На правах рукописи

Кошелев Владимир Константинович

МЕЖПРОЦЕДУРНЫЙ СТАТИЧЕСКИЙ АНАЛИЗ ДЛЯ ПОИСКА ОШИБОК В ИСХОДНОМ КОДЕ ПРОГРАММ НА ЯЗЫКЕ C#

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

Автореферат
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва — 2017

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

Научный руководитель:	**Белеванцев Андрей Андреевич**, канд. физ.-мат. наук, ведущий научный сотрудник
Официальные оппоненты:	**Гергель Виктор Павлович,** доктор технических наук, профессор, декан факультета вычислительной математики и кибернетики, Нижегородского государственного университета имени Н. И. Лобачевского, директор Научно-исследовательского института прикладной математики и кибернетики, руководитель Приволжского научно-образовательного Центра суперкомпьютерных технологий **Павлов Евгений Геннадьевич,** кандидат технических наук, ООО "Исследовательский центр Самсунг", старший инженер программист, руководитель проектов
Ведущая организация:	Вычислительный центр им. А. А. Дородницына Федерального исследовательского центра «Информатика и управление» Российской академии наук

Защита состоится 25 мая 2017 г. в 12:30 на заседании диссертационного совета Д 002.087.01 при Институте системного программирования РАН 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 « » апреля 2017 года.

Ученый секретарь
диссертационного совета Д 002.087.01,
канд. физ.-мат. наук Зеленов С. В.

Общая характеристика работы

Актуальность темы.

Наряду с тестированием и динамическим анализом, статический анализ исходного кода широко используется для поиска дефектов в программах. Многие современные стандарты сертификации требуют, чтобы программное обеспечение было проверено на наличие дефектов, в том числе при помощи инструментов статического анализа. Например, таким стандартом является ГОСТ Р 56939-2016 для разработки безопасного программного обеспечения, который начал действовать в РФ 06.01.2017.

От статических анализаторов для применения в промышленном цикле разработки требуются: высокая скорость анализа (не более 2-3 часов для полного анализа проектов из нескольких миллионов строк кода); высокое качество анализа (высокий процент истинных срабатываний (50-70%) и поддержка поиска популярных классов ошибок, включающая поиск типичных ошибочных ситуации в рамках данных классов). Для соблюдения этих требований статические анализаторы вынужденно используют нестрогий анализ, позволяющий достичь компромисса между временем работы, количеством пропусков дефектов и процентом ложных срабатываний. При этом для поиска сложных межпроцедурных дефектов необходимо использовать анализ программ, учитывающий зависимости по данным (чувствительность к потоку), условия переходов (чувствительность к путям), контексты вызовов (чувствительность к контексту) и работу с динамической памятью.

В течение последних десяти лет язык программирования C# стабильно входит в шестерку самых распространенных языков программирования про версии TIOBE. В настоящий момент чувствительный к контексту, потоку и путям анализ с целью выявления дефектов для языка программирования C# осуществляют лишь закрытые коммерческие инструменты, для которых отсутствует детальное описание используемых подходов и алгоритмов. Существующие открытые решения для поиска дефектов ограничиваются обходом абстрактного синтаксического дерева или простым внутрипроцедурным анализом, не проводя детального анализа потоков управления и данных.

Методы проведения статического анализа программ разрабатываются как во многих научных центрах, таких как университет Стенфорд, университет Беркли, университет Британской Колумбии, ИСП РАН, так и в коммерческих организациях, таких как Coverity, Klocwork, JetBrains. Общепризнанным способом организации масштабируемого межпроцедурного анализа является использование резюме функций, сжато описывающие их поведение. Конкретная форма и алгоритмы построения резюме целиком зависят от использующихся методов внутрипроцедурного анализа. Методов статического анализа, использующихся в открытых инструментах, на поверку оказывается недостаточно для удовлетворения указанных выше требований. В свою очередь, для коммерческих инструментов анализа отсутствует описание использованных методов.

Метод символьного выполнения, являясь чувствительным к потоку и путям, успешно применяется при анализе программ для поиска ошибочных входных данных, используя для этого решатели задачи выполнимости формул. В данной работе предлагается воспользоваться методом символьного выполнения для проведения внутрипроцедурного анализа. Метод символьного выполнения является ресурсоёмким, поэтому для достижения высокой скорости анализа в данной работе предлагается разработать:

- алгоритм объединения состояний символьного выполнения;
- алгоритм упрощения формул, основанный на свойствах доминаторов;
- подход для поддержки циклов с фиксированным числом итераций;
- метод моделирования вызовов с использованием резюме;
- подход для уменьшения числа запросов к решателю за счет доказательства свойств программы методами статического анализа.

Использование данных подходов позволяет достичь требуемой скорости при сохранении высокого качества анализа.

Методы статического анализа применяют различные критерии для выдачи предупреждений о наличии дефектов в зависимости от целей анализатора и использующихся алгоритмов анализа. Для успешного применения символьного выполнения для поиска дефектов необходимо разработать метод определения критерия выдачи предупреждений, учитывающий особенности символьного выполнения и позволяющий соблюсти ба-

ланс между числом пропусков ошибок и количеством ложных предупреждений.

Цели и задачи диссертационной работы: разработка внутрипроцедурных алгоритмов анализа и алгоритмов межпроцедурного анализа, чувствительных к потоку, контексту и путям, метода определения критерия выдачи предупреждений о наличии дефекта, обеспечивающих поиск дефектов в исходном коде программ, написанных на языке C#. Разработанные алгоритмы должны обеспечивать качество анализа, соответствующее современным требованиям для промышленных статических анализаторов, включая масштабируемость, достаточную для анализа программ, состоящих из миллионов строк кода.

