На правах рукописи

Бородин Алексей Евгеньевич

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

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

АВТОРЕФЕРАТ

диссертации на соискание ученой степени кандидата физико-математических наук

Москва – 2016

Работа выполнена в *Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук.*

Научный руководитель:	**Белеванцев Андрей Андреевич**, кандидат физико-математических наук, ведущий научный сотрудник
Официальные оппоненты:	**Терехов Андрей Николаевич**, доктор физико-математических наук, профессор, заведующий кафедрой системного программирования Федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет»,
	Михалкович Станислав Станиславович, кандидат физико-математических наук, доцент Института математики, механики и компьютерных наук им. И. И. Воровича Федерального государственного автономного образовательного учреждения высшего профессионального образования «Южный федеральный университет»
Ведущая организация:	Вычислительный центр им. А. А. Дородницына Федерального исследовательского центра «Информатика и управление» Российской академии наук

Защита состоится «16» июня 2016 г. в 15 часов на заседании диссертационного совета *Д 002.087.01* при *Институте системного программирования РАН*, расположенном по адресу: *109004, Москва, ул. А. Солженицына, д.25.*

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования Российской академии наук.

Автореферат разослан «____» _____ 2016 г.

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

Общая характеристика работы

Актуальность темы исследования.

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

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

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

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

избавиться. Ложные предупреждения связаны с тем, что точное вычисление необходимых свойств программы при статическом анализе является алгоритмически неразрешимой задачей.

Для промышленного применения статического анализатора нужно выполнить следующие требования:

- масштабируемость: анализ больших программных систем (миллионы строк кода) должен длиться не более 4-6 часов (ночная сборка);
- качество: высокий уровень истинных предупреждений (50-70%) вместе с пропуском небольшого количества ошибок;
- понятность: объяснение причин, по которым найденная ситуация является ошибкой. Объяснение включает в себя подробное текстовое описание ошибки, указание места проявления ошибки, описание цепочки событий, приведших к ошибке, а также дополнительную информацию, которая будет полезной пользователю для понимания ошибки;
- полнота: поддержка популярных языков программирования, поддержка реализации детекторов для поиска известных типов ошибок разной критичности;
- расширяемость: простое добавление анализа новых типов ошибок, в том числе специфичных для конкретной программы.

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

Используемые эвристики необходимо регулярно пересматривать при повышении требований к масштабируемости анализа, потому что с ростом размера анализируемой программы увеличивается вероятность обнаружения ранее не встреченных программных конструкций; неправильная работа эвристик для этих конструкций может серьезно ухудшить качество анализа. Кроме того, нужно повышать точность используемых алгоритмов анализа потока данных и управления за счёт большей точности алгоритмов анализа указателей, анализа циклов, учёта контекста вызова функции и влияния отдельных путей выполнения. Таким способом разрабатываются ведущие мировые промышленные анализаторы (инструменты Coverity CMC, Klocwork K10, HP Fortify, инструмент Svace Института системного программирования РАН).

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

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

Для достижения поставленных целей были сформулированы и решены следующие задачи:

- Разработка внутрипроцедурных алгоритмов анализа, позволяющих на основе вычисленной ими информации о потоке данных и управления программы выполнять поиск широкого класса дефектов; доказательство кор-

ректности алгоритмов.

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

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

- Реализация разработанных алгоритмов в инструменте статического анализа Svace.

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

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

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

Разработанные алгоритмы реализованы в статическом анализаторе Svace для программ на языках Си и Си++.

Теоретическая и практическая значимость. Предложены алгоритмы внутрипроцедурного анализа функции для задачи поиска дефектов. На основе внутрипроцедурного анализа функции предложен алгоритм масштабируемого межпроцедурного контекстно-чувствительного анализа.

Все предложенные алгоритмы были реализованы в статическом анализаторе Svace. Кроме того, на основе предложенных алгоритмов реализованы алгоритмы детекторов конкретных типов дефектов – разыменования нулевого указателя, отсутствия освобождения ресурсов, выхода за границы массивов,

неинициализированных переменных. Экспериментальные результаты анализа больших программных систем подтвердили масштабируемость реализованных алгоритмов при высоком качестве анализа – более 60% истинных срабатываний критических детекторов и в разы возросшее количество предупреждений по сравнению с предыдущей версией анализатора. Разработанный анализатор продемонстрировал масштабируемость и качество анализа на уровне лучших мировых аналогов и внедрён для промышленного использования в компании Samsung.

