Федеральное государственное бюджетное учреждение науки
Институт системного программирования им. В.П. Иванникова
Российской академии наук

На правах рукописи

Герасимов Александр Юрьевич

Классификация предупреждений о программных ошибках методом динамического символьного исполнения программ

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

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
канд. физ.-мат. наук
Курмангалеев Шамиль Фаимович

Москва – 2019

Оглавление

Введение

Актуальность темы исследования

Размер и сложность программного обеспечения (ПО) постоянно увеличиваются. Размер исходного текста современных программ и программных систем может достигать сотен миллионов строк. Усложнение приводит к тому, что создание качественного программного обеспечения становится фактически невозможным без применения автоматических средств проверки программ на соответствие функциональным требованиям к программному обеспечению и на отсутствие программных ошибок. Наличие ошибок в программном обеспечении в первую очередь связано с тем, что совершенствование подготовки программистов не успевает за увеличением сложности и размера программ. Несмотря на развитие методов разработки программного обеспечения и инструментальных средств поддержки процесса разработки программ, в выпускаемых программах содержатся ошибки, которые могут приводить к неправильной работе программы, несанкционированному доступу к критическим данным и выполнению злонамеренного кода. По данным портала CVE Details (www.cvedetails.com), предоставляющего статистику зарегистрированных ошибок в программах на основе информации корпорации MITRE с 1999 года, ежегодно регистрируется несколько тысяч критических ошибок в используемых программах.

С начала 2000-х годов наблюдается значительный рост индустрии разработки средств автоматического обнаружения ошибок в программах. Среди них выделяют методы статического и динамического анализа программ. Методы статического анализа программ позволяют обнаруживать ошибки в программном обеспечении без запуска программ на исполнение. Динамические методы анализа программ позволяют обнаружить ошибки в программах в процессе (online) или по результатам (offline, postmortem) исполнения программы. У каждого из методов обнаружения ошибок в программах есть как преимущества, так и недостатки.

Методы статического анализа программ позволяют проанализировать программу целиком, включая части программ, исполняющиеся крайне редко, обладают высокой производительностью анализа, производя полный анализ программ размером в несколько миллионов строк исходного текста, хорошо масштабируются. Но при этом выявление свойств программы методами статического анализа является в общем случае алгоритмически неразрешимой задачей. В связи с этим в процессе создания инструментальных средств статического анализа приходится идти на компромиссы между количеством истинных предупреждений об ошибках и количеством необнаруженных ошибок для удовлетворения требований масштабируемости и производительности анализа.

Оценка качества инструментов, предназначенных для обнаружения программных ошибок, производится путем проверки соотношения *количества истинных предупреждений* об ошибке (принятая гипотеза о наличии ошибки верна), *количества ложных предупреждений* об ошибке (принятая гипотеза о наличии ошибки неверна: ложноположительное предупреждение, или ошибка первого рода) и *количества ненайденных ошибок* (принятая гипотеза об отсутствии ошибки неверна: ложноотрицательный результат анализа, или ошибка второго рода). Точность анализа выражается соотношением количества истинных предупреждений об ошибках и общим количеством предупреждений об ошибках в программе. Уровень ошибок первого рода при анализе таких программных систем, как ОС Android, может достигать 40% от общего количества предупреждений об ошибках, что в абсолютных величинах составляет тысячи предупреждений. Квалифицированному специалисту на анализ одного предупреждения об ошибке может потребоваться значительное время, что, приводит к необоснованным временным затратам и нежелательным финансовым расходам, в связи с большой долей ошибок первого рода в результатах анализатора. Возникает задача автоматической классификации предупреждений об ошибках в программе с целью предварительного

разбиения их на классы: заведомо истинные предупреждения, заведомо ложные предупреждения и предупреждения, истинность или ложность которых в автоматическом режиме гарантированно установить не удается.

Для решения задачи классификации программных ошибок, найденных методами статического анализа программ, предлагается применять методы динамического анализа программ. Методы динамического анализа программ позволяют автоматически обнаруживать ошибки в программах и слабо подвержены влиянию проблемы ошибок первого рода, так как позволяют вычислять наборы внешних (входных) данных программы, на которых можно продемонстрировать ошибку в программе и использовать их для отладки программы.

В данной работе предлагается использовать метод динамического символьного исполнения программ для вычисления внешних данных программы, который заключается в представлении потока данных программы в виде набора символьных (свободных) переменных, значения которых зависят от внешних данных программы и операций над ними. Этот метод позволяет в процессе анализа потока данных программы вычислять значения символьных переменных и генерировать новые внешние данные для исполнения программы по альтернативному пути или для проявления ошибки при исполнении потенциально опасных операций.

Несмотря на отсутствие ошибок первого рода при применении метода динамического символьного исполнения программ для решения задачи вычисления наборов внешних данных программы, данный метод отличается низкой производительностью в связи с тем, что для его реализации требуется «тяжеловесная» инструментация программы с целью сбора трассы исполнения программы и решение задачи определения вычислимости формул в теориях для генерации внешних данных программы. Недостатком данного метода также является экспоненциальный рост количества путей в программе от количества условных переходов, зависящих от потока символьных данных

программы. Поэтому исчерпывающий анализ программ за ограниченное время методами динамического символьного исполнения затруднителен.

В связи с вышеизложенным становится актуальной задача комбинирования методов статического и динамического анализа программ для классификации предупреждений об ошибках в программах, найденных методами статического анализа программ.

Цель и задачи диссертационной работы. Цель диссертационной работы – повысить точность анализа программ путем комбинирования методов статического анализа программ и динамического символьного исполнения программ с целью классификации предупреждений об ошибках в программах. Разработанные алгоритмы должны обеспечить повышение точности результатов статического анализа программ.

Для достижения поставленной цели сформулированы и решены следующие задачи:

1. Разработка модели обнаружения ошибок в программе в процессе символьного исполнения программы.
2. Разработка алгоритмов комбинированного анализа программ, объединяющего статический анализ и динамическое символьное исполнение программ на основе направленного динамического символьного исполнения программ, использующего предупреждение об ошибке, найденной методами статического анализа программ, для вычисления внешних данных программы, приводящих к проявлению ошибки в процессе исполнения программы.
3. Разработка метода классификации предупреждений о программных ошибках при помощи разработанных модели и алгоритмов.

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

1. Разработана модель обнаружения ошибок в программе в процессе символьного исполнения программы.

2. Разработаны алгоритмы комбинированного анализа программ на основе направленного динамического символьного исполнения программ, использующего предупреждение об ошибке, найденной методами статического анализа исходного кода программ, для вычисления внешних данных программы, приводящих к проявлению программной ошибки в процессе исполнения программы.
3. Показана применимость разработанных алгоритмов для решения задачи классификации предупреждений о программных ошибках, найденных методом статического анализа программ.

Теоретическая и практическая значимость. Разработаны алгоритмы направленного динамического символьного исполнения программы. Предложенные алгоритмы реализованы на основе инструментов статического анализа исходного кода программ Svace и инструмента динамического анализа программ Anxiety, разрабатываемых в Институте системного программирования им. В.П. Иванникова РАН. Показана применимость предложенных алгоритмов для решения задачи классификации предупреждений о программных ошибках, найденных методами статического анализа программ. Результаты исследования, изложенные в диссертации, могут быть использованы для дальнейших исследований в области анализа программ, создания учебных курсов, а также при создании промышленных анализаторов программ.

Методология и методы исследования. Результаты диссертации были получены с использованием методов и моделей, применяемых при проведении динамического символьного исполнения программ. Математическую основу данной работы составляют теория графов, теория множеств, теория алгоритмов, математическая логика.

Положения, выносимые на защиту:

- модель обнаружения ошибок в программе в процессе символьного исполнения программы;

- алгоритмы комбинированного анализа программ на основе направленного динамического символьного исполнения программ, использующего предупреждение об ошибке, полученное методами статического анализа исходного кода программ, для вычисления внешних данных программы, приводящих к проявлению программной ошибки в процессе исполнения программы;
- метод классификации предупреждений об ошибках в программе, основанный на применении разработанных модели и алгоритмов комбинированного анализа программ.

Апробация результатов. Основные результаты диссертационного докладывались на следующих конференциях:

1. Открытая конференция ИСП РАН им. В.П. Иванникова. (Россия, Москва, 2016).
2. 11th International Conference on Computer Science and Information Technologies (Armenia, Yerevan, 2017)
3. 60-я научная конференция МФТИ (Россия, Долгопрудный, 2017)

Публикации. Материалы диссертации опубликованы в 9 печатных работах, из них 6 опубликованы в изданиях, входящих в перечень рецензируемых научных изданий ВАК при Минобрнауки РФ [1-3, 6, 8-9], 4 статьи опубликованы в сборниках трудов конференций [4-7]. Пять из девяти статей [4, 6-9] опубликованы в изданиях, индексируемых в Scopus. В совместных работах [1, 3-4, 9] личный вклад автора заключается в описании метода определения достижимости программных дефектов. В работе [6] личный вклад автора заключается в написании введения и описании реализации инструмента Anxiety. Получены 5 свидетельств о государственной регистрации программы для ЭВМ.

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

Глава 1. Программные ошибки и известные подходы к их обнаружению

1.1. Программные ошибки

1.1.1. Определение программной ошибки

О возможности программной ошибки впервые было упомянуто в 1842 году Адой Августой графиней Лавлейс в труде «Очерк об аналитической машине, представленной Чарльзом Бэббиджем»: «... an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.”(с англ.: … анализирующий процесс должен быть одинаково произведен в соответствии с предоставленными Аналитической Машине необходимыми управляющими данными, и это при сём также может быть источником возможной ошибки. Правда в том, что механизм безошибочен в своих процессах, но карты (с управляющими данными – А.Г.) могут давать ошибочные команды» [1].

ГОСТ 56939-2016 «Защита информации. Разработка безопасного программного обеспечения. Общие требования.» [2] определяет следующие понятия:

- «недостаток программы: Любая ошибка, допущенная в ходе проектирования или реализации программы, которая в случае её не исправления может являться причиной уязвимости программы»;
- «уязвимость программы: Недостаток программы, который может быть использован для реализации угроз безопасности информации».

Из ГОСТа очевидно, что понятие *недостатка программы* определяется через понятия ошибок проектирования или реализации программы. При этом в рамках ГОСТа 56939-2016 даётся понятие процесса *экспертизы исходного кода программы*, который заключается в выявлении недостатков программы (потенциально уязвимых конструкций) в исходном коде программы.

В «Стандартной классификации программных аномалий» IEEE 1044-2009 [3] даётся более развёрнутая классификация различных программных аномалий (отклонений от нормы):

- дефект (defect) – несовершенство или недостаток в работе продукта, при котором продукт не удовлетворяет требованиям или спецификациям и нуждается в исправлении или замене;
- ошибка (error) – действие человека, которое привело к некорректному результату;
- сбой (failure) – прекращение возможности выполнения продуктом требуемой функции или невозможность исполнять её в ранее определённых ограничениях;
- повреждение (fault) – сообщение об ошибке в программе.