Для достижения поставленных целей были сформулированы и решены следующие задачи:

1. Разработка внутрипроцедурных чувствительных к потоку и путям алгоритмов анализа, позволяющих на основе вычисленной ими информации выполнять поиск широкого класса дефектов; доказательство корректности алгоритмов.
2. Разработка межпроцедурных чувствительных к потоку, контексту и путям алгоритмов анализа, включающих алгоритм построения резюме метода по результатам внутрипроцедурного анализа и алгоритм применения резюме в точках вызова.
3. Разработка метода определения критерия выдачи предупреждения о наличии дефекта с учетом особенностей проводимого анализа, подходящего для широкого класса дефектов. Разработка конкретных детекторов для приведенного критерия выдачи предупреждения.
4. Оценка на практике характеристик предложенных методов и алгоритмов на соответствие заявленным требованиям.

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

1. Разработаны алгоритмы внутрипроцедурного анализа, обладающие чувствительностью к потоку и путям, не теряющие информацию в точках объединения.
2. Разработаны алгоритмы межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающие результаты внут-

рипроцедурного анализа в виде резюме метода, и применяющие резюме при обработке вызова данного метода.

3. Разработан метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений.
4. Доказательства корректности предложенного алгоритма внутрипроцедурного анализа и соответствия разработанных детекторов выбранному критерию выдачи предупреждений.

Теоретическая и практическая значимость. Предложены алгоритмы чувствительного к путям внутрипроцедурного и межпроцедурного анализа для задачи поиска широкого класса дефектов. Разработан метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям.

Все предложенные алгоритмы были реализованы в статическом анализаторе SharpChecker. Анализатор SharpChecker предполагает промышленное использование при разработке программного обеспечения для раннего обнаружения программных дефектов. Данный анализатор также может быть использован в сертифицирующих центрах для оценки качества сертифицируемого программного обеспечения. Разработанный анализатор SharpChecker внедрён для промышленного использования в компании Samsung.

Методология и методы исследований. Результаты диссертации были получены с использованием методов и моделей, применяемых при проведении статического анализа. Математическую основу данной работы составляют теория графов, теория множеств, математическая логика, теория алгоритмов.

Основные положения, выносимые на защиту:

1. алгоритм внутрипроцедурного анализа, обладающий чувствительностью к потоку и путям, при этом не теряющий информацию в точках объединения;
2. алгоритм межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающий результаты внутрипроцедурного анализа в виде резюме метода и применяющий резюме при обработке вызова данного метода;

3. метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений.

Апробация работы. Основные результаты работы докладывались на следующих конференциях:

1. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия 2014);
2. SPB .Net Meetup №10 (Санкт-Петербург, Россия, 2016);
3. Открытая конференция ИСП РАН 2016 (Москва, Россия, 2016);
4. Научно-исследовательский семинар Института системного программирования РАН.

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

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях [1–6], 5 из которых изданы в журналах, рекомендованных ВАК. Вклад автора в работе [2] заключается в разработке алгоритмов внутрипроцедурного и межпроцедурного анализа и разработке детектора для поиска доступа к нулевому указателю (`null`). В статье [3] вклад автора состоит в разработке алгоритмов оптимизации размера формул и реализации инфраструктуры инструмента SharpChecker. В работе [4] вклад автора состоит в разработке метода определения критерия выдачи предупреждений. В публикации [5] вклад автора заключается в разработке ядра анализа помеченных данных для программ на языках C/C++.

Статический анализатор языка C# SharpChecker, в котором реализованы предложенные методы анализа, включен в единый реестр российских программ для электронных вычислительных машин и баз данных по Приказу Минкомсвязи России от 09.03.2017 №103, Приложение 1, пп. №37, реестровый № 2910.

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 рассматривается задача поиска ошибок в исходном коде программ при помощи методов статического анализа, и производится обзор существующих подходов.

Ключевая сложность разработки статического анализа заключается в поиске компромисса между масштабируемостью, классом обнаруживаемых ошибок, процентом ложных срабатываний и пропуском ошибок и дополнительными ограничениями, накладываемыми на анализируемую программу. Идеальный статический анализатор должен хорошо масштабироваться, обнаруживать все ошибки времени выполнения, не иметь ложных срабатываний и пропусков ошибок и применяться ко всем программам без ограничений. Создание идеального анализатора невозможно в силу алгоритмической неразрешимости задачи по теореме Райса.

В данной работе рассматривается задача поиска дефектов. Дополнительных ограничений на анализируемые программы не накладывается. Задача анализа — поиск дефектов в исходном коде, приводящих к ошибкам времени выполнения. Допускается наличие ложных срабатываний и пропусков ошибок. Сложность разработки заключается в обнаружении практически значимого класса ошибок при сохранении низкого процента ложных срабатываний и хорошей масштабируемости.

Данная задача решалась в ряде научно-исследовательских и промышленных инструментов. В данной главе рассматриваются следующие научно-исследовательские инструменты для анализа программ на C/C++: Prefix, Saturn, Varvel. К сожалению, публикации на тему организации межпроцедурного чувствительного к путям статического анализа для языка C#, решающего задачу поиска дефектов, отсутствуют. Из промышленных статических анализаторов, поддерживающих анализ программ на языке C#, рассматриваются Coverity, Klocwork, PVS-Studio, ReSharper.

В результате анализа возможностей приведенных выше инструментов делается вывод, что для создания масштабируемого инструмента статического анализа необходимо использовать чувствительный к путям внутрипроцедурный анализ в комбинации с межпроцедурным анализом, основанным на резюме.

Инструмент Svace, разработанный в Институте системного программирования РАН, поддерживает межпроцедурный анализ на основе резюме, однако поддержка чувствительности к путям оставлена на усмотрение разработчика детектора. В данной работе рассматриваются межпроцедурный анализ, полностью чувствительный к путям, и метод определения критерия выдачи предупреждения об ошибке.

Во второй главе рассматриваются особенности анализа языка C#, а также вводится внутреннее представление для программ на языке C#.