Положения, выносимые на защиту:

- алгоритм внутрипроцедурного анализа функции, интегрирующий анализ указателей и нумерацию значений и обеспечивающий возможность поиска дефектов разыменования нулевых указателей, выхода за границы массивов, использования неинициализированных переменных, отсутствия освобождения ресурсов (в том числе освобождения памяти), повторной блокировки семафора, несогласованность конструкторов и деструкторов Си++ классов, ошибок использования библиотечных функций;

- алгоритм создания резюме функции, обобщающий результаты внутрипроцедурного анализа функции и обеспечивающий потоковую чувствительность межпроцедурного анализа;

- алгоритм анализа инструкций вызова функций, обеспечивающий масштабируемый контекстно-чувствительный межпроцедурный анализ;

- инструмент статического анализа, реализующий разработанные алгоритмы для программ на языках Си и Си++.

Апробация результатов. Основные результаты диссертации докладывались на следующих конференциях:

1. IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST) (Cleveland, Ohio, USA, 2014);

2. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия, 2014);

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

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

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

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

Содержание работы

Во введении обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, показана практическая значимость полученных результатов, представлены выносимые на защиту научные положения.

В первой главе описывается задача поиска ошибок в исходном коде программ и производится обзор существующих подходов.

Рассматривается задача статического анализа программ, написанных на языках Си и Си++, для поиска ошибок в исходном коде при приемлемом уровне ложных срабатываний и заданном времени анализа. При такой постановке задачи анализ может как пропускать некоторые дефекты, так и выдавать ложные предупреждения. С целью улучшения точности анализа могут использоваться

допущения, которые неверны в общем случае, но выполняются для большей части исходного кода.

В 2012 году для проектов размером в сотни тысяч строк кода среднее количество истинных срабатываний инструмента Svace составляло 50%, а для некоторых детекторов – 70-90%. При переходе в 2014 году к анализу больших проектов (операционные системы Tizen и Android) уровень истинных срабатываний для многих детекторов упал ниже 15%. Такой уровень был недостаточен для промышленного использования инструмента. Анализ ложных срабатываний показал, что основными их причинами является недостаточно точное моделирование памяти, использование эвристик для анализа указателей, неточный анализ циклов.

Одним из условий внедрения инструмента Svace является высокий показатель количества находимых дефектов. При этом недопустимо повышать процент ложных срабатываний. Пропуск некоторых дефектов был следствием фильтрации предупреждений для случаев, когда не было достаточно уверенности, что предупреждение будет истинным. То есть повышение точности анализа могло увеличить количество находимых дефектов. Дополнительно на основе обзора существующих методов и инструментов статического анализа (ARCHER, Prefix, Saturn, Calysto, Coverity, Klocwork) был сделан вывод, что для нахождения большего количества дефектов необходимо увеличить степень контекстной чувствительности, реализовать анализ с чувствительностью к путям и правильно выбрать те части вычисленных анализом данных, которые будут сохранены в резюме.

В настоящей работе были предложены алгоритмы нумерации значений и анализа указателей для задачи поиска ошибок (третья глава), предложены механизмы написания детекторов на основе разработанных алгоритмов анализа, представлен критерий выдачи предупреждений детекторами и способ учета отдельных путей выполнения (четвертая глава), предложен метод повышения контекстной чувствительности анализа, учитывающий при обработке вызова функции влияние фактических параметров вызова на отдельные пути выполнения внутри вызываемой функции (пятая глава). Все предложенные алгоритмы реализованы в инструменте Svace, разрабатываемом в Институте системного программирования РАН (шестая глава).

Во второй главе описывается низкоуровневый типизированный язык svace0, являющийся внутренним представлением анализа, и операционная семантика языка. Язык конструировался с целью упрощения написания детекторов и использования в качестве промежуточного представления для разных языков (Си, Си++, Java). Язык является структурированным и не имеет инструкций безусловного перехода. Инструкции *if*, *while* позволяют осуществлять условное выполнение программы и организовывать циклы.

