На правах рукописи Sign

Сидорин Алексей Васильевич

НАЗВАНИЕ ДИССЕРТАЦИОННОЙ РАБОТЫ

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

Автореферат

диссертации на соискание учёной степени

кандидата технических наук

Москва — 2015

Работа выполнена в Московский государственный технический университет имени Н. Э. Баумана

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

Официальные оппоненты: **Фамилия Имя Отчество,**
доктор физико-математических наук, профессор,
Не очень длинное название для места работы,
старший научный сотрудник

Фамилия Имя Отчество,
кандидат физико-математических наук,
Основное место работы с длинным длинным длинным длинным названием,
старший научный сотрудник

Ведущая организация: Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования с длинным длинным длинным длинным названием

Защита состоится DD mmmmmmmm YYYY г. в XX часов на заседании диссертационного совета NN на базе Название учреждения по адресу: Адрес.

С диссертацией можно ознакомиться в библиотеке Название библиотеки.

Автореферат разослан DD mmmmmmmm YYYY года.

Ученый секретарь
диссертационного совета
NN, д-р физ.-мат. наук Sign Фамилия Имя Отчество

Общая характеристика работы

Актуальность темы. 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 (Svace [5], FindBugs [6]). В отличие от статического анализа, для динамического анализа программы требуется её выполнение — на специ-

---

[1] Matsumoto H. Applying Clang Static Analyzer to Linux Kernel // 2012 LinuxCon Japan. Yokohama: 2012. 6.

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

[3] Marjamaki D. Cppcheck design. 2010. URL: http://www.cs.kent.edu/řothstei/spring_12/secprognotes/cppcheck-design.pdf.

[4] Johnson S. C. Lint, a C Program Checker // COMP. SCI. TECH. REP. 1978. С. 78–90.

[5] 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.

[6] Hovemeyer David, Pugh William. Finding Bugs is Easy // SIGPLAN Not. New York, NY, USA, 2004. Dec.. Т. 39, № 12. С. 92–106. URL: http://doi.acm.org/10.1145/1052883.1052895.

альных входных данных, в виртуальной машине (Valgrind [7]), с использованием инструментации (AddressSanitizer [8], ThreadSanitizer [9]), с использованием дополнительных библиотек или их подменой. Наконец, смешанный анализ представляет собой комбинацию статического и динамического анализа и используется в таких инструментах как Mayhem [10], KLEE [11], а также других автоматических генераторах контрпримеров.

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

---

[7] Nethercote Nicholas, Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // SIGPLAN Not. New York, NY, USA, 2007. Jun.. Т. 42, № 6. С. 89–100. URL: http://doi.acm.org/10.1145/1273442.1250746.

[8] AddressSanitizer: A Fast Address Sanity Checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko [и др.] // Proceedings of the 2012 USENIX Conference on Annual Technical Conference. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012. С. 28–37. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[9] 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. С. 62–71. URL: http://doi.acm.org/10.1145/1791194.1791203.

[10] Unleashing Mayhem on Binary Code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert [и др.] // Proceedings of the 2012 IEEE Symposium on Security and Privacy. SP '12. Washington, DC, USA: IEEE Computer Society, 2012. С. 380–394. URL: http://dx.doi.org/10.1109/SP.2012.31.

[11] 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. С. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

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 и незначительное потребление памяти
- возможность его реализации в компиляторе для выполнения дополнительных проверок и предупреждения программиста о потенциально некорректном поведении компилируемого кода
- 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. 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 [12]. В основе метода лежит идея разбиения входных данных на классы эквивалентности в зависимости от встречаемых по пути выполнения условий. Метод абстрактной интерпре-

[12] King James C. Symbolic Execution and Program Testing // Commun. ACM. New York, NY, USA, 1976. jul. T. 19, № 7. C. 385–394. URL: http://doi.acm.org/10.1145/360248.360252.

тации, предложенный в 1977 году супругами Кузо [13], 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. 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 и Clang Static Analyzer.