Язык программирования C# — объектно-ориентированный статически типизированный язык высокого уровня. Компания Microsoft активно развивает язык C#, выпуская новую версию каждые два-три года. С каждой новой версией в язык C# добавляются новые языковые возможности, что требует от инструментов статического анализа адаптироваться к ним. В данной работе поддерживаются следующие возможности: динамическая память, динамическая проверка типа, исключения, конструкции управления ресурсами. Частично поддерживаются: косвенные вызовы, многопоточность, замыкания, Linq-выражения. Не рассматривается поддержка небезопасных секций и отражений.

В качестве внутреннего представления предлагается использовать набор графов потока управления (ГПУ) для анализируемых методов.

ГПУ состоит из базовых блоков, соединённых рёбрами. Каждый базовый блок содержит последовательность инструкций. Каждой инструкции соответствует дополнительная локальная переменная, значение которой равно результату выполнения этой инструкции. С точки зрения языка C#, дополнительные локальные переменные соответствуют результатам вычисления выражений. Дополнительные локальные переменные могут быть использованы в базовых блоках, доминируемых базовым блоком, содержащим инструкцию. Переопределение дополнительных локальных переменных недопустимо.

Кроме дополнительных локальных переменных, для всего ГПУ определён набор параметров метода и набор локальных переменных. Тип параметров задан прототипом метода в соответствующем классе. Тип локальных переменных определяется по типу переменных, которыми они инициализируются.

В данной работе рассматриваются следующие инструкции внутреннего представления:

$$val := a.f; \quad \texttt{чтение поля} \tag{1}$$
$$val := a[i]; \quad \texttt{чтение элемента массива} \tag{2}$$
$$val := a \triangledown b \quad \texttt{некоторая бинарная операция} \tag{3}$$
$$val := \triangledown a \quad \texttt{некоторая унарная операция} \tag{4}$$
$$val := a.f = b; \quad \texttt{запись в поле} \tag{5}$$
$$val := a = b; \quad \texttt{запись в локальную переменную} \tag{6}$$
$$val := const; \quad \texttt{определение константы} \tag{7}$$
$$val := foo(params); \quad \texttt{вызов метода} \tag{8}$$
$$return\ var; \quad \texttt{возврат из метода} \tag{9}$$
$$throw\ var; \quad \texttt{возникновение исключения} \tag{10}$$

Таким образом, в данной главе определяется внутреннее представление, над которым будет производиться анализ.

В третьей главе рассматривается вопрос использования символьного выполнения для проведения чувствительного к путям и к потоку внутрипроцедурного анализа.

При символьном выполнении будем считать, что каждый метод является точкой входа в программу. Входные параметры методы и состояние памяти могут иметь произвольные значения. Тогда параметризуем начальное состояние в точке входа в метод набором символьных переменных. Символьные переменные являются типизированными в соответствии с типами исходных полей и переменных. Дополнительно для символьных переменных, имеющих ссылочный тип, потребуем, чтобы их значения не являлись псевдонимами друг друга. Данное предположение, разумеется, ограничивает множество рассматриваемых состояний.

Для описания параметризации введем следующие множества:

– V — множество переменных, определяемых в методе;
– $P, P \subseteq V$ — множество параметров метода;
– S — множество символьных переменных;
– $S_R, S_R \subseteq S$ – множество символьных переменных ссылочного типа;

– $\vec{s}$ – вектор, состоящий из всех символьных переменных;

– F — множество полей, по которым осуществляется доступ.

Тогда множество начальных состояний программы на входе в метод будет описываться следующим образом:

– $\mathcal{P}_{entry} : P \to S$ — символьная параметризация параметров метода;

– $\mathcal{H}_{entry} : S_R \times F \rightharpoonup S$ — символьная параметризация состояния кучи.

Здесь и далее *entry* — точка входа в метод.

Пару отображений $\mathcal{P}_{entry}$ и $\mathcal{H}_{entry}$, задающую символьную параметризацию на входе в метод, будем называть начальным символьным состоянием или $State_{entry}$. Подставляя вместо символьных переменных, содержащихся в $\vec{s}$, их допустимые конкретные значения, получим множество способов запустить метод. Обозначим данное множество как Σ. Предполагается, что значения символьных переменных полностью определяют результаты вычисления всех булевых выражений, а, следовательно, и путь выполнения. Если результат булевых выражений зависит от неизвестных значений, то для данных значений также введём символьные переменные, определяемые в точке входа. Так как вектор конкретных значений $\vec{\sigma} \in \Sigma$ однозначно определяет путь выполнения, то обозначим как $L(\vec{\sigma})$ путь выполнения, соответствующий вектору конкретных значений $\vec{\sigma}$.

Введем вспомогательные обозначения:

– $L(\vec{\sigma})_e$ — значения переменных и состояние памяти на ребре e пути выполнения $L(\vec{\sigma})$;

– Σ_l — множество векторов конкретных значений, для которых $L(\vec{\sigma})$ соответствует пути на ГПУ l.

Для определения символьного состояния в произвольной точке ГПУ введем SE — множество символьных выражений. Символьными выражениями являются символьные переменные и константы, а также выражения, построенные из символьных выражений с помощью унарных, бинарных и условного операторов.

Для задания ограничений, учитывающих условия переходов, в символьном состоянии явно вводится предикат пути. Предикатом пути будем называть символьное выражение $\mathcal{G}$ логического типа, задающее множество допустимых значений вектора символьных переменных — $\Sigma_{\mathcal{G}}$, $\vec{\sigma} \in \Sigma_{\mathcal{G}} \iff \mathcal{G}(\vec{\sigma})$.

Тогда символьное состояние в конкретной вершине пути ГПУ будем описывать набором $State = \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle$, где:

– $\mathcal{V} : V \to SE$ — параметризованные значения локальных переменных;

– $\mathcal{H} : S_R \times F \rightharpoonup SE$ — параметризованное состояние кучи;

– $\mathcal{G}$ — предикат пути.