В 1993 году Национальный институт стандартов и технологий США (National Institute of Standards and Technology) выпустил руководство «Анализ программных ошибок» [4], в котором даются следующие определения:

- аномалия (anomaly) – любое состояние, которое отличается от ожидаемого;
- дефект (defect) – любое несоответствие для использования или несоответствие спецификации;
- ошибка (error):
    - отличие между вычисленным, наблюдаемым или измеренным значением и истинным, указанным или теоретически корректным значением или состоянием;
    - некорректный шаг, процесс или определение данных;
    - некорректный результат;
    - действие человека, приведшее к некорректному результату;
- повреждение (fault) – некорректный шаг, процесс или определение данных в компьютерной программе (см. также *ошибка*);
- сбой (failure) – различие между внешним результатом исполнения программы и требованием к программному продукту.

При этом в руководстве делается оговорка, что трактовка терминов базируется на серии работ [5-9] и на самом деле различия между понятиями «дефект», «ошибка» и «сбой» нет, так как трактуются данные термины в сообществе по-разному (даже в стандартах, использующих данные термины). Например, в работе, посвященной методам статического анализа исходного кода [10], также утверждается, что единого мнения о том, что следует называть ошибкой, нет, и приводятся такие определения:

- ошибка – некоторое место в исходном коде программы, из-за которого на определённых внешних данных программа может аварийно завершиться либо вывести некорректные выходные данные;
- дефект – место, указывающее на недостаток исходного кода, который необязательно приведёт к некорректной работе программы, но может ухудшить её эксплуатационные характеристики (например, утечка памяти);
- уязвимость – ошибка в программе, которая может быть использована атакующим для намеренного краха программы, выполнения произвольного кода, утечки конфиденциальных данных либо для других нарушений безопасности системы.

В связи с тем, что единой трактовки таких терминов, как «дефект», «ошибка», «сбой», нет, то в данной работе термины «ошибка» и «дефект» будут использоваться как синонимы термина «недостаток программы», а термин «сбой» будет использоваться для обозначения состояния аварийного завершения работы программы. Термин «уязвимость программы» будет использоваться в значении получения несанкционированного доступа к данным, обрабатываемым программой, или в значении несанкционированного исполнения программного кода.

1.1.2. Статистика обнаружения уязвимостей в программах

По данным интернет-портала CVE Details [11] с 1999 по 2017 годы зарегистрировано более 94000 критических ошибок в программах, выпущенных на рынок. На *рис. 1* изображена диаграмма зарегистрированных

ошибок по годам, сформированная на основе данных портала CVE Details, предоставляющем статистику базы CVE (англ. Common Vulnerabilities and Exposures) – базе общих уязвимостей информационной безопасности. Исходя из данных портала, начиная с 2005 года, ежегодно регистрируется не менее 4000 новых уязвимостей и тенденция к снижению количества регистрируемых ошибок в год не наблюдается. В 2017 году было зарегистрировано 14712 уязвимостей, что более чем в два раза превышает среднегодовое количество уязвимостей, регистрируемых ежегодно за предшествующие десять лет.

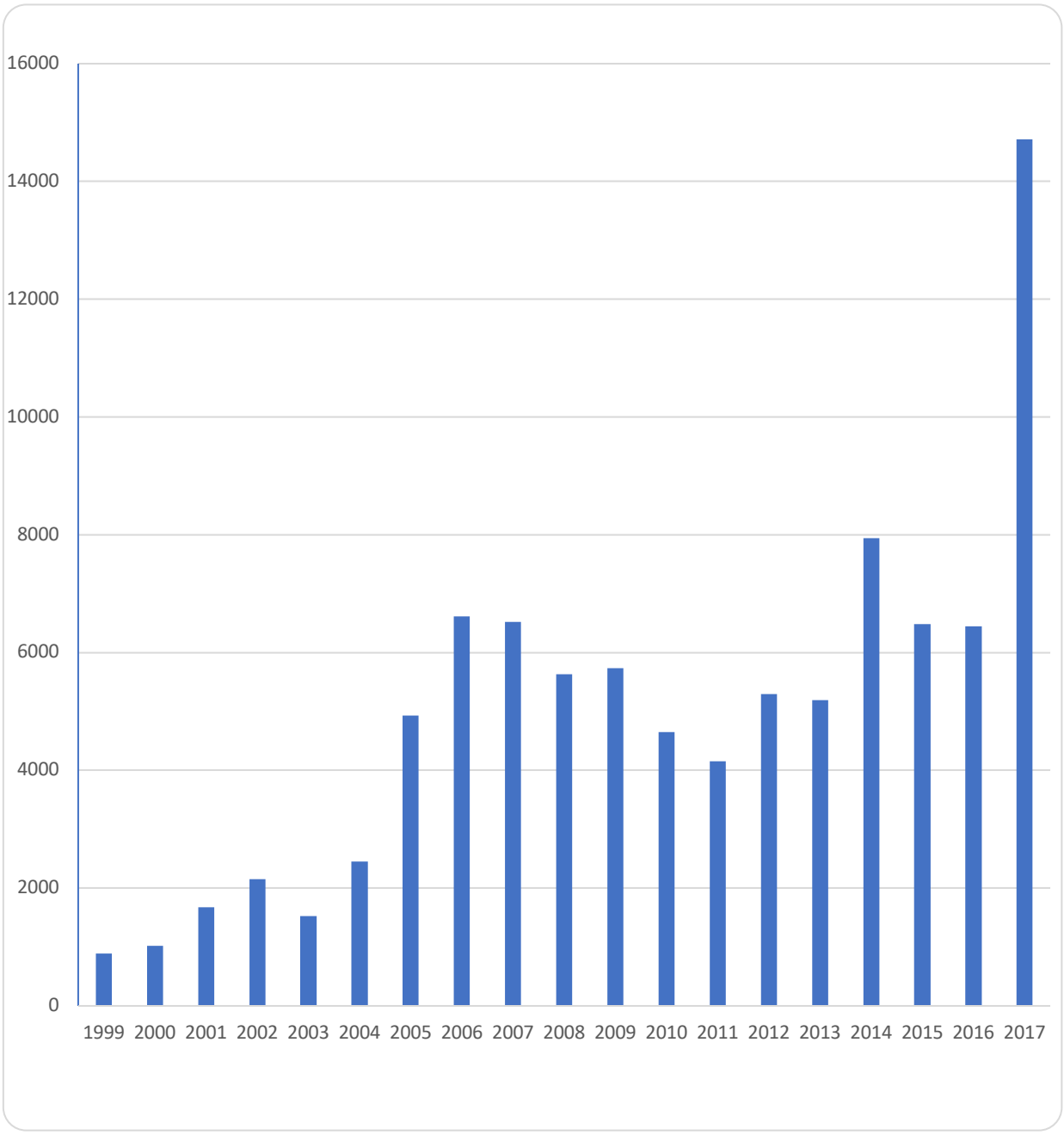


Рисунок 1. Количество зарегистрированных ошибок в программах

Статистика по типам вредоносного воздействия на программу (рис. 2) показывает, что наиболее часто регистрируемыми уязвимостями являются:

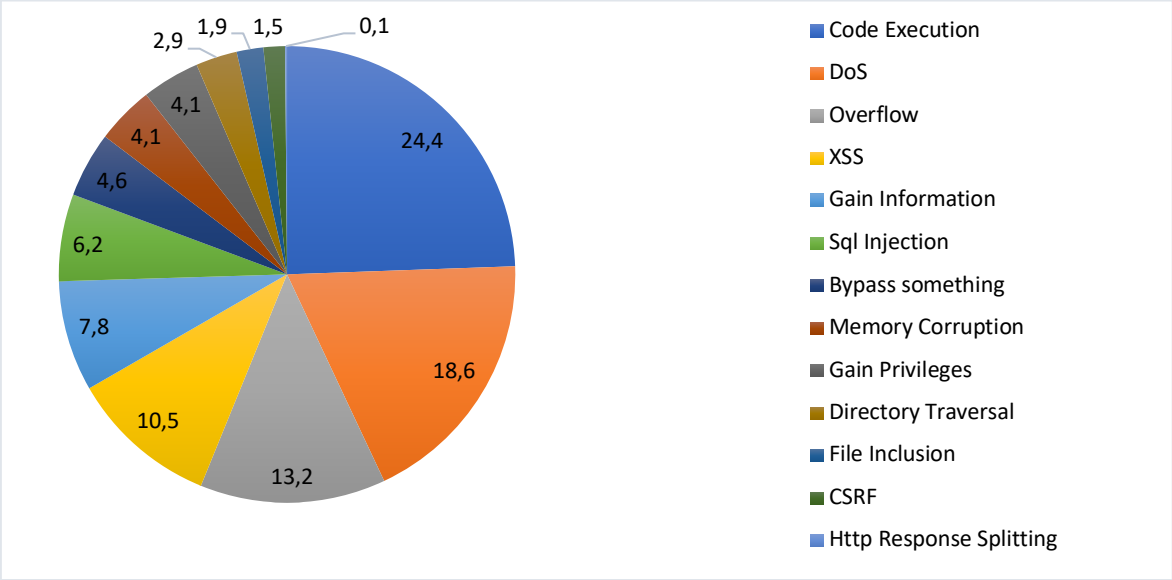


Рисунок 2. Статистика по типам зарегистрированных ошибок

- исполнение злонамеренного кода (code execution);
- аварийное завершение программы (DoS – Denial of Service);
- переполнение (overflow);
- исполнение злонамеренного кода на стороне клиента (XSS);
- получение неавторизованного доступа к данным (gain information);
- внедрение в запрос SQL (SQL Injection).

1.1.3. Классификация программных ошибок

На текущий момент существуют различные способы классификации программных ошибок. В «Стандартной классификации программных аномалий» Института инженеров электротехники и электроники (IEEE) [12] описывается общий метод классификации программных ошибок, который среди прочего требует определения целей классификации и стандарта определения, какое из поведений программы считается ошибочным. Во всеобщем перечне недостатков в программном обеспечении, созданном и поддерживаемом корпорацией MITRE [13], содержится 714 описаний возможных недостатков в программном обеспечении (по данным на 16 января

2018 года). На портале cwe.mitre.org предоставляется возможность просмотра всех зарегистрированных типов недостатков в программном обеспечении в виде различных классификаций: шаблоны программных ошибок (англ. Software Fault Patterns) [14], первые десять наиболее критичных рисков для веб-приложений Открытого проекта по безопасности веб-приложений (англ. Open Web Application Security Project) [15], Семь разрушительных царств (англ. Seven Pernicious Kingdoms) [16] и др.

