживает дефект: на одном из путей файл `f` закрывается дважды, что приводит к неопределённому поведению. Этому пути соответствует система $a = 0, b > 2$. Таким образом дефекту сопоставляется множество условий, при котором он проявляется при выполнении программы.

Основным преимуществом метода символьного выполнения является простота и очевидность концепции, на которой он основан: метод использует идею «симуляции» выполнения программы, так, как это делает программист. Метод символьного выполнения получил распространение не только в инструментах статического, но и смешанного анализа: так, хорошо зарекомендовали себя инструменты, использующие подход concolic testing [35] (символьно-конкретное — concrete+symbolic). Concolic testing — это метод поиска дефектов, осуществляющий генерацию тестовых данные, при использовании которых программа проявляет некорректное поведение, на основе символьного выполнения.

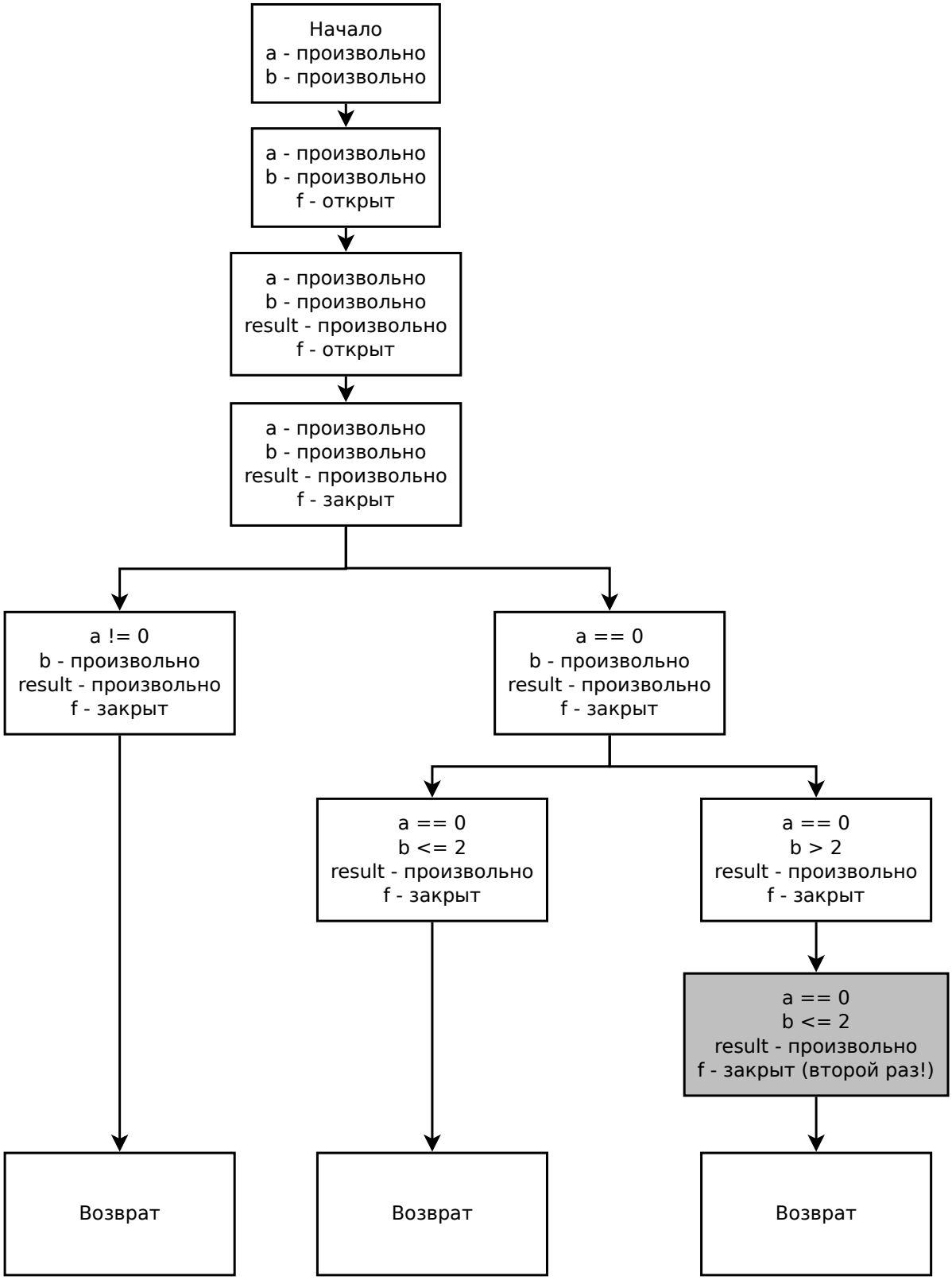

Рисунок 1.3 — Граф выполнения программы чтения из файла

Наряду с преимуществами, метод имеет ряд недостатков. Так, существует проблема экспоненциального роста количества проходимых путей (path explosion), приводящая к проблемам с масштабируемостью метода. Есть также проблемы при моделировании циклов, поскольку зачастую количество итераций цикла точно неизвестно — оно также является символьной величиной. Тем не менее, метод символьного выполнения активно применяется, в том числе, целым набором широко используемых инструментов анализа программ. Таким образом, разработка подходов для улучшения данного метода является актуальной и практически важной задачей.

Основные проблемы масштабируемости метода связаны с двумя факторами.

1. При моделировании циклов время анализа линейно зависит от количества итераций, проходимых программой при выполнении цикла. Даже если число итераций известно, но велико, анализ программы выполняется длительное время, резко возрастающее при наличии в программе вложенных циклов. При использовании смешанного анализа обычно эту проблему анализом выполнения программы для установления реального количества итераций. При статическом анализе наиболее распространённым решением является ограничение количества итераций циклов каким-либо максимальным константным значением. Этот подход позволяет ограничить время анализа, однако не решает проблему роста времени анализа при наличии вложенных циклов. Кроме того, это ограничение приводит к потере точности моделирования, что, в свою очередь, приводит к ложным срабатываниям или отсутствию срабатываний в тех случаях, когда они ожидаются.
2. Время анализа быстро растёт при использовании межпроцедурного анализа.

Метод символьного выполнения позволяет реализовывать анализ, чувствительный к потоку управления, к пути выполнения, и объектно-чувствительный анализ.

Отличие межпроцедурного анализа (МПА) от внутрипроцедурного (ВПА) заключается в том, что анализатор позволяет использовать доступные определения пользовательских функций для моделирования эффектов их вызовов. МПА используется для решения двух основных проблем, связанных с использо-

ванием внутрипроцедурного анализа. Во-первых, межпроцедурный анализ позволяет определить эффект, оказываемый на состояние программы в результате вызова из анализируемой функции другой функции. В отсутствие межпроцедурного анализа вызов функции можно моделировать либо самостоятельно (с помощью спецификаций эффектов), либо приближённо. Первый подход, как правило, используется для функций, эффекты которых специфицированы. Таковы, например, функции различных публичных интерфейсов взаимодействия. Наиболее распространено такое моделирование для Posix API, встречаются также реализации, моделирующие вызовы Windows API. Кроме того, хорошими кандидатами на специфицирование являются функции, принадлежащие стандартной библиотеке языка, поскольку она, как правило, стандартизирована, реже — функции других распространённых библиотек (например, STL и др.).

Во-вторых, в случае, если полная спецификация функции недоступна, при внутрипроцедурном анализе может использоваться приближённое моделирование. В этом случае считается, что вызов функции может произвести любые действия с данными, которые доступны внутри функции (для языков, имеющих операции арифметики с указателями, например, C/C++, в общем случае можно считать, что программа может модифицировать любые данные), и вернуть произвольное значение. Для уточнения эффектов может использоваться анализ атрибутов доступной декларации функции, например, информация о модификаторах типов аргументов функции, атрибуты аргументов и самой функции. Так, например, функция, объявленная с GNU-атрибутом `__attribute__((pure))`, не имеет прав на изменение глобальной памяти и аргументов, а функция с атрибутом `__attribute__((noreturn))` никогда не вернёт управление в вызывающую функцию. Могут также использоваться различные эвристики. Вместе с тем, приближённое моделирование может решить проблему анализа лишь частично. Из-за невозможности оценки влияния вызова на состояние программы анализатор может сделать некорректные выводы о текущем состоянии выполнения программы, что может привести как к ложным срабатываниям анализатора (ошибка первого рода), так и к отсутствию срабатывания в условиях, когда анализатор должен выдавать диагностическое предупреждение (ошибка второго рода).

1.3 Межпроцедурный анализ для метода символьного выполнения

Межпроцедурный анализ при использовании метода символьного выполнения обычно выполняется с использованием метода встраивания [36]. Метод встраивания заключается в моделировании операторов вызываемой функции оператор за оператором, в порядке, определяемом потоком управления функции. В начале моделирования вызова устанавливается соответствие имён, внутренних для вызываемой функции, и имён в контексте вызывающей функции: так, значениями формальных параметров вызываемой функции становятся значения её фактических аргументов в вызывающей функции, объектам ключевых слов `this` или `self` ставится в соответствие объект, вызов метода которого моделируется, и т. д.. Затем функция моделируется так же, как и вызывающая, формируя дальнейший путь выполнения. По окончании моделирования вызова значение, возвращаемое оператором `return`, становится значением, возвращаемым функцией в контексте вызывающей функции, после чего продолжается моделирование вызывающей функции.

Метод встраивания достаточно прост в реализации и, кроме того, является естественным для смешанного анализа, поскольку соответствует непосредственному выполнению анализируемой программы. Его дополнительное преимущество заключается в сохранении всех видов чувствительностей анализа:

1. этот метод сохраняет контекстную чувствительность, поскольку на значения переменных при моделировании вызова функции накладываются ограничения, существующие в вызывающей функции;
2. метод сохраняет чувствительность к пути выполнения, поскольку при вложенном вызове также учитываются условные операторы внутри функции;
3. метод сохраняет чувствительность к потоку, поскольку операторы функции выполняются в соответствии с потоком управления функции.

Однако метод встраивания имеет недостаток, связанный с масштабируемостью: каждый раз при моделировании вызова функции она анализируется заново, в различных контекстах. Это приводит к большим расходам как процессорного времени, затрачиваемого на анализ, так и к расходам памяти, поскольку информацию, получаемую при моделировании вызова на каждом его шаге,

и зачастую необходимую для последующего анализа, необходимо сохранять в памяти. В результате сравнения времени анализа с использованием межпроцедурного анализа методом встраивания и без использования межпроцедурного анализа были получены значения времени time-1 и time-2 (для тестирования использовался программный комплекс Clang Static Analyzer). Эти данные показывают заметное увеличение времени анализа. При этом время анализа зависело от размера исходного файла следующим образом: графики.

Таким образом, нашей задачей является построение такого метода межпроцедурного анализа для метода символьного выполнения, который будет обладать следующими свойствами:

1. сохранение всех видов чувствительности анализа, т. е. метод МПА должен сохранять контекстную чувствительность, чувствительность к пути и чувствительность к потоку;
2. высокая масштабируемость: разрабатываемый метод должен позволять анализ крупных программных систем за приемлемое время;
3. достаточная точность анализа: реализация разрабатываемого метода не должна уступать методу встраивания в количестве корректных срабатываний (и, по возможности, не допускать пропусков дефектов) и в отношении количества корректных срабатываний к общему количеству срабатываний.

1.4 Улучшения метода символьного выполнения

Низкая масштабируемость метода символьного выполнения в его оригинальном описании является известной проблемой, и многие исследователи пытались решить её различными способами. Для уменьшения времени анализа и улучшения покрытия, а также для решения других возникающих проблем, предлагались различные решения.

Часть исследователей пыталась решить проблему роста количества путей, уменьшая количество ветвлений состояния программы. В этом отношении интересны работы [37; 38]. В этих работах получаемые в результате выполнения пути выполнения программы вместе с соответствующими им классами экви-

валентности по возможности объединяются в один сразу после ветвления. В результате объединения ветвей выполнения анализатором рассматривается не отдельное состояние программы в процессе выполнения, а суперпозиция состояний, получаемых на различных путях выполнения программы. Применение данного подхода уменьшает количество анализируемых путей программы, что позволяет снизить время анализа, однако, поскольку при анализе одного объединённого пути анализируются несколько фактических, анализ происходит без потери анализатором покрытия кода.

Использование событийного подхода при программировании приложения создаёт ряд проблем для анализа программы, поскольку, во-первых, обычно код, отвечающий за низкоуровневую обработку событий находится не в приложении, а в системной или прикладной библиотеке, а, во-вторых, порядок и состав приходящих событий не может быть определён заранее. Это приводит к комбинаторному взрыву вариантов выполнения программы. Попытка решения этой проблемы представлена, в частности, в [39].

Метод символьного выполнения имеет ряд недостатков, связанных с анализом циклов в программе. Во-первых, количество итераций цикла может быть заранее неизвестным, и представлять собой символьное значение, не вычисляемое в константное при произвольном множестве значений входных данных. В этом случае невозможно предсказать или рассчитать количество итераций цикла, которое необходимо проанализировать без потери точности и полноты анализа. Во-вторых, время анализа цикла растёт в зависимости от количества итераций (как правило, линейно), и при значительном количестве итераций цикла анализ программы становится слишком длительным. Эта проблема становится серьёзнее в случае наличия вложенных циклов, для которых время анализа растёт в зависимости от количества итераций каждого из циклов, а также при использовании межпроцедурного анализа, который может значительно увеличивать длину пути выполнения каждой итерации цикла.

Поскольку циклические конструкции являются неотъемлемой частью любой программы, проблемы, связанные с анализом циклов, привлекали исследовательский интерес. Кроме того, циклы обычно используются для обработки контейнерных типов и для работы с рекурсивными структурами данных, поэтому без решения данной проблемы невозможно построить анализатор, пригод-

ный для промышленного использования. Для решения проблем моделирования циклов предлагались и используются различные методы и эвристики.

Первый подход использует моделирование заранее фиксированного количества итераций цикла. Этот подход достаточно хорошо работает в случае, если потенциальное количество итераций невелико, однако проблемы производительности, связанные с вложенными циклами, данный подход решает хуже, и резкий рост времени анализа в этом случае сохраняется, хотя время анализа и ограничивается сверху. Кроме того, ограничение количества итераций приводит к уменьшению покрытия путей программы, что, в свою очередь, может привести к пропуску дефектов. Данный подход применяется, в частности, в Clang Static Analyzer. Другие подходы к анализу циклов предполагают вычисление фиксированной точки цикла или некоторого приближения к ней. Этот метод используется в Coverity Prevent [40], где для приближённого расчёта фиксированной точки выполняется одна итерация цикла с символьными значениями, что позволяет сохранить высокую скорость анализа. Ещё один подход заключается в составлении резюме цикла как оператора и его последующем применении вместо моделирования итераций [41; 42]. В [43] предложен ряд техник для моделирования циклов, затрагивающих вопросы достижимости операторов цикла (в т. .ч., на основе резюме) и работой с памятью в циклах.

Некоторые из предложенных исследователями улучшений затрагивают обработку и анализ отдельных распространённых типов данных. Это касается как самих данных, так и их представления в памяти программы, а также способов их обработки в анализируемых программах. Многие анализаторы поддерживают вычисления лишь с целочисленными данными как с наиболее простыми и однозначно интерпретируемыми. Ряд работ, однако, предлагают механизмы обработки других типов данных — так, в [44; 45] и в других работах рассматривается проблема обработки арифметических операций с плавающей запятой. Попытка решения проблемы моделирования векторных операций с плавающей запятой была предпринята авторами [46] для набора встроенных функций SSE (Streaming SIMD Extensions). Другие работы посвящены проблеме работы с массивами и строками [47; 48], поскольку моделирование строковых операций и массивов является критичным для безопасности приложений и позволяет обнаруживать некоторые виды атак и инъекций. Интересным представляется подход на основе символьных автоматов и символьных преобразова-

телей [49;50], позволяющий формально описывать преобразования строк алфавита произвольной размерности (возможно и описание операций над бесконечными алфавитами); кроме того, представляется возможным обобщение данного подхода на более общие случаи анализа, в т. ч., анализ разнотипных данных.

Ряд исследователей для улучшения метода предпринимали попытки модификации методов межпроцедурного анализа. Одной из наиболее часто используемых техник улучшения производительности МПА является моделирование вызовов функций при помощи их спецификаций. Данный подход хорошо работает в случае, если действия функции документально определены или стандартизированы — в частности, стандартные библиотеки языков (например, STL), стандарты программирования (Posix) или предоставляемый и документированный программный интерфейс (Windows API). Использование этого подхода можно встретить, в частности, в статическом анализаторах Svace, где используются спецификации в виде реализаций моделируемых функций, содержащих отслеживаемые сигнатуры анализатора. Если говорить о методе символьного выполнения, использование данного подхода можно найти в Clang Static Analyzer, где функции можно специфицировать двумя способами: построив их упрощённое синтаксическое дерево с использованием механизма BodyFarm или моделируя эффекты функции в проверяющем модуле (за что отвечает функция обратного вызова `evalCall()`. В случае, если определения моделируемых функций доступны анализатору, использование их спецификаций вместо определений позволяет не только задать учитываемые эффекты, но и заметно увеличить скорость анализа, особенно если моделируемая функция содержит циклы или вложенные вызовы. Основным недостатком метода является достаточно его затруднительное использование для нестандартизированных, т. е. пользовательских функций. В случае пользовательских функций их спецификации необходимо создавать вручную, что снижает эффективность использования анализатора.

Для улучшения качества анализа и устранения ложных срабатываний статический анализ нередко комбинируют с другими методами анализа, результатом чего являются гибридные анализаторы. Одним из вариантов дополнения метода символьного выполнения является реальное выполнение анализируемой программы (concolic testing), призванное устранить ошибки анализа, связанные с неидеальностью используемого решателя и допущений, используемых при

анализе. Такой подход оказывается естественным в случае анализа машинного или промежуточного кода. Впервые эта идея была использована в работе [51], где описывается инструмент DART (Directed Automated Random Testing). Этот инструмент позволяет автоматически генерировать тестовые данные, использование которых в качестве входных приводит к аварийному завершению анализируемой программы. В работе, описывающий генератор тестов CUTE [52], этот подход был расширен для случаев использования сложных структур данных. Наиболее известными промышленно используемыми инструментами, использующими гибридное выполнение, являются KLEE [12] (основанный на LLVM) и SAGE (Microsoft) [53].

Возможность использования резюме функций для повторного использования результатов их анализа также вызывало интерес у исследователей. Наиболее интересной работой в данной области представляется работа Патриса Годфруа (Patrice Godefroid), основного автора системы DART, посвящённая композиционному символьному выполнению [54]. В рамках данной работы автор использует символьное выполнение для автоматической генерации тестов с помощью инструмента SMART (Systematic Modular Automated Random Testing), являющегося расширением инструмента DART. При этом при анализе программы генерируются резюме функций. Само резюме ϕ_f функции f описывается как формула логики высказываний для некоторого исчисления ограничений T, при этом высказывания и ограничения выразимы в T. При этом ϕ_f может быть вычислено с помощью последовательной итерации и определяется как дизъюнкция формул ϕ_w вида $\phi_w = pre_w \wedge post_w$, где pre_w — конъюнкция ограничений на множество входных данных (вход) f, а $post_w$ — конъюнкция ограничений на выход f. Каждая из конъюнкций ϕ_w может быть вычислена из ограничений соответственно пути w как краткое описание. Сами входные данные функции f определены как любой адрес памяти, который может быть прочитан внутри функции f во время её выполнения, а множество выходных данных (выход) функции f определяется как любой адрес, в который может быть произведена запись во время выполнения f и которые могут быть прочитаны в программе после возврата из f. Предусловия резюме функции описываются в терминах входных данных функции, а не программы с целью избежать дублирования резюме для различных контекстов вызова. Каждое из предусловий определяет

собственный класс эквивалентности для выполнений программы с конкретными данными.

Резюме функции вычисляется последовательными итерациями при её выполнении. Каждый раз при выходе из функции ограничения (т. е., при завершении одного из путей выполнения w), вычисляемые системой DART, используются для построения предусловия pre_w для этого пути: pre_w получается упрощением конъюнкции условий ветвлений относительно входа функции, которые были верны для пути w. Постусловия вычисляются следующим образом. Если выполнение функции завершается оператором `return`, постусловие $post_w$ может быть вычислено как конъюнкция ограничений, соответствующих регионам памяти $m \in Write(f, \overrightarrow{I_f}, w)$, в которые производилась запись во время выполнения пути w функции f при контексте (наборе входных данных) $\overrightarrow{I_f}$. Точнее,

$$post_w = \bigwedge_{m \in Write(f, \overrightarrow{I_f}, w)} (m = evaluate\_symbolic(m, \mathcal{M}, \mathcal{S}))$$

В противном случае, если функция завершается оператором вызова `halt` или `abort`, её множество выходных данных объявляется пустым: $post_w = false$ для последующего использования в контексте вызова.

Следовательно, резюме отдельного пути w функции f является конъюнкцией предусловия и постусловия: $\phi_w = pre_w \wedge post_w$. Полное резюме функции является объединением резюме отдельных путей для каждого из смоделированных путей и определяется как дизъюнкция резюме путей: $\phi_f = \bigvee_w \phi_w$.

Автор работы заявляет о переходе от экспоненциального роста количества исследуемых путей программы к полиномиальному. В этой же работе автор доказывает, что для выбранного базового метода анализа с помощью системы DART множество находимых дефектов для случая резюме (SMART) и для случая встраивания (DART без дополнений) совпадают. Автором также предлагается ряд техник, которые могли бы потенциально уменьшить время анализа.

Впоследствии подход, обозначенный в данной работе, был расширен другими авторами. Так, совместно с Сасватом Анандом (Saswat Anand) в работе [55] автором описывается подход, использующий выбор ветви для анализа при межпроцедурном анализе методом символьного выполнения («Demand-Driven Symbolic Execution»). Описываемый подход заключается в определении

достижимости не всех ветвей выполнения, а только отдельных, которые могут быть интересны для анализа какими-либо видами проверок (непосредственно в примерах, приведённых в работе, авторами описывается определение достижимости assert-утверждений и строковой функции). В [56] композиционное символьное выполнение используется для анализа достижимости состояний программы и их последовательностей («may-must»-анализ) с целью верификации и поиска ошибок в драйверах устройств, разрабатываемых для ОС Windows 7.