Введём функцию конкретизации для символьного состояния — $\varphi(State = \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle, \vec{\sigma})$, которая в случае, если $\vec{\sigma} \in \Sigma_{\mathcal{G}}$, строит конкретное состояние для локальных переменных и памяти, подставляя в $\mathcal{V}$ и $\mathcal{H}$ вместо символьных переменных конкретные значения, соответствующие $\vec{\sigma}$.

Задача символьного выполнения заключается в построении символьного состояния для выбранного пути на ГПУ. Формально данная задача ставится следующим образом: для заданного пути l на ГПУ, начинающегося в $entry$ и заканчивающегося ребром e, необходимо вычислить символьное состояние $State_e$ такое, что:

$$\mathcal{G} = \Sigma_l \wedge \forall \vec{\sigma} \in \Sigma_l (L(\vec{\sigma})_e = \varphi(State_e, \vec{\sigma})). \tag{11}$$

Здесь и далее операция $\uplus$ обозначает переопределение отображения для заданных значений:

$$(A \uplus B)(x) = \begin{cases} B(x), & \textit{если } x \in domain(B); \\ A(x), & \textit{иначе} \end{cases} \tag{12}$$

Для построения символьных состояний для точек на пути l определим правила интерпретации инструкций относительно начального состояния в соответствии с их семантикой. Пусть e_{pred}, e_{succ} — ребра пути на ГПУ до и после инструкции $\mathcal{I}$ соответственно. Тогда введем следующие правила интерпретации для соответствующих инструкций.

$$\frac{\langle e_{pred} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle \quad \mathcal{I} = \text{"}a = b;\text{"}}{\langle e_{succ} \rangle \models \langle \mathcal{V} \uplus \{a \mapsto \mathcal{V}(b)\}, \mathcal{H}, \mathcal{G} \rangle}$$

$$\frac{\langle e_{pred} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle \quad \mathcal{I} = \text{"}a = const;\text{"}}{\langle e_{succ} \rangle \models \langle \mathcal{V} \uplus \{a \mapsto const\}, \mathcal{H}, \mathcal{G} \rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b \diamond c;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto \mathcal{V}(b) \diamond \mathcal{V}(c)\}, \mathcal{H}, \mathcal{G}\rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = \nabla b;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto \nabla b\}, \mathcal{H}, \mathcal{G}\rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "assume(c);"}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \wedge \mathcal{V}(c))\rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b.f;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus a \mapsto \mathcal{H}(\mathcal{V}(b), f), \mathcal{H}, \mathcal{G}\rangle)}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a.f = b;"}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H} \uplus \langle \mathcal{V}(a), f\rangle \mapsto \mathcal{V}(b), \mathcal{G}\rangle)}$$

При задании правил интерпретации не были рассмотрены инструкции вызова метода и доступа к массиву. Интерпретация инструкций вызова с учетом резюме вызванных методов будет рассмотрена далее.

В данной главе рассматривается подход, позволяющий проводить анализ циклов с фиксированным числом итераций. Данный подход предлагает подменять результаты арифметических операций в внутри цикла на новые неизвестные символьные значения. Таким образом, становится возможным выход из циклов с фиксированным числом итераций.

Лемма 3.1. Для всех символьных состояний, полученных в результате применения правил интерпретации, начиная из начального символьного состояния для заданного пути на ГПУ l, выражение (11) верно.

Так как символьное выполнение может проанализировать лишь определённый набор путей выполнения, сразу определим пути, которые будут подвергаться анализу. Для этого введем понятие графа развертки.

Графом развертки для ГПУ $G = \langle V,E,v_{entry},v_{exit}\rangle$ будем называть ациклический ориентированный связанный граф с выделенными истоком и стоком $G' = \langle V',E',v'_{entry},v'_{exit}\rangle$, для которого определена функция соответствия $\varphi : V' \to V$ такая, что верно:

$$\varphi(v'_{entry}) = v_{entry} \quad \varphi(v'_{exit}) = v_{exit} \tag{13}$$

$$\forall v' \in V' \setminus \{v'_{entry}, v'_{exit}\} : \varphi(v') \notin \{v_{entry}, v_{exit}\} \tag{14}$$

$$\langle v', u'\rangle \in E' \implies \langle \varphi(v'), \varphi(u')\rangle \in E' \tag{15}$$

$$\langle v',u'\rangle \in E' \wedge \langle v',w'\rangle \in E' \wedge u' \neq w' \implies \varphi(u') \neq \varphi(w') \tag{16}$$

Для перехода от анализа конкретного пути ГПУ к анализу графа развертки необходимо определить операции объединения символьных состояний, а также переопределить правила вывода для чтения и записи в память. Не нарушая общности, можно считать, что объединяются всегда лишь два состояния. В том случае, если в базовый блок входит более двух рёбер, добавлением пустых базовых блоков можно добиться того, что объединению будут подвергаться не более двух блоков.

Тогда задачу объединения символьных состояний сформулируем следующим образом. Пусть дана тройка вершин $\langle l,r,j\rangle$ таких, что есть ребра $\langle l,j\rangle$ и $\langle r,j\rangle$. Для вершин l и r известны состояния $\langle \mathcal{V}_l, \mathcal{H}_l, \mathcal{G}_l\rangle$ и $\langle \mathcal{V}_r, \mathcal{H}_r, \mathcal{G}_r\rangle$. Необходимо построить символьное состояние для вершины j, объединяющее информацию о путях, пришедших по ребрам $\langle l,j\rangle$ и $\langle r,j\rangle$.

Предикат при объединении вычисляется следующим образом: $\mathcal{G}_j = \mathcal{G}_l \vee \mathcal{G}_r$. По построению данный предикат допускает все возможные значения начальных состояний, соответствующих предикатам $\mathcal{G}_l$ и $\mathcal{G}_r$.