Всё множество программных ошибок, приводящих к некорректному поведению программы во время исполнения, можно разделить на следующие классы по видам вредоносного воздействия:

- ошибки, приводящие к порче пользовательских данных в процессе обработки: целочисленное переполнение, порча данных в оперативной памяти, обращение к неинициализированному блоку памяти, обращение к памяти по неинициализированному или висящему указателю (англ. – dangling pointer), фальсификация данных (англ. - request forgery) и др.;
- ошибки, приводящие к неавторизованному доступу к пользовательским данным: получение неавторизованного доступа к базе данных, получение неавторизованного доступа к информации в оперативной или постоянной памяти вычислительного устройства, получение повышенного уровня привилегий доступа к данным и др.;
- ошибки, приводящие к исчерпанию системных ресурсов, таких как память на куче, файлы, сокеты и др.;
- ошибки, приводящие к аварийному завершению исполнения программы: доступ к области памяти, не принадлежащей программе, деление на ноль и др.;
- ошибки, приводящие к исполнению злонамеренного кода: перехват потока управления злонамеренным кодом, исполнение злонамеренного кода на стороне клиента, внедрение в исполнение команды в командной строке и др.

1.1.4. Причины появления программных ошибок

В работах [17] и [18] приведен краткий обзор причин появления ошибок в программах, среди которых можно выделить следующие:

1. Влияние квалификации команды разработчиков программы:
   - плохой или недостаточный уровень проработки архитектуры программы: плохо спроектированная архитектура программы может привести к внесению ошибок в процессе развития программной системы;
   - пренебрежительное отношение программистов к результату своей работы: вместо тщательного планирования изменений в программе применяется метод быстрого кодирования с последующим исправлением ошибок, найденных путём тестирования программы.
2. Уделяется мало внимания различным уровням тестирования программы в процессе разработки:
   - недостаточное количество и качество автоматических тестов на компоненты (модульных тестов), что приводит к позднему обнаружению нарушения контракта использования модуля и функции;
   - недостаточность или отсутствие интеграционных тестов, что приводит к интеграционным ошибкам в процессе изменения кода компонентов, в том числе разделяемых компонентов операционной системы, не учитывающих контекст их использования;
   - недостаточный уровень тестирования программы на устойчивость при обработке некорректных внешних данных.
3. Влияние изменений, вносимых в среду исполнения программы. Многие современные программы используют разделяемые библиотеки, как правило, входящие в комплект поставки операционных систем или используемые совместно различными программами. Изменение в коде разделяемых библиотек при недостатке интеграционного тестирования может привести к неправильной работе программы.

Всё множество причин появления ошибок в программах в итоге сводятся к одной – недостаточной квалификации разработчиков программного обеспечения. В связи с этим становится очевидной необходимость создания автоматических средств обнаружения ошибок в программном обеспечении, которые позволят обнаруживать ошибки на различных этапах жизненного цикла ПО.

1.2. Обзор методов обнаружения программных ошибок

1.2.1. Программные ошибки, как мера качества программного обеспечения

Свойство наличия или отсутствия ошибок в программе может быть использовано как показатель качества программы. Несмотря на то, что понятие качества предполагает бинарную классификацию продуктов на качественные и некачественные, и, применительно к программному обеспечению, наличие или отсутствие ошибок в программе может являться критерием оценки качества ПО, исторически оценка качества программ производится либо в виде оценки числа ошибок на тысячу строк исходного текста программы – метрика плотности ошибок или, как среднее время наработки программы на отказ [19]. При этом автор работы приводит критику различных способов оценки качества программ.

На ранних этапах развития вычислительной техники применялся процесс «Кодирование и исправление» [20], который заключался в кодировании программы с последующим этапом избавления программы от ошибок (англ. debugging) программистом, написавшим программу. В итоге применение этого метода привело к тому, что в программном обеспечении спутника Mariner 1 осталась ошибка, которая привела к потере спутника. После этого случая было принято решение применять инспекцию исходного кода несколькими программистами при разработке программ для управлении военно-воздушных сил США [21]. Позже стали применяться методы тестирования программ для подтверждения того, что программа делает именно то, что требуется, и не делает ничего, для чего она не предназначена

[22]. При этом тесты составлялись таким образом, чтобы проверить программу на всех возможных путях исполнения со всеми возможными наборами внешних данных, то есть произвести *исчерпывающее* тестирование. Но метод исчерпывающего тестирования был признан практически неприменимым [23] в связи с тем, что в сколь-нибудь сложной программе количество путей исполнения и множество значений внешних данных слишком велико. Э. В. Дейкстра в лекции «О надёжности программ» утверждает, что тесты могут показать наличие ошибок в программе, но не могут доказать их отсутствие [24]. В выводах статьи [25] утверждается, что методы верификации программ недостаточно эффективны для гарантированного обеспечения высокого качества программ, и приводится статистика, согласно которой в процессе прохождения инспекции исходного текста в программе обнаруживается от 25% до 50% ошибок, а на этапе тестирования программы – от 30% до 60%. При этом в час обнаруживается от 1 до 2,5 ошибок.

Инспекция кода и тестирование относятся к двум классам методов обнаружения ошибок в программах, отличающихся способом обнаружения ошибок в программе: статические методы – без запуска программы на исполнение, динамические методы – по результатам или в процессе исполнения программы.

1.2.2. Статические методы

В статье, посвященной истории создания языка Си [26], Деннис Ритчи (Dennis M. Ritchie) отмечает: несмотря на то, что в первой редакции книги «Язык программирования Си» было указано большинство правил, которые привели систему типов языка Си к его современной форме, многие программы были написаны в старом, более свободном стиле, и компиляторы это позволяли. Для того чтобы обратить внимание разработчиков программ на официальные правила языка, обнаруживать разрешенные, но подозрительные конструкции и помочь найти несоответствие интерфейсов, не обнаруживаемое простыми механизмами в процессе раздельной компиляции, Стив Джонсон (Steve Johnson) адаптировал свой компилятор *pcc* [27] для создания

инструмента *Lint* [28] (англ. lint – выщипывать пушинки), который сканировал файлы исходного кода программы и отмечал сомнительные конструкции. Особенностью программы *Lint* являлась возможность сравнивать соответствие и находить отсутствие противоречий во всей программе путём сканирования множества исходных файлов и сравнения типов аргументов функций в месте вызова с типами аргументов функции, указанных в определении.

Согласно утверждениям технического отчёта «Следующее поколение статического анализа» [29], программу *Lint* можно назвать первым инструментом *автоматического* статического анализа кода, но на самом деле *Lint* не разрабатывался с целью обнаружения дефектов, которые приводят к ошибкам времени исполнения. Скорее, целью создания инструмента было обнаружение подозрительных и непортируемых конструкций в коде для помощи разработчикам в создании более согласованного кода. Под «подозрительными конструкциями» подразумевается технически корректный исходный текст программы с точки зрения спецификации языка, который может привести к такому выполнению программы, которое не подразумевалось разработчиком. Проблема заключалась в том, что такой подозрительный код мог исполняться и часто исполнялся корректно. Из-за технологических ограничений инструмента *Lint*, который был построен на основе синтаксического анализатора кода, уровень шума (ложноположительных предупреждений об ошибках) был экстремально высок, часто превышая соотношение шума и предупреждений о реальных программных ошибках 10:1.

С начала 1990-х годов наблюдается развитие инструментов автоматического анализа исходного кода программ. Инструменты, аналогичные *lint,* такие как *Blast* [30], *MAGIC* [31], *RATS* [32], *FlowFinder* [33], *ITS4* [34], *Splint* [35] и *UNO* [36], используют синтаксический анализ исходного текста программ для поиска ограниченного набора программных ошибок, что плохо отражается на качестве результатов анализа согласно

отчёту о сравнении инструментов с использованием *Juliet Test Suite* (пер. с англ.: тестовый набор Джульетта) проекта *SAMATE* (англ. Software Assurance Metrics and Tool Evaluation – оценка показателей программного обеспечения и инструментов) [37]. Среди них выделяются специализированные инструменты, нацеленные на поиск ошибок определённого вида, таких как состояние гонки при выполнении файловых операций (TOCTOU или TOCTTOU: от англ. – time-of-check to time-of-use, от момента проверки к моменту использования) [38] или ошибки переполнения буфера в памяти *BOON* [39], *Archer* [40], анализа потоков помеченных (англ. tainted*)* данных для обнаружения состояния гонки, взаимной блокировки или ошибочной последовательности использования потоковых функций *CQUAL* [41], ошибок утечки памяти, проблем блокировки и разыменования нулевых указателей *SATURN* [42]. В то же время часть из указанных инструментов не являются автоматическими средствами анализа, такие как *MAGIC*, *Blast*, *Splint*, *CQUAL*, которые в процессе работы полагаются на аннотации исходного кода, созданные разработчиками программ вручную.

В середине 2000-х стали появляться инструменты статического анализа программ второго поколения: *Coverity Prevent* [29], *Klocwork inSight* [43], *Svace* [44] и др. [45]. Основной особенностью данных инструментов было смещение фокуса с поиска отдельных подозрительных конструкций в исходном коде программ на создание сред анализа исходного текста программ, поддерживающих создание широкого спектра детекторов программных ошибок, что потребовало комбинации анализа представления программы в виде дерева абстрактного синтаксиса, нагруженного семантической информацией, изощрённого анализа путей исполнения с межпроцедурным (контекстно зависимым) анализом с целью определения поведения программ при передаче управления от одной функции к другой внутри программной системы [46]. В связи с постоянным поиском компромисса между точностью и масштабируемостью анализа инструменты второго поколения могли находить ограниченный набор дефектов с высокой точностью, но либо анализ

не масштабировался на миллионы строк исходного кода, либо, работая сравнительно быстро, обладал малой точностью аналогично инструменту *Lint*, привнося ложноположительные предупреждения об ошибках в результаты свой работы [46, 47]. Если для инструментов первого поколения проблема шума в результатах работы препятствовала их внедрению в индустрии, то инструменты второго поколения сдвинули развитие инструментов статического анализа программ с технологической мёртвой точки, позволяя обнаруживать осмысленные дефекты в программах. Несмотря на это, разработчики программного обеспечения до сих пор ожидают более высокой точности результатов анализа [47, 48].

В связи с развитием алгоритмов символьного исполнения и появлением инструментов для решения формул в теориях (англ. solver) [49] в последнее время стали появляться инструменты статического анализа третьего поколения, которые объединяют классические методы анализа путей исполнения программы и символьное исполнение программы в комбинации с применением решателей. Согласно исследованию Чельфа и Чу [29] применение подобной комбинации позволило получить дальнейшие технологические преимущества для статического анализа и снизить высокие уровни предупреждений ошибок первого рода (ложноположительных предупреждений) на 15%.

В обзоре применения методов статического анализа при разработке программ [50] даётся деление инструментов с точки зрения поддерживаемых языков программирования: для платформы Microsoft .Net – *StyleCop* [51] и *FxCop* [52], для программ на языке Java – *Java FindBugs* [53], *Jlint* [54], *ESC/Java* [55] и поддерживающие несколько языков программирования – *Klocwork inSight*, *Coverity Prevent*, *Hammurapi*, *RATS*, *Understand*.