Продемонстрируем свойства языка на примере. Рис. 1 содержит код функции на языке Си, и соответствующий оттранслированный код на языке svace0.

```
 1: int calc(int f) {
 2:     int a;
 3:     int b;
 4:     int*p;
 5:     a = 1;
 6:     b = 2;
 7:     p = f?
 8:           &a
 9:         : &b;
10:
11:
12:     int x = *p;
13:     *p = 3;
14:
15:     int y = *p;
16:
17:     return x + y; }
```

```
 1: def calc(f) =
 2:     a = alloca();
 3:     b = alloca();
 4:     p = alloca ref int();
 5:     store 1, a;
 6:     store 2, b;
 7:     if(f!=0,
 8:         store a, p,
 9:         store b, p
10:     );
11:     t1 = load p;
12:     x = load t1;
13:     store 3, t1;
14:     t2 = load p;
15:     y = load t2;
16:     t3 = x + y;
17:     ret t3;
```

Рис. 1. Код на языке Си и соответствующий код svace0

Переменные языка svace0 соответствуют адресам переменных языка Си и являются неизменяемыми псевдорегистрами. На рисунке переменные языка svace0 a, b и p обозначают адреса переменных a, b и p языка Си соответственно. Инструкция *alloca* используется для выделения памяти, при этом возвращается указатель, который не имеет алиасов среди всех созданных указателей. После выполнения инструкции *alloca* на строке 1, переменная a будет указывать на вновь созданную ячейку памяти, имеющую неопределённое значение. Инструкции *load* и *store* позволяют манипулировать переменными исходного языка через их адреса. В частности, строка 5 осуществляет запись константы 1 в память, на которую указывает a. После выполнения инструкции *load* на строке 11, значение, содержащееся в ячейке памяти, на которую указывает регистр

p, будет присвоено регистру $t1$. В данном примере регистр $t1$ будет равен либо a, либо b. Временный регистр $t3$ используется для реализации сложения.

В третьей главе описывается ядро анализа функции. В ходе анализа выполняется нумерация значений, то есть определение равных между собой значений переменных и ячеек памяти. Для обозначения набора равных значений в анализе вводится понятие *идентификатора значения*.

Нумерация значений на основе идентификаторов значений похожа на нумерацию значений, используемую оптимизируемыми компиляторами, но имеет следующие отличия:

- При обходе определения переменной в цикле анализатор присвоит тот же самый идентификатор значения, только если сможет доказать, что значение переменной не изменилось. Подобная организация анализа позволяет поддерживать свойство *мобильности*: на идентификатор значения можно ссылаться в любой точке программы после создания.

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

- Описываемый анализ использует некоторые предположения, которые некорректны в общем случае. При использовании излишне агрессивной нумерации значений в случаях нарушения предположений ошибка распространится на все переменные одного класса эквивалентности. Нумерация значений производится только для присваиваний, арифметических операций и сдвигов указателей.

- Нумерацию значений, используемую компиляторами, можно рассматривать как трансформацию промежуточного представления программы. В

результате будет получено другое промежуточное представление с меньшим количеством переменных. В описываемом анализе нельзя выделить фазу нумерации значений. Нумерация значений выполняется одновременно с анализами указателей и эффектов от вызова функций.

Рассматривается задача анализа только для некоторых состояний на входе в функцию языка svace0. Разрешёнными состояниями будут такие состояния на входе в функцию, где отсутствуют алиасы среди переменных и ячеек памяти. Используется неконсервативное предположение, что выполнение функции начинается с разрешённых состояний. В общем случае такое предположение некорректно, но оно справедливо для большинства функций.

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

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

Для анализа строится граф потока управления функции. В качестве вершин используются инструкции, а не базовые блоки. Выполняется обход вершин графа потока управления. С каждым ребром графа ассоциируется абстрактное состояние. Анализ продвигает состояние по графу потока управления. Если вершина имеет несколько входящих рёбер, то выполняется операция объединения абстрактных состояний. Ациклические графы обходятся в топологическом порядке.

Циклы анализируются особым образом. Производится несколько итераций анализа инструкций цикла. Для первой итерации цикла игнорируются обратные рёбра, для последующих итераций учитываются только обратные рёбра. В конце анализа цикла производится объединение абстрактных состояний, созданных на разных итерациях, для рёбер, выходящих из цикла. Множество анализируемых путей включает только пути, которые обходятся описанным выше

способом.