Для построения $\mathcal{V}_j$ и $\mathcal{H}_j$ необходимо найти такое символьное выражение C, что одновременно:

$$\Sigma_{\mathcal{G}_l} \subset \Sigma_C, \quad \Sigma_{\mathcal{G}_r} \cap \Sigma_C = \varnothing \tag{17}$$

Если выполнение дошло до вершины j из начального состояния, соответствующего значениям $\vec{\sigma}$, то можно гарантировать, что если $\vec{\sigma} \in \Sigma_C$, то было пройдено ребро $\langle l,j\rangle$, иначе $\langle r,j\rangle$. Тогда состояние памяти и переменных может быть выражено при помощи условного оператора от C следующим образом:

$$\mathcal{V}_j(v) = \begin{cases} \mathcal{V}_l(v), \ \textit{если } \mathcal{V}_l(v) = \mathcal{V}_r(v) \\ (C)?(\mathcal{V}_l(v)) : (\mathcal{V}_r(v)), \ \textit{иначе} \end{cases} \tag{18}$$

$$\mathcal{H}_j(s,f) = \begin{cases} \mathcal{H}_l(s,f), \ \textit{если } \mathcal{H}_l(s,f) = \mathcal{H}_r(s,f) \\ (C)?(\mathcal{H}_l(s,f)) : (\mathcal{H}_r(s,f)), \ \textit{иначе} \end{cases} \tag{19}$$

В качестве C может быть использован предикат $\mathcal{G}_l$. Действительно, для предиката $\mathcal{G}_l$ выполняется требование (17).

Введём операцию $Decompose : SE \to 2^{S \times SE}$, которая для символьного выражения ссылочного типа строит набор пар из символьной перемен-

ной и условия, при котором исходное выражение равно данной символьной переменной. Все условия являются попарно несовместными. Результат *Decompose* может быть вычислен с помощью обхода дерева выражений, содержащих условные операторы.

Введём операцию взятия поля $DEREF : SE \times F \to SE$, которая производит чтение из символьного выражения SE по полю f:

$$Decompose(se) = \{\langle s_i, c_i\rangle | i \in [1 \dots n]\}$$

$$|Decompose(se)| = n$$

$$h_i = \mathcal{H}(s_i, f)$$

$$DEREF(se, f) = \begin{cases} ite(c_1, h_1, ite(c_2, h_2, ...ite(c_{n-1}, h_{n-1}, h_n))), \textit{для } n > 1 \\ h_1, \textit{для } n = 1 \end{cases}$$

При помощи данных операций зададим интерпретацию для инструкций чтения и записи поля.

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b.f;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto DEREF(\mathcal{V}(b), f)\}, \mathcal{H}, \mathcal{G}\rangle} \tag{20}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a.f = b;" \quad Decompose(\mathcal{V}(a)) = \{s_i, c_i\}}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H} \uplus_{i=1}^{n} \{\{s_i, f\} \mapsto ite(c_i, \mathcal{V}(b), \mathcal{H}(s_i, f))\}, \mathcal{G}} \tag{21}$$

Будем говорить, что символьное состояние $\langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle$ относительно точки e определено корректно, если $\Sigma_{\mathcal{G}}$ содержит в точности все $\vec{\sigma}$, из которых e достижима, и для каждого $\vec{\sigma} \in \Sigma_{\mathcal{G}}$ верно, что $\varphi(\langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle, \vec{\sigma}) = L(\vec{\sigma})_e$

Теорема 3.1. Определённые выше правила интерпретации строят корректные символьные состояния.

Алгоритм 3.1. Выберем конкретную вершину в графе развертки — u. Будем считать, что все вершины, которые топологически меньше u, уже обработаны и их предикаты пути посчитаны. Найдем для u её область постдоминирования. Путь u не постдоминирует точку входа. Рассмотрим граф вершин, топологически меньших либо равных u, назовём его G'. Построим разрез $\langle S,T\rangle$ такой, что к S относятся все вершины G', которые

в исходном графе не постдоминирует данная вершина, а к T — все вершины из области постдоминирования. Пусть $\langle s_i, t_i \rangle$ — список ребер, лежащих на разрезе. Тогда предикат пути для u построим следующим образом: $\langle \vee_i (\mathcal{G}(s_i) \wedge Cond(s_i,t_i)), \rangle$, где $\mathcal{G}(s_i)$ — предикат пути для вершины s_i, а $Cond(s_i,t_i)$ — условие перехода по ребру. В данном случае под условием перехода по ребру понимается условие в инструкции `assume` базового блока, в который входит ребро. Если у блока отсутствует `assume`, то условие считается тождественным истине.

Теорема 3.2. Алгоритм 3.1 корректно вычисляет предикат пути.

Алгоритм 3.2. Пусть dom — это непосредственный доминатор вершин lhs и rhs, тогда рассмотрим множество вершин, топологически меньше либо равных lhs и больше либо равных dom, назовём его L. Рассмотрим аналогичное множество для rhs, назовем его R. Рассмотрим тогда $L \cap R$ и $L \setminus R$, без ограничения общности будем считать, что $L \setminus R$ не пусто. Тогда $Interpol = \vee_i (\mathcal{G}_{dom,u_i} \wedge Cond(u_i,v_i))$, где $\langle u_i,v_i \rangle$ — ребра, лежащие на разрезе $\langle L \cap R, L \setminus R \rangle$, а $\mathcal{G}_{dom,u_i}$ — условие того, что путь выполнения дойдет до u_i, пройдя dom.

Теорема 3.3. Алгоритм 3.2 корректно вычисляет условие $Interpol$.

В четвертой главе рассматривается вопрос учета вызовов при символьном выполнении. Выделяются три типа вызовов: прямые вызовы известных методов, косвенные вызовы и вызовы методов из сторонних библиотек.

Для моделирования прямых вызовов необходимо разработать процедуру построения резюме метода. В данной работе предлагается хранить в резюме состояние памяти до и после выполнения метода.

$$\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE}, \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle \tag{22}$$

Для построения резюме такого вида могут быть использованы результаты внутрипроцедурного символьного выполнения. В качестве $\mathcal{V}_{PRE}$ и $\mathcal{H}_{PRE}$ предлагается использовать начальное символьное состояние, а в качестве $\mathcal{V}_{POST}$ и $\mathcal{H}_{POST}$ — символьное состояние, полученное в точке выхода.