В итоге инструменты статического анализа кода можно классифицировать по следующим признакам:

- способ анализа: автоматический или полуавтоматический;

- язык программирования: один, несколько, анализ промежуточного представления, анализ исполняемого кода;
- типы обнаруживаемых программных ошибок: фиксированный список, возможность создания дополнительных детекторов ошибок;
- модель кода программы: текстовый поиск, синтаксический анализ, анализ дерева абстрактного синтаксиса, анализ путей исполнения, межпроцедурный анализ.

1.2.3. Динамические методы

Фаззинг

Первые официально опубликованные результаты исследований, посвящённые тестированию поведения программ при использовании псевдослучайных внешних данных, проводились в начале 80-х годов XX века в Юго-Западном исследовательском институте в Сан-Антонио и в Техасском университете в Далласе [56, 57]. При этом, по воспоминаниям Джеральда Вайнберга, работавшего в НАСА на проекте «Меркурий» по запуску первого человека в космос, в 50-х годах XX века тестирование разрабатываемого программного обеспечения проводилось на пачках перфокарт, вынутых из мусорных корзин или случайно перемешанных [58]. В 1970 году была представлена работа [59], в которой описывается метод псевдослучайной генерации по заданной грамматике синтаксически корректных программ для тестирования компиляторных фронтендов. В 1983 году Стив Кэппс разработал инструмент *The Monkey* (англ. monkey – обезьяна) для тестирования программы *MacPaint*. Через специальный программный интерфейс инструмент подключался к программе и с высокой скоростью, как будто компьютером управляла злая обезьяна, генерировал псевдослучайные последовательности нажатий клавиш на клавиатуре и воздействий на манипулятор «мышь» [60, с. 22]. Фактически *The Monkey* является первым известным примером промышленного инструмента тестирования программы на устойчивость к ошибочным (англ. malformed – плохо сформированный, уродливый) внешним данным программы.

Термин «фазз» (англ. fuzz – неточный, расплывчатый) впервые упомянут в названии исследовательского проекта «Operating System Utility Program Reliability – The Fuzz Generator» (пер. с англ. Программа оценки надежности утилит операционной системы – Фазз-генератор), который проводился в Висконсинском университете в Мадисоне с 1988 года под руководством профессора Барта Миллера [61]. В статье [62], посвящённой описанию предложенного метода и результатам исследования, рассказывается, что на идею исследования техники случайного отрицательного тестирования программ (англ. negative testing – проверка программы на способность обрабатывать некорректные внешние данные) натолкнула ситуация, когда один из авторов подключился по телефонной линии с использованием модема к удалённой компьютерной системе в ночь, когда был ураган с дождём. Дождь сильно влиял на качество связи и приводил к тому, что на вход утилитам командной строки подавались неправильные аргументы в связи со случайно внедрёнными неправильными символами. Подобный шум из-за влияния дождя на линию был ожидаем, но совершенно неожиданным наблюдением стало то, что получение на вход некорректных аргументов командной строки приводило к аварийному завершению утилит операционной системы. Данное наблюдение натолкнуло авторов исследования на идею автоматического тестирования программ путём генерации псевдослучайных внешних данных, или фаззинга. Авторы исследования описывают свой метод не как замену тестированию или формальной верификации, а как недорогой механизм обнаружения ошибок с целью увеличения надёжности системы в целом. При этом использовался очень грубый показатель корректности: программа считалась повреждённой, если в процессе её исполнения происходило аварийное завершение работы программы или обнаруживалось её зависание. Если представить программу как сложный конечный автомат, то предложенный механизм можно описать как случайный обход пространства состояний этого конечного автомата с целью поиска неопределённых состояний [62, с. 32].

В 1995 году Миллер и соавторы повторили исследование [63], но расширили метод фаззинга с целью поддержки сетевых программ и программ с пользовательским интерфейсом. Авторы отметили:

- преобладание ошибок в утилитах операционных систем, что оказалось неожиданным в связи с технически лёгкой доступностью техники фаззинга для программ этого класса;
- высокое качество сетевых программ и сервера событий графических приложений X-Window;
- низкое качество клиентских приложений X-Window.

Предложенный метод псевдослучайного тестирования программ был расширен и привёл к созданию специализированных сред фаззинг-тестирования программ *SPIKE* [64] и *PROTOS* [65]. Вдохновлённые исследованием Висконсинского университета и идеей синтаксического тестирования Бориса Бейцера [66], группа безопасного программирования университета Оулу (Oulu University Secure Programming Group) в 1999 году запустила проект *PROTOS*, который должен был помочь поставщикам программного обеспечения и оборудования в обнаружении ошибок. Идея заключалась в том, чтобы автоматически сгенерировать тестовые наборы, демонстрирующие ошибки в реализации различных протоколов, и предоставить их сначала поставщикам оборудования и программных систем, а после того как поставщики исправят найденные ошибки – сообществу. В результате были разработаны тестовые наборы для различных сетевых протоколов: WAP-WSP, HTTP, LDAP, SNMP, SIP, DNS и др. Реализация метода «Mini-Simulation», на котором основана работа генератора фазз-тестов, подробно изложена в работе Каксонена [67]. В основе метода «Mini-Simulation» лежит использование спецификации протоколов в виде грамматики и правил, описывающих взаимодействие с внешними сущностями (коммуникационные правила), и конструкции, которые сложно описать в виде грамматики, такие как длины и контрольные суммы (семантические правила). Далее производится симуляция исполнения модели протокола с мутациями

спецификации, которые позволяют внедрить аномальные элементы (аномалии) с целью нахождения ошибок в реализации протокола. Метод, предложенный в проекте *PROTOS*, впоследствии получил название – *генерация внешних данных на основе грамматик* (англ. grammar-based input generation), описывающих формат внешних данных.

В 2002 году Дэйв Айтел, вдохновлённый результатами проекта *PROTOS*, выпускает фаззер под названием *SPIKE* с открытым исходным кодом под лицензией GPL. Среда фазз-тестирования *SPIKE* предоставила возможности фаззинга протоколов, описывая их реализацию на основе блоков переменной длины [68], каждый из которых представляет собой описание длины и последовательности байт. Внешние данные представлялись как структурированный набор блоков с различной гранулярностью, что позволяло снизить трудоёмкость построения фаззеров путём создания новых алгоритмов генерации данных поверх имеющихся. Среда *SPIKE* позволяла не только генерировать псевдослучайные тестовые данные, но и предоставляла библиотеку сгенерированных примеров, которые с высокой вероятностью приводили к аварийному завершению плохо написанных программ. К тому же предоставлялись реализации функций, способных генерировать последовательности данных для общеизвестных протоколов и форматов. Среди прочих предоставлялись реализации таких коммуникационных технологий для удалённых вызовов процедур, как Sun RPC и Microsoft RPC. *SPIKE* стал одним из первых промышленных фаззеров, опубликованных для всеобщего использования [69, с. 24]. Метод, предложенный в фаззере *SPIKE*, позволил разбить (произвести декомпозицию) задачу генерации внешних данных для программ на блоки и позволил строить новые фаззеры на основе готовых блоков, представляющих комбинацию простых генераторов внешних данных.

Почти одновременно с *SPIKE* Дэйв Айтел выпустил фаззер *sharefuzz*. Отличительной особенностью этого фаззера являлась возможность использования переменных окружения операционной системы как источника

внешних данных программы путём переопределения библиотечных функций получения значений переменных окружения операционной системы, возвращавших значения переменных окружения, с целью обнаружения уязвимостей, связанных с переполнением буфера. Таким образом сформировалось деление инструментов фаззинга по виду транспортной системы передачи сгенерированных внешних данных.

Развитие методов фаззинга программ привело к созданию специфических инструментов, таких как *mangleme* [70] Михала Залевски (Michał Zalewski), предназначенный для фаззинга веб-браузеров, или *CSSDIE* [71], созданный группой исследователей для тестирования каскадных стилевых таблиц (англ. Cascaded Style Sheets, CSS).

На странице во всемирной сети World-wide Web, посвященной инструменту *AFL* (англ. – American Fuzzy Lop)[72], Михаль Залевски описывает историю реализации инструмента. В конце 2000-х Залевски обратил внимание на убедительные результаты, с точки зрения найденных уязвимостей, предоставленные Трэвисом Орманди, который применил технику статической инструментации программы для контроля покрытия базовых блоков программы при помощи инструмента *gcov* [73] с последующим выбором оптимальных тестовых наборов как «затравки» («зерна», англ. seed) для фаззинга. Залевски создал инструмент под названием *Bunny-the-fuzzer* [74], алгоритм которого пытался установить отношение между различными битами потока внешних данных программы и её внутренним состоянием с целью получения каких-то дополнительных преимуществ от этого. Но оказалось, что идея сложна в реализации и не приводит к хорошим практическим результатам. Орманди в итоге пришел к фаззингу методом «серого ящика», совмещающему методы фаззинга и динамического символьного исполнения [75]. К подобным же выводам пришли Габриэль Компана (Gabriel Campana) с инструментом *Fuzzgrind* [76] и исследовательская группа в Microsoft Research с инструментом *SAGE* (англ. Scalable, Automated, Guided Execution – масштабируемое автоматизированное

направленное исполнение) [77]. Но, несмотря на успехи в исследовании методов динамического символьного исполнения, Залевски отмечает их крайне низкую эффективность при поиске ошибок и уязвимостей в реальных программах [78]. Залевски при разработке фаззера *AFL* предложил идею объединения методик выбора стратегии генерации внешних данных, которая основывается на контроле покрытия базовых блоков программы и генетических алгоритмах генерации внешних данных для программы таким образом, чтобы в каждом следующем поколении сгенерированных внешних данных достигнутое покрытие по базовым блокам не уменьшалось, а увеличивалось. За данным методом закрепился термин *эволюционный фаззинг*. Идея метода заключается в том, что каждое новое поколение внешних данных оценивается функцией приспособленности (англ. – fitness function) к желаемому результату фаззинга на основе оценки прироста покрытия базовых блоков программы.

В диссертационной работе De Mott [79], посвящённой методам автоматического обнаружения ошибок и анализа программ, приводится анализ методов проведения фаззинга программ по трем компонентам: способу влияния на поток внешних данных, методу оценки исполнения программы и способу влияния на процесс фаззинга. Даётся определение фаззинга как техники тестирования программ в процессе исполнения путем передачи на вход полунеправильных внешних данных с отслеживанием состояния программы, как правило, при помощи инструментов отладки с целью обнаружения исключительных ситуаций, таких как нарушение доступа к памяти.

Мутационный фаззинг использует в качестве начальных значений корректные внешние данные, которые в дальнейшем изменяются (мутируются) с целью обнаружения полунеправильных внешних данных.