В текущей версии анализатора выполняется три итерации обхода циклов. В реальных программах циклы могут иметь значительно больше итераций. Чтобы анализировать такие циклы, на первой итерации не используется точная нумерация значений. В результате для переменных цикла создаются новые идентификаторы значений, имеющие произвольные свойства. На последующих итерациях их свойства будут уточнены. Таким образом можно проанализировать свойства циклов, которые выполняются более трёх раз и не имеют длинных межитерационных зависимостей по данным.

Среди множества идентификаторов значений анализ выделяет *ссылки*. Ссылки обозначают непересекающиеся ячейки памяти и создаются только в тех случаях, когда можно гарантировать свойство непересечения. Ссылки используются для отслеживания значений ячеек памяти и для моделирования отношений указывания.

Абстрактным состоянием анализа будет пара функций $< val, pt >$. Пусть V – множество идентификаторов значений, A – множество ссылок, R – множество переменных. Функция $val : R \cup A \to V$ описывает взаимосвязь переменных и ссылок с идентификаторами значений. Функция $pt : V \to \wp(A)$ для каждого идентификатора значений возвращает множество ссылок, на которые он может указывать. Абстрактное состояние накладывает ограничение на форму памяти и равенство переменных в описываемых конкретных состояниях.

Передаточная функция для каждой инструкции трансформирует состояние на входе в состояние на выходе. Для обозначения передаточной функции для инструкции $instr$, трансформирующей состояние на входе $<val_1,\ pt_1>$ в состояние на выходе $< val_2, pt_2 >$, используется следующая нотация:

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

В точках слияния путей анализ для каждой переменной и ссылки считывает идентификаторы значения на входных рёбрах. Если идентификаторы значения одинаковые, то этот идентификатор значения используется и для выходного ребра. Если они разные, то создаётся новый идентификатор значения, отличный от всех предыдущих, и сопоставляется переменной или ссылке на выходном ребре. Функцию создания нового идентификатора значения на основе двух других назовём $joinval$.

Для инструкции *alloca* известно, что возвращается указатель на ячейку памяти, которая не пересекается со всеми созданными ячейками памяти. Для таких ячеек памяти анализ создаёт ссылку. Для каждого входного параметра функции также создаётся ссылка, обозначающая его разыменование. Остальные указатели могут указывать только на созданные таким образом ссылки.

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

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

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

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

Корректность анализа формулируется теоремой 1: конкретизация всех созданных в ходе анализа абстрактных состояний будет включать возможные конкретные состояния во время выполнения функции при заданных предположениях о разрешённых состояниях и множестве анализируемых путей.

Теорема 1. *Пусть после анализа инструкции на некоторой итерации анализ сопоставил выходному ребру инструкции абстрактное состояние s и множество анализируемых путей p. Тогда для любого пути $l \in p$, имеющего разрешённые конкретные состояния в точке входа в процедуру, его конкретное состояние ν в текущей точке должно принадлежать конкретизации абстрактного состояния.*

Для более точного анализа используется нумерация значений для разы-

менования указателя. Абстрактное состояние расширяется до тройки $<val, pt, cache>$, где функция $cache : \wp(V) \to V$ используется для возможности сильных обновлений для результатов ϕ-функций. Для указателя p анализ формирует пару $< v_p, V_p >$, где $v_p = val(p)$ – идентификатор значения указателя p, множество $V_p = val(pt(v_p))$ – упорядоченное множество идентификаторов значений для ссылок A_p, на которые указывает p. Этой паре сопоставляется идентификатор значения $v_n = val(v)$ для $*p$. После анализа инструкции *store* вычисляется $cache_{out} = cache_{in}[< v_p, V_p > \mapsto v_n]$. При следующем доступе к памяти через указатель вновь вычисляется пара $< v'_p, V'_p >$, где $v'_p = val'(p)$, $V'_p = val(pt(v'_p))$. Если функция *cache* определена для этой пары, значит, не было изменений в указываемых ячейках памяти, и можно использовать предыдущее значение $v_n = cache(< v'_p, V' >)$.

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

Четвёртая глава посвящена детекторам для поиска ошибок. Анализ разделён на ядро и расширения. Конкретные детекторы реализуются в виде расширений. Ядро выполняет общие действия, необходимые детекторам: обход графа потока управления, анализ указателей, создание идентификаторов значений и ссылок. Подобное разделение позволяет как упростить реализацию отдельных детекторов, так и ускорить анализ за счёт однократного выполнения общих операций.