Однако отсутствие дополнительных ограничений на размер приведет к неограниченному разрастанию как резюме, так и символьных состояний,

что приведет к существенной деградации производительности анализатора. Следовательно, необходимо ограничить максимальный размер резюме. В данной работе предлагается ввести ограничения на размер графа памяти $\mathcal{H}$ и на размер графа символьных выражений. Далее рассматриваются алгоритмы, позволяющие произвести сжатие данных графов в соответствии с рассмотренными ограничениями.

Для применения резюме в точке вызова метода необходимо построить соответствие между памятью в точке вызова и состояниями памяти $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$ и $\langle \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle$. Пусть метод вызывается с набором параметров $\{se_i\}$, где se_i — символьное выражение, соответствующее i-тому аргументу вызова. Будем считать, что построено отображение $Formal2Actual$, отображающее символьные переменные, соответствующие формальным параметрам метода, в символьные выражения, с которыми данный метод был вызван.

Опишем процедуру, позволяющую построить сопоставление между символьными выражениями вызванного и вызывающего методов. Для начала сопоставим начальное состояние вызываемого метода, заданное парой $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$, и текущее символьное состояние $\langle \mathcal{V}, \mathcal{H} \rangle$. Для этого воспользуемся алгоритмом, приведённым на листинге 1.

Использующаяся функция $DEREF$ аналогична по своей семантике функции (20).

Благодаря построенному отображению из символьных переменных вызванного метода в символьные выражения вызывающего, возможно осуществить трансляцию символьных выражений вызванного метода в символьные выражения вызывающего. Для этого достаточно заменить все вхождения символьных переменных вызванного метода на соответствующие выражения вызывающего. Функцию, осуществляющую такую трансляцию, определим как $Translate : SE_{callee} \to SE_{caller}$. В случае, если для символьного выражения вызванного метода нет аналогичного символьного выражения в вызывающей, то заводится новая неизвестная символьная переменная, соответствующая выражению вызванного метода.

Для завершения применения резюме необходимо обновить состояние памяти в точке вызова в соответствии с $\langle \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle$. Для этого предлагается использовать алгоритм, аналогичный 1. Он позволяет привести содержимое памяти в соответствие с обновлениями, произошедшими в вы-

```
PROC GetCallee2Caller(Callee2Caller : Array of SE x SE,
  CallerHeap: S x F -> SE, CalleeHeap : S x F -> S) : SE -> SE
BEGIN
  result : SE -> SE = Empty
  visited: Set of SE = Empty
  queue: Queue of <SE, SE> = Empty
  FOREACH <callee, caller> in Callee2Caller; DO
    visited.Add(callee)
    queue.Add(<callee, caller>)
    result.Add(<callee, caller>)
  DONE
  WHILE Queue.Size > 0 DO
    <callee, caller> = Queue.Dequeue();
    FOREACH field in CallerHeap.Keys.Where(s == caller) DO
      pointsToCallee = CalleeHeap.Get(<callee, field>)
      pointsToCaller = DEREF(CalleeHeap, caller, field)
      result.Add(pointsToCallee, pointsToCaller);
      visited.Add(pointsToCallee);
      queue.Enqueue(<pointsToCallee, pointsToCaller>);
    DONE
  DONE
  return result
END
```

Листинг 1 Алгоритм построения функции *Translate*

званном методе. После выполнения данного обновления, основное резюме метода считается применённым, и начинается применение вспомогательных частей резюме, определённых анализаторами.

В этой главе также рассматривается подход к анализу чистых методов. Метод считается чистым, если он не меняет динамическую память и для одного и того же входного состояния возвращает одно и то же значение. Такая поддержка требуется, например, при анализе использования коллекций, и в частности массивов. Поддержку чистых методов предлагается осуществлять с помощью кеширования состояний, с которыми данные методы были вызваны. Если метод уже вызывался с некоторым символьным состоянием, то результат вызова берется из кеша.

Для межпроцедурного анализа помеченных данных предлагается использовать подход, комбинирующий слайсинг и внутрипроцедурный чувствительный к путям анализ. Слайсинг предлагается осуществлять, сводя задачу распространения помеченных данных к IFDS-задаче. Результатом слайсинга является набор межпроцедурных путей в межпроцедурном

ГПУ, по которым помеченные данные распространяются от истока к стоку без учета условий переходов. Далее данные пути подвергаются описанному чувствительному к путям внутрипроцедурному анализу, в инструкциях вызова трансляция символьных состояний осуществляется по аналогии с алгоритмами применения резюме.

В пятой главе рассматривается метод определения критерия выдачи предупреждения о наличии дефекта. Для выбранного критерия предлагаются алгоритмы эффективного поиска соответствующих дефектов на примере ошибок типа доступ по нулевому указателю и утечка ресурсов. Показывается соответствие обнаруживаемых алгоритмами ошибок выбранному критерию.

Для построения критерия выдачи предупреждений, рассмотрим множество $\{\Sigma_i\} \subseteq 2^{\Sigma}$, где $\Sigma_i \subseteq \Sigma$. Элемент Σ_i назовём *абстракцией*, а $\{\Sigma_i\}$ — множеством абстракций. Будем говорить, что в вершине графа развертки B содержится ошибка, если найдется такая абстракция Σ_i, что на всех наборах значений переменных $\vec{\sigma} \in \Sigma_i$ произойдёт ошибка в точке B.