Умный фаззинг (англ. intelligent fuzzing) использует некоторую модель, описывающую внешние данные, для генерации полунеправильных внешних данных. Данный вид фаззинга показал хорошие результаты при фаззинге

сетевых протоколов и форматов данных. В качестве примера умных фаззеров приводятся *Sulley Fuzzing Framework* [80] и *Peach Fuzzing Framework* [81]. Модель данных для фаззера *Sulley* описывается на языке *Python* [82], на котором и написан сам фаззер. Модель данных для фаззера *Peach* описывается в формате XML (англ. Extencible Markup Language – расширяемый язык разметки) [83].

В основе ***Фаззинга с подкреплением*** (англ. – feedback fuzzing) лежат алгоритмы адаптации процесса изменения потока внешних данных на основе информации, полученной от анализируемой программы. Эволюционный фаззинг – один из примеров фаззинга с подкреплением.

Фаззинг «стеклянного» («белого») ящика (англ. glass -box fuzzing, white-box fuzzing), как правило, реализован в виде динамического символьного исполнения программ. Этот метод будет подробнее рассмотрен позже.

Распределённый фаззинг (англ. distributed fuzzing) – фаззинг одной программы, проводящийся на нескольких вычислительных узлах одновременно.

В итоге методы фаззинга программ можно классифицировать по признакам, представленным в таб. 1.

Таблица 1. Признаки классификации методов фаззинга

Генерация данных	Поддерживаемый транспорт	Вид подкрепления
• псевдослучайная; • генетические алгоритмы; • по грамматике; • по модели	• аргументы; • командной строки; • файлы; • переменные окружения; • сетевые сокеты	• без подкрепления; • покрытие по базовым блокам; • покрытие трасс исполнения

Динамическое символьное исполнение

В середине 70-х годов XX столетия были опубликованы результаты исследований, посвящённые автоматизации процесса генерации внешних данных при помощи символьного исполнения программ [84, 85, 86]. В основе предложенного метода лежит представление программы в виде операций над символьными переменными, соответствующим данным, обрабатываемых программой. Набор операций и условных переходов на пути исполнения программы формирует *условие пути*, которое накладывает ограничения на значения символьных переменных и, следовательно, набор внешних данных. Также вводится понятие *дерева символьного исполнения* [86], которое характеризует пути исполнения программы. Каждому узлу дерева ставится в соответствие исполняющаяся операция, а каждому переходу к следующей операции ставится в соответствие направленная дуга. Каждой операций ветвления ставится две дуги: одна соответствует переходу исполнения при вычислении условия в «истину», другая – переходу исполнения при вычислении условия в «ложь». Подобное представление программы обладает следующими свойствами:

1. Для каждого листового узла дерева и, соответственно, завершённого пути исполнения программы существует конкретный несимвольный набор внешних данных программы, который, будучи поданным на вход программе при её реальном (несимвольном) исполнении, приведёт к исполнению по тому же пути (исполнению той же последовательности инструкций).
2. Условия пути для двух разных листовых узлов дерева различны. Если два пути из одного корня ведут к двум различным листовым узлам, то это означает, что на пути была операция ветвления и одной ветке соответствует вычисление условия ветвления в значение «истина», а другой – в значение «ложь».

Представление программы в виде модели операций над символьными переменными позволяет, исходя из семантики операций, применяемых к

символьным переменным, вычислить допустимое значение внешних данных, соответствующее ограничениям в условии пути. Таким образом, имея некоторый начальный набор внешних данных для программы и инструмент, который позволяет получать трассу исполнения программы, можно производить:

- перебор путей исполнения программы путём инвертирования значения условного выражения в операции ветвления для вычисления новых внешних данных программы;
- проверку различных свойств программы при помощи добавления предикатов к условию пути в различных точках пути исполнения. Например, если на пути исполнения встречается операция деления, то можно добавить предикат проверки значения делителя на равенство нулю. Если результирующее условие пути окажется вычислимым в «истину», то это означает, что можно вычислить такой набор внешних данных, при котором произойдёт исключительная ситуация деления на ноль, которая может привести к аварийному завершению программы.

Изначально методы символьного исполнения программ применялись для генерации тестовых наборов для программы с целью повышения тестового покрытия базовых блоков программы методом «белого ящика», или структурного тестирования программы [87]. Но, начиная с середины 2000-х годов, наблюдается взрывной рост инструментов и сред, предназначенных для динамического анализа программ с целью проверки различных свойств программы. В статье, посвящённой инструменту *CUTE* [88], впервые вводится термин «конколик» (конкретно-символьное) исполнение (от англ. *concolic:* CONCrete and symbOLIC execution – конкретное и символьное исполнение), но сам метод впервые описан в статье, посвящённой инструменту *DART* [89] (англ. Directed Automated Random Testing – направленное автоматизированное случайное тестирование). Оба инструмента решают задачу увеличения тестового покрытия программы.

Перед началом процесса генерации тестового покрытия для программы производится *инструментация* программы (англ. instrumentation – добавление специальных инструментов, измеряющих и записывающих различные характеристики в процессе испытаний), то есть добавление в код программы функциональности, задача которой состоит в сборе информации о выполненных программой инструкциях над внешними данными, представляющимися в виде символьных переменных. Далее производятся псевдослучайная генерация начального набора внешних данных для программы и запуск программы с целью сбора последовательности выполненных инструкций, которая в дальнейшем преобразуется в условие пути исполнения. После этого производится добавление в условие пути предиката, который позволяет инвертировать значение условия для одного из условных переходов. Для выбора условного перехода в указанных инструментах используется эвристика поиска «в глубину» (англ. depth-first – сначала глубокий) с одним отличием: для инструмента *CUTE* применяется ограниченный обход «в глубину» с целью исключения ситуации генерации тестового покрытия, которое соответствует покрытию тестовыми наборами бесконечных циклов в программе. Дополненное условие пути передается для решения решателю, который определяет выполнимость дополненного условия пути и в случае его выполнимости позволяет вычислить значения символьных переменных в условии пути, которые удовлетворяют ограничениям в условии пути и в итоге, преобразуются во внешние данные для программы с целью исполнения программы по новому пути. Указанные действия производятся итеративно. Данный метод получил название офлайн (англ. off-line – неподключенный) или «посмертное» (лат. post mortem – посмертный) динамическое символьное исполнение.

В работе [90] описывается метод, который впоследствии был реализован в инструменте *EXE* [91] (англ. Execution generated Executions – исполнения, порождённые исполнением), который является дальнейшим развитием метода динамического символьного исполнения с целью генерации тестового

покрытия программы. Отличие метода заключается в том, что, во-первых, в процессе инструментации программы для операций условного перехода добавляется код, который порождает процесс с копированием состояния программы (при помощи системного вызова *fork()*), и для порожденного процесса производится вычисление нового набора внешних данных, который должен провести исполнение программы по альтернативному пути исполнения для данного условного перехода. Если дополненное условие пути оказывается несовместным, то порождённый процесс уничтожается. Данная техника проведения символьного исполнения получила название онлайн (англ. online – подключенный) динамическое символьное исполнение. Во-вторых, в данном инструменте была впервые применена техника выбора следующего пути для анализа среди доступных путей исполнения в программе, которая базируется на проверке условия целенаправленного проявления ошибки в программе. Благодаря реализации данной функциональности в инструменте *EXE* впервые была продемонстрирована возможность целенаправленной генерации внешних данных, которые приводят к аварийному завершению программы (в терминологии статьи – «вход смерти» от англ. input of death).

Вышеприведённые инструменты основаны на технике инструментации исходного кода программ и построены на основе среды анализа и трансформации исходного текста программ на языке Си *CIL* [92].

В работе, посвященной инструменту *SAGE* [77], зависимость от языка программирования указывается как один из недостатков приведённых методов и описывается метод, в котором генерация условия пути производится на основе офлайн-анализа трасс исполнения программы, представленных в виде инструкций процессора. Трассы записываются при помощи среды динамической инструментации *Nirvana* [93], а решение символьных ограничений производится при помощи решателя *Dissolver* [94], созданных в корпорации Microsoft. Как дополнительные преимущества данного метода указываются:

- поддержка анализа программ, написанных на разных языках программирования;
- возможность анализа программ, доступ к исходному коду которых отсутствует;
- возможность поиска ошибок внесенных не только в исходный код программы, но и ошибок, внесённых компилятором;
- отсутствие значительного влияния инструментационного кода на исполнение программы в связи с отложенной генерацией условий пути – после исполнения программы по трассе исполнения.

Несмотря на достоинства предложенного метода, он также обладает и недостатками. Прежде всего данный метод, как и представленные ранее, не производит анализ системных и библиотечных функций, заменяя анализ кода функций конкретными значениями, полученными в процессе исполнения программы, или с применением генерации псевдослучайных значений. В связи с возможностью генерации псевдослучайных значений может возникать ситуация, при которой новый набор внешних данных программы, вычисленный по трассе исполнения программы, не приводит к исполнению программы по желаемому пути. Авторы называют такую ситуацию *дивергенция*. Также отмечается, что последовательное исследование всех достижимых путей исполнения программы не масштабируется на исследование больших реалистичных программ. Как возможный вариант решения авторы предлагают применять композиционный метод путём адаптации техники, применяющейся для межпроцедурного статического анализа программ, к методу динамического символьного исполнения. Суть метода описывается в докладе [95] и сводится к описанию алгоритма *SMART* (англ. Systematic Modular Automated Random Testing – систематическое модульное автоматизированное случайное тестирование), который по сути предлагает исследовать отдельные функции программы изолированно друг от друга и заменять их символьными формулами, описывающими предусловия и постусловия их вызовов в виде символьных ограничений на входные

параметры и выходные значения, которые в дальнейшем могут использоваться для подстановки в местах их вызова.

Метод, аналогичный инструменту *SAGE*, применяется в инструменте *Avalanche* [96], который построен на основе среды динамической трансляции исполняемого кода программ *Valgrind* [97] и решателя *STP* [98]. В *Avalanche* проводится символьное исполнение с отслеживанием потока данных программы только для помеченных (англ. tainted – запятнанный, marked – помеченный) данных, так же как в инструментах *Flayer* [75], *CatchConv* [99] и *TaintCheck* [100]. Помечается поток данных программы, полученных из окружения программы (внешних данных программы) и непосредственно порожденных из них. Дополнительно строятся предикаты безопасности, которые проверяют опасные условия, например, деление на ноль, разыменование нулевого указателя и других, в случае выполнимости которых вычисляются внешние данные программы, приводящие к её аварийному завершению.