Исследователи из научно-технического университета Китая применили подход резюме для символьного выполнения для поиска утечек памяти в коде программ на языка C [57]. Различные исследователи пытались использовать подход резюме для решения различных узких задач: от поиска дефектов, связанных с многопоточностью [58], до изучения активности объектов кучи [59]. Исследователи из Исследовательского центра IBM T.J. Watson и университета Тель-Авива смогли применить прототип своего фреймворка, использующего точные и полные (precise and concise) резюме, к анализу программ на языке Java [60]. Метод композиционного символического анализа с использованием резюме байткод-методов был реализован в виде расширения инструмента Symbolic PathFinder [61]. Вариация метода динамического символьного выполнения, использующая т.н. «резюме данных», была разработана для использования для анализа кода на языке JavaScript с помощью инструмента MultiSE [62].

2 Межпроцедурный анализ на основе резюме для метода символьного выполнения

2.1 Математическая модель. Разработанный алгоритм метода резюме для символьного выполнения

Рассмотрим основные источники затрат при встраивании и при использовании резюме. При встраивании функций вызываемая функция анализируется каждый раз при её вызове. При этом в случае контекстно-чувствительного анализа функция анализируется не полностью: анализируются лишь те пути выполнения, которые являются достижимыми при контексте на момент вызова. Суммарное время, затраченное на анализ функции при допустимой степени вложенности, равной 1, можно вычислить по формуле:

$$T_{\text{встраивания}} = \sum_{i=0}^{n} t_i \tag{2.1}$$

где i — номер вызова, t_i — время, затраченное на анализ i-го вызова (с учётом контекста). С учётом того, что сама вызываемая функция также анализируется отдельно, формула приобретает вид:

$$T_{\text{встраивания полное}} = T_{\text{анализа}} + \sum_{i=0}^{n} t_i \tag{2.2}$$

При использовании подхода, основанного на резюме, временные затраты вычисляются следующим образом. Вызываемая функция анализируется один раз, но полностью (независимо от вида анализа). Однократный характер также носят затраты, связанные с составлением резюме функции. Применение резюме выполняется в каждой точке вызова функции. Таким образом, общие затраты рассчитываются по формуле:

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + \sum_{i=0}^{n} t_{i\text{применения}} \tag{2.3}$$

что, с учётом примерного равенства времён применения резюме (поскольку резюме имеет не меняющийся размер), приближённо равно

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + nt_{i\text{применения}} \tag{2.4}$$

Таким образом, выигрыш от применения резюме будет получен при выполнении следующего соотношения:

$$T_{\text{сбора}} + nt_{i\text{применения}} < \sum_{i=0}^{n} t_i \tag{2.5}$$

откуда следует, что для получения ускорения необходимо выполнение соотношения

$$t_{\text{ср. применения}} < t_{\text{ср.}} \tag{2.6}$$

Пусть $s_1, \ldots, s_n$ — некоторая последовательность операторов программы. Каждый оператор имеет набор эффектов, который он оказывает на состояние выполнения программы. Таким образом, каждый оператор программы представляет собой передаточную функцию: $p_i = s_i(p_{i-1})$, где p_{i-1} — состояние программы непосредственно перед выполнением оператора, p_i — состояние программы непосредственно после выполнением оператора s_i (и перед выполнением оператора s_{i+1}). Тогда в результате выполнения всего блока операторов программа из начального состояния p_0 перейдёт в состояние p_n: $p_n = s_n(s_{n-1}(\ldots s_i(\ldots(s_1(p_0))\ldots))$. Тогда суммарный эффект последовательности операторов можно представить в виде композиции их передаточных функций:

$$s = s_1 \circ s_2 \circ \ldots \circ s_n \tag{2.7}$$

Данная формула справедлива для последовательности операторов без переходов, то есть для базовых блоков программы. Кроме того, формула справедлива для последовательности операторов, содержащей безусловный переход, поскольку такая последовательность также представляет собой путь выполнения без ветвлений. Однако при наличии условных переходов в блоке последовательности операторов, оказывающих эффект на выполнение программы, могут различаться. Это означает, что суммарный эффект выполнения блока зависит от пути выполнения внутри блока, а следовательно, и от значения выражения в условии.

Пусть c_j — условие, принадлежащее анализируемому блоку, $0 \leqslant j \leqslant m$, в ветках `if` и `else` которого находятся непрерывные последовательности операторов $s_0, \ldots, s_k$ и $s_{k+1}, \ldots, s_n$ соответственно, возможно, пустые. Тогда будет справедливо следующее соотношение:

$$s_c = \begin{cases} s_1 \circ \ldots \circ s_k & \text{при } c_j \equiv true \\ s_{k+1} \circ \ldots \circ s_n & \text{при } c_j \equiv false \end{cases} \tag{2.8}$$

С использованием данных правил можно строить композиции эффектов произвольных последовательностей операторов.

Поскольку тело функции также является последовательностью операторов языка, эффект от вызова функции можно рассчитать по тем же правилам. На основе зависимости полученного эффекта от условий на пути выполнения внутри функции и строится резюме. При этом в резюме сохраняются не все эффекты, производимые операторами, содержащимися в теле функции, а лишь те из них, которые сохраняются после выхода из неё и могут повлиять на дальнейшее выполнение программы после выхода из функции. Таким образом, сократить время анализа при использовании резюме в сравнении со встраиванием можно получить за счёт отсутствия необходимости затрачивать время на анализ эффектов, действия которых локальны или не учитываются при дальнейшем анализе. Так, связывание символьного значения с выражением имеет только локальный эффект, поскольку все выражения становятся неактивными при выходе из контекста анализа функции. Аналогично, локальный эффект имеют записи в локально видимую память и т. д. Кроме того, модель анализатора заведомо допускает упрощения, поскольку анализатор не может досконально смоделировать поведение программы. Это означает, что ряд эффектов операторов не будет учтён, т. е. на моделирование некоторых эффектов операторов будет затрачено время, однако результат этого моделирования не будет отражён в изменении состояния. Учёт этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, однако, что некоторые эффекты самого применения резюме не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Но, поскольку набор эффектов, получаемых в результате применения резюме, включается строго или совпадает

с набором эффектов, моделируемых при анализе методом встраивания, время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания.

2.2 Алгоритм метода резюме для символьного выполнения

В результате проведённого анализа в данной работе построен алгоритм метода межпроцедурного анализа с помощью резюме для метода символьного выполнения.

1. Провести анализ вызываемой функции, получив в результате её граф выполнения.
2. Для каждого конечного узла графа выполнения функции осуществить сбор эффектов, оказываемых на выполнение программы при выполнении данной ветви выполнения. Полученным результатом является набор ветвей резюме.
3. В каждой точке вызова проанализированной функции создать новые узлы графа выполнения (узлы применения резюме) со следующими характеристиками:
   - Дуги графа выполнения ведут из узла, соответствующего вызову функции (узел вызова) в каждый из узлов применения резюме.
   - Каждая точка применения резюме соответствует листу графа выполнения вызываемой функции и, соответственно, своей ветви резюме.
   - Состояние программы в каждой точке применения резюме есть композиция состояния программы в узле вызова и функции, описываемой соответствующей ветвью резюме.

Таким образом, множество узлов графа выполнения вызываемой функции отображается в множество узлов применения резюме. Поскольку множество всех узлов графа выполнения, как правило, многократно превосходит по количеству элементов множество листов графа, данный метод имеет значительно большую потенциальную масштабируемость.

2.3 Модель анализатора

В терминологии Clang Static Analyzer, разработанной на основе [63], выполнение программы представляет собой множество последовательных переходов между состояниями из одного состояния в другие. Каждому состоянию соответствует точка выполнения (`ProgramPoint`), для которого это состояние актуально. (Здесь и далее в скобках приводятся названия соответствующих классов из фреймворка Clang Static Analyzer). Переходы между состояниями обусловлены либо эффектами интерпретации отдельных выражений, определёнными стандартом языка, либо событиям, связанных с выполнением проверок проверяющими модулями (`Checker`). Из одной точки выполнения может идти не один переход в другое, а более — это происходит в случаях, когда условие перехода невозможно однозначно разрешить в пользу выбора какой-либо одной ветви выполнения, например, при обработке условных операторов (это и есть разделение на классы эквивалентности). Кроме того, проверки также могут разделять состояние программы, сохраняя в разных структурах состояния различающиеся данные состояния. Результирующее множество узлов в виде состояний и переходов из одного состояния программы в другое образует граф выполнения программы (`ExplodedGraph`).

За базовое моделирование процесса выполнения, без каких-либо проверок корректности исходного кода, отвечает ядро анализатора. Ядро анализатора представляет собой во-первых, набор методов, связанных с построением графа состояний выполнения программы (класс `CoreEngine`), а, во-вторых, набор методов, отвечающие за моделирование эффектов, специфичных для языка программирования, т. е. моделирование эффектов выражений и правил их выполнения (класс `ExprEngine`). Кроме того, в процедуре построения графа выполнения могут принимать участие проверяющие модули, анализируя события, наступающие в процессе выполнения. Эти проверки могут останавливать построение графа на заданном пути в случае обнаружения критического дефекта, разделять состояние программы и вносить в него дополнительную информацию для работы проверяющего модуля.

Структура состояния является представлением состояния программы в точке выполнения. Структура состояния включает следующие данные:

1. Содержимое памяти программы (модель памяти — *RegionStore*) [64], представляемое как отображение между регионами памяти и символьными значениями, связанными с этими регионами. Для создания записи в модели памяти необходимо произвести непосредственное связывание региона памяти и его значения, например, при обработке оператора присваивания. Операциями, изменяющими содержимое модели памяти, являются непосредственное связывание региона со значением (происходящее, например, при присваивании переменной значения), пометка регионов памяти как не содержащих первоначальное значение (инвалидация) и удаление имеющихся привязок, происходящее при потере регионом памяти активности, а также иногда используемое вместо инвалидации. В случае, если необходимо получить символьное значение для региона памяти, не имеющего записи в модели памяти (например, для аргументов функции), происходит неявное связывание с помощью символьного значения специального вида, при этом записи в модели памяти не создаётся.
2. Окружение (*Environment*) ставит в соответствие активным выражениям их символьные значения, как левосторонние, так и правосторонние: левосторонними значениями выражений являются абстрактные области памяти кода программы, где располагаются выражения, а правосторонними — вычисленные символьные значения выражений. Значения выражений, ставших неактивными, удаляются из окружения.
3. Нетипизированное хранилище (*GDM*, Generic Data Map) является контейнером для хранения данных проверок, а также используется для хранения некоторых специфических структур данных ядра анализатора, связанных с состоянием программы. Наиболее важными такими данными является карта соответствия символов и их диапазонов возможных значений, с помощью которой производится анализ достижимости. GDM позволяет хранить произвольные структуры данных, однако наиболее часто используются простейшие типы — указатели и целочисленные типы, а также неизменяемые словари, наборы и списки (`ImmutableMap`, `ImmutableSet` и `ImmutableList` соответственно), имеющие специальную поддержку, упрощающую их использование. За помещение данных в GDM и удаление их оттуда отвечают непосредственно

использующие эти данные модули — проверки и модули ядра анализатора (в частности, ответственный за арифметические и логические вычисления решатель — `ConstraintManager`).

Символьное значение, в терминологии Clang Static Analyzer, является абстракцией переменного или константного значения какого-либо типа данных. Абстрактным значением можно представить, например, значение выражения, содержимое области памяти, саму область памяти и указатель на неё. В модели Clang Static Analyzer символьное значение может иметь несколько основные видов.

Первым видом символьных значений являются целочисленная константа. Clang Static Analyzer на данный момент может корректно работать лишь со значениями, представляемыми с помощью целочисленных типов. Константа имеет знаковость, разрядность и значение. Целочисленными константами также представляются символы строк (`char`-типы). Константы могут быть как правосторонними значениями (`nonloc::ConcreteInt`), так и левосторонними (`loc::ConcreteInt`). Система арифметики позволяет производить вычисления над константами с использованием бинарных и унарных операторов с получением новой константы в качестве результата.

Вторым видом символьного значения является символьное значение (`SymbolVal`). Символьное значение является представлением для *символьного выражения*, или *символа* — абстрактного неконстантного выражения. Символьным выражением могут быть:

- атомарный символ — атомарное правостороннее значение, которое нельзя однозначно определить как константу. Различные виды атомарных символов используются анализатором для различных целей. Существуют следующие виды атомарных символов:
  - `SymbolRegionValue` — символ, являющийся значением некоторого региона памяти по умолчанию, если его исходное значение неизвестно.
  - `SymbolExtent` — символ, представляющий размер соответствующего региона памяти в байтах, если этот регион имеет неконстантный размер.
  - `SymbolConjured` — символ, являющийся результатом выражения в случае, если правило вычисления данного выражения

неивестно. В частности, `SymbolConjured` может быть получен в качестве возвращаемого результата функции, определение которой недоступно анализатору.

  - `SymbolMetadata` — символ, описывающий некоторый регион памяти. Такие символы обычно используются проверяющими модулями для хранения информации об отслеживаемом регионе
  - `SymbolDerived` — символ, представляющий значение некоторого региона памяти, чей родительский регион имеет символьное значение.

- бинарное отношение символьного выражения и константы, выражаемое с помощью бинарных операторов (в CSA им соответствуют классы `SymIntExpr` и `IntSymExpr`);
- бинарное отношение двух символьных выражений, представленную в виде бинарного выражения с помощью бинарных операторов (`SymSymExpr`).

Символьное выражение, таким образом, в общем случае представляет собой вычисляемое дерево, листьями которого являются атомарные символы и константы.

Ещё одним видом символьных значений являются значение вида региона памяти (`MemRegionVal`) — значение, являющееся *регионом памяти*, т. е. абстрактным представления некоторого последовательного набора байт в памяти программы. Регион памяти является левосторонним выражением, будучи контейнером для некоторого значения, расположенного в памяти. Таким образом, регион памяти может быть как ключом для хранения в Store, так и его значением — например, значением указателя. В виде регионов памяти представляются левосторонние значения выражений (например, переменных, объектов). Каждый регион памяти, кроме *областей памяти*, имеет родительский регион. Таким образом, регионы памяти образуют иерархию памяти, корневыми регионами которой являются области памяти, а прочие регионы памяти являются их подрегионами и подрегионами друг друга. Память делится на области согласно расположению объектов в оперативной памяти (таблица 2.1)

Сами подрегионы разделяются на классы в зависимости от назначения (актуальные для C и C++ классы регионов перечислены в таблице 2.2. Вместе они образуют иерархию наследования. Классы регионов памяти играют важ-

Таблица 2.1

Виды областей памяти

Название региона	Хранимые данные
NonStaticGlobalSpaceRegion	Глобальные переменные внешней области видимости
StaticGlobalSpaceRegion	Глобальные статические переменные
HeapSpaceRegion	Переменные, располагающиеся в «куче»
StackArgumentsSpaceRegion	Переменные, являющиеся аргументами функции и располагающиеся на стеке
StackLocalsSpaceRegion	Переменные, локальные для функции
UnknownSpaceRegion	Вспомогательная область памяти для случаев, когда область хранения неизвестна

ную роль при построении и применении резюме функций, поскольку несут в себе важную служебную информацию, необходимую для определения новых регионов в заданном контексте.

Для экономии ресурсов при обработке операций присваивания сложных объектов используются символьные значения отложенной обработки (`LazyCompoundVal`). Их задача заключается в отображении крупных блоков хранилища на другой регион памяти до момента, пока в новый регион памяти не будет произведена запись. Таким образом производится увеличение скорости работы анализатора за счёт уменьшения объёмов копируемой и хранимой в структуре состояния информации.

Вспомогательными видами символьных значений является неизвестное значение (`UnknownVal`) и неопределённое значение (`UndefinedVal`). Неизвестное значение является может являться результатом выражения, которое анализатор не может корректно смоделировать — например, битовые операции над неконстантными значениями; кроме того, неизвестное выражение может получиться, если операндом выражения является неизвестное выражение. Нередко вместо неизвестного значения создаётся новый атомарный символ без наложенных на него ограничений, что позволяет восстановить контекстную чувствительность для символьных операций. Неопределённое значение является результатом операций, результат которых не определён стандартом — в частности, разыменование нулевого указателя, а также при выполнении операций, операндом которого является другое неопределённое значение. Использование

Таблица 2.2

Виды регионов памяти

Название региона	Хранимые данные
AllocaRegion	Память на стеке, выделенная функцией `alloca()`
SymbolicRegion	Регион памяти, располагающийся по адресу, заданному указателю. Не имеет определённого размера и типа, они задаются его подрегионами
BlockDataRegion	Данные блоковых конструкций языка C и лямбда-выражений C++
BlockTextRegion	Код блоковых конструкций языка C и лямбда-выражений C++
FunctionTextRegion	Регион кода функции
CompoundLiteralRegion	Регион составного литерала
CXXBaseObjectRegion	Регион базового подобъекта класса. Задаётся определением базового класса
CXXTempObjectRegion	Регион временного объекта в C++
CXXThisRegion	Регион, на который указывает указатель `this` (C++). Используется при анализе методов класса вне контекста
FieldRegion	Регион поля структуры или класса. Задаётся определением поля
VarRegion	Регион переменной
ElementRegion	Регион элемента массива
StringRegion	Регион строкового литерала

неопределённых значений также является одним из видов потенциальных проверок, и введение для этого отдельного типа символьного значения упрощает работу с анализатором.

Символьное значение можно получить следующими способами.

Левостороннее символьное значение можно получить при обращении к выражению, результат которого является левосторонним выражением — переменная, элемент массива, поле объекта и т. д. При этом результирующее значение будет связано с объявлением той области памяти, к которой происходит обращение.

Символьное значение можно также получить в результате выполнения загрузки символьного значения из региона памяти (при преобразовании левостороннего выражения в правостороннее выражения, lvalue-to-rvalue cast). При этом, если регион уже имел непосредственную привязку, то будет получено символьное выражение, связанное с этой привязкой. В противном (и более частом) случае произойдёт создание атомарного символа, являющимся значением этого региона по умолчанию, без каких-либо наложенных на него ограничений, или возврат уже созданного символа.

Наконец, символьное значение можно получить, осуществив бинарную операцию над другими символьными значениями.

2.4 Эффекты, учитываемые в резюме функции

Каждый оператор при своём выполнении производит эффект, заключающийся в изменении состояния программы. В случае анализа речь идёт о моделировании эффектов операторов, то есть о моделировании действия, которое моделируемый оператор оказывает на модель состояния программы.

С учётом описанной модели анализатора, сократить время анализа при использовании резюме в сравнении со встраиванием можно получить за счёт отсутствия необходимости затрачивать время на анализ эффектов, действия которых локальны или не учитываются при дальнейшем анализе. Например, связывание символьного значения с выражением имеет только локальный эффект, поскольку все выражения становятся неактивными при выходе из контек-

ста анализа функции. Аналогично, локальный эффект имеют записи в локально видимую память и т. д. Кроме того, модель анализатора заведомо допускает упрощения, поскольку анализатор не может досконально смоделировать поведение программы. Это означает, что ряд эффектов операторов не будет учтён, т. е. на моделирование некоторых эффектов операторов будет затрачено время, однако результат этого моделирования не будет отражён в изменении состояния. Учёт этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, однако, что некоторые эффекты самого применения резюме не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Тем не менее, время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания. Это объясняется тем, что набор эффектов, получаемых в результате применения резюме, включается строго или совпадает с набором эффектов, моделируемых при анализе методом встраивания.

В данной работе рассмотрен следующий набор эффектов, оказывающих влияние на состояние анализируемой программы в процессе её выполнения:

1. Принятие решений о выборе пути выполнения. Выбор пути выполнения сопровождается наложением ограничений на символьные значения, относительно которых принимается решение о выборе пути. Если эти символьные значения содержат ссылки на внешние по отношению к вызываемой функции регионы памяти, накладываемые ограничения должны быть отражены в резюме. Кроме того, как было показано выше, каждое принятие решения влияет на присутствие и порядок операторов в последовательности выполнения, а следовательно, и на набор эффектов, включаемых в резюме. Наконец, наложение ограничений на входные данные функции в зависимости от выбора пути выполнения позволяет сохранить контекстную чувствительность при анализе, поскольку определённые пути выполнения могут быть достижимы лишь при ограниченном наборе входных значений аргументов функции и значений, находящихся во внешней по отношению к ней памяти.
2. Модификация регионов памяти с областью видимости, отличной от локальной, то есть находящихся в статической или глобальной области

видимости, принадлежащих куче, а также модификация аргументов, переданных по неконстантному указателю или неконстантной ссылке, и областей памяти, относящихся к ним (возможно, с использованием арифметики указателей).

3. Инвалидация регионов памяти, то есть пометка некоторых регионов как изменивших значение на неизвестное. Данное действие обычно выполняется при моделировании оператора, все эффекты которого учесть по каким-либо причинам невозможно — например, при вызове функции с недоступным определением.
4. Возврат вызываемой функцией некоторого значения. Это значение связывается с выражением вызова функции как элемент окружения.
5. Пометки проверяющих модулей:
   (a) пометки символов, регионов памяти и символьных значений;
   (b) события, которые необходимо проверить отложенно, когда контекст вызываемой функции станет достаточно определён для того, чтобы утверждать наличие потенциального дефекта;
   (c) иные действия, связанные с процедурами проверок (в зависимости от логики работы проверяющего модуля).

Поскольку проверяющие модули самостоятельно отвечают за свои данные, логику обработки резюме для проверок имеет смысл включать непосредственно в логику работы этих модулей.

2.5 Порождение новых ветвей выполнения программы и отсечение недостижимых путей

Одним из результатов сбора резюме являются пары «регион памяти — символьное значение». В результате актуализации символьных значений из резюме могут получиться символьные значения, имеющие диапазон, отличный от диапазона этого символьного значения в контексте вызывающей функции. Это является следствием того, что при моделировании условий внутри вызываемой

функции может произойти разделение входных данных функции (аргументов и внешних переменных) на классы эквивалентности.

Рассмотрим пример. Пусть вызывается функция:

```
1  void f(int a) {
2    if (a > 10) {
3      ...
4    } else {
5      ...
6    }
7  }
```

В результате анализа этой функции в её резюме войдут две ветви выполнения. В первой ветви `a` будет иметь диапазон конкретных значений от `INT_MIN` до 10, во второй — от 11 до `INT_MAX`.

Пусть происходит вызов функции при `a` от 5 до 13. Тогда в первой создаваемой ветви выполнения `a` будет иметь диапазон от 5 до 10, а во второй — от 11 до 13.

Далее, пусть в вызывающей функции `a` имеет диапазон конкретных значений от 5 до 9. Тогда в первой ветви выполнения `a` будет иметь диапазон от 5 до 9, а во второй ветви выполнения множество значений будет пустым. Наличие символического значения с пустым множеством допустимых значений означает, что вторая ветвь является недостижимой и может не рассматриваться. Действительно, при вызове функции при заданном `a` выполняется только `else`-ветвь, но не `if`-ветвь.