Одним из способов увеличения производительности и покрытия при использовании метода символьного выполнения является использование *резюме* при межпроцедурном анализе. Этот подход позволяет повторное использование данных анализа функции для моделирования её вызовов. Исследованию этого метода посвящён ряд наиболее близких теме настоящей диссертации работ, в частности, [14,15,16].

Одними из наиболее актуальных целевых языков для статического анализа традиционно являются языки C и C++. 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

---

[13] 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. С. 238–252. URL: http://doi.acm.org/10.1145/512950.512973.

[14] Godefroid Patrice. Compositional Dynamic Test Generation // SIGPLAN Not. New York, NY, USA, 2007. jan. Т. 42, № 1. С. 47–54. URL: http://doi.acm.org/10.1145/1190215.1190226.

[15] 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. С. 367–381. URL: http://dl.acm.org/citation.cfm?id=1792734.1792771.

[16] Compositional May-must Program Analysis: Unleashing the Power of Alternation / Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani [и др.] // SIGPLAN Not. New York, NY, USA, 2010. Jan.. Т. 45, № 1. С. 43–56. URL: http://doi.acm.org/10.1145/1707801.1706307.

с неправильной работой с указателями. Во-вторых, стандарты языков трактуют достаточно большое количество ситуаций как не имеющих определённого или зависимого от реализации поведения, 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, таких как компоненты операционных систем и драйверы, 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
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)

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.

Достоверность 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 [1–6], 4 из которых изданы в журналах, рекомендованных ВАК [1–4], 2 — в тезисах докладов [5; 6].

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, приводится обзор научной литературы по изучаемой проблеме, формулируется цель, ставятся задачи работы, сформулированы научная новизна и практическая значимость представляемой работы.

В **первой главе** рассмотрен вопрос статического анализа компьютерных программ и, в особенности, метод символьного выполнения как один из наиболее актуальных и используемых на практике. Рассмотрены основные проблемы, возникающие при использовании данного метода, и методы их решения. Представлены и проанализированы работы, посвящённые улучшениям метода символьного выполнения, в т. ч. наиболее близкие к тематике работы.

Во **второй главе** рассмотрен и подробно описан метод межпроцедурного анализа программ с использованием резюме для метода символьного выполнения. Разработанный алгоритм метода межпроцедурного анализа с помощью резюме для метода символьного выполнения выглядит следующим образом:

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 для каждой ветви резюме создать новый узел графа выполнения (узел применения резюме) и соединить эти узлы с узлом, соответствующим вызову функции (узлом вызова). Состояние программы в каждой точке применения резюме есть композиция состояния программы в узле вызова и функции, описываемой соответствующей ветвью резюме.

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. 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 из-за оссобенностей анализатора. 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. Ограничения учитываются при *анализе достижимости* ветвей выполнения программы.
2. Модификация регионов памяти с нелокальной областью видимости.
3. Инвалидация регионов памяти, то есть пометка некоторых регионов как изменивших значение на неизвестное. Данное действие обычно выполняется при моделировании оператора, все эффекты которо-

го 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.
5. Пометки проверяющих модулей, в т. ч. пометки состояния программы и пометки отложенных проверок

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

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.

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:

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

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 мы отслеживаем событие потери актуальности (активности) и определяем 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. 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.

Под инвалидацией региона памяти мы понимаем 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 автома-

тически, и дополнительных действий для обработки инвалидаций регионов памяти не требуется.

Для обработки возврата функцией значения 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.

В результате сбора резюме 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 объекта, метод которого вызывается. Кроме того, с регионом памяти в контексте вызываемой функции может быть связано значение, не имеющее в своей основе регион аргумента, например, константа. 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 полу-

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

1. Для всех регионов памяти, содержащихся в символьном значении, создать новый регион, являющийся актуализацией данного региона и заменить в символьном значении исходный регион актуализированным.
2. Для всех символов, содержащихся в символьном значении, полученном на шаге 1, проверить, не вычисляется ли символ в константу, и, если символ вычисляется в константу, заменить данный символ вычисленной константой.