Альтернативный метод, направленный на преодоление ограничений, связанных с анализом программ пользовательского режима и проблемы взрывного роста количества путей для анализа, представлен в работе, посвящённой инструменту S^2E [101]. Данный инструмент построен поверх среды эмуляции аппаратных платформ *QEMU* [102], которая позволяет проводить *полносистемный анализ* – анализ исполнения кода ядра операционных систем совместно с анализом кода прикладных программ. Возможности *QEMU* позволяют контролировать выполнение программы с высокой степенью точности, например, учитывать такие факторы, как влияние планировщика исполнения вычислительных потоков процессора на исполнение программы в процессе его детерминированного воспроизведения. Использование *QEMU* позволяет преодолеть дивергенцию, связанную с ошибками эмуляции поведения системных и библиотечных функций, позволяя получить полную трассу исполнения программы. В качестве платформы для символьного исполнения программы инструмент S^2E

использует платформу символьного исполнения программ *KLEE* [103], построенную поверх представления программы в виде бит-кода компиляторной среды для анализа и трансформации программ *LLVM* [104]. Использование *KLEE* позволяет производить символьное исполнение программы по нескольким путям одновременно. Проблема взрывного роста количества путей для анализа решается применением техники выборочного символьного исполнения, заключающейся в замене части символьных переменных конкретными значениями. Система S^2E, благодаря возможностям *KLEE*, может выборочно исполнять различные части программы символьно или конкретно, таким образом управляя размером дерева путей при символьном исполнении программы, например, исключая некоторые библиотеки из символьного исполнения и трактуя выходные значения функций данных библиотек как конкретные значения. Как и любой метод, S^2E обладает и недостатками, а именно – низкой производительностью анализа. В статье указывается, что замедление исполнения анализируемой программы в режиме конкретного исполнения составляет до 6 раз по сравнению с обычным исполнением программы в *QEMU*, а для символьного – до 78 раз [101]. Одновременно даётся ремарка, что количество инструкций, исполняющихся конкретно, в 30 000 раз больше, чем инструкций, требующих символьного исполнения, если анализируется приложение пользовательского уровня. S^2E позволяет управлять процессом символизации исполнения программы, предоставляя механизм применения различных моделей исполнения, которые определяют, какие части системы должны исполняться конкретно, а какие символьно. В зависимости от целей анализа можно найти компромисс между производительностью, полнотой и точностью анализа. Таким образом, методы динамического символьного исполнения программ можно классифицировать по признакам, представленным в таб. 2.

Таблица 2. Признаки классификации методов динамического символьного исполнения

Способ Инструментации	Способ обработки путей исполнения	Символьное исполнение
• предварительная инструментация исходного кода; • предварительная инструментация исполняемого кода; • динамическая инструментация исполняемого кода; • эмуляция аппаратных платформ	• онлайн – исполнение достижимых путей «на лету» через распараллеливание процесса исполнения программы в точках ветвления; • оффлайн – последовательное исполнение путей исполнения программы	• полное символьное исполнение программы; • символьное исполнение потока помеченных данных программы; • выборочное символьное исполнение

1.2.4. Преимущества и недостатки методов обнаружения ошибок

Статический анализ исходного кода программ

При оценке инструментов статического анализа исходного кода программ нередко используются три метрики [105]:

- полнота анализа (англ. recall), показывающая отношение количества найденных ошибок к полному количеству ошибок в программе;
- точность анализа (англ. precision) – показывающая отношение количества истинных предупреждений об ошибках к количеству ложных предупреждений об ошибках найденных инструментом;
- производительность (англ. performance) – количество вычислительных ресурсов, требующихся для получения результата.

Эти три метрики взаимосвязаны и изменяются противоположно друг другу. Несложно создать инструмент с потрясающей производительностью и точностью, который не сообщит ни об одном дефекте. Также нетрудно создать инструмент с хорошей полнотой и высокой производительностью, который на каждой строчке исходного кода будет сообщать обо всех возможных ошибках. Теоретически можно создать инструмент, который обнаруживает ошибки в программах с хорошей полнотой и точностью, но при наличии достаточного количества вычислительных ресурсов и времени. В результате все инструменты статического анализа исходного кода программ создаются в условиях компромисса между точностью, полнотой, производительностью и учётом требования масштабируемости анализа.

Среди причин подобного компромисса можно перечислить следующие:

1. *Ограничение количества исследуемых путей*. Если в программе *n* условных переходов, то количество путей для анализа в пределе равно 2^n. Хотя на практике некоторые из путей неисполнимы в связи с несовместностью условий, но реальное значение всё равно близко к предельному. Если в программе присутствует циклическая обработка внешних данных, то количество путей в программе ничем не ограничено и исследование всех возможных путей исполнения потребует бесконечно большого времени. Если в программе присутствует обработка асинхронных операций, таких как прерывания и исключительные ситуации (англ. exception – исключение), или вычисление происходит в параллельных вычислительных потоках, то количество путей ещё больше увеличивается. Таким образом, производителям инструментов статического анализа приходится бороться с проблемой анализа большого количества путей, используя технику игнорирования некоторых ситуаций, например, обработки исключений, ограничивать количество анализируемых итераций циклов, игнорировать при анализе рекурсивные вызовы функций или вызовы

функций по указателю. Эта же причина указывается и в диссертационной работе А. Е. Бородина [106, с. 11].

2. *Использование абстрактных доменов*. Предположим, что в процессе символьного исполнения для 8-битной переменной x используется два абстрактных состояния $x = 0$ и $x > 0$. Несмотря на наличие $2^8 = 256$ состояний у подобной переменной в процессе реального исполнения, в процессе символьного анализа будут учитываться только два состояния. Выразительность абстрактного домена является очень важным фактором, который влияет на точность и полноту анализа, но в свою очередь излишняя выразительность может негативно влиять на производительность. Ключевой характеристикой любого абстрактного домена является нижняя грань, которая определяет границу, после которой анализ не имеет никакой полезной информации о текущем значении. То есть нижняя грань – это абстрактное значение, которое соответствует всем возможным значениям. В итоге, достижения нижней грани невозможно избежать для любой нетривиальной абстракции, и в общем случае требует решения *проблемы останова*, которая теоретически неразрешима вследствие теоремы Тьюринга о проблеме останова [107]. При достижении нижней грани у инструмента есть возможность оценить значение и как потенциально опасное, тем самым увеличив полноту анализа, и как потенциально безопасное, тем самым увеличив точность. Также анализ может отсекать по нижней грани операции, в которых используется более двух переменных, и терять точность анализа для таких операций или работать только с целочисленными операциями и не принимать во внимание операции над числами с плавающей точкой.
3. *Работа с недоступным исходным текстом*. Если анализируемая программа использует сторонние библиотеки и их исходный текст недоступен, то в случае вызова функций из таких библиотек наиболее

популярным решением является предположение, что функция ничего не делает и возвращает неизвестное значение. Несмотря на то, что функция может разыменовывать параметры, менять значения глобальных переменных или прерывать исполнение программы. Как правило, в данном случае существует возможность проанализировать исполняемый код библиотечных функций, что добавляет требование анализа объектного кода, либо аннотировать вручную функции наиболее популярных библиотек.

Несмотря на вышеперечисленные проблемы методов статического анализа исходного кода программ, стоит отметить, что данные методы позволяют проанализировать весь доступный исходный код программы, даже тот, который будет исполняться крайне редко или при крайне редком условии. Методы статического анализа исходного кода программ позволяют находить ошибки, которые крайне сложно, если вообще возможно, найти методами динамического анализа программ, например произвести проверку исходного кода на соответствие различным промышленным или корпоративным стандартам, таким как стандарты кодирования Компьютерной группы реагирования на чрезвычайные ситуации (англ. Computer Emergency Response Team (CERT)) [108] – стандарты кодирования Ассоциации надёжности программного обеспечения автомобильной промышленности (англ. Motor Industry Software Reliability Association (MISRA)) [109] – стандарты кодирования Программы создания ударного истребителя (англ. Joint Strike Fighter Air Vehicle (JSF AV)) [110] – стандарт кодирования Лаборатории реактивного движения НАСА (англ. NASA Jet Propulsion Laboratory) [111] и др.

Фаззинг

При анонсе проекта *Springfield* компании Microsoft [112] Элиссон Линн приводит утверждение ведущего исследователя Microsoft Research Париса Годефруа (Patrice Godefroid), что более трети всех ошибок, обнаруженных в процессе всего цикла тестирования операционной системы Windows 7, было

найдено инструментом с элементами фаззинг-тестирования *SAGE*. С другой стороны, разработчик инструмента *AFL* Михал Залевски оценивает количество уязвимостей удалённого исполнения кода, найденных методами достаточно простого фаззинга, в примерно 70% от всего количества найденных уязвимостей за несколько лет до 2015 года [78].

При этом исследователи [113, 114] отмечают и недостатки данного метода обнаружения ошибок и уязвимостей безопасности в программах. Прежде всего, это случайный характер обнаружения ошибок и уязвимостей методами фаззинга. Даже применение фаззинга с обратной связью, при котором собирается информация о прохождении исполнения через определённые базовые блоки в программе (покрытие по базовым блокам) или по определенным путям (покрытие по путям), не гарантирует целенаправленного обнаружения ошибок и уязвимостей. Информация о покрытии программы учитывается в алгоритмах генерации или мутации внешних данных косвенно, то есть не позволяет целенаправленно изменить поток внешних данных таким образом, чтобы гарантированно расширить покрытие или найти ошибку в программе. Также стоит отметить проблему обнаружения позиции и значения ключевых данных в потоке внешних данных, то есть данных, которые приводят к изменению потока управления программы и исполнению по новым путям и базовым блокам. Учитывая косвенный характер влияния информации обратной связи на работу инструментов фаззинга, достаточно сложно подобрать значение внешних данных для условных переходов, зависящих от сравнения с константой. Третья проблема методов фаззинга – это прохождение потока исполнения через функции проверки внешних данных на корректность. Такие методы, как сравнение с константным значением и вычисление контрольных сумм в потоке внешних данных, нередко проводят исполнение программы по пути обработки ошибочных внешних данных и не позволяют провести исполнение программы по путям основного алгоритма решения задачи.

Динамическое символьное исполнение

В работе, посвящённой инструменту *SAGE* [77], упоминаются фундаментальные проблемы динамического символьного исполнения. Во-первых, это проблема взрывного роста количества путей (англ. path explosion problem – проблема взрыва путей), которые необходимо пройти в процессе проведения анализа. Во-вторых, высокая сложность или невозможность применения динамического символьного исполнения для анализа с достаточной точностью и за разумное время специфических функций с интенсивными математическими вычислениями, адресной арифметикой, а также библиотечных функций и функций операционной системы, поведение которых приходится эмулировать при неполносистемном анализе. Как отмечалось ранее, применение псевдослучайных и конкретных значений по результатам вызовов функций с последующей подстановкой их как константных значений в символьные формулы в процессе исполнения программы позволяет обойти часть из указанных ограничений, но может приводить к неожиданному результату, который авторы статьи назвали дивергенцией, когда поведение программы отличается от предполагаемого и исполняется путь, исполнение которого не ожидалось.