Введём следующие обозначения:

- n – количество ветвей выполнения, полученных в резюме анализа вызываемой функции,
- i – номер ветви выполнения, где $0 \leqslant i < n$,
- p_i – количество символьных правосторонних входных значений в i-ой ветви выполнения,
- j – номер символьного значения, где $0 \leqslant j < p_i$,
- s_{ij} – символьное значение с номером j в i-ой ветви выполнения,
- $r_{\text{входные } ij}$ – множество значений для s_{ij} в контексте вызывающей функции в точке непосредственно перед вызовом функции,

– $r_{\text{резюме } ij}$ – множество значений для s_{ij} в контексте вызываемой функции,
– $state_{\text{входное}}$ — состояние программы в контексте вызывающей функции в точке непосредственно перед вызовом функции,
– $state_{\text{выходное}}$ — состояние программы после вызова функции (после применения резюме).

Тогда при применении i-й ветви выполнения резюме:

$$\forall i \in [0; n], \forall j \in [0; p_i] : \; r_{\text{выходные } ij} = r_{\text{входные } ij} \cap r_{\text{резюме } ij},$$

то есть результирующее множество является пересечением множеств входных конкретных значений символического значения и множества конкретных значений символического значения из применяемой ветви резюме.

В случае, если результирующее множество конкретных значений является пустым хотя бы для одного символического значения, то данная ветвь выполнения является недостижимой и не принимается в дальнейшее рассмотрение, что может быть выражено формулой 2.5.

$$(\exists i, j : r_{\text{входные } ij} \cap r_{\text{резюме } ij} = \varnothing) \Rightarrow (state_{\text{входное}} \nrightarrow state_{\text{выходное}})$$

2.6 Сбор данных для создания резюме

Пусть некоторое значение относится к региону памяти. Поскольку при передаче аргумента в функцию не по значению его значение может измениться, необходимо различать входное значение региона до его изменения в функции и выходное значение. Для получения информации о разбиении данных на классы эквивалентности можно отслеживать события ветвлений (assume), однако данный подход неудобен на практике, поскольку, во-первых, требует отдельного отслеживания событий изменений региона для разделения входных и выходных значений, и, во-вторых, требует активного участия сборщика резюме в процессе принятия решения о ветвлении. Вместо этого в настоящем решении предложен и использован подход, основанный на проверке активно-

сти символов и регионов. Поскольку входные данные, внешние по отношению к анализируемой функции, всегда являются символическими, отслеживая событие потери актуальности (активности), можно определить диапазон возможных значений символа, связанного с данным регионом, в данной ветви выполнения. При этом первое событие потери активности соответствует диапазону входных значений, а последнее соответствует диапазону выходных значений, причём если они совпадают, это означает, что никаких присваиваний или инвалидаций региона входного символа не было, и входной диапазон является также выходным диапазоном. Данный метод позволяет избежать использования сложных алгоритмических схем для сбора входных и выходных значений аргументов функций, а также входных значений глобальных регионов памяти.

Сбор выходных значений также бывает необходимо проводить по окончании пути выполнения функции. Это необходимо для обработки символьных значений, привязанных к региону памяти непосредственным присваиванием или иным видом связывания. Для этого используется итерация по хранилищу с сохранением диапазонов внешних по отношению к функции регионов памяти в резюме.

Инвалидация региона памяти в терминологии Clang Static Analyzer означает связывание с данным регионом нового символа, без наложенных на него ограничений, т. е. способного принимать произвольные значения. Поскольку символьные значения, связанные с регионами памяти, обрабатываются при завершающем проходе по хранилищу, а значения регионов, актуальные до инвалидации, обрабатываются по событию потери активности, непосредственно предшествующему событию связывания нового значения, инвалидация обрабатывается автоматически, и дополнительных действий для обработки инвалидаций регионов памяти не требуется.

Обработка события возврата функцией значения достаточно тривиальна. Результирующее символьное значение сохраняется в резюме целиком, а ограничения, накладываемые на него и на его части, обрабатываются отдельно.

За хранение данных проверок проверяющие модули отвечают самостоятельно. Основными видами данных проверок является отметка отложенной проверки и данные состояния проверки. Отложенные проверки используются для выдачи предупреждений в тех ситуациях, когда из-за отсутствия данных о контексте вызова невозможно однозначно утверждать наличие дефекта или его

отсутствие. Данные состояния используются для построения нового состояния проверки при применении резюме.

2.7 Актуализация символьных значений

В результате сбора резюме на предыдущем шаге мы получаем некоторое множество регионов памяти, с которыми связаны некоторые символьные значения. Кроме того, регионы памяти сами могут входить в символьные значения как их составная часть. Однако, полученные регионы памяти адресуются в контексте объявлений имён внутри функции. В контексте вызывающей функции эти регионы могут иметь уже другое значение, то есть регионы, используемые внутри функции, являются относительными по отношению к вызывающей функции. Так, например, в контексте вызываемого метода класса регион памяти, связанный с указателем `this`, будет адресоваться безотносительно какого-либо объекта, а в контексте вызывающей функции этот регион будет регионом, в котором находится объект, метод которого вызывается. Аналогично, аргумент функции, фигурирующий в ней как самостоятельная переменная, (и, соответственно, как самостоятельный регион памяти), может быть подрегионом в контексте вызывающей функции — полем структуры, элементом массива. Кроме того, с регионом памяти в контексте вызываемой функции может быть связан не символ, относящийся к региону памяти, а константа, символьное выражение или иное значение, не имеющее в своей основе регион аргумента. Всё это означает, что для корректного применения резюме необходимо производить актуализацию символьных значений, то есть их перевод из контекста имён и значений вызываемой функции в контекст имён и значений вызывающей функции.

Идея актуализации заключается в следующем. Пусть имеется символьное значение. В его состав могут входить символы, регионы памяти и константы. Они, согласно модели анализатора, образуют дерево. Непосредственно актуализации подвергаются только регионы памяти. Таким образом, в символьном значении происходит подмена регионов памяти, содержащихся в нём, на актуализированные. Затем, если это возможности, символы полученного символьного

выражения вычисляются в константы, заменяя исходные поддеревья. Данную процедуру можно выразить следующим алгоритмом:

1. Для всех регионов памяти, содержащихся в символьном значении:
    (a) Создать новый регион, являющийся актуализацией данного региона
    (b) Заменить в символьном значении исходный регион актуализированным
2. Для всех символов, содержащихся в символьном значении, полученном на шаге 1:
    (a) Проверить, не вычисляется ли символ в константу
    (b) Если символ вычисляется в константу, заменить данный символ вычисленной константой.

В соответствии с типами символьных значений, можно производить актуализацию символьных значений в зависимости от их типа.

Константные значения являются неизменяемыми при их актуализации, поскольку они не содержат элементов, зависящих от контекста. Вместе с тем, типы константного значения в контексте вызываемой функции и в контексте вызывающей функции могут быть различающимися, поэтому может понадобиться осуществление дополнительного приведения типов для константы.

Символьные значения, относящиеся к регионам памяти, можно актуализировать, используя следующие правила для различных категорий регионов памяти.

2.7.1 Регионы, относящиеся к пространству аргументов вызываемой функции

Можно выделить два различных случая передачи, в зависимости от того, является ли передаваемый тип ссылочным или типом указателя, или не является.

В случае, если аргумент передаётся по ссылке, левостороннее значение фактического аргумента становится левосторонним значением формального параметра, а правостороннее значение фактического параметра становится право-

сторонним значением формального параметра. Это значит, что значение объекта в результате выполнения функции может отличаться от значения на момент вызова, поскольку в результате передачи по ссылке с фактическим аргументом может быть связано другое значение.

Таким образом, при передаче аргумента по ссылке:

1. Адрес формального параметра является адресом фактического аргумента.
2. Левостороннее значение формального параметра является левосторонним значением фактического аргумента.
3. Правостороннее значение формального параметра является правосторонним значением фактического аргумента.
4. Базовым регионом для построения подрегионов доступа, относящихся к региону памяти формального параметра, является левостороннее значение фактического аргумента.

В случае, если аргумент передаётся по указателю, правостороннее значение фактического параметра становится правосторонним значением формального параметра. Для левосторонних значений данное утверждение неверно: выполнение присваивания формальному аргументу внутри функции не влияет на значение фактического аргумента. Однако в случае, если указываемый тип не является константным, в результате вызова функции может измениться привязка региона памяти, адрес которой задаёт указатель, и его субрегионов.

Таким образом, при передаче аргумента по указателю:

1. Правостороннее значение формального параметра является правосторонним значением фактического аргумента.
2. Регион памяти с адресом, задаваемым указателем формального аргумента является регионом памяти с адресом, задаваемым указателем фактического аргумента (поскольку значения указателей совпадают). Привязки этого региона и его субрегионов могут изменяться в зависимости от модификатора константности их типов.
3. Базовым регионом для построения подрегионов доступа, относящихся к региону памяти с адресом, задаваемым указателем формальным параметром, является регион памяти с адресом, задаваемым указателем фактического аргумента.

В случае, если аргумент передаётся по значению правостороннее значение фактического параметра становится правосторонним значением формального параметра.

В случае передачи в качестве аргумента структурных типов по ссылке или указателю, правила изменения их полей аналогичны. Так, в случае передачи по значению структуры, содержащей ссылку в качестве поля, данное поле можно считать передающимся по ссылке. Фактически, в случае структурных типов можно считать, что передаётся набор аргументов по их типам с тем отличием, что при потере актуальности их окружающего региона памяти эти поля также могут потерять актуальность.

2.7.2 Регионы памяти внешней области видимости

Помимо передаваемых аргументов, вызываемая функция может иметь доступ к ряду других данных — переменных, имеющих области видимости выше, чем функция. К этим регионам относятся глобальные переменные и члены класса и его предков, если вызываемой функцией является метод класса. Их изменения и наложения ограничений на них также необходимо отслеживать.

Регионы глобальных переменных (включая статические) сохраняются неизменными, дополнительные действия по из актуализации предпринимать не нужно. Это так, поскольку регионы глобальных переменные не зависят от контекста вызова и связаны лишь с объявлением соответствующих переменных.

Методам класса, включая конструкторы, деструкторы и операторы-члены класса, могут быть доступны для чтения и записи поля как самого класса, так и его предков в иерархии наследования. При актуализации регионы памяти, относящиеся к статическим полям класса, не изменяются, поскольку они не относятся к конкретному объекту поля и, следовательно, их адресация не зависит от контекста вызова. Нестатические поля в контексте вызываемого метода адресуются относительно условного объекта, связанного с указателем `this`, и, поскольку в контексте вызываемой функции фактическим объектом будет объект, метод которого вызывается, при актуализации эти поля становятся соответствующими полями вызываемого объекта. Если определение нестатического поля

принадлежит классу, определение которого находится ниже по иерархии наследования, то поле отображается в соответствующее поле родительского класса объекта в соответствии с компоновкой полей дочернего класса.

2.7.3 Актуализация составных и служебных символьных значений

Актуализация символьных значений, обозначающих бинарные операции над символами (бинарные символьные значения), выполняется следующим образом. Сначала выполняется актуализация правой и левой частей символьного значения. Если результатом бинарной операции является константа, эта константа становится результирующим символьным значением. В противном случае результатом актуализации является новое символьное значение в виде бинарной операции. Фактически, бинарные символы актуализируются рекурсивно, с возможными упрощениями в виде свёртывания отдельных элементов или всего выражения в константу. Это свёртывание объясняется уменьшением диапазонов входных значений каждого из символьных значений, входящего в состав бинарного символьного значения, при уточнении контекста вызова.

Актуализация метасимволов является отдельной подзадачей. Под метасимволами понимают символические значения, относящиеся к объекту анализа, но не являющиеся его непосредственной характеристикой. Выделение метасимволов связано с тем, что источником метаданных является не ядро анализатора, а сами проверяющие модули, которые связывают необходимую информацию в виде метасимволов с интересующими их регионами памяти и выражениями. Соответственно, регионы памяти и выражения могут иметь более одного связанного с ними метасимвола. Поскольку каждый проверяющий модуль может реализовывать свой подход к использованию метаинформации и обозначениям интересующих его объектов, для актуализации метасимволов используется отдельное событие, на которое должны подписываться проверяющие модули, использующие метаданные. В результате проверяющие модули самостоятельно обновляют представление связанных с ними символьных значений и нарушение принципа инкапсуляции данных не происходит. В качестве примера можно привести проверку, связанную с длиной строки. Длина строки не входит в число

данных, о которых известно самому анализатору — за её моделирование отвечает проверяющий модуль, связывающий символьное значение (метасимвол) с регионом памяти данной подстроки. Таким образом, задачей проверяющего модуля при актуализации метазначения, связанного с регионом, является поиск существующего метазначения для этого региона и его возврат.

Построение и актуализация сложных структур данных в резюме производится с помощью сохранения цепочки родительских регионов. Для каждого региона памяти, имеющего связанное с ним символьное значение (как явно, так и неявно), строится упорядоченный список родительских регионов, начиная от региона верхнего уровня — $M_0 \dots M_n$. При этом регионы $M_1 \dots M_n$ может быть только регионами элемента массива, регионами поля структуры или регионами данных базового класса. Этот список сохраняется в резюме и используется для актуализации значений по следующему алгоритму.

1. Актуализация региона M_0.
2. Для всех регионов $M_1 \dots M_n$ согласно их положению в списке:
   (a) если M_i — регион элемента массива, то в резюме сохраняется символьное значение индекса, а результатом актуализации является элемент массива от региона, полученного на $(i-1)$-ом шаге алгоритма, с символьным значением индекса, полученным в результате актуализации сохранённого значения индекса.
   (b) если M_i — регион поля структуры, то в резюме сохраняется объявление поля структуры, а результатом актуализации является поле структуры от региона, полученного на $(i-1)$-ом шаге алгоритма, с тем же определением.
   (c) если M_i — регион данных базового класса, то в резюме сохраняется ссылка на определение базового класса, а результатом актуализации является подрегион данных базового класса от региона, полученного на $(i-1)$-ом шаге, с тем же определением.

Цепочка родительских регионов может строиться как явно — при построении резюме для региона памяти, так и неявно — при сохранении и последующем разборе символьного значения, для которого строится резюме.

2.7.4 Актуализация литеральных регионов

Литеральные регионы, т. е. регионы, относящиеся к строковым или составным литералам, актуализируются с использованием выражения языка, по которому они были построены. Поскольку эти регионы хранят исходное выражение как служебную информацию, именно это выражение записывается в резюме, и по нему строится актуализированное выражение. Фактически, литералы не изменяются вместе с контекстом, и обычно хранятся в статической памяти, являясь видом констант, что упрощает работу с ними при использовании метода резюме.

2.8 Применение резюме проверяющими модулями

Схемы применения резюме, описанные выше, затрагивают отсечение недостижимых ветвей выполнения программы и уточнение множества конкретных значений для символьных значений. Однако для того, чтобы анализатор имел возможность выполнять проверки при вложенном вызове функции, необходима доработка проверяющих модулей. Для получения проверяющими модулями возможностей анализа при использовании резюме мы вводим две дополнительных функции обратного вызова. Первая из них (названная `evalSummaryPopulate`) вызывается для сбора резюме проверяющим модулем, вторая (названная `evalSummaryApply`) вызывается при применении резюме.

Проверяющий модуль, имеющий возможность выполнения действий при событии `SummaryPopulate`, должен сохранить информацию, которая может понадобиться для обновления состояния или для выполнения отложенной проверки. Информация, содержащая в резюме, не освобождается до окончания работы анализатора, поэтому проверяющий модуль может использовать произвольный формат хранения данных, лучшим образом отвечающий задаче проверки. Как показала практика модификации проверяющих модулей, для каждой проверки, проводимой модулем, в GDM обычно помещается две дополнительных записи, которые затем будут использоваться для заполнения резюме — для обновления

состояния и для отложенной проверки. В качестве примера рассмотрим проверку двойного закрытия файлового дескриптора. В резюме помещаются две секции: первая отвечает за обновление состояния дескриптора (является ли он открытым или закрытым), а вторая — за выполнение отложенной проверки: в ней запоминаются события закрытия дескрипторов, исходное состояние которых неизвестно.

При обработке события `SummaryApply` проверяющий модуль должен произвести обновление состояния в соответствии с информацией, хранящейся в выбранной ветви резюме. Так, если при выполнении вызова функции дескриптор был закрыт, он должен быть помечен как закрытый в состоянии вызывающей функции. Если же в контексте вызывающей функции уже известно, что дескриптор закрыт, то отложенная проверка должна выдать предупреждение.

Кроме того, если проверяющий модуль использует метаданные для хранения информации о состоянии контролируемых объектов, он может потребовать реализации функциональности актуализации метасимвола. Для этого вводится функция обратного вызова `evalSummarySVal`. Реализующие эту функциональность модули должны определять, на какое символьное значение метаданных отображается актуализированный метасимвол, и возвращать это символьное значение в качестве результата функции обратного вызова. Пример использования данной функции приведён в описании реализации проверки переполнения строки C++ (раздел 2.9.5).

Интерес представляет сравнение двух методов обработки данных проверяющими модулями. Определим два вида проверок формально.

Проверку при использовании метода встраивания определим следующим образом. Пусть имеется граф выполнения программы в виде дерева. Алгоритм поиска осуществляет обход дерева от корня к листу по пути w без возврата, собирая ограничения, наложенные на входные данные, и отслеживая изменение состояния программы в каждом узле графа выполнения. Алгоритм поиска выдаёт срабатывание в случае обнаружения нарушения заданного для него условия корректности состояния программы в заданной точке, таким образом представляя дефект конъюнкцией $pre_w \wedge node_w$, т. е. отображением множества значений входных данных функции на узел графа выполнения, состояние которого нарушает условие корректности программы.

Проверку при использовании метода резюме определим как проверку при использовании метода встраивания для функции верхнего уровня и отложенную проверку при использовании межпроцедурного вызова. Под отложенной проверкой понимается дополнительная проверка всех узлов графа выполнения вызываемой функции в узле применения резюме с актуализацией состояния в данных узлах с учётом контекста вызова и актуализированных ограничений на момент выхода из вызываемой функции. Отложенная проверка и обычная проверка производятся с использованием одного критерия корректности состояния программы.

Теорема 1. Если при использовании проверки с помощью метода встраивания результатом проверки является предупреждение, выданное в некотором узле графа выполнения программы, то при использовании метода резюме для той же функции верхнего уровня и того же набора вызываемых функций результатом проверки методом резюме также является предупреждение, выданное в узле графа выполнения вызывающей функции или в одном из узлов графов выполнения вызываемых функций.

Доказательство. В результате применения резюме в идеальном случае состояние в каждом из создаваемых узлов применения резюме будет аналогичным состоянию одного из узлов выхода из функции графа выполнения программы при использовании метода встраивания. Из данного утверждения следует, что обычная проверка функции верхнего уровня и проверка при использовании метода встраивания будут иметь одинаковый результат для всех общих узлов графа выполнения функции верхнего уровня, если дефект находится в одном из таких узлов. Докажем теперь, что отложенная проверка выдаёт срабатывания для остальных случаев срабатывания проверки метода встраивания. Пусть состоянию программы на момент вызова функции St_{caller} соответствует набор ограничений множества входных данных pre_w, а узел срабатывания $node_{w'}$ достижим на пути w' вызываемой функции при ограничениях множества входных данных функции $pre_{w'}$. Тогда узел срабатывания графа при вызове функции достижим при входных данных

$$pre_{node} = pre_w \wedge pre_{w'} \tag{2.9}$$

Данные ограничения выполняются непосредственно для метода встраивания. В случае использования метода резюме узлу вызываемой функции могут соответ-

ствовать n ветвей выполнения, где $n \geqslant 1$. Поскольку эти ветви имеют общего предка, ограничения множества входных данных этих ветвей графа выполнения функции образуют набор ограничений, для которого выполняется условие

$$pre_{w'} = \bigvee_{i=1}^{n} pre_{w'_i}, \tag{2.10}$$

откуда

$$\forall i \in [1, n] : pre_{w'_i} \subseteq pre_{w'} \tag{2.11}$$

Поскольку отложенная проверка проводится с ограничениями произвольно выбранной ветви графа выполнения $pre_{w'_i}$, актуализированные ограничения будут соответствовать предусловиям

$$pre_{w_i} = pre_w \wedge pre_{w'_i}, \tag{2.12}$$

откуда, с учётом 2.9 и 2.11, получаем

$$pre_{w_i} \subseteq pre_{node}. \tag{2.13}$$

Неравенство 2.13 означает, что ограничения при отложенной проверке входят в ограничения при обычной проверке метода встраивания, и входят в класс эквивалентности данных ограничений. Если нарушение правила корректности обнаружено при входных ограничениях pre_{node}, оно будет обнаружено при любых входных данных заданного класса эквивалентности. Следовательно, если нарушение правила происходит при использовании метода встраивания, оно будет обнаружено и при использовании метода резюме. Теорема доказана.

Обратное утверждение, вообще говоря, не является верным. Рассмотрим пример.

```
1  int callee(int *p) {
2    int t = *p;
3    if (p)
4      return t;
5    return 0;
6  }
```

```
 7
 8  int caller(int *p) {
 9    return callee(p);
10  }
```

Пусть задачей проверяющего модуля является проверка на разыменование нулевого указателя. В случае использования МПА методом встраивания дефект обнаружен не будет, поскольку на момент разыменования `p` неизвестно, является ли он нулевым или нет. Однако при использовании МПА методом резюме в резюме функции будет записаны две ветви: $(p = 0 \wedge 0) \vee (p \neq 0 \wedge *p)$. Поскольку на момент разыменования неизвестно, является ли указатель нулевым или нет, для обеих ветвей резюме планируется отложенная проверка. При применении первой ветви резюме значение указателя актуализируется в 0, поскольку ограничений в вызывающей функции на него не имелось, и отложенная проверка даст срабатывание на разыменование нулевого указателя. Причина подобного различия в поведении заключается в том, что при использовании МПА методом резюме на момент вызова функции имеется больше информации об ограничениях множества входных данных, которые используются в вызываемой функции.

Теорема 2. При использовании отложенной проверки в конце анализа функции верхнего уровня при использовании МПА методом встраивания и МПА методом резюме множества срабатываний совпадают.