Все свойства анализа представляются в виде атрибутов. Значения атрибута должны образовывать полурешётку[1]. Атрибуты расширяют состояние ана-

[1] Значения атрибута являются частично упорядоченным множеством с оператором объединения, определяющим для каждой пары элементов наименьший верхний элемент. Оператор объединения используется в точках слияния путей.

лиза. К тройке $< val, pt, cache >$ добавляются функции для каждого атрибута.

В ходе анализа ядро оповещает расширения о всех событиях и позволяет создавать и ассоциировать атрибуты с идентификаторами значений и точками графа потока управления. Для детектора анализируемая программа состоит из событий, генерируемых ядром анализа, а реализация детекторов заключается в описании передаточных функций для событий. Рис. 2 содержит список части событий. Событие $deref$ генерируется для инструкций, непосредственно разыменовывающих указатели: $load$ и $store$. Если разыменование выполняется внутри вызываемой функции, то событие не генерируется. Событие $assume$ генерируется для каждого ребра инструкций ветвления. Событие $const(v, c)$ генерируется для уведомления детекторов, что значение идентификатора значения v равно константе c.

$$const(v, c)$$
$$deref(v)$$
$$assume(a = b)$$
$$assume(a \neq b)$$
$$assume(a \geq b)$$
$$assume(a \leq b)$$
$$assume(a > b)$$
$$assume(a < b)$$

Рис. 2. Виды событий. v – идентификаторы значений, c – константы; a, b – идентификаторы значений либо константы.

Анализ осуществляет проход по графу потока управления сразу для всех детекторов. Поэтому входное состояние перед каждой инструкцией можно не хранить после обхода инструкции, что позволяет сократить потребление памяти.

В разделе 4.2 описывается *критерий выдачи предупреждений*. При написании функции программисты часто подразумевают некоторый контекст использования, в котором возможно вызывать функцию. Функция на рис. 3 имеет 3 аргумента x, y и z, и использует 2 глобальных указателя p и q. Ничего не известно про взаимосвязь параметров, возможно, не все их комбинации разрешены.

Предположим, что для каждого отдельного пути внутри функции есть критерий определения того, что путь содержит ошибку (задаётся детектором). Рассмотрим некоторое ребро графа потока управления такое, что все пути, проходящие через данное ребро, содержат ошибку. Если ребро достижимо, то при

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

Рис. 3. Иллюстрация критерия выдачи предупреждений.

выполнении программы будет ошибка. Недостижимое ребро будет соответствовать либо недостижимой инструкции, либо лишней инструкции ветвления. Программисты, как правило, не пишут лишних проверок и инструкций, которые никогда не выполняются. Таким образом, *предупреждение выдаётся, если все пути, проходящие через некоторое ребро в графе потока управления функции, содержат ошибку.*

Критерий нельзя применять к инструкциям, сгенерированным препроцессором Си для макросов. Макросы могут содержать проверки, которые при подстановке в тело конкретной функции будут лишними.

Применим критерий для функции foo на рис. 3. Все пути, проходящие через инструкцию (2), проходят через инструкцию (4) и содержат разыменование нулевого указателя. Поэтому согласно критерию следует выдать предупреждение.

В разделе 4.5 описывается детектор разыменования нулевого указателя. Рассматриваются только ситуации, где есть инструкция сравнения указателя с нулевым значением.

Для анализа создадим двоичный атрибут $Null$, имеющий значения: $null$ – указатель был сравнен с нулём, и результат сравнения является истиной, и any – указатель может иметь любое значение. Решётка атрибута $null \sqsubseteq any$. Передаточная функция для события $assume(p = 0)$ ассоциирует с идентификатором значения для p атрибут $null$. В точках слияния путей используется оператор объединения: если оба идентификатора значения имеют значение $null$, то и результат получит такое значение. Если для события разыменования с указателем ассоциирован идентификатор значения, которому сопоставлено значение $null$, то выдаётся предупреждение, т. к. если данное событие достижимо, то произой-

дёт разыменование указателя, который может иметь только нулевое значение.