Как отмечалось в статье [115], при применении анализа потока помеченных данных в процессе динамического символьного исполнения программы возникают проблемы излишней помеченности потока данных программы. В этом случае говорят, что данные перепомечены (англ. overtainted). Или возникает проблема недостаточной помеченности потока данных программы, в этом случае говорят о недопомеченности (англ. undertainting) потока данных программы. Перепомеченность потока символьных данных программы приводит к падению производительности анализа в связи с построением символьных формул большой размерности. Недопомеченность потока данных программы может приводить к ложным утверждениям о поведении программы, что в итоге снижает точность анализа,

приводя к прекращению распространения потока помеченных данных в потоке данных программы.

Тем не менее метод динамического символьного исполнения позволяет целенаправленно генерировать внешние данные для направления анализа в интересующие части программы и для обнаружения дефектов в программах.

1.2.5. Комбинированные методы обнаружения ошибок в программах

В связи с вышеуказанными недостатками отдельных методов анализа программ становится очевидной идея комбинирования различных методов анализа с целью компенсации их недостатков.

Одним из первых примеров комбинирования фаззинга, статического анализа и динамического символьного исполнения является инструмент *TaintScope* [116], созданный в 2010 году. Для предотвращения выхода программы по результатам неуспешной проверки контрольной суммы в инструменте *TaintScope* методами статического анализа исполняемого кода производится поиск условных переходов по результатам сравнения вычисленного значения со значением контрольной суммы и замена их безусловными переходами на алгоритм обработки данных, который исполнится в случае получения внешних данных, прошедших проверку посредством вычисления контрольной суммы. Потом производится фаззинг программы для обнаружения внешних данных, на которых происходит аварийное завершение программы. После этого производится запуск программы на полученных внешних данных, которые приводят к аварийному завершению программы, с целью сбора трассы исполнения для вычисления условий получения правильной контрольной суммы и генерации внешних данных, которые пройдут проверку путем вычисления контрольной суммы и приведут к аварийному завершению программы. Последним этапом производится проверка сгенерированных наборов внешних данных. Используя данную технику, инструмент позволяет обойти проблему применения интенсивных математических вычислений, предотвращающую использование инструментов динамического символьного исполнения.

Однако в статье, посвящённой описанию метода, не приведен подробный алгоритм вычисления корректных внешних данных, проходящих проверку методом вычисления контрольной суммы, и утверждается, что применение криптографических алгоритмов для вычисления контрольных сумм (хэш-функций) не поддерживается инструментом.

Чтобы решить проблему прохождения условных переходов со сложными проверками (англ. complex checks в терминологии статьи), для которых фаззер не может подобрать значения внешних данных программы за приемлемое время, в инструменте *Driller* [117] применяется комбинирование методов фаззинга и динамического символьного исполнения. В случае, когда при помощи методов фаззинга не удаётся расширить покрытие (найти переходы в следующее состояние, англ. state transitions), то есть мутации внешних данных программы не приводят к расширению покрытия при достаточно большом количестве итераций, инструмент переключается в режим динамического символьного исполнения. В режиме символьного исполнения для всех условных переходов, которые не удалось пройти в альтернативном направлении, производятся попытки вычислить внешние данные программы, которые сделают возможным переход в альтернативном направлении, тем самым расширяется покрытие программы. Все сгенерированные внешние данные, расширяющие покрытие программы, используются на последующих итерациях фаззинг-тестирования программы как затравка.

Аналогичный метод приеняется в инструменте *Munch* [118], который комбинирует методы фаззинга и динамического символьного исполнения. В инструменте используются два режима:

- запуск фаззинга с последующим запуском направленного динамического символьного исполнения (англ. guided symbolic execution) для достижения функций в программе, которые не были достигнуты в процессе фаззинга;

- запуск динамического символьного исполнения с последующим запуском фаззера для повышения глубины проникновения в логику работы программы.

Инструмент *Check'n'Crash*, описанный в работе [119] в 2005 году, совмещает метод статического анализа программ на языке Java при помощи инструмента *ESC/Java* [55], который позволял находить ошибки в программе в процессе компиляции и генерацию модульных тестов для публичных методов классов при помощи инструмента *JCrasher* [120] с целью проверки публичных методов классов на устойчивость к получению псевдослучайных данных. Комбинирование методов позволило значительно увеличить производительность генерации тестов в программах на языке Java для демонстрации программных ошибок, таких как деление на ноль, разыменование нулевых ссылок на объекты, попытки размещения массивов с отрицательной длиной, выход за границы массивов и др.

Развитие метода инструмента *Check'n'Crash* произошло в инструменте *DSD-Crasher* [121], который последовательно применяет технику динамического вычисления инвариантов контекста для функций в виде предусловий и постусловий, впоследствии дополняет предположения статического анализатора *ESC/Java* вычисленными предусловиями для снижения уровня ошибок первого рода, связанных с недостатком информации о контексте вызова публичных методов классов.

Инструмент гибридная система обнаружения уязвимостей (англ. Hybrid Vulnerability Detection System) [122] представляет собой ещё один метод совмещения статического и динамического анализа программ. Система построена на инфраструктуре *Tree-SSA* [123], позволяющей строить статические анализаторы на универсальном промежуточном представлении *GIMPLE* [124] компилятора *GCC* [125], и системе проверки моделей для конечных автоматов с магазинной памятью *Moped* [126]. Идея инструмента состоит в том, чтобы методами статического анализа найти пути в программе, на которых происходит нарушение свойств безопасности программы, с

последующим статическим построением предиката пути и вычислить внешние данные программы для проверки нарушения свойств программы в процессе её исполнения.

В статье, посвящённой описанию инструмента *DyTa* [127], описывается метод совмещения статического анализа программ на языке C# при помощи инструмента *Code Contracts* [128] и инструмента динамического символьного исполнения для генерации тестового покрытия *Pex* [129]. Метод основан на предварительной инструментации CLR-кода проверками утверждений (англ. assert) для предусловий пути на входе в функцию и условий реализации ошибок, полученных от статического анализатора. В процессе динамического символьного исполнения отбрасываются пути, для которых проверка предусловий пути не прошла, и вычисляются условия реализации ошибок. Таким образом, реализуются алгоритм направленного анализа в процессе динамического символьного исполнения, который позволяет сократить количество путей для анализа, и проверка реализуемости ошибок, найденных методами статического анализа программ.

Альтернативный метод объединения техники статического анализа и динамического символьного исполнения описывается в статье [130], посвящённой подтверждению ошибок использования указателя на выделенную память после её освобождения (англ. Use After Free – использование после освобождения). Метод основан на экскавации срезов графа потока исполнения, содержащих операции выделения, использования (обращения по указателю к блоку памяти) и освобождения динамической памяти с последующим применением направленного динамического анализа, который в качестве эвристики выбора пути использует для каждого условного перехода на пути оценку, основанную на расстоянии, с целью выбора наиболее короткого пути на каждой следующей итерации анализа. В процессе анализа пути для операции обращения к блоку памяти, которая помечена инструментом статического анализа исполняемого кода как потенциально использующая указатель на освобожденный блок памяти, производится

вычисление условия реализации ошибки. Если условие оказывается вычислимым, то ошибка обращения к освобожденному блоку памяти подтверждается. Использование эвристики близости или дистанции до заранее определённой точки в программе для реализации направленного динамического символьного исполнения программы на основе предварительного статического анализа программы также используется для генерации регрессионного тестового покрытия для изменённых частей программы [131, 132], построения внешних данных для воспроизведения обнаруженных аварийных завершений программы [133] и достижения определённой точки в программе [134].

Исходя из обзора методов анализа программ, наиболее перспективным направлением развития методов обнаружения ошибок в программах представляется направление совмещения различных методов анализа программ с целью повышения точности результата и производительности инструментов анализа программ. В данной работе будет рассмотрен метод комбинирования статического анализа программ и динамического символьного исполнения программ с целью повышения точности результата анализа программ при сохранении высокой производительности анализа.

Глава 2. Методы анализа программ

Исходя из выводов первой главы, наиболее перспективным направлением исследования является исследование методов комбинирования статического анализа программ и динамического символьного исполнения программ с целью точного обнаружения ошибок в программах. В данной главе будут описаны методы статического анализа программ и динамического символьного исполнения программ, лежащие в основе разработанного метода.

2.1. Статический анализ программ

Методы статического анализа программ характеризуются тем, что проводят анализ без запуска программы на исполнение. Современные инструменты статического анализа программ проводят анализ модели программы, полученной из исходного или исполняемого кода программы.

2.1.1. Анализ абстрактного синтаксического дерева

Дерево абстрактного синтаксиса, или абстрактное синтаксическое дерево (АСД) (англ. – Abstract Syntax Tree, AST), представляет собой структуру данных, в которой каждый внутренний узел дерева соответствует оператору, а его дочерние узлы представляют операнды этого оператора [135, с. 110]. Фактически данная структура описывает структуру исходного кода программы. С использованием данной структуры возможно построение анализаторов, которые выполняют обход узлов абстрактного синтаксического дерева или таблицы символов [10, с. 49] и направлены на проверку простейших синтаксических правил. При этом анализ абстрактного синтаксического дерева является относительно быстрым методом анализа программ [10, с. 51, теорема 2.1] и позволяет создавать анализаторы на специализированных декларативных языках [136]. При помощи анализа абстрактного синтаксического дерева возможно построение анализаторов, проверяющих не только такие достаточно простые правила кодирования, как проверка правил именования переменных (например, на соответствие венгерской нотации) или отступов в исходном коде программ, но и ошибок, влияющих на логику исполнения программы. На примере анализа исходного

кода программ на языках Си или Си++ при помощи анализа абстрактного синтаксического дерева возможно построение анализаторов для обнаружения таких ошибок, как:

- цикл с пустым телом (правило 14.8 стандарта MISRA-C:2004) (листинг 1);

Листинг 1. Пример кода для правила 14.8 стандарта MISRA-C:2004

Ошибка	Нет ошибки

`while(a == true);`
`{`
`  ...`
`}`
	`while(a == true)`
	`{`
	`  ...`
	`}`

- использование оператора присваивания вместо операции сравнения на равенство в условных выражениях (правило 13.1 стандарта MISRA-C:2004) (листинг 2);

Листинг 2. Пример кода для правила 13.1 стандарта MISRA-C:2004

Ошибка	Нет ошибки

`while(a = true)`
`{`
`  ...`
`}`
	`while(a == true)`
	`{`
	`  ...`
	`}`

- использование одинаковых выражений в левой и правой части оператора сравнения) (листинг 3);

Листинг 3. Пример кода для ошибки сравнения одинаковых подвыражений

Ошибка	*Нет ошибки*

`if(a.x == a.x)`
`{`
`  ...`
`}`
	`if(a.x == a.y)`
	`{`
	`  ...`
	`}`

и других.

Предупреждения об ошибках, найденных при помощи модели программы в виде абстрактного синтаксического дерева, как правило, представляются одной точкой в программе, указывающей на конструкцию языка программирования, которая содержит ошибку.

Анализ абстрактного синтаксического дерева имеет ограниченные возможности для обнаружения ошибок в программах. Для ошибок.

требующих анализа потока данных программы и потока управления, используется граф потока программы.