Доказательство. Поскольку в идеальном случае актуализация состояния при использовании резюме даёт набор состояний, соответствующих состояниям программы после межпроцедурного анализа методом встраивания, в конце анализа функции верхнего уровня множества классов эквивалентности разбиения входных данных функции и листьев графа выполнения должны совпадать. В конце выполнения функции ряд классов эквивалентности, соответствующий листьям-наследникам узла отложенной проверки, будет входить в класс эквивалентности, при котором был обнаружен дефект при использовании метода резюме. Следовательно, при повторной проверке будет установлено наличие дефекта.

2.9 Методы реализации резюме для различных видов проверок

Как говорилось выше, проверяющие модули должны самостоятельно обеспечивать поддержку резюме. В настоящем разделе дано описание разработанных в рамках данной работы и апробированных методов обеспечения резюме в проверяющих модулях различного назначения. Данные проверяющие модули были ранее разработаны в Московском исследовательском центре Samsung в рамках исследовательских работ и адаптированы к МПА методом резюме в рамках данной работы. На примере этих проверяющих модули продемонстрируем ряд разработанных общих техник для эффективного использования резюме в проверяющих модулях при использовании межпроцедурного анализа. Комбинируя данные техники, можно эффективно реализовывать метод резюме для проверок различного назначения, от проверок ошибок при работе с памятью до проверки корректности работы с механизмом исключений и поиска ошибок, связанных с многопоточностью.

2.9.1 ConstModifiedChecker — проверка модификации константных данных

Задачей данной проверки является определение записи в область памяти, имеющую константный квалификатор, с использованием приведения типа к неконстантному (CERT EXP05-C согласно классификации CERT [65]). Данная проверка преследует две цели. Во-первых, модификация данных таким образом представляет собой плохой стиль програмирования и указывает на необходимость пересмотра интерфейсов программы; в случае константного метода класса (C++) в ряде случаев нужный эффект можно получить, используя ключевое слово `mutable` для определения члена класса, который может быть изменён при вызове семантически константного метода. Во-вторых, попытка записи в память, инициализируемую компилятором как константную, с использованием приведения типов, приводит к неопределённому поведению и к ошибке

сегментации, поскольку объект может быть помещён в память только для чтения [66].

Исходный принцип действия проверки следующий. Проверка отслеживает все регионы памяти, встречаемые в процессе анализа программы и имеющие изначально константный тип. Данные регионы записываются в GDM. В случае выполнения явного приведения типов (с использованием одного из выражений `ExplicitCastExpr`) — с использованием приведения типов в стиле C, операторов `const_cast` или `reinterpret_cast` — от константного типа к неконстантному, регионы, для типа которых было выполнено приведение, записываются в отдельный список. Предупреждение выдаётся анализатором при обработке события записи в регион памяти, если регион записи или его родительский регион занесены в список константных регионов, для которых было выполнено приведение типа при условии, что в иерархии регионов не имеется регион члена класса, объявленного как `mutable`.

В соответствии с исходным механизмом работы проверяющего модуля, выделим изменения, которые необходимо учесть при переходе от МПА методом встраивания к МПА с использованием резюме функции.

1. В вызываемой функции может произойти запись в регион памяти, имеющей неконстантный тип в контексте вызываемой функции. В контексте вызывающей функции данный регион, однако, может иметь изначально константный тип. В этом случае при использовании МПА методом встраивания произойдёт выдача предупреждения анализатора, и аналогичное срабатывание должно быть выдано при использовании метода резюме.
2. Ряд регионов в результате вызова функции может потерять константный тип из-за приведения внутри вызываемой функции. Такие регионы должны отслеживаться в вызывающей функции после вызова, поскольку запись в них должна приводить к выдаче предупреждения.
3. В вызываемой функции могут встретиться регионы, имеющие константный или неконстантный тип. Информацию о них нужно сохранить в вызывающей функции для последующего отслеживания обращений к ним.

Таким образом, в резюме функции помещаются следующие данные:

1. список регионов памяти, в которые была произведена запись при моделировании функции;
2. список регионов памяти, с типа которых был снят константный квалификатор;
3. список регионов памяти, имеющих константный тип;
4. список регионов памяти, имеющих неконстантный тип.

В резюме записываются только те регионы памяти, которые не принадлежат стеку, т. е. не относятся к переменным, локальным для функции или к региону аргументов функции. Данные регионы не доступны из вызывающей функции непосредственно, и данное ограничение позволяет уменьшить как объём памяти, затрачиваемой на хранение резюме функции, так и избегать потенциально ресурсоёмкой работы по актуализации этих регионов при применении резюме. Кроме того, в резюме не хранятся типы регионов памяти, относящиеся к коду функций, а именно, `BlockTextRegion` и `FunctionTextRegion`.

Резюме данным проверяющим модулем применяется согласно следующему алгоритму:

1. все списки регионов памяти из резюме вызываемой функции проходят актуализацию согласно методу, описанному выше;
2. для всех регионов из актуализированного списка регионов, в которые была произведена запись, производится проверка, не входит ли этот регион в список регионов, с типа которых убран константный квалификатор, вызывающей функции. Если какой-либо регион или его родительский регион входят в данный список, проверяющий модуль выдаёт предупреждение о записи в память, имеющую константный квалификатор. В противном случае, если регион записи не является локальным для вызывающей функции, он добавляется в список регионов с записью для вызывающей функции;
3. все регионы из актуализированного списка константных регионов добавляются в список константных регионов вызывающей функции в случае, если они или их родители не присутствуют в списке неконстантных регионов;
4. все регионы из актуализированного списка неконстантных регионов добавляются в список неконстантных регионов вызывающей функции в

случае, если они или их родители не присутствуют в списке константных регионов;

5. все регионы из актуализированного списка регионов, с типа которых убран константный квалификатор, добавляются в аналогичный список регионов вызывающей функции.

Таким образом, для поддержки резюме в данном модуле потребовался лишь один дополнительный список в структуре состояния программы — список нестековых регионов памяти, в которые производилась запись внутри функции. Результирующий проверяющий модуль с поддержкой резюме сохраняет исходное поведение, и результирующая проверка эквивалентна исходной.

2.9.2 IntegerOverflowChecker — проверка на целочисленное переполнение

Задачей данной проверки является определение ситуаций, когда при выполнении бинарной операции умножения, сложения или вычитания происходит целочисленное переполнение. Ситуация переполнения может возникнуть в соответствии с предположениями программиста о выполнении программы, например, при выполнении операции сложения или умножения по модулю. Однако в случае, если использование результата операции, приведшей к переполнению, не предусмотрено алгоритмом, результатом переполнения могут стать искажение данных, с которыми работает программа, некорректное поведение программы при использовании искажённых данных и даже аварийное завершение программы (например, в случае, если переполненное значение используется в качестве индекса массива); кроме того, при целочисленном переполнении в случае знаковых операндов поведение программы не определено [66]. При использовании специальных опций компилятора может допускаться и иное поведение — например, при использовании опции `-ftrapv` компилятора gcc при целочисленном переполнении вызывается перехватчик [67]. Кроме того, ряд опций компилятора могут модифицировать его поведение при оптимизации целочисленных операций, что может затруднить ручной поиск дефекта (например, опция `-fwrapv` компилятора gcc указывает компилятору, что результат целочислен-

ного переполнения при использовании знаковых операндов должен браться по модулю степени двойки). Данный класс ошибок определён в классификации CWE как CWE-190.

Исходный принцип работы данного проверяющего модуля следующий. При выполнении операции сложения проверяющий модуль выполняет проверку, может ли результат быть меньше, чем любое из слагаемых (для случая беззнакового сложения) или является ли результат неопределённым (для знакового сложения), а также допустим ли при выполнении операции иной результат, без переполнения. Выдача предупреждения происходит в случае, если ограничения на символьные значения, участвующие в операции, гарантируют наличие переполнения. Аналогичные проверки проводятся и для других бинарных операций.

Отличие МПА методом резюме от МПА методом встраивания заключается в том, что при анализе функции вне контекста вызова часто нельзя однозначно утверждать, возникнет ли переполнение или нет, поскольку вызывающая функция может наложить на символьные значения дополнительные ограничения. Таким образом, возникает необходимость в выполнении отложенной проверки при применении резюме вызывающей функцией.

Применение резюме для проверки целочисленного переполнения мы производим следующим образом. Если при выполнении бинарной операции возможно как переполнение, так и его отсутствие, проверяющий модуль сохраняет в резюме функции запись, содержащую оба символьных значения, код операции и узел графа выполнения, соответствующий этой операции. При применении резюме оба сохранённых символьных значения актуализируются вместе с привязанными к ним диапазонами возможных значений, после чего выполняется повторная проверка бинарной операции на наличие переполнения. Если ситуация, при которой переполнение не возникнет, становится невозможной, проверяющий модуль выдаёт диагностическое сообщение, строя отчёт по узлу графа выполнения, ссылка на который сохранена в резюме по алгоритму, описанному в следующей главе. Если переполнение невозможно, отложенная проверка для данной операции более не нужна. В противном случае оба актуализированных символьных значения, код операции и узел применения резюме сохраняются в резюме проверяющей проверяющей функции для отложенной проверки в вызывающих функциях более высокого уровня.

2.9.3 AtomicityChecker — проверка атомарности доступа к разделяемым данным

Данный проверяющий модуль осуществляет проверку на атомарность доступа к разделяемым данным в многопоточной среде при использовании Posix-мьютексов (`pthread_mutex_t`) и мьютексов стандартной библиотеки C++11 (`std::mutex`). Задачей проверяющего модуля является отслеживание ситуаций, при которых в процессе доступа к разделяемым данным имеется промежуток, при котором мьютекс не захвачен, из-за чего разделяемые данные могут быть изменены другим потоком непредусмотренным образом. Данный класс ошибок отличается крайне высокой сложностью ручного поиска в связи с трудностью их воспроизведения, поскольку в многопоточной среде порядок выполнения потоков не детерминирован. Кроме того, подобные ошибки достаточно трудны для поиска с использованием динамических анализаторов, поскольку в данном случае к разделяемому объекту может не быть одновременного обращения. Этот класс дефектов определён в классификации CWE как CWE-567.

Пример неатомарного доступа представлен в листинге ниже.

```
// Global variable
std::string global_string;

// Function-local
pthread_mutex_lock(&lock);
int index = global_string.find("some_substring");
pthread_mutex_unlock(&lock);
...
pthread_mutex_lock(&lock);
global_string[index] = 's';
pthread_mutex_unlock(&lock);
```

Исходный принцип работы данного проверяющего модуля следующий. При обработке события записи в локальный для функции регион памяти проверяющий модуль производит проверку, что значение, записываемое в данный регион памяти, не было получено с использованием разделяемых данных. В

случае, если разделяемые данные использовались и при инициализации был захвачен примитив синхронизации, данный локальный регион памяти заносится в список регионов, инициализированных с помощью разделяемых данных. Вместе с ним заносится список регионов разделяемой памяти, с помощью которых он был проинициализирован, а также список захваченных при инициализации примитивов синхронизации.

Каждому примитиву синхронизации ставится в соответствие счётчик, увеличивающийся на единицу каждый раз при захвате данного примитива, и состояние вида «свободен-захвачен». Состояние примитива синхронизации обновляется при обнаружении вызовов функций соответствующих библиотек (`pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `std::mutex::lock()`, `std::mutex::try_lock()`, `std::mutex::unlock()`).

При обращении к глобальному региону памяти производится просмотр регионов памяти, которые могут быть получены из выражения доступа. Если какой-либо из этих регионов содержится в списке регионов, проинициализированных с использованием разделяемых данных, и в списке разделяемых регионов данного локального региона присутствует обрабатываемый разделяемый, производится проверка, равно ли текущее значение счётчика инициализации примитива синхронизации, с которым был инициализирован данный регион памяти, его значению на момент инициализации. В случае, если они не равны (что означает, что примитив синхронизации был освобождён, как минимум, один раз), проверяющий модуль выдаёт диагностическое предупреждение об обнаружении дефекта.

Согласно принципу работы данного проверяющего модуля, в резюме вызываемой функции необходимо учитывать следующие эффекты и изменения состояния:

1. изменения состояния примитивов синхронизации (захват и освобождение) необходимо отслеживать, чтобы иметь информацию об их состоянии для моделирования выполнения программы после вызова функции;
2. запись в регионы памяти, не глобальные, но внешние по отношению к вызываемой функции, необходимо отслеживать, поскольку они могут быть локальными для другой функции;
3. чтения из разделяемых регионов памяти, в выражении которых могут участвовать регионы памяти, локальные для вызывающей функции,

также необходимо учитывать в резюме, поскольку на момент анализа вызываемой функции данных для решения о наличии дефекта или его отсутствия может быть недостаточно.

Соответственно, в резюме функции вносятся следующие данные:

1. для всех примитивов синхронизации, состояние которых было изменено в процессе моделирования выполнения функции, в резюме заносится их состояние на момент выхода из функции (в конце пути внутри графа выполнения функции) и счётчик захватов и освобождений;
2. для всех выражений, содержащих чтения из разделяемых регионов памяти, и для каждого разделяемого региона памяти, чтение из которой выполняется в данном выражении, в резюме заносится ссылка на выражение и список регионов памяти, которые получены из переменных-аргументов, переданных по ссылке и указателю;
3. для всех записей в регионы, полученные из переменных-аргументов, передаваемых по ссылке или указателю, заносится список примитивов синхронизации, захваченных на момент записи (вместе со счётчиком захватов-освобождений), а также список нелокальных регионов памяти, которые могут быть получены из подвыражений выражения записи.

Для каждого примитива синхронизации его счётчик захватов в различных узлах графа выполнения вызываемой функции требует актуализации. Пусть его актуализируемый счётчик захватов в вызываемой функции равен N. Если в вызывающей функции примитив синхронизации захватывался M раз, где $M \neq 0$, актуализированным значением счётчика становится $M + N$. В случае, если на момент вызова функции примитив синхронизации был захвачен, актуализированное значение счётчика дополнительно увеличивается на 1.

При моделировании вызова функции её резюме применяем следующим образом.

1. для списка чтений из глобальных регионов производится отложенная проверка:
    (a) актуализируются сам регион и структуры данных, связанные с ним в резюме (список потенциально локальных регионов памяти, получаемых из выражения доступа, регионы памяти примитивов синхронизации, захваченных при выполнении операции доступа и их счётчики захвата-освобождения. Если

актуализированный регион является глобальным, отложенная проверка не нужна;

(b) если актуализированный регион является локальным для вызывающей функции, а какие-либо из счётчиков примитивов синхронизации на момент инициализации локальной переменной и на момент обращения к ней не совпадают, выдаётся диагностическое предупреждение;

(c) если актуализированный регион не является локальным для функции, но получен из переменных-аргументов, актуализированные данные добавляются в список отложенной проверки вызывающей функции.

2. актуализируется состояние примитивов синхронизации: их состояние захвата устанавливается в состояние захвата на момент выхода из вызываемой функции согласно информации, содержащейся в резюме;
3. актуализируются счётчики захвата примитивов синхронизации (способом, описанным выше).

2.9.4 MissingLockChecker — проверка на несериализованный доступ к разделяемой памяти

Данный проверяющий модуль относится к классу проверок выполнения в многопоточной среде. Его задачей является поиск операций чтения и записи к разделяемой памяти, не защищённых с помощью примитивов синхронизации. Отсутствие синхронизации при операции доступа к неконстантному объекту может приводить к неатомарному обновлению состояний объектов программы, к гонкам при выполнении операций чтения и записи. При поиске подобных дефектов обычно хорошие результаты показывают динамические анализаторы (например, ThreadSanitizer), однако подобную проверку с некоторыми ограничениями можно реализовать и в статическом анализаторе. Этот класс дефектов определён в классификации CWE как CWE-414.

Принцип работы данного проверяющего модуля заключается в сборе статистики обращений к разделяемым регионам памяти. Для каждого разделяемо-

го региона памяти, обращение к которому обнаруживается при моделировании выполнения функции, в состояние программы заносится список примитивов синхронизации, захваченных на момент обращения к данному региону. В конце анализа графа выполнения для всех регионов производится подсчёт случаев сериализованного и несериализованного доступа. В случае, если к региону памяти имелись как сериализованные доступы, так и несериализованные доступы, проверяющий модуль выдаёт диагностическое сообщение с указаниями выражений сериализованного и несериализованного доступа. Предупреждение также не выдаётся в случае, если относительное количество несериализованных доступов не превышает задаваемого пользователем порога. Это дополнительная эвристика, позволяющая частично исключить выдачу предупреждений для регионов памяти, сериализация доступа к которым не планируется. Таким образом, данный проверяющий модуль по принципу проверки является статистическим, а символьное выполнение программы ему требуется для сбора статистики по обращениям к разделяемой памяти.

В соответствии с принципом работы, проверяющий модуль отслеживает следующие события:

1. изменение состояния примитивов синхронизации (захват и освобождение) с помощью отслеживания вызовов соответствующих функций, список которых аналогичен списку функций, перечисленному в описании проверяющего модуля AtomicityChecker (раздел 2.9.3);
2. обращения к разделяемым регионам памяти (запись и чтение).

Соответственно, в резюме вызываемой функции проверяющий модуль сохраняет следующие данные:

1. список всех обращений к нелокальным регионам памяти, которые в вызываемой функции были не защищены примитивом синхронизации при обращении, поскольку они могут быть защищены примитивом синхронизации в вызывающей функции. Для каждого региона также сохраняется список примитивов синхронизации, разблокированных на момент обращения к региону;
2. состояния примитивов синхронизации на момент выхода из функции.

При моделировании вызова функции резюме применяется следующим образом:

1. производится актуализация регионов обращения и регионов примитивов синхронизации, получаемых из резюме функции;
2. для всех регионов памяти из списка отложенной проверки выполняется отложенная проверка:
    (a) если регион является глобальным и если на момент вызова функции имелись захваченные примитивы синхронизации, не содержащиеся в списке разблокированных для проверяемого региона, увеличиваем его счётчик обращений с захватом примитива синхронизации на 1;
    (b) иначе, если регион обращения не является локальным для функции, то: если вызывающая функция не является функцией верхнего уровня, добавить его в список отложенной проверки вызывающей функции; если вызывающая функция является функцией верхнего уровня, то увеличить счётчик обращений без захвата примитива на 1.

Приведённый алгоритм позволяет корректно получать статистику обращений к памяти при использовании МПА методом резюме, поскольку моделирует как изменения состояния примитивов синхронизации, так и обращения к памяти.

2.9.5 BasicStringBoundChecker — проверка на использование корректного индекса при обращении к элементам строки

Данный проверяющий модуль выполняет проверку корректности индекса при обращении к строковым примитивам стандартной библиотеки C++ — экземплярам класса `std::basic_string`. Для обращения к одиночным элементам строки STL можно использовать метод `at()` и оператор `[]`. Индекс при обращении к элементам строки не должен превышать длины строки (включая завершающий элемент). В противном случае, при использовании метода `at()` должно быть сгенерировано исключение; при использовании оператора `[]` проверка индекса на корректность не выполняется, и поведение программы в этом случае не определено. Следствием подобного дефекта может стать аварийное

завершение программы и искажение данных, с которыми работает программа, а также нарушения безопасности, что позволяет классифицировать данный дефект как критический. Этот класс дефектов определён в классификации CERT как STR53-CPP.

Механизм работы модуля заключается в отслеживании длин строк, с которыми анализатор встречается при моделировании выполнения программы. Для этого проверяющий модуль моделирует большое количество функций, операторов и методов, при вызове которых возможно изменение длины строки. Формально моделирование заключается в связывании с регионами памяти, принадлежащим строковым объектам, метаданных — символьного значения, обозначающего длину строки. Проверяющий модуль производит проверку индекса при вызове метода `at()` или индексного оператора. Проверка заключается в сравнении символьного значения длины строки и символьного значения индекса. В случае, если символьное значение индекса больше, чем символьное значение длины строки, проверяющий модуль выдаёт диагностическое предупреждение.

При вызове функции может измениться строковый объект, переданный в неё, что необходимо учитывать при дальнейшем моделировании. Кроме того, при анализе вызываемой функции значения как длины строки, так и индекса могут быть определены недостаточно для однозначного вывода о наличии или отсутствии строкового переполнения при обращению к элементу строки. Это означает, что может потребоваться произвести отложенную проверку операции обращения к элементу строки.

Таким образом, в резюме проверяющего модуля необходимо хранить следующие данные:

1. символьные значения, соответствующие длинам нелокальных для функции строковых объектов;
2. для всех операций взятия элемента, для которых при проверке вне контекста вызова не удалось однозначно установить наличие или отсутствие строкового переполнения, в резюме сохраняется запись, состоящая из региона памяти строкового объекта (который её однозначно идентифицирует), символьное значение его длины и символьное значение индекса.

Поскольку символьное значение длины строки является метаданными, применение резюме данного проверяющего модуля использует описан-

ный выше метод актуализации символьных значений метаданных. Актуализация метаданных выполняется с использованием функции обратного вызова `evalSummarySVal()` следующим образом. При актуализации символьного значения длины строки (точнее, метасимвола, который может содержаться в символьном значении. В случае константной длины строки дополнительных действий по актуализации символьного значения не требуется) выполняется актуализация региона памяти, к которому он привязан. Проверяющий модуль проверяет, связано ли с данным регионом памяти символьное значение в контексте вызывающей функции, и в случае обнаружения возвращает найденное символьное значение, которое становится значением длины строки в контексте вызываемой функции.

Данный проверяющий модуль применяет своё резюме следующим образом.

1. Для всех проверок из списка отложенной проверки производится актуализация соответствующего региона памяти, символьного значения длины и индекса, после чего они сравниваются повторно. В случае, если символьное значение индекса больше значения длины строки, проверяющий модуль выдаёт диагностическое сообщение об обнаруженном дефекте. В случае, если информации для принятия однозначного решения недостаточно, актуализированные значения добавляются в список отложенных проверок вызывающей функции.
2. Для всех регионов памяти из списка нелокальных строк производится актуализация самих регионов памяти и их символьных значений, после чего данная информация сохраняется в состоянии вызывающей функции.

В результате реализации резюме поведение проверяющего модуля при использовании МПА методом резюме и методом встраивания оказывается идентичным.

2.9.6 ThrowWhileCopyChecker — проверка на безопасность обработки исключений в функциях копирования

Задачей данного проверяющего модуля является проверка соответствия уровня безопасности обработки исключений уровню не ниже базового для копирующего конструктора и копирующего конструктора. Гарантия уровня безопасности обработки исключений для этих специальных членов является общепринятой практикой языка C++, поскольку отсутствие подобных гарантий в случае выброса исключения может привести к появлению объекта, находящегося в неконсистентном или в непредусмотренном состоянии, в результате чего любая операция с ним, включая вызов деструктора, приводит к неопределённому поведению. (Базовая гарантия безопасности обработки исключений означает сохранение инвариантов состояния программы и отсутствие утечек, сильная гарантия подразумевает отсутствие эффектов выполнения функции, сгенерировавшей исключение, на состояние программы.) Результатом появления подобного дефекта может стать не только повреждение данных объекта, но и утечка ресурсов. Данный класс дефектов соответствует категории CERT ERR56-CPP.