Передаточные функции детектора:

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

$Null \| assume(p \neq 0) \| \mapsto Null[p \mapsto any]$

Детектор находит инструкции, разыменовывающие переменные, которые на всех путях могут иметь только нулевые значения. Критерий выдачи предупреждений позволяет осуществлять поиск более сложных дефектов. Согласно критерию, не обязательно, чтобы выполнение конкретной инструкции всегда приводило к ошибке. Достаточно, чтобы все пути, проходящие через некоторое ребро, содержали ошибку. При этом ошибка может происходить в разных инструкциях программы.

Рассмотрим более сложный детектор, который выдаёт предупреждение, если переменная на некотором ребре может иметь только нулевое значение, и в дальнейшем всегда будет разыменована. Чтобы собрать информацию о том, что произойдёт после выполнения некоторой инструкции, используется *обратный анализ*, описываемый **в разделе 4.6**. Во время прямого анализа для каждой инструкции создается передаточная функция для обратного анализа. После завершения прямого анализа выполняется обратный, который использует созданные передаточные функции. Передаточные функции для обратного анализа на вход получают состояние на выходном ребре инструкции, и возвращают состояние для входного ребра инструкции. Производится всего одна итерация обратного анализа. Одного обхода недостаточно для полного анализа поведения функции, но экспериментально было установлено, что одного обхода хватает на практике для большинства ситуаций.

Добавим атрибут $Deref$ со значениями $deref$ – указатель точно будет разыменован и $other$ – остальное. Для его распространения выполним обратный анализ. Тогда в точках графа потока управления, где с указателем будут ассоциированы значения атрибутов $null$ и $deref$, будет выдано предупреждение, т. к. в этой точке указатель может иметь только нулевое значение, и в дальнейшем указатель точно будет разыменован. Подобный анализ позволяет находить несогласованное использование указателей.

Ещё более сложные дефекты могут быть найдены с помощью анализа с **чувствительностью к путям** выполнения программы. Такой анализ различа-

ет пути, по которым выполнение могло достигнуть некоторой точки и способен отсеять несуществующие пути, проходящие через условия, которые не могут быть истинными на одном пути. Реализации чувствительности к путям позволяет значительно повысить количество находимых дефектов без повышения уровня ложных срабатываний. У детектора без чувствительности недостаточно информации для анализа путей, выполнимость которых зависит от условных выражений. В этих случаях детектор не выдаёт предупреждение, чтобы сохранить хороший показатель истинности срабатываний за счёт пропуска ряда ошибок.

Свойства анализа формулируются в виде формул логики высказываний и представляются с помощью атрибутов. Формулы имеют вид, показанный на рис. 4, где Val – идентификаторы значений, $Const$ – константы. Для каждой литеральной формулы $Atom$ определена инструкция отрицания, возвращающая другую литеральной формулу.

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

Рис. 4. Используемые формулы.

Для реализации используется ациклический граф, узлами которого являются операции конъюнкции и дизъюнкции, а листьями – литеральные формулы. Используется алгоритм хеширования, гарантирующий, что одинаковым формулам будет сопоставлен один граф. При такой реализации граф более сложной формулы ссылается на графы её частей.

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

```
void func(char*p, int f) {
        if(!p && f)return;
        if(f)*p = 0;//f ∧ ¬p ∧ f
}
```

Атрибут на основе $FFormula$ позволяет описать ещё более сложные условия, но и требует более сложного анализа для определения разрешимости формулы. Выбор конкретного типа атрибутов зависит от детектора.

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

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

Теорема 2. $\forall A : F \Rightarrow F \bigtriangledown A$.

Согласно теореме 2, формула F^+ является приближением формулы F, т. е. формула F^+ является более слабой. Теорема позволяет выбирать приближённую формулу для описания точного условия.

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

Рассмотрим уже описанную задачу поиска разыменований нулевых указателей. Условие будем ассоциировать с идентификаторами значений. Атрибут $NullCond$ для идентификатора означает ассоциированные условия того, что значение указателя было сравнено с нулём. Атрибут $RchCond$ обозначает необходимые условия.

$$NullCond \| assume(p = 0) \| \ \mapsto NullCond[p \mapsto True]$$
$$NullCond \| assume(p! = 0) \| \ \mapsto NullCond[p \mapsto False]$$
$$NullCond \| r = \phi(a_0, .., a_k) \| = \bigcup (NullCond_{in_k}(a_k) \wedge RchCond(in_k))$$

Для инструкции разыменования $*q$ условие ошибки описывается формулой $NullCond_p(v_q) \wedge RchCond(p) \wedge v_q = 0$.