Пусть Σ_i выражается некоторым образом через символьные переменные $\vec{s}$. Тогда обозначим как $P_B^{\Sigma_i}(\vec{s})$ формулу от символьных переменных, задающую условие того, что $\vec{s}$ одновременно принадлежит абстракции Σ_i, и управление дойдёт до точки B. Как $Err_B(\vec{s})$ обозначим условие того, что в точке B произойдет ошибка. Тогда дадим определение ошибки для абстракции Σ_i следующим образом:

$$Err_B^{\Sigma_i} = (\exists \vec{s} : P_B^{\Sigma_i}(\vec{s})) \wedge (\forall \vec{s} : P_B^{\Sigma_i}(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{23}$$

Тогда наличие ошибки в точке B определим как существование абстракции, на которой произойдёт ошибка:

$$\exists \Sigma_i : Err_B^{\Sigma_i} \tag{24}$$

Определение (24) будем называть общим определением ошибки, $\langle \{\Sigma_i\}, Err_B \rangle$ — определением ошибки, полученным подстановкой $\{\Sigma_i\}$ и Err_B в общее определение ошибки. В зависимости от выбора множества абстракций $\{\Sigma_i\}$ будут различаться множества обнаруживаемых ошибок.

В некоторых случаях можно показать, что имеет место вложение множеств обнаруживаемых ошибок. Рассмотрим два различных множества аб-

стракций $\mathcal{A}' = \{\Sigma'_i\}$ и $\mathcal{A}'' = \{\Sigma''_i\}, \mathcal{A}', \mathcal{A}'' \in 2^{\Sigma}$ таких, что:

$$\forall i \exists \vec{J} : \Sigma'_i = \bigcup_{j \in \vec{J}} \Sigma''_j \tag{25}$$

Лемма 5.1. Пусть $Err_B^{\mathcal{A}'}$ множество ошибок в точке B при использовании абстракций $\mathcal{A}'$, а $Err_B^{\mathcal{A}''}$ – при использовании абстракций $\mathcal{A}''$, тогда верно:

$$\forall B : Err_B^{\mathcal{A}'} \subseteq Err_B^{\mathcal{A}''} \tag{26}$$

Рассмотрим различные критерии ошибочных ситуаций, получающихся в зависимости от выбора множества абстракций.

Построим абстракцию, соответствующую задаче поиска ошибочных входных данных. Для этого пронумеруем все значения множества $\Sigma = \{\vec{\sigma}_i\}$ и определим множество абстракций как $\Sigma_i = \vec{\sigma}_i$. Для данных абстракций определение ошибки (24) будет записано в виде: $\exists \vec{\sigma_i} : \mathcal{G}_B(\vec{\sigma}_i) \wedge Err_B(\vec{\sigma}_i)$.

С другой стороны, рассмотрим $\Sigma_1 = \Sigma$. Тогда определение (24) принимает вид:

$$\exists \vec{s}(\mathcal{G}_B(\vec{s})) \wedge \forall \vec{s}(\mathcal{G}_B(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{27}$$

Определение (27) утверждает, что если точка B такова, что она достижима хотя бы на одном конкретном состоянии и в ней всегда происходит ошибка, то точка B ошибочна. Данное определение является самым строгим из рассматриваемых, поэтому ему свойственен пропуск реальных ошибок.

Введем набор булевых переменных $b_j = b_j(\vec{s})$ для каждого выражения в графе развертки, имеющего логический тип. Пронумеруем все пути в графе развертки $\{l_i\}$. Для конкретного пути l_i обозначим набор переменных b_j, вычисленных в истину, как B_i^+, а набор переменных, вычисленных в ложь, как B_i^-.

$$\Sigma_i = \bigwedge_j \begin{cases} b_j, & b_j \in B_i^+ \\ \neg b_j, & b_j \in B_i^- \\ true, & \text{иначе} \end{cases} = \vec{l_i}(\vec{s})$$

Таким образом, каждая Σ_i определяет результат вычисления всех булевых выражений, которые в свою очередь определяют путь l'_i. Формула (24) для данной абстракции имеет следующий вид:

$$\exists \vec{l_i} : (\exists \vec{s} : (\mathcal{G}_B^{\vec{l_i}}(\vec{s}))) \wedge \forall \vec{s} : ((\mathcal{G}_B^{\vec{l_i}}(\vec{s}) \rightarrow Err_B(\vec{s})) \quad (28)$$

где $\mathcal{G}_B^{\vec{l_i}}(\vec{s}) = (\forall j : \vec{l}_{i,j} = \vec{l}_{i,j}(\vec{s})) \wedge \mathcal{G}_B$.

Также рассматриваются абстракции для критической точки $\Sigma_i = \mathcal{G}_B$ и значений вектора $\vec{b}$ — $\Sigma_i = \vec{b_i}$.

Заметим, что для введённых абстракций верно следующее соотношение:

$$Err^{\Sigma_i=\Sigma} \subseteq Err^{\Sigma_i=\mathcal{G}_{B_i}} \subseteq Err^{\Sigma_i=\vec{l_i}} \subseteq Err^{\Sigma_i=\vec{b_i}} \subseteq Err^{\Sigma_i=\vec{\sigma}_i} \quad (29)$$

Рассматриваются примеры ошибочных ситуаций для введённых абстракций.

Далее в данной главе рассматриваются чувствительные к путям алгоритмы поиска ошибок доступа к `null` и утечки ресурсов. Идея алгоритма поиска доступа к `null` заключается в построении для всех символьных выражений для каждой точки программы e формул $IsNull_e^{se}(\vec{b_i})$ таких, что $IsNull_e^{se}(\vec{b_i}) \rightarrow se == \texttt{null}$. Тогда, если в точке, где происходит доступ к se, формула $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b_i})$ выполнима, то выдается ошибка о возможном доступе к `null`.

Теорема 5.1. Из выполнимости формулы $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$ следует наличие ошибки по определению $Err^{\Sigma_i=\vec{b_i}}$.

Пусть $IsNull_{e,l}^{se}(\vec{b_i})$ условие такого, что в точке e для всех $\vec{\sigma} \in \Sigma_l$, $se == \texttt{null}$.