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.

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 состоя-

нии вызывающей функции. Если же в контексте вызывающей функции уже известно, что дескриптор закрыт, то отложенная проверка должна выдать предупреждение.

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.

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

Важным элементом работы статического анализатора является построение отчёта. 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 применении метода резюме, в отличие от метода встраивания, возникает проблема потери информации о части пути, проходимом внутри вызываемой функции, поскольку явного построения поддеревьев графа выполнения программы более не происходит. Для построения отчёта при межпроцедурном анализе методом резюме в настоящей работе предлагается сохранять в узлах применения резюме ссылки на соответствующий лист графа выполнения вызываемой функции и ссылки узлы отложенной проверки (для проверяющих модулей).

В **третьей главе** приводится описание разработанного метода межпроцедурного анализа программ с использованием непосредственно исходного кода, а также рассматриваются проблемы, возникающие при межпроцедурном анализе крупных программных комплексов, и методы их решения. 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, в т. ч. 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 огра-

ниченной и затрагивает лишь малую часть программного проекта, в число которых входят C и C++.

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.

Единицей анализа в анализаторах исходных кодов обычно 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.

В данной работе реализован трёхфазный анализ с сохранением промежуточных результатов в файлах в директории проекта. Анализ разделяется на фазу сборки, фазу предобработки данных и непосредственно сам анализ исходных кодов. На *фазе сборки* специальными инструментами собирается информацию о транслируемых модулях, которые должны быть проанализированы. Для этого сначала инструментом, использующим утилиту `strace`, производится перехват вызовов компилятора во время построения проекта. Затем для каждой обнаруженной команды сборки другой инструмент, использующий API Clang, записывает сигнатуры 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. Определения функций различаются по тройке <путь к файлу, сигнатура функции, целевая архитектура>, которая впоследствии используется для выбора нужного определения функции и его импорта.

Фаза предобработки данных необходима для обработки служебной информации, собранной на фазе сборки. На этой фазе строится соответствие между сигнатурами импортируемых функций. Затем строится глобальный граф вызовов с использованием локальных графов вызовов, сгенерированых на предыдущей фазе. После этого выполняется топологическая сортировка построенного глобального графа вызовов. При этом сортируются не сами функции, а файлы, их содержащие, поскольку анализ отдельных функций из файла означает многократную загрузку одних и тех же файлов.

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. 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 отвечает за слияние синтаксических деревьев различных транслируемых модулей и 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 сигнатуры этой функции в сгенерированном отображении и, если сигнатура функции была найдена, загружается синтаксическое дерево файла, содержащего определение этой функции. Затем эта функция импор-

тируется в основное синтаксическое дерево вместе с необходимыми определениями и объявлениями.

Задача импорта фрагментов синтаксического дерева обычно решается с помощью поиска определения в целевом AST. Если аналогичное определение (или объявление) не найдено, оно создаётся в целевом AST 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 AST.

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. 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, за исключением 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. Это вызывает необходимость переупорядочения полей структур после того, как определение структуры было полностью импортировано, в соответствии с их порядком в импортируемой структуре, для сохранения порядка объявлений и модели раскладки в памяти.

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.

В **четвертой главе** приведены результаты и методики тестирования программного обеспечения, реализующего разработанные методы.

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.

В **заключении** приведены основные результаты работы, которые заключаются в следующем:

1. Результат номер один.
2. Результат номер два.
3. Результат номер три.

Публикации автора по теме диссертации

1. *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.

2. *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.

3. *Сидорин А.В., Романова Т.Н.* Реализация межмодульного анализа для языков C и C++ в статическом анализаторе, использующем для анализа исходный код программы // *Наука и образование: электронное научно-техническое издание. Информатика, вычислительная техника и управление.* — 2015. — № 9.

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

5. *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.

6. 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.