В пятой главе описывается межпроцедурный анализ. Используется анализ на основе "резюме". При таком анализе каждая функция представлена кратким описанием поведения функции резюме. Функции программы обходятся по графу вызовов, начиная с листьев, таким образом, чтобы вызываемая функция обходилась до вызывающей. Циклы в графе вызовов разрываются. После анализа каждой функции строится её резюме, описывающее интересующие эффекты функции, которое затем используется для анализа инструкций вызова функции. Резюме функции описывается набором функций val, pt и $Attr$ для каждого атрибута.

Анализ разделяет все значения в функции на два вида: внешние (предзначения) и внутренние. Для вызываемой функции внешние значения – это фактические значения из произвольного контекста вызова, про которые ничего не известно. Внутренние значения создаются внутри функции и на стороне вызывающей функции им не соответствуют никакие объекты. Поэтому в резюме помещаются все внешние значения и только те внутренние значения, до которых можно дотянуться через внешние. Последнее возможно только, если внутри функции была запись в память, на которую ссылается внешнее значение.

Для создания резюме функции используется состояние анализа в единственной инструкции выхода из функции. В резюме добавляются идентификаторы значения для входных параметров функции, после чего производится несколько обходов состояния, добавляющие в резюме зависимые ссылки и идентификаторы значения на основе функций val и pt. Если идентификатор значения v добавлен в резюме на i-й итерации обхода, то ссылки из $pt(v)$ будут добавлены на $(i+1)$-итерации. Если ссылка a добавлена на i-й итерации обхода, то идентификатор значения $val(a)$ будет добавлен на $(i+1)$-итерации. При трансляции резюме в контекст вызывающей функции происходит сопоставление формальных и фактических параметров. После чего добавляются зависимые элементы на основе функций val и pt.

Рассмотрим ссылку a в точке слияния путей. Пусть $v_1 = val_1(a)$ – идентификатор значения ссылки на одном пути, а $v_2 = val_2(a)$ – идентификатор

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

Для более точного анализа предложен механизм *отложенного слияния*. Идея механизма заключается в том, чтобы в точках слияния запоминать все предзначения, и ассоциировать их с результатом. Выделим подмножества из множества V: PV для обозначения предзначений, $JV = \{joinval(v_1, v_2) | v_1, v_2 \in V\}$ для обозначения результата слияния путей, $SV = V \setminus JV$ для простых значений.

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

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

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

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

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

Пусть $trans$ обозначает карту трансляции идентификаторов значения из резюме в сопоставленные идентификаторы значения вызывающей функции. В момент трансляции резюме в вызывающий контекст, сохранённые предзначения заменяются на соответствующие идентификаторы значения на стороне вызова: $delay'(v) = \{trans(w) | w \in delay(v)\}$. Слияние атрибутов производится для атрибутов соответствующих элементов контекста вызова $delay'(v)$ и атрибутов резюме для $ival(v)$.

На рис. 5 показан пример кода, где отложенное слияние позволяет проводить более точный анализ. Атрибут $ValueInterval$ будем использовать для анализа интервала целочисленных значений. При анализе функции set ничего не известно про значения $*p$. Поэтому значение $ValueInterval$ для $*p$ в начале функции будет $[-\infty; +\infty]$. В точке слияния (1) для условия новое значение будет $[-\infty; +\infty] \sqcup [11; 11] = [-\infty; +\infty]$, т. е. информация о том, что в функции переменной $*p$ может быть присвоено значение 11 будет потеряна. При использовании отложенного слияния для функции set будет создан идентификатор значения со ссылками на предзначение v_{dp} для $*p$ и v_{11}, при этом $ValueInterval(v_{dp}) = [-\infty; +\infty]$, $ValueInterval(v_{11}) = [11; 11]$. В контексте вызова функции set $ValueInterval(v_x) = [10; 10]$, новое значение для x будет $r = joinval(v_x, v_{11})$. Значение $ValueInterval$ для r будет вычислено как $ValueInterval_{use}(r) = ValueInterval_{use}(v_x) \sqcup ValueInterval_{set}(v_{11}) = [10; 10] \sqcup [11; 11] = [10; 11]$. Заметим, что для операции слияния используются атрибуты как контекста функции use, так и резюме для функции set.