Теорема 5.2. Ошибочные ситуации, соответствующие выполнимости формулы $\mathcal{G}_e^l \wedge IsNull_{e,l}^{se}(\vec{b})$ в точках доступа к символьному выражению se для всех возможных путей l, совпадают с ситуациями по определению (28).

Используя результат теоремы 5.2, в каждой точке доступа можно построить условие $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$ такое, что его выполнимость эквивалентна (28). Далее в данной главе рассматриваются практические аспекты реализации такой проверки и вопросы межпроцедурного поиска доступа к `null`. Межпроцедурный поиск предлагается организовывать, сохраняя в резюме условия, при которых к символьным переменным осуществляется доступ.

Алгоритм поиска утечек ресурсов устроен схожим образом. Для каждого ресурса, представленного символьной переменной, вычисляется условие его утечки — $LC_s(\vec{b_i})$. В случае, если в момент выхода из метода формула $LC_{se}(\vec{b_i}) \wedge \mathcal{G}$ выполнима, выдается предупреждение об утечке ресурса.

Рассматриваются причины, по которым построенный алгоритм поиска является нестрогим:

- развертка циклов;
- замена результатов арифметических выражений в цикле на неизвестные значения;
- предположение об отсутствии псевдонимов в точках входа;
- неточность построения резюме.

В шестой главе описывается инструмент статического анализа SharpChecker, в котором реализованны предлагаемые в данной работе алгоритмы. Проводится оценка производительности и анализ результатов инструмента SharpChecker.

Среди алгоритмов поиска ошибок, реализованных в инструменте SharpChecker, есть как использующие исключительно синтаксический анализ, так и анализ потоков данных, а также наиболее мощный, описанный в данной работе, чувствительный к контексту и путям выполнения межпроцедурный анализ.

В данной главе особое внимание уделяется особенностям реализации, связанным непосредственно с языком C#, а именно построению графа вызовов и графа потока управления. Обсуждаются особенности построения резюме для внешних методов и стандартной библиотеки языка C#. Рассматривается построение по символьным выражениям запросов к SMT-решателю.

Приводится выполненная оценка эффективности оптимизации размера формул при помощи алгоритмов 3.1 и 3.2. На проекте Lucene.net суммарное время формирования и выполнения запросов к SMT-решателю в однопоточном режиме работы при использовании наивных алгоритмов построения предикатов составило 12 минут 35 секунд, а с использованием алгоритмов 3.1 и 3.2 — 8 минут 36 секунд. Таким образом, время, проведённое в решателях, сократилось на 32%.

Инструмент SharpChecker был применен к набору проектов с открытым исходным кодом. Размеры проектов варьировались от пятидесяти тысяч строк кода до полутора миллионов. Тестирование проводилось на компьютере с процессором Intel Core i7 6700 и 32 гигабайтами оперативной памяти. Результаты тестирования приведены в таблице 1. Для оценки качества инструмента, для групп дефектов NRE и утечка ресурсов были сфор-

Название проекта	Размер	Время анализа	NRE	Утечка ресурсов
Jil	50 тыс.	3 мин.	32	2
CSParser	60 тыс.	1 мин.	7	0
CSharpUtils	108 тыс.	2 мин.	21	107
Lucene.net	256 тыс.	19 мин.	33	876
SharpDevelop	1213 тыс.	23 мин.	140	182
CodeContracts	1534 тыс.	14 мин.	64	23

Таблица 1 — Результаты тестирования инструмента SharpChecker.

мированы выборки, по которым проводилась оценка отношения истинных и ложных срабатываний. Оценка показала, что для группы NRE процент истинных срабатываний находится в 95-процентном доверительном интервале 57-74%, а для группы утечка ресурсов — в интервале 72-87%.

В **заключении** сформулированы результаты диссертационной работы и приводятся направления дальнейших исследований.

Основные результаты диссертационной работы

1. Разработан алгоритм внутрипроцедурного анализа, обладающий чувствительностью к потоку и путям, при этом не теряющий информацию в точках объединения.
2. Разработан алгоритм межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающий результаты внутрипроцедурного анализа в виде резюме метода и применяющий резюме при обработке вызова данного метода.
3. Разработан критерий выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений. Для данного критерия разработаны детекторы, позволяющие обнаружить ошибки вида «доступ по нулевому указателю» и «утечка ресурсов».
4. Доказана корректность предложенного алгоритма внутрипроцедурного анализа и соответствие разработанных детекторов выбранному критерию выдачи предупреждений.

5. Разработан инструмент статического анализа, реализующий разработанные алгоритмы для анализа программ на языке C#. Для данного инструмента проведена экспериментальная оценка его характеристик на соответствие заявленным требованиям. Предложенные алгоритмы и методы позволяют проводить анализ проектов, состоящих из более миллиона строк кода, в отведённое время. При этом качество результатов анализа соответствует заявленным требованиям (более 50% истинных срабатываний для реализованных детекторов).

Публикации на тему диссертации

1. *Кошелев В. К.* Формализация определения ошибок при статическом символьном выполнении // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 5. — С. 105–118.

2. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя / Кошелев В.К., Дудина И.А., Игнатьев В.Н., Борзилов А.И. // *Труды Института системного программирования РАН.* — 2015. — Т. 27, № 5. — С. 59–86.

3. *В.К. Кошелев, В.Н. Игнатьев, А.И. Борзилов.* Инфраструктура статического анализа программ на языке C# // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 1. — С. 21–40.

4. *И.А. Дудина, В.К. Кошелев, А.Е. Бородин.* Поиск ошибок доступа к буферу в программах на языке C/C++ // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 4. — С. 149–168.

5. *Koshelev V. K., Izbyshev A. O., Dudina I. A.* Interprocedural Taint Analysis for LLVM-bitcode // *Programming and Computer Software.* — 2015. — Vol. 41, no. 4. — Pp. 237–245.

6. *Кошелев В. К.* Статический масштабируемый межпроцедурный анализ помеченных данных // Материалы Международного молодежного научного форума «ЛОМОНОСОВ-2014». — 2014.