ThrowWhileCopyChecker является примером проверяющего модуля, работающего с обработкой исключений с помощью резюме. Исходный принцип работы проверяющего модуля следующий. В случае, если анализируемой функцией является конструктор копирования или оператор копирующего присваивания, модуль отслеживает ряд операций:

1. связывания регионов памяти с новым значением отслеживаются с целью определения, имелись ли в при вызове специального метода записи в память инициализируемого объекта, а также для определения восстановления первоначального значения поля объекта в том случае, если программист производит операцию восстановления состояния программы самостоятельно. Для отслеживания записей в память вводится специальный словарь, состоящий из записей «регион памяти — символьное значение региона до первой записи в него». В случае, если записываемое в регион памяти значение символьное равно исходному, запись для данного региона удаляется из словаря;

2. операции выделения памяти отслеживаются для определения отсутствия утечек памяти при обработке исключений;
3. операции выброса и перехвата исключения (операторы `throw`, `try` и `catch` и соответствующие им выражения AST Clang — `CXXThrowExpr`, `CXXTryStmt`, `CXXCatchStmt`) обрабатываются с целью определения неперехваченных в специальном методе исключений, которые могут прервать выполнение специального метода и оставить объект в неконсистентном состоянии.

В случае, если проверяющий модуль обнаруживает, что исключение покидает специальный метод, и хотя бы одно поле объекта при этом было изменено или произошло выделение памяти, проверяющий модуль выдаёт диагностическое сообщение, в котором показываются операция, приведшая к изменению состояния объекта или утечке памяти, и оператор `throw`, приведший к выходу из специального метода.

Таким образом, для реализации МПА с помощью метода резюме необходимо хранить в резюме данного проверяющего модуля следующие записи:

1. для методов классов необходимо сохранять операции записи в регионы памяти полей объекта;
2. для всех функций необходимо сохранять операции записи в регионы памяти аргументов, переданных по ссылке или указателю, поскольку они потенциально могут модифицировать поля других объектов;
3. в резюме необходимо сохранять операции выделения памяти для отслеживания возможных утечек;
4. если выход из функции произошёл в результате генерации неперехваченного исключения, это также необходимо отметить в резюме для построения корректного пути выполнения в вызывающей функции.

При моделировании вызова функции необходимо применить её резюме следующим образом:

1. актуализировать регионы памяти из списков операций записи;
2. если вызывающая функция является методом класса, и если актуализированный регион памяти из списка записей является регионом поля объекта, для которого вызывается метод класса (регион памяти, на который ссылается указатель `this`), то сохранить данную операцию

записи в состоянии вызывающей функции как операцию записи в поле объекта;

3. если актуализированный регион памяти передан в вызывающую функцию по ссылке или указателю, сохранить его в резюме вызывающей функции, поскольку данная операция записи может модифицировать поле другого объекта;
4. сохранить в структуре состояния вызывающей функции список операций выделений памяти, произошедших при вызове функции;
5. если выход из функции произошёл в результате генерации неперехваченного исключения:
    (a) если для данного исключения в вызывающей функции имеется перехватчик, сгенерировать узел обработки исключения, чтобы далее продолжить анализ внутри обработчика исключения;
    (b) если для исключения не имеется обработчика, пометить в резюме вызывающей функции выход в результате генерации исключения и сгенерировать лист графа выполнения вызывающей функции;
    (c) если для исключения не имеется обработчика, вызывающая функция является копирующим конструктором или оператором присваивания, а в структуре состояния имеются записи в регион памяти объекта, сгенерировать диагностическое предупреждение.

Применение резюме данным образом позволяет как строить корректный путь выполнения программы при моделировании обработки исключений, так и отслеживать изменения состояния программы, сохраняя таким образом логику работы проверяющего модуля при использовании МПА методом резюме.

2.9.7 SimpleStreamChecker — проверка операций с файловыми дескрипторами

Задачей данного проверяющего модуля является проверка корректности работы с частью Posix API, связанной с файлами. SimpleStreamChecker обнаруживает операции записи и чтения из закрытого файлового дескриптора, двойное закрытие файлового дескриптора, а также утечки файловых дескрипторов. Чтение и запись в закрытый или не открытый файловый дескриптор приводят к неопределённому поведению (часто следствием является ошибка сегментации или аварийное завершение программы). Утечка дескриптора, т. е. потеря активности переменной, хранящей дескриптор открытого файла, является одним из видов утечки ресурсов и может привести к потере данных, записанных в файл, к исчерпанию системных ресурсов и аварийному завершению программы и . В задачи данного проверяющего модуля входит как поиск дефектов, так и моделирование вызовов отслеживаемых функций Posix, таких как `fopen`, `fclose`, `fwrite` и др, включая возвращаемый результат. Данный тип проверки соответствует классификации CERT FIO42-C.

Исходный принцип работы данного проверяющего модуля следующий. SimpleStreamChecker отслеживает вызовы функций открытия и закрытия файла, помечая в своём состоянии в GDM файловый дескриптор как открытый или закрытый соответственно, используя для этого карту «символ типа `FILE *` — статус файла». Обработчик функции открытия файла также разделяет состояние на состояние с удачным открытием файла с ненулевым возвращамым результатом и на неудачное с нулевым возвращаемым указателем. Обработчик, связываемый с функциями обращения к содержимому файла, проверяет, является ли файл, переданный ему, закрытым, и, если он является закрытым, выдаёт диагностическое предупреждение об обращении к закрытому дескриптору. Обработчик события потери символом активности проверяет, не входит ли символ в число символов открытых файловых дескрипторов. Потеря активности символом открытого дескриптора является утечкой, и в случае обнаружения утечки дескриптора модуль выдаёт диагностическое предупреждение об обнаружении утечки файлового дескриптора.

Для реализации резюме для данного проверяющего модуля в соответствии со стандартной процедурой выделим хранимые изменения состояния и необходимые отложенные проверки.

1. Проверяющий модуль должен хранить изменения состояния файловых дескрипторов, не локальных для анализируемой функции. В резюме сохраняется состояние нелокального дескриптора на момент выхода из функции (в конце трассы выполнения) в виде карты «дескриптор–состояние».
2. В случае, если файловый дескриптор не является локальным и на момент анализа операции обращения к файлу точно не известно, был ли файл закрыт или открыт, проверяющий модуль планирует в своём резюме отложенную проверку события обращения и связывает её с текущим узлом графа выполнения. При этом для экономии вычислительных ресурсов для каждого дескриптора с неизвестным состоянием имеет смысл хранить лишь одно обращение, поскольку для одного дескриптора отложенная проверка всех обращений должна показать или одновременное наличие эффекта, или его отсутствие. При этом предпочтительным для сохранения является первое обращение в процессе моделирования выполнения, поскольку в случае обнаружения дефекта оно обеспечит трассу наименьшей длины.
3. В случае, если файловый дескриптор не является локальным и на момент анализа события утечки дескриптора точно не известно, был ли файл открыт, проверяющий модуль планирует в своём резюме отложенную проверку события утечки и связывает её с текущим узлом графа выполнения.
4. Действия модуля по разделению состояния и связыванию символьного значения с её выражения являются локальными и в резюме функции не сохраняются.

При моделировании вызова функции проверяющий модуль применяет её резюме следующим образом:

1. производится актуализация всех символов, содержащихся в структурах данных резюме функции;
2. для каждой отложенной проверки события чтения из файлового дескриптора производится проверка состояния актуализированного сим-

вола дескриптора на момент вызова функции. Если файловый дескриптор был закрыт, в вызываемой функции произошло обращение к содержимому закрытого файла, и в этом случае выдаётся диагностическое предупреждение; если состояние файлового дескриптора на момент вызова неизвестно, событие обращения добавляется в список отложенной проверки вызывающей функции. Если дескриптор находится в открытом состоянии, произошло обращение к открытому дескритору. В этом случае дефект отсутствует, и дальнейших действий не требуется;

3. для каждой отложенной проверки события утечки файлового дескриптора производится проверка состояния актуализированного символа дескриптора на момент вызова функции. Если файловый дескриптор был открыт, в вызываемой функции произошла утечка дескриптора, и в этом случае выдаётся диагностическое предупреждение; если состояние файлового дескриптора на момент вызова неизвестно, событие обращения добавляется в список отложенной проверки вызывающей функции. Если дескриптор находится в закрытом состоянии, дефект однозначно отсутствует;
4. для всех актуализированных символов дескрипторов, имеющихся в списке изменивших состояние в резюме вызываемой функции, данное состояние устанавливается в качестве текущего в вызывающей функции.

Проведённые изменения, связанные с реализацией резюме, позволяют полностью сохранить функциональность проверяющего модуля SimpleStreamChecker.

2.10 Построение отчёта о дефекте

Построение отчёта является важным элементом работы статического анализатора. Недостаточно просто найти дефект и указать место его возникновения. В случае анализа путей выполнения программы между различными значимыми для данного дефекта точками программы (например, открытие и закрытие файла) может проходить длинный путь через большое количество операто-

ров. При этом путь выполнения может включать в себя условные операторы, циклы и вызовы других функций. Однако субъективная сложность отслеживания пути выполнения быстро растёт при увеличении длины пути. Если дефект не локализован в пределах нескольких строк кода или небольшой функции, разработчику становится практически невозможно определить путь выполнения, на котором проявляется дефект. Таким образом, при анализе программы методом анализа её путей выполнения необходимо выполнять построение подробного отчёта, однозначно указывающего условия, при которых проявляется дефект, и соответствующий им путь выполнения.

При использовании межпроцедурного анализа методом встраивания путь, проходимый внутри функции, отображается в графе выполнения как часть общего пути выполнения, поэтому проблем при построении пути при генерации отчёта не возникает. Однако при применении метода резюме возникает проблема потери информации о части пути, проходимом внутри вызываемой функции, поскольку явного построения поддеревьев графа выполнения программы более не происходит. В результате замены встраивания функции на применение её резюме граф выполнения программы изменяется следующим образом. Вместо подграфа вложенного вызова в графе выполнения появляются отдельные точки выполнения, условно соответствующие конечным точкам подграфа вложенного вызова функции. В связи с этим при построении отчёта о найденном дефекте нельзя традиционным образом указать путь, который прошло выполнение программы при вызове функции. Для построения отчёта при межпроцедурном анализе методом резюме в настоящей работе предложен следующий метод.

Задачей проверяющих модулей является отложенная проверка: на основе резюме вызываемой функции и состояния на момент вызова проверяющий модуль должен сделать вывод о необходимости выдачи предупреждения (срабатывании). Имеется два варианта срабатывания проверки. В первом случае выдача предупреждения происходит внутри вызываемой функции. Допустим, что уровень вложенности вызова равен единице. Такое допущение допустимо, поскольку к нему можно свести стек вызовов произвольного уровня вложенности. В этом случае конечной точкой трассы является узел графа выполнения вызываемой функции. В случае, если этот узел известен, от него можно построить трассу до корневого узла графа выполнения. Данная трасса-подграф будет

являться путём, проходимым внутри функции при выполнении программы до критической точки.

Второй случай предполагает построение пути при необходимости показать путь внутри вызванной функции целиком, от точки входа до точки выхода. Данная задача сводится к первой при условии, что в качестве конечной точки выбирается лист графа выполнения, соответствующий выбранной ветви выполнения резюме.

Таким образом, для корректного построения пути при моделировании вложенного вызова функции достаточно иметь информацию или об узле графа выполнения вызываемой функции, для которого выдаётся срабатывание, или о листе графа выполнения, который относится выбранный путь выполнения при применении резюме. Для этого достаточно хранить ссылку на этот узел. В случае построения пути изнутри вызываемой функции за хранение ссылки может отвечать проверяющий модуль, генерирующий срабатывание. В случае построения полного пути по вложенному вызову ссылку на лист графа выполнения вызываемой функции можно хранить в узле применения резюме в качестве дополнительной информации.

Поясним описанный метод примером. Рассмотрим функцию следующего вида:

```
1  void close_file(bool flag, FILE *f) {
2    ...
3    if (flag)
4      fclose(f);
5  }
```

Пусть задачей проверяющего модуля является проверка двойного закрытия файлового дескриптора. Резюме функции `close_file()` будет состоять из двух ветвей: в первой ветви ($flag \equiv false$) никакой дополнительной информации не хранится, во второй ($flag \equiv true$) имеется событие закрытия файла. При анализе приведённой функции вне контекста вызывающей функции проверяющий модуль не может доказать, что внутри этой функции дескриптор закрывается во второй раз, поскольку на всех путях выполнения дескриптор закрывается не более одного раза. Отсюда следует необходимость выполнения отложенной проверки при применении резюме этой функции, когда контекст

вызова позволит однозначно определить состояние файлового дескриптора при вызове `fclose(f)` внутри `close_file()`.

Далее, пусть имеется функция, вызывающая `close_file()` и имеющая следующий вид.

```
1  void double_close(FILE *file) {
2    fclose(file);
3    close_file(true, file);
4  }
```

На момент вызова функции `close_file()` (и применения резюме) дескриптор `file` является закрытым, т. е. его состояние однозначно определено. Это значит, что может выполниться отложенная проверка, которая покажет, что происходит закрытие уже закрытого дескриптора. В этом случае происходит срабатывание, путь к которому нужно построить изнутри вызываемой функции. Рис. 2.1 поясняет применение этой схемы в данном примере. Поток управления внутри функции показан сплошными линиями, отношение отложенной проверки в заданном узле — штрихпунктирной, а пунктиром показана трасса построения отчёта.

Рассмотрим более общий случай произвольной вложенности вызовов. В случае построения пути изнутри вызываемой функции информации об узле срабатывания может оказаться недостаточно, поскольку по одному узлу невозможно восстановить всю цепочку вложенных вызовов. Это объясняется тем, что резюме функции верхнего уровня хранит ссылки только на узлы следующего уровня вложенности. Для решения этой задачи проверяющий модуль должен самостоятельно хранить стек вызовов, для которого строится путь выполнения.

Рассмотрим данный случай на примере. Модифицируем представленный в предыдущем примере набор функций, добавив в него промежуточный вызов:

```
1  void close_file(bool flag, FILE *f) {
2    ...
3    if (flag)
4      fclose(f);
5    ...
6  }
```

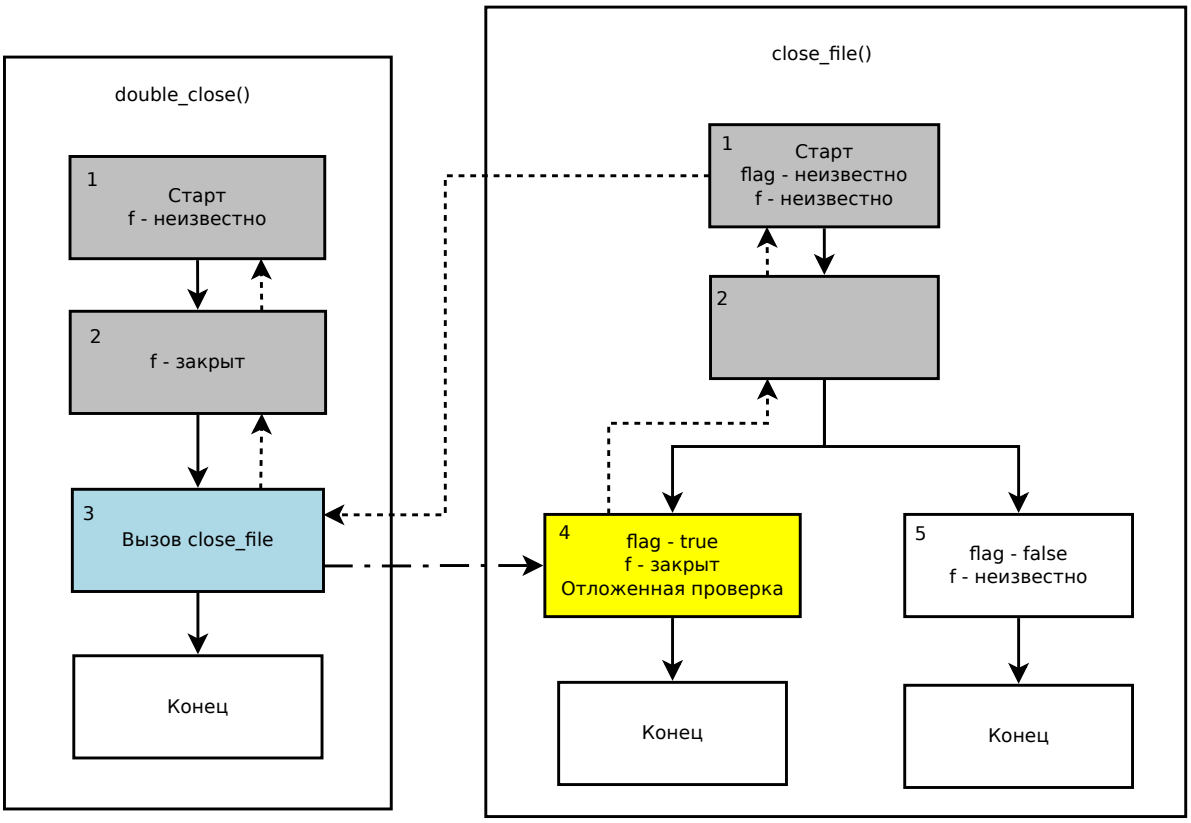

Рисунок 2.1 — Построение отчёта при использовании резюме вложенного вызова функции

```
 7
 8  void potential_double_close(bool flag, FILE *file) {
 9    close_file(flag, file);
10  }
11
12  void double_close(FILE *file) {
13    fclose(file);
14    potential_double_close(true, file);
15  }
```

Резюме функции `close_file()` аналогично предыдущему случаю. Резюме функции `potential_double_close()` состоит из двух ветвей: в первой ветви ($flag \equiv false$) никакой дополнительной информации не хранится, во второй ($flag \equiv true$) имеется отложенная проверка события закрытия файла, содержащая ссылку на узел №4 графа выполнения функции `close_file()`. В качестве точки отложенной проверки для функции `potential_double_close()`, при этом в качестве целевого узла указывается не узел применения резюме, а

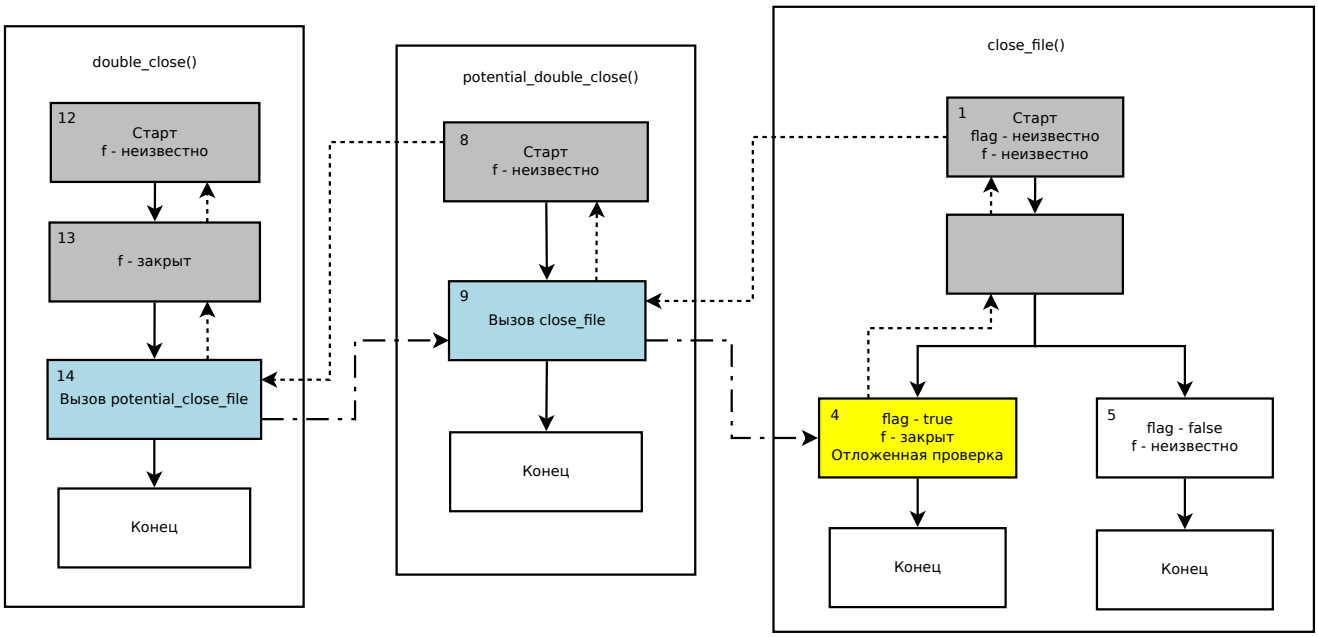

Рисунок 2.2 — Построение отчёта при использовании резюме вызова функции многократной вложенности

его предшественник. Это делается для упрощения, поскольку узел применения резюме на момент сохранения информации ещё не создан (сохранение информации проверяющих модулей является одним из действий по его созданию, и к этому моменту проведены не все действия по применения резюме), но информация об узле предвызова и целевом узле вызываемой функции позволяет однозначно идентифицировать ветвь резюме (или группу ветвей), затрагиваемую при прохождении пути до заданной точки.

Схема построения пути в приведённом случае показана на рис. 2.2.

На рис. 2.3 показано построение пути при использовании полного пути выполнения внутри вызываемой функции.

При показе пути внутри вложенного графа возникает проблема именования, поскольку внутри графа выполнения функции схема именования является локальной и не связана со схемой именования вызывающей функции. Данную проблему можно решить уже описанным способом с помощью переименования именованных символьных значений из контекста вызываемой функции в контекст вызывающей. При этом можно выполнять переименование не для всех символьных значений, имеющихся внутри пути выполнения, а только для тех, информация о которых непосредственно представляется пользователю. Это уменьшает временные затраты, поскольку актуализация символьного значения может быть достаточно дорогой операцией.