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

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

Рис. 5. Мотивация для отложенного слияния идентификаторов значений

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

Инструмент Svace осуществляет статический поиск ошибок в программах, написанных на языках Си, Си++, Java и Си#. Описанные в работе алгоритмы были реализованы в анализаторе, осуществляющем анализ на основе резюме

для языков Си и Си++. Основной анализатор Svace при обработке программ на Си и Си++ получает на вход файлы биткода LLVM и информацию о компоновке программы. Биткод LLVM генерируется компилятором, основанным на Clang версии 3.4. По сравнению с открытой версией Clang выполнено более 500 модификаций, направленных на максимальное сохранение информации об исходном коде в генерируемом биткоде LLVM. Далее инструкции биткода преобразуются в язык svace0.

Таблица 1. Оценка предупреждений для Android 4.4 для двух версий Svace

Дефект	svace март 2015		svace март 2016	
	Всего	Истинных, %	Всего	Истинных, %
Утечки памяти	328	26	1662	46
Утечки ресурсов	93	71	234	87
Разыменование нулевых указателей	143	48	858	54
Разыменование после сравнения с нулём	90	51	955	62
Разыменование без проверки результата malloc	495	97	605	94
Сравнение с нулём после разыменования	64	81	251	74
Некорректное освобождение памяти (new/free, new[]/delete)	7	85	30	80
Использование потенциально отрицательного значения	79	68	449	76
Отсутствие va_end после va_start	23	91	25	96
Использование tainted-целых	90	81	143	80
Использование tainted-строк	27	81	47	87
Использование неинициализированных переменных	31	80	818	58
Использование указателя на переменную как массив	16	43	251	52
Проверка индекса после доступа к массиву	3	100	17	70
Доступ к массиву с неправильной проверкой индекса	71	42	249	67
Неконсистентность конструкторов и деструкторов, приводящая к утечке	28	75	196	74

Целью работы было увеличение количества выдаваемых предупреждений при сохранении высокого уровня истинных срабатываний. Для оценки результатов работы было произведено сравнение предупреждений, выданных текущей версией инструмента Svace (март 2016), с предупреждениями, выданными старой версией Svace (март 2015). Табл. 1 содержит результаты сравнения. Как

видно из таблицы, количество выдаваемых предупреждений существенно возросло, при этом процент истинных срабатываний был сохранён, а в некоторых случаях улучшен.

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

В Заключении формулируются результаты диссертационной работы и приводятся направления дальнейших исследований. В дальнейшем планируется повышать уровень истинных срабатываний с помощью увеличения степени контекстной чувствительности и разработки вспомогательных алгоритмов (для определения взаимосвязей между переменными; анализа функций условно завершающих программу; девиртуализации вызова виртуальных функций в Си++ и др).

Основные результаты диссертационной работы:

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

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

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

Список публикаций автора по теме диссертации

1. Аветисян А. И., Бородин А. Е. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок // Труды ИСП РАН. 2011. Т. 21. С. 39–54.
2. Аветисян А. И., Белеванцев А. А., Бородин А. Е., Несов В. С. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды ИСП РАН. 2011. Т. 21. С. 23–38.
3. Иванников В. П., Белеванцев А. А., Бородин А. Е. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды ИСП РАН. 2014. Т. 26, № 1. С. 231–250.
4. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code // Programming and Computer Software. 2014. Vol. 40, no. 5. P. 265–275.
5. Бородин А. Е. Статический поиск ошибок повторной блокировки семафора // Труды ИСП РАН. 2014. Т. 26, № 3. С. 103–112.
6. Бородин А. Е., Белеванцев А. А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Труды ИСП РАН. 2015. Т. 27, № 6. С. 111–134.
7. Borodin A. Summary Based Static Analysis for Practical Search for Defects in C Programs and Libraries // Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on / IEEE. Cleveland: 2014. P. 231–232.
8. Бородин А. Е. Анализ на основе аннотаций для практического поиска дефектов в программах и библиотеках, написанных на языке Си // Труды XXI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2014». Москва: 2014. С. 100–102.
9. Бородин А. Е. Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Открытая конференция по компиляторным технологиям. Москва: 2015.