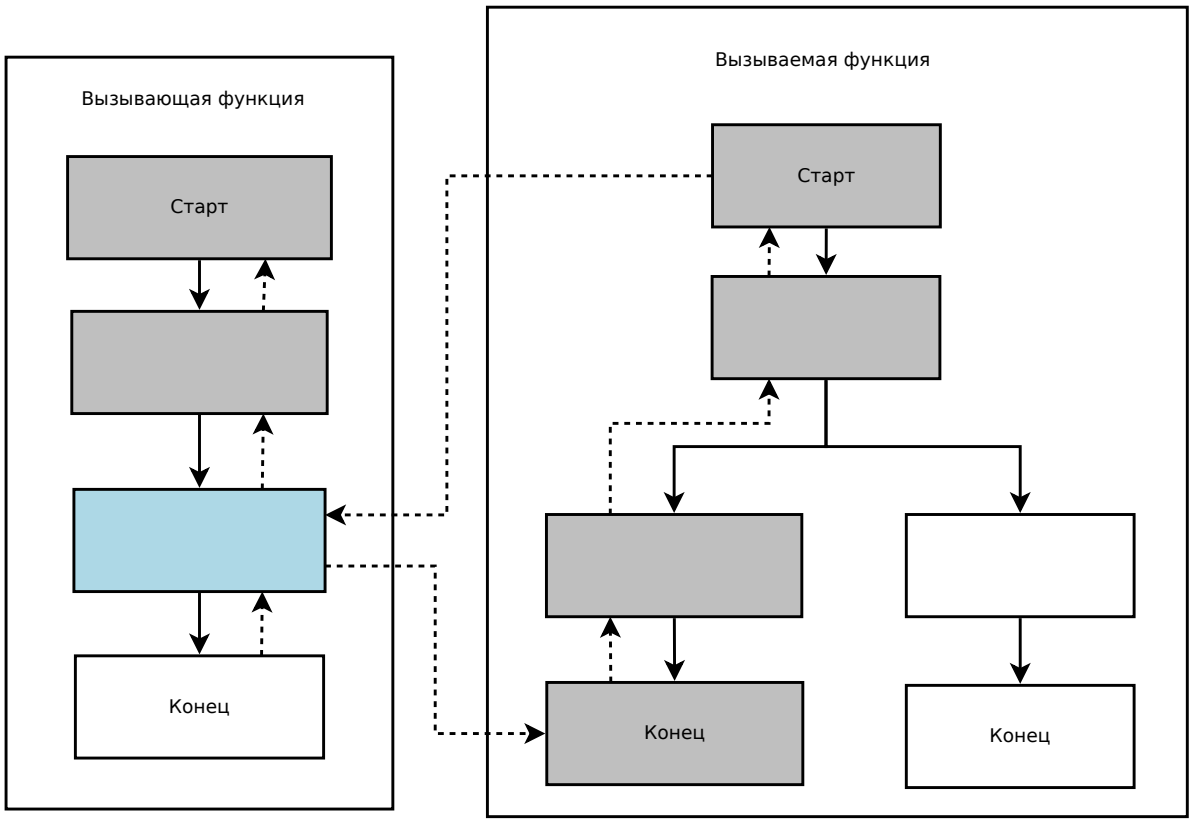

Рисунок 2.3 — Построение отчёта при использовании резюме: случай полной трассы внутри функции

3 Межмодульный анализ

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

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

Различные инструменты анализа кода реализуют межмодульный анализ по-разному. В случае динамического анализа анализируются не определения функций, а сгенерированный код: объектный код или промежуточное представление. В этом случае код вызываемых функций непосредственно становит-

ся доступным анализатору или в момент загрузки анализируемого модуля, или в процессе выполнения программы при загрузке подгружаемых модулей.

Большинство статических анализаторов, имеющих возможность межмодульного анализа, используют в качестве входных данных анализа промежуточное представление. Так, статический анализатор Svace [6] использует для анализа байт-код LLVM, Coverity SAVE [5] — компилятор Edison Design Group, и строят глобальный граф вызовов анализируемого проекта, производя разбор байт-кода, предварительно сгенерированного из исходных файлов проекта. Clang Static Analyzer, в отличие от многих других инструментов анализа исходного кода методом символьного выполнения, использует в качестве входных данных не промежуточное представление или объектный код, а непосредственно исходные файлы. Одной из основных причин этого является удобная и удачно спроектированная реализация абстрактного синтаксического дерева, в котором представлена вся информация о программе без каких-либо предварительных оптимизаций или потерь информации. Это обстоятельство требует применения иных подходов к межмодульному анализу, нежели в Coverity SAVE или Svace. В связи с этим в данной работе для проведения межмодульного анализа с использованием фреймворка Clang Static Analyzer предлагается выполнять слияние синтаксических деревьев различных файлов, связанных вызовами функций.

В результате проведённого исследования была разработана архитектура анализатора, позволяющего производить межмодульный анализ программы на языках C и C++ и использующем для анализа непосредственно исходный код программы. Разработанная схема межмодульного анализа интересна тем, что позволяет использовать различные алгоритмы межпроцедурного анализа, т. е. как анализ методом встраивания, так и анализ методом резюме, без каких-либо дополнительных модификаций как самого алгоритма межпроцедурного анализа, так и проверяющих модулей. «Прозрачный» межмодульный анализ позволит в дальнейшем провести корректное сравнение различных алгоритмов межпроцедурного анализа при использовании межмодульного подхода. Данное сравнение представляет интерес, поскольку при использовании межмодульного анализа количество анализируемых путей внутри программы быстро растёт, что может представлять сложность при использовании межпроцедурного анализа.

3.1 Реализация межмодульного анализа в статическом анализаторе, использующем для анализа непосредственно исходный код программы

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

Именно Clang Static Analyzer является целевым анализатором для реализации межмодульного анализа в данной работе, поскольку ранее автором был реализован ряд других проектов с использованием данного анализаторного фреймворка, и реализация межмодульного анализа является их завершающей частью. Кроме того, использование межмодульного анализа резко увеличивает количество требующих анализа путей программы и представляет интерес при сравнении производительности и качества межпроцедурного анализа методом встраивания и разработанного автором метода резюме. Таким образом, в данной работе рассматривается решение проблемы межмодульного анализа для случая использования анализатором непосредственно исходного кода программы.

Для решения проблемы определения местоположения анализируемых функций, в данной работе реализован трёхфазный анализ с сохранением промежуточных результатов в файлах в директории проекта. Таким образом, анализ разделяется на три фазы: фаза сборки, фаза предобработки данных и непосредственно сам анализ исходных кодов. На рисунке 3.1 представлена схема взаимодействия инструментов, используемых на различных фазах анализа, в виде диаграммы IDEF0. На этой схеме модули, реализованные в данной работе полностью, обозначены белым цветом, а серым обозначены модули, существо-

вавшие ранее и доработанные для использования при межмодульном анализе: `clang` является непосредственно статическим анализатором, использованным для реализации межмодульного анализа, а Perl-скрипт `ccc-analyzer` входит в состав вспомогательного пакета `scan-build` и служит для формирования командной строки запуска статического анализатора `clang` и его запуска.

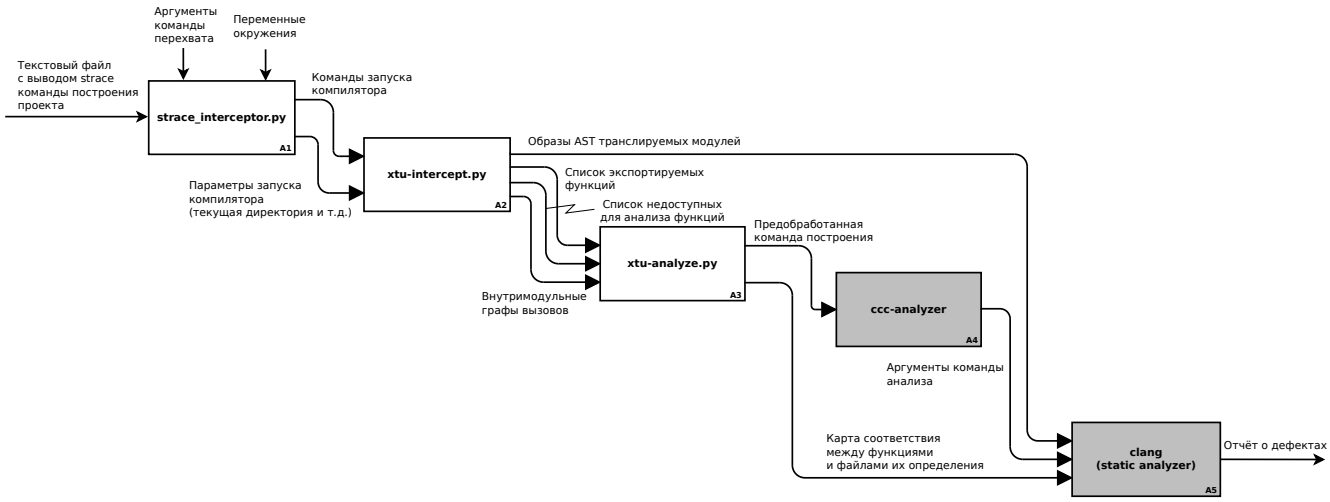

Рисунок 3.1 — IDEF0-диаграмма взаимодействия разработанных программных инструментов при межмодульном анализе

3.2 Фаза сборки

На фазе сборки специальный инструмент (`strace_interceptor`), использующий программу `strace`, собирает информацию о транслируемых модулях, которые должны быть проанализированы. Программа `strace` является утилитой ОС Linux, задачей которой является трассировка указываемого процесса (и, опционально, его потомков) и вывод информации о системных вызовах трассируемых процессов в стандартный поток вывода. Инструмент `strace_interceptor` реализован с использованием языка Python. Данный инструмент был разработан в рамках данной работы с целью поддержки различных систем сборки, в том числе, использующих сборку проекта в «песочнице», без изменения сборочных конфигураций. В число поддерживаемых систем сборки входят Makefile, Open Build Service (OBS — при локальном построении проекта с использованием клиентского приложения `osc`) [68] и Git Build System

(GBS) [69], причём поддерживается, в том числе, сборка в «песочнице» другой архитектуры с использованием эмуляторов (например, Qemu). Разработанный инструмент выполняет разбор вывода утилиты strace и использует информацию о системных вызовах `chroot()`, `chdir()`, `vfork()` и `execve()` для поиска вызовов компилятора и записывает текущую директорию, корневую директорию, команду сборки и переменные окружения в файл. Корневая директория отличается от стандартной («/») в тех случаях, когда был выполнен вызов `chroot()`. Текущая директория задаётся относительно корневой. Задачей этого же инструмента является поиск директорий с включаемыми файлами, связанными с запускаемым при сборке компилятором. Затем для каждой обнаруженной команды сборки другой инструмент, использующий программный интерфейс (API) Clang (`clang-func-mapping`), записывает сигнатуры видимых извне (экспортируемых) определений функций данного модуля трансляции и сигнатуры используемых (импортируемых) функций, определения которых в данном модуле трансляции недоступны, в служебные файлы в директории проекта. Этот же инструмент для каждой обнаруженной функции строит локальный граф вызовов, который также сохраняется в служебный файл. На фазе сборки дополнительно создаются образы абстрактных синтаксических деревьев для всех модулей трансляции, что упрощает дальнейший импорт и позволяет избежать повторных вызовов компилятора для каждого импортируемого файла.

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

1. Определения функций для различных архитектур могут не совпадать непосредственно, в отношении текстового сравнения компилируемого исходного кода. Это возможно, поскольку для разных архитектур мо-

гут быть использованы различные фрагменты кода в директивах условной компиляции.

2. По тем же причинам для разных архитектур может не совпадать окружение функции: определения используемых типов, зависимые определения. Кроме того, для разных архитектур могут использоваться различные включаемые файлы.
3. Даже при совпадении окружения и текста функций, для различных архитектур могут не совпадать определения основных типов. Так, тип char является по умолчанию знаковым (`signed char`) для платформы x86 и беззнаковым (`unsigned char`) для платформы ARM.

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

Все вышеперечисленные факторы означают, что выбирать модуль трансляции для импорта определения функции надо крайне осторожно. Для решения этой проблемы можно предложить несколько способов. В настоящей работе было использовано менее общее, но более простое в реализации решение. Для такого приближённого учёта архитектур достаточно знать, объектные модули каких архитектур могут компоноваться друг с другом. Например, объектные модули, скомпонованные для архитектур «arm» и «thumb» могут быть использованы для компоновки в один объектный файл при использовании определённых опций компилятора, тогда как «arm» и «x86» не могут. После построения такой матрицы, можно различать определения функций по тройке <путь к файлу, сигнатура функции, целевая архитектура>, и использовать эту тройку для выбора нужного определения функции и его импорта. Этот подход, хотя и прост в реализации, однако, не решает проблемы, связанной с использованием различных опций компилятора для одной и той же архитектуры, поскольку выбираться будет только один файл. Другим, и более общим решением видится поддержка «дерева сборки». В этом случае на фазе сборки отслеживается полное дерево компиляции, ассемблирования и компоновки (и, возможно, сборки в бинарный образ) для объектных файлов верхнего уровня. Это позволяет точно определить, какая функция из какого исходного файла должна быть

использована для моделирования межфайлового вызова. Кроме того, это частично решает проблему модулей трансляции, которые собираются несколько раз в рамках сборки для одной архитектуры, поскольку для каждой из сборок становится известен высокоуровневый модуль, для которого она производится. Вместе с тем, этот подход имеет и некоторые проблемы. Во-первых, при его использовании необходимо отслеживать не только вызовы компилятора, но также вызовы ассемблера и компоновщика. Во-вторых, некоторые файлы собираются очень часто. Так, при построении образа ОС Android встречаются файлы, которые собираются 38 раз. Это может означать необходимость множественного повторного анализа уже проанализированных модулей трансляции. Таким образом, выбор между этими вариантами нельзя назвать однозначным, поскольку у каждого из них есть свои преимущества и недостатки.

3.3 Фаза предобработки данных

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

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

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

3.4 Фаза анализа. Слияние синтаксических деревьев

На вход фазы анализа поступает файл с упорядоченным набором файлов для анализа. Все эти файлы добавляются в очередь анализа в порядке следования в исходном файле, после чего полученная очередь начинает обрабатываться пулом рабочих процессов анализатора. Данная схема позволяет осуществлять анализ с очень высокой степенью параллелизма, т. к. различные процессы не используют разделяемых ресурсов, и её производительность линейно растёт с увеличением количества процессоров (проверена масштабируемость до 32-х процессоров включительно). Каждый анализатор из пула анализирует свой транслируемый модуль, подгружая определения функций и структур данных, от которых они зависят, по мере необходимости.

В данной разработке был реализован межмодульный анализ с использованием реализации класса `ASTImporter`, который является частью интерфейса сериализации синтаксических деревьев компилятора Clang и отвечает за слияние синтаксических деревьев различных транслируемых модулей. Импорт фрагментов синтаксического дерева (т. е. данный класс) уже был частично реализован в Clang. Реализованная функциональность была расширена, т. к. значительная часть необходимых функций не была реализована ранее. В результате появилась возможность полноценного импорта фрагментов синтаксических деревьев функций в основной контекст синтаксического дерева. Когда анализатор обнаруживает функцию с недоступным определением, производится поиск сигнату-

ры этой функции в сгенерированном отображении. Если в результате поиска сигнатура функции была найдена, загружается синтаксическое дерево файла, содержащего определение этой функции. Затем эта функция импортируется в основной контекст синтаксического дерева вместе с необходимыми определениями и объявлениями.

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

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

Первой проводимой операцией при импорте объявления из исходного контекста является поиск похожего объявления в целевом синтаксическом дереве. Этот поиск часто включает в себя рекурсивный обход вложенных объявлений для определения, являются ли два объявления структурно эквивалентными. В данной работе, однако, испытан ряд простых и легковесных эвристик, ускоряющих поиск за счёт частичного отказа от рекурсивного обхода. Рекурсивная проверка структурной эквивалентности выполняется только в случае, если эти эвристики не смогли однозначно показать различие или эквивалентность.

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

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

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

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

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

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

всё равно имеет смысл проанализировать, поскольку его работоспособность, как правило, проверяется при тестировании. Вторая стратегия заключается в разрешении конфликтов между определениями. Её недостаток заключается в том, что у анализатора может не быть данных о программе для корректного разрешения конфликта.

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

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

Ниже приведён пример подобных конфликтующих определений, найденный в библиотеке `zlib` [70]. В заголовочном файле zlib.h находится определение следующего вида:

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

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

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

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

Ещё одним примером является компиляция с различными опциями препроцессора (такими как определения символов препроцессора) различных исходных файлов, использующих один и тот же включаемый файл, что приводит к появлению различающихся определений одной и той же структуры данных в результате условной компиляции. Например, из-за опций препроцессора могут быть объявлены дополнительные поля структуры данных. Эти определения являются различными для компоновщика, поскольку они не эквивалентны побайтово. С другой стороны, выбор между различными конфликтующими определениями подобного рода не является тривиальным на уровне компилятора (а это тот уровень, на котором работает Clang Static Analyzer и анализ на уровне исходных кодов), поскольку эти определения должны быть помещены в одну область видимости. Данная проблема, возможно, может быть решена с помощью внутреннего переименования конфликтующих определений. Пример подобной структуры приведён ниже.

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

В приведённом случае проблема возникает при условиях, аналогичном предыдущему случаю: если при слиянии импортирующий модуль имеет определение структуры без поля, а импортируемый модуль содержит определение структуры с полем и код программы, это поле использующий, т. е. при слиянии транслируемого модуля, в котором символ препроцессора не определён,

и модуля, в котором он определён. Проблемы возникает и в обратном случае, поскольку оба определения структуры имеют различные смещения полей, находящихся после опционального поля.

Ещё одной причиной несоответствия определений является тот факт, что некоторые элементы определений структур, согласно стандарту языка C++, создаются «по требованию», т. е. лишь в том случае, если они реально используются в коде [71]. К таким необязательным определениям относятся конструкторы по умолчанию, конструктор копирования и деструктор класса, которые создаются лишь в том случае, если они, во-первых, используются в коде, и, во-вторых, не имеют перегруженных определений. В отношении анализа программы данная проблема становится особенно важной в случаях, когда поля класса сами имеют нетривиальные конструкторы и деструкторы, поскольку в этом случае генерируемые компилятором специальные методы должны вызывать конструкторы и деструкторы полей класса. В результате допустима ситуация, при которой импортируемое определение класса имеет созданный компилятором специальный метод, а аналогичное определение в импортирующем модуле трансляции его не имеет. Решением этой проблемы является либо импорт специальных методов в случае обнаружения подобного несоответствия, либо самостоятельное создание недостающих специальных методов в синтаксическом дереве. В данной работе необходимые специальные методы создаются с использованием методов класса `Sema` — реализации семантического анализатора в составе компилятора Clang.

4 Тестирование разработанного программного комплекса

Разработанная система, реализующая описанные методы, первоначально испытывалась с использованием синтетических тестов, разработанных специально для тестирования корректности реализации. Однако синтетические тесты зачастую не отражают реальное качество анализатора в связи с ограниченностью возможных синтаксических конструкций, а также ввиду ограниченного объёма тестирующего кода. Кроме того, синтетические тесты не позволяют исследовать такие показатели разработанных методов, как масштабируемость и производительность анализатора. Наконец, поскольку разработанный комплекс предполагается к внедрению и промышленному использованию, имеет смысл произвести тестирование на наиболее характерных проектах. В связи с этим возникает необходимость проведения тестирования с использованием кода реальных проектов.

4.1 Выбор тестовых проектов

Для тестирования системы и оценки её количественных и качественных характеристик можно выбрать ряд пакетов различного размера. Желательно также, чтобы среди выбранных проектов присутствовал ряд проектов, не проходивших ранее проверку с использованием известных статических анализаторов. Поскольку после проверки другими статическими анализаторами и исправления найденных ошибок уменьшается количество потенциальных положительных срабатываний, результаты тестирования будут искажены. С другой стороны, вопрос взаимодействия с другими статическими анализаторами также интересен и представляет практический интерес, поскольку для поиска дефектов в разрабатываемом программном коде зачастую используется не один, а несколько анализаторов, причём как статических, так и динамических. Особенный интерес представляет возможность нахождения дефектов после проверки другими анализаторами, поскольку это свидетельствует о возможности дополнения существующих анализаторов или их замены. Это позволит прове-

сти сравнение результатов разработанного комплекса на проектах, прошедших статический анализ ранее, и проектах, его не проходивших.

Что касается крупных програмных комплексов, то на данную роль была отобрана ОС Android. Данная ОС включает в себя 389 пакетов, связанный между собой, и имеет суммарный объём анализируемого кода на языках C и C++ около 1,2 млн. SLoc. Кроме того, данный программный комплекс включает многие из уже описанных ранее пакетов, что позволяет заменить данным комплексом остальные, которые уже входят в его состав. Особый интерес для тестирования представляет то обстоятельство, что ОС Android может собираться для разных архитектур (x86, x86_64, ARM и MIPS), причём исполняемые файлы различных архитектур могут генерироваться во время одной сборки. Большое количество межфайловых связей делают проект интересным для межмодульного анализа и исследования масштабируемости разработанных методов межпроцедурного анализа. Общие характеристики исходного кода ОС Android приведены в таблицах 4.1 и 4.2.

Таблица 4.1

Характеристики тестовой базы ОС Android

Характеристика	Значение
Количество строк кода	1,2 млн
Количество файлов исходного кода	31038
Количество транслируемых модулей	20635
Количество архитектур на построение	2
Количество пакетов	389

Таблица 4.2

Код ОС Android на языках C и C++

Тип файла	Файлов	Пустых строк	Комментариев	Строк кода
Код C++	18007	950783	952630	5036304
Код C	13031	783549	1056897	4711475
Header	26669	690283	1375129	2659402
Всего	57707	2424615	3384656	12407181

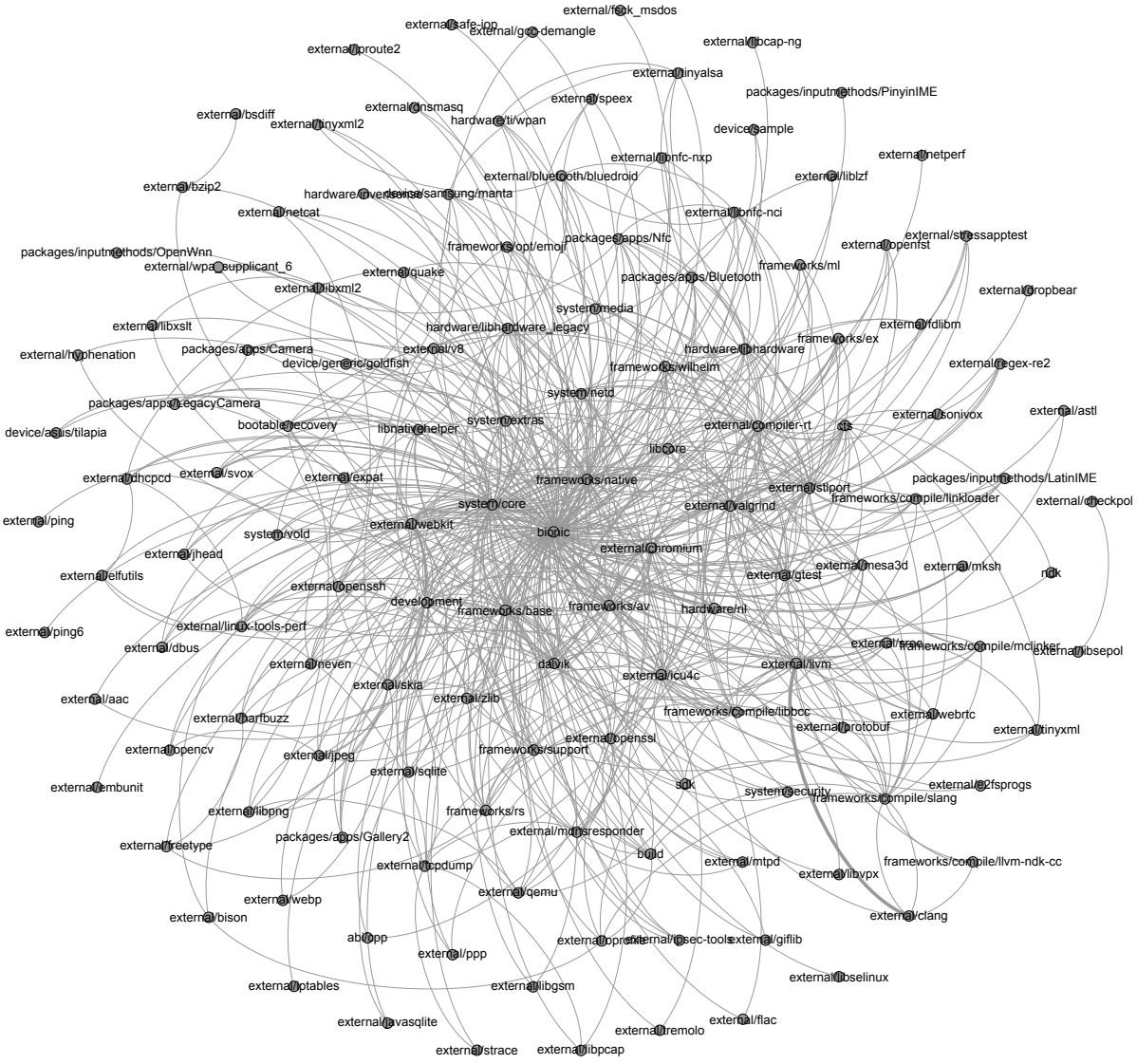

Рисунок 4.1 — Граф межпакетных вызовов ОС Android 4.2

4.2 Методика тестирования

Для тестирования использовался сервер, конфигурация которого описана в таблице 4.3:

Таблица 4.3

Характеристики тестового стенда

Характеристика	Значение
Модель процессора	Intel Xeon E5-2600 2.60 ГГц
Количество физических процессоров	2
Количество физических ядер процессора	8
Количество виртуальных ядер процессора	16 (Hyper-Threading)
Количество виртуальных ядер системы	32
Объём оперативной памяти	96 Гб
Тип оперативной памяти	DDR3

Процессы анализаторов запускались параллельно на всех виртуальных процессорах. Время анализа измерялось от момента старта фазы анализа до момента завершения последнего процесса анализатора и выдачи финального отчёта.

В качестве базовой версии Clang Static Analyzer использовалась версия 3.4.1.

Анализатор (Clang Static Analyzer) имеет ряд настроек, непосредственно влияющих как на качество анализа, так и на его время. Ниже в таблице 4.4 перечислены опции анализатора, при которых производилось сравнение.

Параметр `max-nodes` непосредственно влияет на полноту анализа, поэтому в дальнейшем измерения будут проходить с его учётом. Он станет одним из варьируемых параметров.

Таблица 4.4

Тестовые настройки анализатора

Имя параметра	Назначение параметра	Значение
`mode`	Режим анализа. Влияет на максимальный размер функции, для которой выполняется анализ вложенного вызова	`deep`
`c++-inlining`	Анализ вызовов членов классов (C++)	`destructors`
`cfg-temporary-dtors`	Анализ вызовов деструкторов временныз объектов	`false`
`c++-stdlib-inlining`	Анализ вызовов стандартной библиотеки C++	`true`
`ipa-always-inline-size`	Размер функций, для которых всегда выполняется МПА	3
`c++-allocator-inlining`	Анализ аллокаторов (C++)	`false`
`c++-template-inlining`	Анализ шаблонов функций (C++)	`true`
`c++-container-inlining`	Анализ контейнерных классов (C++)	`true`
`c++-shared_ptr-inlining`	Анализ указателей `shared_ptr` (C++)	`false`
`max-times-inline-large`	Максимальное количество анализа вложенного вызова функции	32
`max-nodes`	Максимальное количество анализируемых узлов графа выполнения функции	*варьируется*

4.3 Тестирование покрытия и производительности

В качестве критерия производительности при сравнении межпроцедурного анализа методом встраивания и методом резюме можно взять количество узлов графа выполнения, обрабатываемых в единицу времени. Данный параметр может использоваться для режима встраивания непосредственно, однако для режима резюме он не отражает реальной производительности системы, поскольку каждому из узлов применения резюме в результирующем графе выполнения соответствует полноценный путь внутри одной или нескольких функций. Однако, количество узлов графа, соответствующих узлам применения резюме, можно вычислить рекурсивно, способом, схожим со способом построения отчёта при вложенном вызове функции:

1. Установить начальный счётчик узлов `cnt` $= 0$, $Proceed = \varnothing$ — набор учтённых узлов
2. Для всех заданных конечных точек графа выполнения выполнить шаги 3 и 4.
3. Пусть N — узел применения резюме, M — узел графа выполнения функции, на который хранится указатель в N.
4. Пока M — не корневой узел графа выполнения, и M не учтён:
    (a) Увеличить `cnt` на 1
    (b) Добавить M в $Proceed$
    (c) Если M — узел применения резюме, выполнить для M шаги алгоритма 1–3
    (d) Установить M равным первому родительскому узлу M

Таким образом, можно осуществить подсчёт количества *эквивалентных* узлов графа выполнения, обрабатываемых в единицу времени. Для графа верхнего уровня вместо списка конечных узлов используется список всех узлов графа, поскольку часть ветвей выполнения графа верхнего уровня анализируется не полностью.

При измерениях необходимо учитывать, что часть времени расходуется не анализатором на анализ кода программы, а компилятором — на разбор исходного файла и построение синтаксического дерева. Данные затраты не меняются

Лимит количества узлов	Время анализа	Количество проанализированных узлов графа	Узлов графа в секунду
Метод встраивания			
8000	0:11	$2{,}70 \cdot 10^8$	$1{,}12 \cdot 10^6$
16000	0:17	$5{,}00 \cdot 10^8$	$8{,}33 \cdot 10^5$
32000	0:28	$9{,}25 \cdot 10^8$	$7{,}34 \cdot 10^5$
64000	0:50	$1{,}72 \cdot 10^9$	$6{,}66 \cdot 10^5$
128000	1:30	$3{,}21 \cdot 10^9$	$6{,}44 \cdot 10^5$
256000	2:51	$6{,}01 \cdot 10^9$	$6{,}10 \cdot 10^5$
512000	5:27	$1{,}13 \cdot 10^{10}$	$5{,}89 \cdot 10^5$
Метод резюме			
2000	0:09:30	$1{,}94 \cdot 10^9$	$1{,}29 \cdot 10^7$
4000	0:12	$3{,}90 \cdot 10^9$	$1{,}30 \cdot 10^7$
8000	0:19	$8{,}49 \cdot 10^9$	$1{,}18 \cdot 10^7$
16000	0:32	$1{,}68 \cdot 10^{10}$	$1{,}12 \cdot 10^7$
32000	0:52	$3{,}31 \cdot 10^{10}$	$1{,}22 \cdot 10^7$
64000	1:38	$6{,}70 \cdot 10^{10}$	$1{,}22 \cdot 10^7$
128000	3:00	$1{,}37 \cdot 10^{11}$	$1{,}32 \cdot 10^7$

Таблица 4.5

Результаты измерений количества узлов графа, обрабатываемых в единицу времени, при внутримодульном анализе для методов встраивания и резюме

от прогона к прогону и составляют, в среднем, 7 минут, и это время должно быть вычтено из общего времени анализа.

Результаты измерения нижней границы для ОС Android для внутримодульного и межмодульного анализа в различных режимах анализа представлены в таблицах 4.5 и 4.6 соответственно.

Объяснить результаты

Лимит количества узлов	Время анализа	Количество проанализированных узлов графа	Узлов графа в секунду
Метод встраивания			
8000	0:26	$1{,}58 \cdot 10^8$	$1{,}39 \cdot 10^5$
16000	0:33	$3{,}09 \cdot 10^8$	$1{,}98 \cdot 10^5$
32000	0:47	$5{,}96 \cdot 10^8$	$2{,}49 \cdot 10^5$
64000	1:13	$1{,}14 \cdot 10^9$	$2{,}88 \cdot 10^5$
128000	2:07	$2{,}18 \cdot 10^9$	$3{,}04 \cdot 10^5$
256000	3:56	$4{,}22 \cdot 10^8$	$3{,}07 \cdot 10^5$
Метод резюме			
2000	0:28	$4{,}01 \cdot 10^8$	$3{,}18 \cdot 10^5$
4000	0:41	$7{,}54 \cdot 10^8$	$3{,}70 \cdot 10^5$
8000	0:53	$1{,}40 \cdot 10^9$	$5{,}10 \cdot 10^5$
16000	1:23	$2{,}85 \cdot 10^9$	$6{,}27 \cdot 10^5$
32000	2:18	$5{,}43 \cdot 10^9$	$6{,}92 \cdot 10^5$
64000	4:41	$1{,}07 \cdot 10^{10}$	$6{,}52 \cdot 10^5$

Таблица 4.6

Результаты измерений количества узлов графа, обрабатываемых в единицу времени, при межмодульном анализе для метода резюме

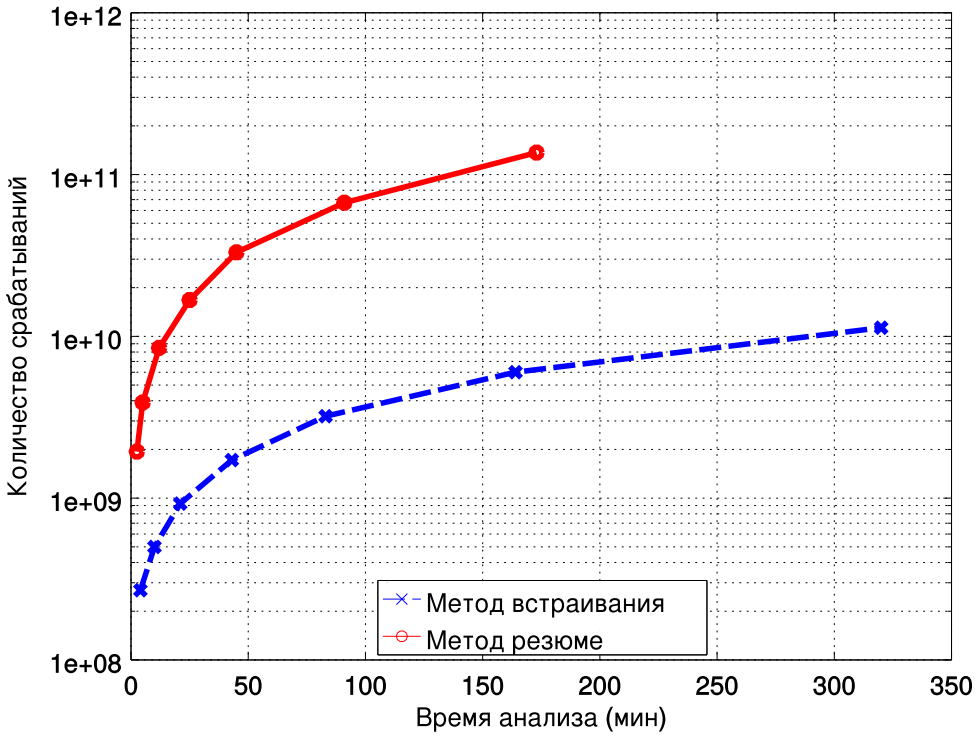

Рисунок 4.2 — Количество анализируемых узлов графа при внутримодульном анализе для методов встраивания и резюме

4.4 Обнаружение дефектов

Основной целью прогона статического анализатора является обнаружение дефектов. Поэтому, помимо покрытия анализа, необходимо проанализировать сообщения о дефектах, выданные анализатором при выполнении анализа программы.

Для проверки использовались проверяющие модули ConstModifiedChecker и IntegerOverflowChecker, поскольку срабатывания прочих проверяющих модулей на данной кодовой базе были единичными или отсутствовали вообще. При сравнении результатов анализа необходимо учитывать, что анализатор может выдавать отчёты на одном и том же операторе при анализе различных путей выполнения программы. Отчёты, относящиеся к одному оператору (точнее, к одной и той же позиции в исходном коде) и к одному и тому же проверяющему модулю, обычно унифицируются, поскольку пользователю не имеет смысла изучать один и тот же дефект несколько раз. С другой стороны, общее количество срабатываний без учёта унификации может коррелировать с покрытием

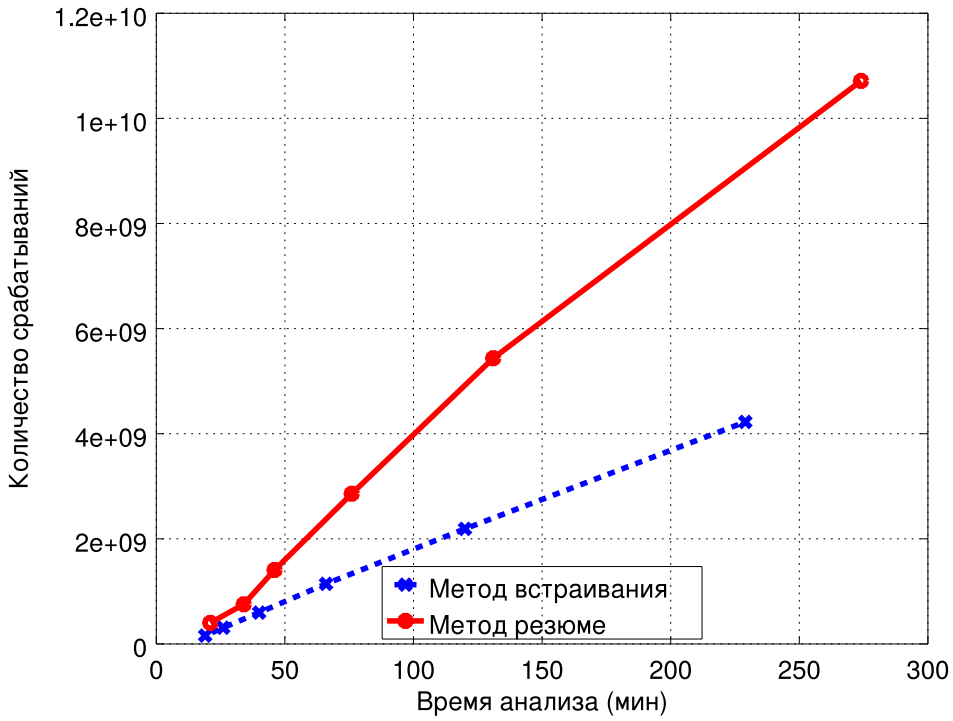

Рисунок 4.3 — Количество анализируемых узлов графа при межмодульном анализе для методов встраивания и резюме

анализатором путей программы. Следовательно, имеет смысл рассматривать как уникальные срабатывания анализатора, так и срабатывания без унификации.

Часть срабатываний была проверена вручную. По результатам проверки построена таблица **??**. Как видно из таблицы, качество анализа у использованных проверяющих модулей при использовании метода резюме осталось примерно таким же, как и при использовании метода встраивания. Это свидетельствует об эквивалентности двух методов проверки для этих проверяющих модулей. Кроме того, часть срабатываний, обнаруженных при межмодульном анализе, также является межмодульными, т. е. их трасса включает в себя несколько файлов исходного кода. Эти срабатывания не могли быть обнаружены при внутримодульном анализе, и это свидетельствует об увеличении полноты анализа.

Таблица TP/FP

Как можно наблюдать по результатам тестирования, при межмодульном анализе затрачиваемое время увеличивается примерно в три раза при использовании обоих методов МПА. Причиной этого является, во-первых, увеличение времени, затрачиваемого на дисковый ввод-вывод при загрузке дополнитель-

Лимит количества узлов	Время анализа	Отчётов анализатора	Уникальных отчётов	Отчётов в секунду
Метод встраивания				
8000	0:11	81861	1506	6,28
16000	0:17	164551	1616	2,70
32000	0:28	318558	1703	1,35
64000	0:50	624176	1771	0,69
128000	1:30	1199886	1828	0,37
256000	2:51	2242228	1895	0,19
512000	5:27	4189537	1933	0,10
Метод резюме				
2000	0:09:30	209038	1457	9,71
4000	0:12	366782	1591	5,30
8000	0:19	658994	1696	2,36
16000	0:32	1174869	1773	1,18
32000	0:52	2358712	1834	0,68
64000	1:38	4465568	1886	0,35
128000	3:00	8661287	1911	0,19

Таблица 4.7

Количество отчётов о дефектах при внутримодульном анализе для методов встраивания и резюме

Лимит количества узлов	Время анализа	Отчётов анализатора	Уникальных отчётов	Отчётов в секунду
Метод встраивания				
8000	0:26	5246	1769	1,55
16000	0:33	5773	1907	1,22
32000	0:47	6257	2048	0,85
64000	1:13	6644	2143	0,54
128000	2:07	6946	2194	0,30
256000	3:56	7335	2281	0,17
Метод резюме				
2000	0:28	10259	1715	1,36
4000	0:41	11439	1900	0,93
8000	0:53	13439	2066	0,75
16000	1:23	14191	2275	0,50
32000	2:18	14465	2326	0,30
64000	4:41	14939	2430	0,15

Таблица 4.8

Количество отчётов о дефектах при межмодульном анализе для методов встраивания и резюме

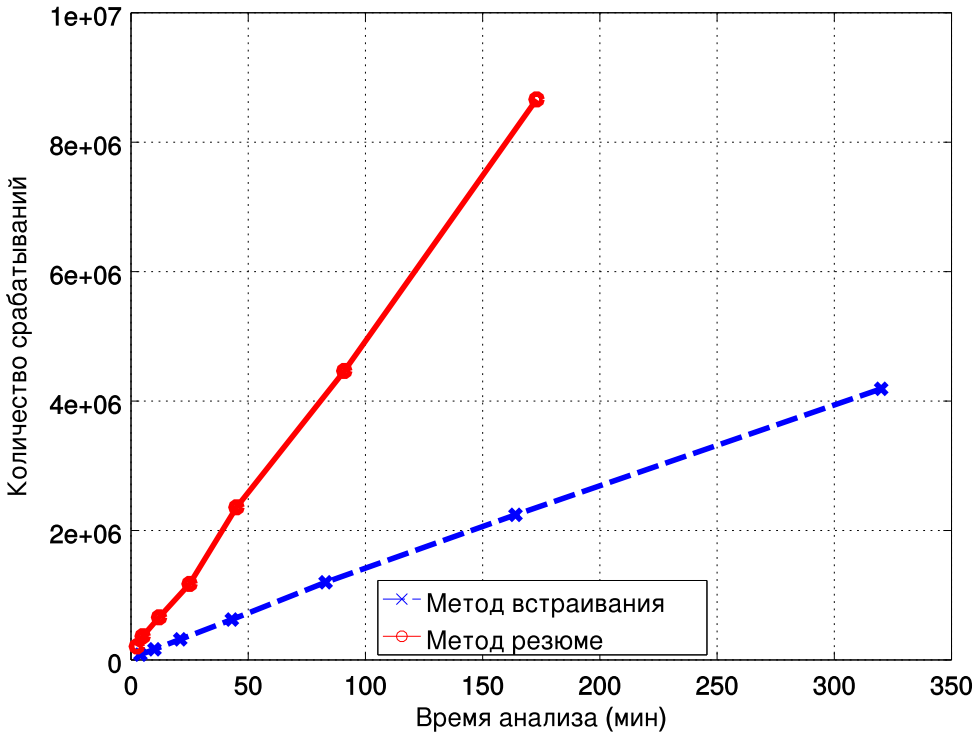

Рисунок 4.4 — Количество найденных дефектов при внутримодульном анализе для методов встраивания и резюме

ных синтаксических деревьев и обращении к поиску определений загружаемых функций; часть времени также затрачивается на объединение синтаксических деревьев. Во-вторых, в текущей версии межмодульного анализа на данный момент не реализовано повторное использование результатов анализа вызываемых функций при анализе различных единиц трансляции. Для решении этой проблемы можно предложить два способа

1. Объединение синтаксические деревья единиц трансляции, которые используют общие вызываемые функции. Данный подход, хотя и не слишком сложен в реализации, на практике имеет два крупных недостатка. Во-первых, его использование приводит к усложнению управления памятью анализатора, вызываемое необходимостью длительно хранить в памяти и удалять резюме функций, а также прогнозировать, какие резюме будут использованы, а какие — нет. Во-вторых, данный подход усложняет распараллеливание анализатора.
2. Лучшим вариантом является сериализация резюме и его сохранение на диск и последующее чтение. В качестве недостатка подхода стоит отметить увеличение количества операций дискового ввода-вывода. Кроме

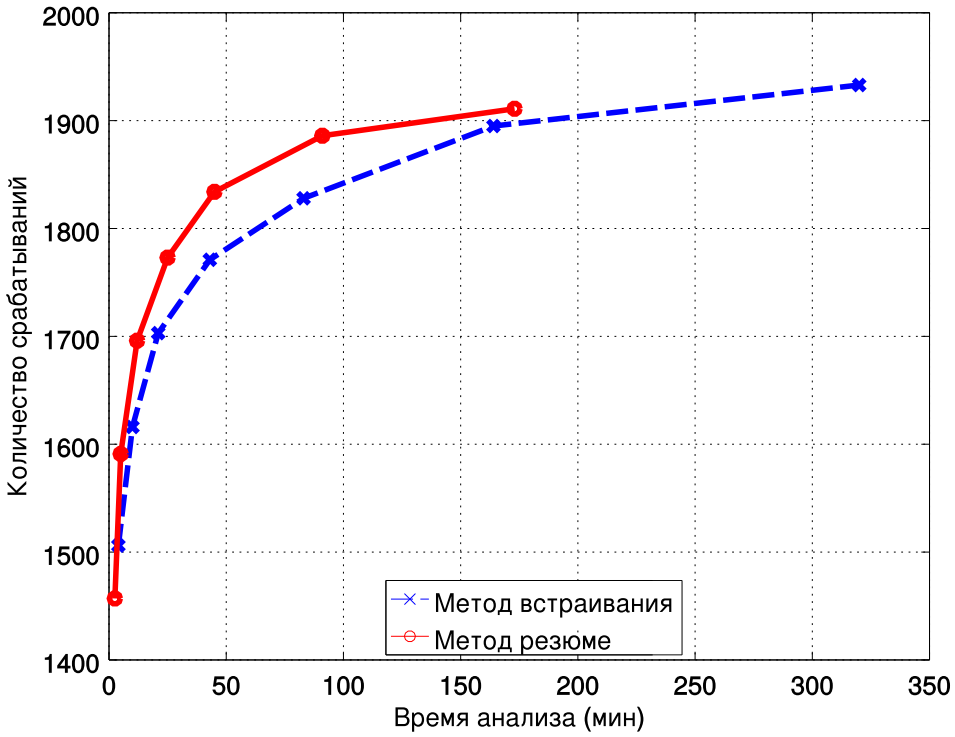

Рисунок 4.5 — Количество уникальных дефектов при внутримодульном анализе для методов встраивания и резюме

того, размер резюме слабо прогнозируем, и затраты на дисковые операции трудно определить заранее.

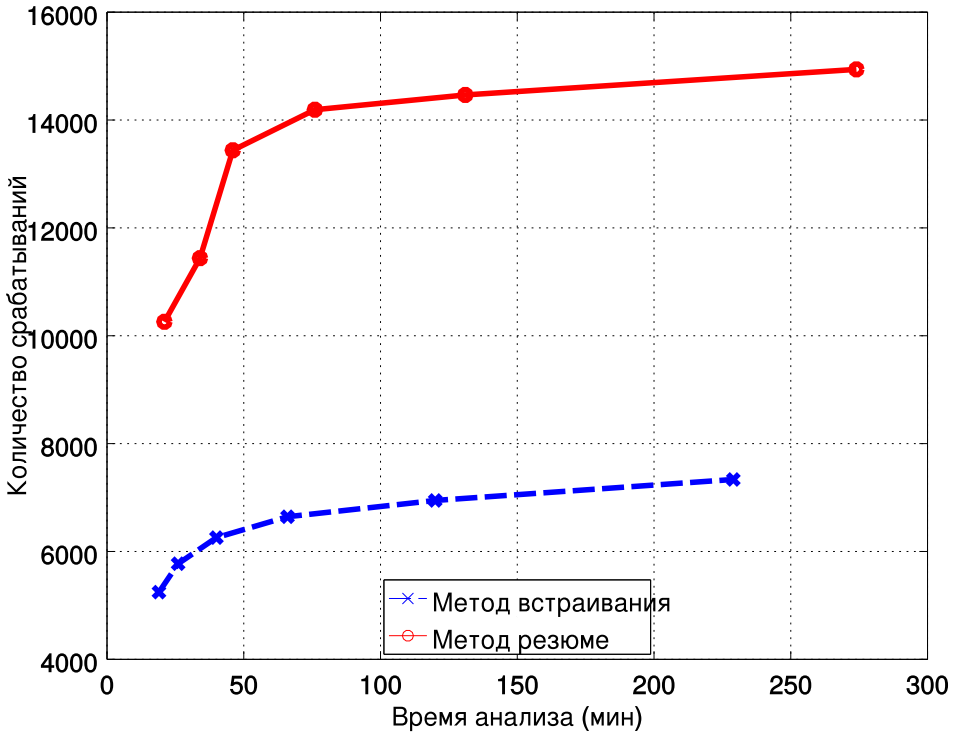

Рисунок 4.6 — Количество найденных дефектов при межмодульном анализе для методов встраивания и резюме

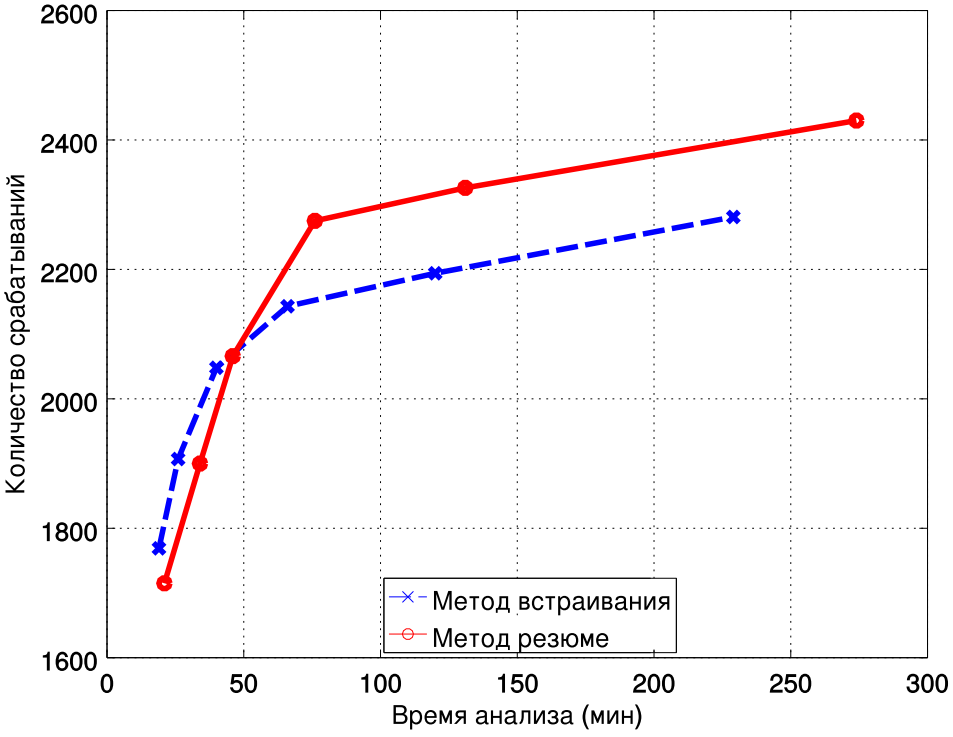

Рисунок 4.7 — Количество уникальных дефектов при межмодульном анализе для методов встраивания и резюме

Заключение

Основные результаты работы заключаются в следующем.

1. На основе анализа ...
2. Численные исследования показали, что ...
3. Математическое моделирование показало ...
4. Для выполнения поставленных задач был создан ...

И какая-нибудь заключающая фраза.

Список литературы

1. *Matsumoto Hiroo.* Applying Clang Static Analyzer to Linux Kernel // 2012 LinuxCon Japan. — Yokohama: 2012. — 6.

2. Описание PVS-Studio. http://www.viva64.com/ru/pvs-studio.

3. *Marjamaki Daniel.* Cppcheck design. — 2010. http://www.cs.kent.edu/~rothstei/spring_12/secprognotes/cppcheck-design.pdf.

4. *Johnson S. C.* Lint, a C Program Checker // COMP. SCI. TECH. REP. — 1978. — Pp. 78–90.

5. *Almossawi Ali, Lim Kelvin, Sinha Tanmay.* — Analysis Tool Evaluation: Coverity Prevent. Final Report, 2006. — 6.

6. Статический анализатор Svace для поиска дефектов в исходном коде программ / В.П. Иванников, А.А. Белеванцев, А.Е. Бородин и др. // *Труды Института системного программирования РАН (электронный журнал).* — 2014. — Т. 26, № 1. — С. 231–250.

7. *Hovemeyer David, Pugh William.* Finding Bugs is Easy // *SIGPLAN Not.* — 2004. — Dec.. — Vol. 39, no. 12. — Pp. 92–106. http://doi.acm.org/10.1145/1052883.1052895.

8. *Nethercote Nicholas, Seward Julian.* Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // *SIGPLAN Not.* — 2007. — Jun.. — Vol. 42, no. 6. — Pp. 89–100. http://doi.acm.org/10.1145/1273442.1250746.

9. AddressSanitizer: A Fast Address Sanity Checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov // Proceedings of the 2012 USENIX Conference on Annual Technical Conference. — USENIX ATC'12. — Berkeley, CA, USA: USENIX Association, 2012. — Pp. 28–37. http://dl.acm.org/citation.cfm?id=2342821.2342849.

10. *Serebryany Konstantin, Iskhodzhanov Timur.* ThreadSanitizer: Data Race Detection in Practice // Proceedings of the Workshop on Binary Instrumentation

and Applications. — WBIA '09. — New York, NY, USA: ACM, 2009. — Pp. 62–71. http://doi.acm.org/10.1145/1791194.1791203.

11. Unleashing Mayhem on Binary Code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Proceedings of the 2012 IEEE Symposium on Security and Privacy. — SP '12. — Washington, DC, USA: IEEE Computer Society, 2012. — Pp. 380–394. http://dx.doi.org/10.1109/SP.2012.31.

12. *Cadar Cristian, Dunbar Daniel, Engler Dawson.* KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — Berkeley, CA, USA: USENIX Association, 2008. — Pp. 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756.

13. *King James C.* Symbolic Execution and Program Testing // *Commun. ACM.* — 1976. — jul. — Vol. 19, no. 7. — Pp. 385–394. http://doi.acm.org/10.1145/360248.360252.

14. *Cousot Patrick, Cousot Radhia.* Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '77. — New York, NY, USA: ACM, 1977. — Pp. 238–252. http://doi.acm.org/10.1145/512950.512973.

15. *Романова Т.Н., А.В. Сидорин.* Метод резюме для разработки универсального многоцелевого анализатора кодов программ с возможностью обнаружения различных классов дефектов в программах, созданных с использованием языков C и C++ // *Вестник МГТУ им. Н.Э. Баумана, серия «Приборостроение».* — 2015. — № 5. — С. 73–93.

16. *Сидорин А.В., Романова Т.Н.* Новая модификация метода анализа кодов программ на основе резюме для тестирования сложных программных комплексов // *Наука и образование: электронное научно-техническое издание. Информатика, вычислительная техника и управление.* — 2015. — № 8. — С. 281–300.

17. *Сидорин А.В., Романова Т.Н.* Реализация межмодульного анализа для языков C и C++ в статическом анализаторе, использующем для анали-

за исходный код программы // *Наука и образование: электронное научно-техническое издание. Информатика, вычислительная техника и управление.* — 2015. — № 9.

18. Summary-based inter-unit analysis for Clang Static Analyzer / A.B. Сидорин, A.B. Дергачёв, Ю.С. Трофимович, Е.Г. Павлов. — 2015. — С. 239–241. https://drive.google.com/file/d/0B2ad_Dq_2eJxeTQ3QnZ0Yk53aDQ/view?pli=1.

19. *Сидорин А.В.* Модификация метода межпроцедурного анализа с использованием резюме для метода символьного выполнения // Материалы XII Международной научно-практической конференции «Инновации на основе информационных и коммуникационных технологий». — Инновации на основе информационных и коммуникационных технологий: Материалы международной научно-практической конференции. — Москва: НИУ ВШЭ, 2015. — С. 239–241. https://drive.google.com/file/d/0B2ad_Dq_2eJxeTQ3QnZ0Yk53aDQ/view?pli=1.

20. Summary-based inter-unit analysis for Clang Static Analyzer / Aleksei Sidorin, Artem Dergachev, Iuliia Trofimovich, Evgeny Pavlov // ???? — ??? — ???: ???, 2015. — Pp. ??–?? https://drive.google.com/file/d/0B2ad_Dq_2eJxeTQ3QnZ0Yk53aDQ/view?pli=1.

21. ГОСТ Р ИСО/МЭК 12207-2010 Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. — М.: Стандартинформ, 2011. — 105 с.

22. ГОСТ 34.601-90 Автоматизированные системы. Стадии создания. — М.: Стандартинформ, 2009. — 7 с.

23. *Липаев В.В.* Проектирование и производство сложных заказных программных продуктов. — Москва: СИНТЕГ, 2011. — 408 с.

24. *Bell T. E., Thayer T. A.* Software Requirements: Are They Really a Problem? // Proceedings of the 2Nd International Conference on Software Engineering. — ICSE '76. — Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. — Pp. 61–68. http://dl.acm.org/citation.cfm?id=800253.807650.

25. *Кент Бек.* Экстремальное программирование. Разработка через тестирование. — СПб: Питер, 2003. — 224 с.

26. Simplified Implementation of the Microsoft SDL. — 2015. http://www.microsoft.com/en-gb/download/details.aspx?id=12379.

27. *Davis Noopur, Mullaney Julia.* The Team Software Process (TSP) in Practice: A Summary of Recent Results: Tech. Rep. CMU/SEI-2003-TR-014. — Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6675.

28. *Липаев В.В.* Программная инженерия. Методологические основы. — Москва: Издательство «ТЕИС», 2006. — 608 с.

29. *Сикорд С.С.* Безопасное программирование на C и C++. Второе издание. — Москва: ООО «И.Д. Вильямс», 2015. — 496 с.

30. Common Weakness Enumeration. — 2015. http://www.cwe.mitre.org.

31. SEI CERT Coding Standards. — 2015. https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards.

32. MISRA-C:2004 – Guidelines for the use of the C language in critical systems, 2004.

33. Lockheed Martin Corporation. — Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, 2RDU00001 Rev C edition, 2005. — dec.

34. *Hammer Christian, Snelting Gregor.* Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs // *International Journal of Information Security.* — 2009. — Oct.. — Vol. 8, no. 6. — Pp. 399–422. http://dx.doi.org/10.1007/s10207-009-0086-1.

35. *Sen Koushik, Marinov Darko, Agha Gul.* CUTE: A Concolic Unit Testing Engine for C // Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — ESEC/FSE-13. — New York, NY, USA: ACM, 2005. — Pp. 263–272. http://doi.acm.org/10.1145/1081706.1081750.

36. Компиляторы. Принципы, технологии и инструментарий / Альфред В. Ахо, Моника С. Лам, Рави Сети, Ульман Джеффри Д. — Вильямс, 2003. — 1184 с.

37. Efficient State Merging in Symbolic Execution / Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, George Candea // *SIGPLAN Not.* — 2012. — jun. — Vol. 47, no. 6. — Pp. 193–204. http://doi.acm.org/10.1145/2345156.2254088.

38. *Ibing Andreas.* Demand-driven Compositional Symbolic Execution // SECURWARE 2014 : The Eighth International Conference on Emerging Security Information, Systems and Technologies. — IARIA, 2014. — Pp. 180–185.

39. *Anand Saswat.* Techniques to Facilitate Symbolic Execution of Real-World Programs: Ph.D. thesis / Georgia Institute of Technology. — The address of the publisher, 2012. — aug.

40. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World / Al Bessey, Ken Block, Ben Chelf et al. // *Communications of the ACM.* — 2010. — Feb.. — Vol. 53, no. 2. — Pp. 66–75. http://doi.acm.org/10.1145/1646353.1646374.

41. Loop Summarization and Termination Analysis / Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, Daniel Kroening // Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software. — TACAS'11/ETAPS'11. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 81–95. http://dl.acm.org/citation.cfm?id=1987389.1987400.

42. *Godefroid Patrice, Luchaup Daniel.* Automatic Partial Loop Summarization in Dynamic Test Generation // Proceedings of the 2011 International Symposium on Software Testing and Analysis. — ISSTA '11. — New York, NY, USA: ACM, 2011. — Pp. 23–33. http://doi.acm.org/10.1145/2001420.2001424.

43. *Trtík Marek.* Symbolic Execution and Program Loops: Ph.D. thesis / Faculty Masaryk of University Informatics. — Brno, 2013.

44. *Botella Bernard, Gotlieb Arnaud, Michel Claude.* Symbolic Execution of Floating-point Computations // *Software Testing, Verification & Reliability.*

— 2006. — Jun.. — Vol. 16, no. 2. — Pp. 97–121. http://dx.doi.org/10.1002/stvr.v16:2.

45. FloPSy: Search-based Floating Point Constraint Solving for Symbolic Execution / Kiran Lakhotia, Nikolai Tillmann, Mark Harman, Jonathan De Halleux // Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems. — ICTSS'10. — Berlin, Heidelberg: Springer-Verlag, 2010. — Pp. 142–157. http://dl.acm.org/citation.cfm?id=1928028.1928039.

46. *Collingbourne Peter, Cadar Cristian, Kelly Paul H.J.* Symbolic Crosschecking of Floating-point and SIMD Code // Proceedings of the Sixth Conference on Computer Systems. — EuroSys '11. — New York, NY, USA: ACM, 2011. — Pp. 315–328. http://doi.acm.org/10.1145/1966445.1966475.

47. *Bjørner Nikolaj, Tillmann Nikolai, Voronkov Andrei.* Path Feasibility Analysis for String-Manipulating Programs // Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. — TACAS '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 307–321. http://dx.doi.org/10.1007/978-3-642-00768-2_27.

48. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities / Xiang Fu, Xin Lu, Boris Peltsverger et al. // Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01. — COMPSAC '07. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 87–96. http://dx.doi.org/10.1109/COMPSAC.2007.43.

49. Symbolic Finite State Transducers: Algorithms and Applications / Margus Veanes, Pieter Hooimeijer, Benjamin Livshits et al. // Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '12. — New York, NY, USA: ACM, 2012. — Pp. 137–150. http://doi.acm.org/10.1145/2103656.2103674.

50. *Veanes Margus, Bjørner Nikolaj, De Moura Leonardo.* Symbolic Automata Constraint Solving // Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. — LPAR'10. — Berlin,

Heidelberg: Springer-Verlag, 2010. — Pp. 640–654. http://dl.acm.org/citation.cfm?id=1928380.1928425.

51. *Godefroid Patrice, Klarlund Nils, Sen Koushik.* DART: Directed Automated Random Testing // Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '05. — New York, NY, USA: ACM, 2005. — Pp. 213–223. http://doi.acm.org/10.1145/1065010.1065036.

52. *Sen Koushik, Marinov Darko, Agha Gul.* CUTE: A Concolic Unit Testing Engine for C // *SIGSOFT Softw. Eng. Notes.* — 2005. — Sep.. — Vol. 30, no. 5. — Pp. 263–272. http://doi.acm.org/10.1145/1095430.1081750.

53. *Godefroid Patrice, Levin Michael Y., Molnar David.* SAGE: Whitebox Fuzzing for Security Testing // *Queue.* — 2012. — jan. — Vol. 10, no. 1. — Pp. 20:20–20:27. http://doi.acm.org/10.1145/2090147.2094081.

54. *Godefroid Patrice.* Compositional Dynamic Test Generation // *SIGPLAN Not.* — 2007. — jan. — Vol. 42, no. 1. — Pp. 47–54. http://doi.acm.org/10.1145/1190215.1190226.

55. *Anand Saswat, Godefroid Patrice, Tillmann Nikolai.* Demand-driven Compositional Symbolic Execution // Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — TACAS'08/ETAPS'08. — Berlin, Heidelberg: Springer-Verlag, 2008. — Pp. 367–381. http://dl.acm.org/citation.cfm?id=1792734.1792771.

56. Compositional May-must Program Analysis: Unleashing the Power of Alternation / Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, Sai Deep Tetali // *SIGPLAN Not.* — 2010. — Jan.. — Vol. 45, no. 1. — Pp. 43–56. http://doi.acm.org/10.1145/1707801.1706307.

57. *Xu Zhenbo, Zhang Jian, Xu Zhongxing.* Melton: a practical and precise memory leak detection tool for C programs // *Frontiers of Computer Science in China.* — 2015. — Vol. 9, no. 1. — Pp. 34–54.

58. *Qadeer Shaz, Rajamani Sriram K., Rehof Jakob.* Summarizing Procedures in Concurrent Programs // *SIGPLAN Not.* — 2004. — Jan.. — Vol. 39, no. 1. — Pp. 245–255. http://doi.acm.org/10.1145/982962.964022.

59. Summary-based inference of quantitative bounds of live heap objects / Víctor Brabermana, Diego Garbervetskya, Samuel Hymc, Sergio Yovinea // *Science of Computer Programming.* — 2013. — Vol. 92. — Pp. 56–84.

60. *Yorsh Greta, Yahav Eran, Chandra Satish.* Generating Precise and Concise Procedure Summaries // *SIGPLAN Not.* — 2008. — Jan.. — Vol. 43, no. 1. — Pp. 221–234. http://doi.acm.org/10.1145/1328897.1328467.

61. *Rojas José Miguel, Păsăreanu Corina S.* Compositional Symbolic Execution through Program Specialization // BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation. — 2013. — mar.

62. MultiSE: Multi-path Symbolic Execution Using Value Summaries / Koushik Sen, George Necula, Liang Gong, Wontae Choi // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. — ESEC/FSE 2015. — New York, NY, USA: ACM, 2015. — Pp. 842–853. http://doi.acm.org/10.1145/2786805.2786830.

63. *Reps Thomas, Horwitz Susan, Sagiv Mooly.* Precise Interprocedural Dataflow Analysis via Graph Reachability // Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '95. — New York, NY, USA: ACM, 1995. — Pp. 49–61. http://doi.acm.org/10.1145/199448.199462.

64. *Xu Zhongxing, Kremenek Ted, Zhang Jian.* A Memory Model for Static Analysis of C Programs // Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I. — ISoLA'10. — Berlin, Heidelberg: Springer-Verlag, 2010. — Pp. 535–548. http://dl.acm.org/citation.cfm?id=1939281.1939332.

65. SEI CERT Coding Standards. — 2015. https://www.securecoding.cert.org/.

66. International Standard for Programming Language C++. ISO/IEC 9899:201x, 2009.

67. Free Software Foundation. — gcc - GNU project C and C++ compiler manual page, 2014.

68. wiki:Portal:Build Service. — 2015. https://en.opensuse.org/Build_Service.

69. Git Build System. — 2015. https://source.tizen.org/documentation/reference/git-build-system?langredirect=1.

70. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. — 2014. http://www.zlib.net.

71. Working Draft, Standard for Programming Language C++. ISO/IEC N4296, 2014.

Список рисунков

Список таблиц