2.1.2. Анализ потока программы

Граф потока программы представляет собой структуру, в которой узлами являются базовые блоки, а рёбра указывают порядок следования базовых блоков. Базовый блок – это последовательность команд, имеющая одну точку входа и одну точку выхода. Анализ потока программы позволяет вычислять свойства потока данных программы с учётом потока управления, что позволяет обнаруживать наиболее интересные ошибки в программе, связанные с последовательностью операций над данными программы.

Одной из возможных реализаций анализа потока программы является представление потока операций программы в виде промежуточного представления на основе графа потока управления (англ. control-flow graph) программы, в котором базовые блоки состоят из операций представленных в трехадресном коде [135, с. 145]: операция представляется в виде четверок <*op, arg1, arg2, res*>, где *op* – операция, *arg1* – первый аргумент операции, *agr2* – второй аргумент операции, *res* – результат операции. При этом у каждого базового блока есть адрес, который может быть использован для осуществления передачи управления в операциях условного и безусловного перехода. В процессе обхода графа потока управления программы осуществляется анализ потока данных программы, заключающийся в том, что для каждой операции вычисляются свойства аргументов и результата операций в зависимости от семантики операции. В зависимости от вида анализа различают потоково-чувствительный анализ, анализ, чувствительный к путям, контекстно-чувствительный (межпроцедурный) анализ [137].

Анализ, чувствительный к потоку, с консервативным подходом при принятии решений

Основной особенностью консервативного анализа является принятие безопасных решений при отсутствии каких-либо предположений о диапазоне значений переменных и контексте вызова подпрограммы [135, с. 712].

Например, во фрагменте программы, представленном на листинге 4, никакой дополнительной информации о подпрограмме *getDataPointer()* нет, так как она импортируется из внешней библиотеки, в связи с чем при проведении консервативного анализа должно учитываться предположение, что указатель указывает на любую переменную программы – и, следовательно, операция на строчке 4 может, с одной стороны, изменить значение любой переменной программы, с другой – привести к разыменованию нулевого указателя.

Листинг 4. Пример неопределённости при консервативном анализе

```
1: extern int getDataPointer();
2: int *p;
3: p = getDataPointer();
4: *p = 12;
```

Поэтому при проведении консервативного анализа для данного фрагмента может быть сделан вывод о разыменовании нулевого указателя, что в общем случае неверно. В данном примере появляется трасса событий в предупреждении об ошибке, которая будет содержать два события:

1. Строка 3: возможная инициализация переменной *p* значением *NULL*;
2. Строка 4: разыменование переменной *p*, возможно содержащей *NULL*.

С другой стороны, для некоторых случаев консервативный анализ даёт точный результат. Например, во фрагменте на листинге 5 независимо от значения переменной *n* код базового блока на строках 4-6 недостижим в связи с несовместностью условий на строках 1 и 3,

Листинг 5. Пример недостижимого кода

```
1: if(m && n)
2: {             // m = true and n = true
3:     if(!n)
4:     {         // m = true and n = true and n <> true
5:         ...   // n = true and n <> true => ∅
6:     }
7: }
```

в связи с чем консервативный анализ обнаружения заведомо недостижимых базовых блоков может быть использован для уменьшения графа потока программы в процессе предварительного анализа за счёт исключения

недостижимых базовых блоков. В данном примере в трассе предупреждения об ошибке будут указаны три события:

1. Строка 1: операнд *n* условного выражения должен быть вычислен в значение *истина*.
2. Строка 3: операнд !*n* должен быть вычислен в истину, значит, *n* должен иметь значение *ложь*.
3. Строка 4: составной оператор недостижим из-за несовместности условий на строках 1 и 3.

Использование потоково-чувствительного анализа позволяет обнаруживать ошибки в подпрограммах, которые реализуются на всех путях исполнения, содержащих ошибочную инструкцию. В качестве примера можно привести фрагмент кода с ошибкой, проявляющейся на всех путях, содержащих операцию разыменования (листинг 6). В приведённом примере невозможно определить значение указателя *p* без учета контекста вызова подпрограммы, но если исполнение проходит по пути, содержащему операцию на строке 5, то независимо от контекста вызова подпрограммы произойдёт разыменование нулевого указателя, потому что в обратном случае подпрограмма завершит исполнение на строке 4.

Листинг 6. Пример ошибки, реализующейся независимо от условий пути

```
1: void emptystring(char * p)
2: {
3:     if(p)
4:         return;              // p <> NULL
5:     *p = '\0';               // p = NULL and deref(p)
6:  }
```

Чувствительный к путям анализ

Анализ потока без использования дополнительного анализа условий выполнения путей не всегда может быть применён для точного определения наличия ошибки в программе. Классическим примером для иллюстрации необходимости проведения анализа, чувствительного к путям исполнения, является последовательное исполнение двух условных операторов, или ромбовидного графа потока управления (листинг 7):

Листинг 7. Пример программы, требующей анализа путей

```
1: char buffer[10]
2: void init(char **p, int a)
3: {
4:     if(a > 0)
5:         *p = &buffer[0];    // *p <> NULL if a > 0
6:     if(a >= 0)
7:         **p = '\0';         // deref(*p) if a == 0
8:  }
```

Для того чтобы утверждать об отсутствии ошибки разыменования нулевого указателя на строке 7, необходимо провести анализ путей исполнения подпрограммы *init* и собрать условия, при которых значение указателя не равно нулю.

Межпроцедурный анализ

Размещение полного графа потока программы в оперативной памяти в процессе анализа не представляется возможным, если в программе присутствуют рекурсивные вызовы в программе, кроме случая хвостовой рекурсии, которую можно преобразовать в цикл. Но даже при отсутствии рекурсивных вызовов в программе размещение полного графа потока программы в оперативной памяти затруднительно для сколь-нибудь больших программ. В связи с этим возникает необходимость проведения анализа потока программы по частям с целью масштабирования анализа. Одним из наиболее удачных и широко применяемых методов разбиения графа потока программы на части является межпроцедурный анализ, или анализ, чувствительный к контексту вызова подпрограмм.

Идея межпроцедурного анализа заключается в разбиении программы на блоки, соответствующие подпрограммам, и проведение анализа каждого блока отдельно. Стоит отметить, что простого анализа каждой подпрограммы в отдельности может быть недостаточно для обнаружения определённого типа дефектов.

Рассмотрим пример на листинге 8.

Листинг 8. Пример межпроцедурной трассы ошибки

```
1: char * init(size_t n)              // init может вернуть ноль
2: {
3:     char *p = malloc(n);           // p может быть равно нулю
4:     if(0 != p)                     // Если p не равен нулю
5:         *p = '\0';                 // Обнулить строку
6:     return p;                      // возможен возврат нуля
7: }
8: int main(int argc, char* argv[])
9: {
10:    char* buf;
11:    buf = init(strlen(argv[0]));   // buf может быть равен нулю
12:    strcpy(buf, argv[0]);          // разыменование возможного
13:}                                  // нуля
```

Подпрограмма *malloc* может вернуть ноль вместо адреса выделенного блока памяти, если выделить память не удалось. В подпрограмме *init* стоит соответствующая проверка перед разыменованием указателя *p*, что гарантирует отсутствие ошибки разыменования нулевого указателя в подпрограмме *init*. Однако значение указателя *p*, возвращаемое в вызывающую подпрограмму, может быть равно нулю и будет разыменовано в подпрограмме *strcpy* на строке 12. Следовательно, если информация о контексте вызова подпрограммы *strcpy*, а именно возможность равенства указателя *buf* нулю, в процессе анализа не будет учтена, то ошибка на строке 12 не будет обнаружена. Анализ программ, учитывающий контекст вызова подпрограмм и позволяющий распространять информацию о возможных значениях переменных по пути, проходящему через тела подпрограмм, называют *контекстно-чувствительным межпроцедурным анализом программ*.

Для решения задачи контекстно-чувствительного межпроцедурного анализа программ в процессе анализа граф вызовов программы сортируется в обратном топологическом порядке с разрывом циклов, образованных рекурсивными вызовами, и производится анализ каждой подпрограммы, начиная от листовых вершин получившегося направленного ациклического графа, к его корню, являющемуся точкой входа в вычислительный поток. Одним из способов распространения информации о возможных значениях

аргументов подпрограмм и возвращаемого значения является механизм аннотаций подпрограммы. Аннотация представляет собой ограничения на множество значений аргументов и возвращаемого значения подпрограммы, которые зависят от графа потока подпрограммы. Таким образом, в местах вызова подпрограмм в контекст вызова подставляются ограничения на множество значений аргументов подпрограммы, а в постусловиях учитывается влияние вызова подпрограммы на её аргументы и возвращаемое значение, взятые из аннотации, вместо повторного анализа подпрограммы. Для примера приведённого выше на листинге 8, в точке вызова подпрограммы *strcmp* формула для вычисления возможного дефекта разыменования нулевого указателя может выглядеть так:

deref(strcpy.arg1) && deref(strcpy.arg2) && (buf may be 0).

После подстановки аргументов формулы будет выглядеть так:

deref(buf) && deref(argv[0]) && (buf may be 0),

что даёт основание для сообщения об ошибке типа «Возможное разыменование нулевого указателя» с межпроцедурной трассой исполнения программы, содержащей точку инициализации ошибочной ситуации в подпрограмме *init* на строке с вызовом подпрограммы *malloc* и точкой реализации ошибочной ситуации в подпрограмме *main* на строке с вызовом подпрограммы *strcpy*.

2.1.3. Представление результатов статического анализа программ

Исходя из вышеизложенных основ обнаружения ошибок в программах при помощи статического анализа, можно привести обобщённое представление предупреждения об ошибке в программе. В общем случае предупреждение об ошибке, найденной методами анализа абстрактного синтаксического дерева и методами анализа потока программы, представляется в виде тройки <*i*, *E*, *s*>:

- ***i*** – точка инициализации ошибочной ситуации. Представляет собой операцию в программе, выполнение которой приводит к созданию предусловия возникновения ошибки в программе;

- ***E*** – множество точек событий, влияющих на реализацию ошибки в программе. Представляет собой набор операций программы, исполнение которых в итоге приводит к ошибке в работе программы;
- ***s*** – точка проявления ошибки. Операция в программе, выполнение которой приводит к проявлению ошибки.

В общем случае точку инициализации и проявления ошибки в программе также можно включить в множество точек событий в программе, приводящих к ошибке при исполнении программы. В вырожденном случае предупреждение об ошибке в программе может быть представлено только операцией, исполнение которой без каких-либо дополнительных условий приведёт к проявлению ошибки в программе.

2.2. Динамический анализ программ

Динамические методы анализа программ характеризуются тем, что проводят анализ программ в процессе или по результатам запуска программы на исполнение. Независимо от способа реализации в процессе анализа выделяются несколько этапов:

- сбор трассы исполнения программы в виде операций, преобразование операций в формулы над данными программы;
- преобразование формул для вычисления новых внешних данных;
- вычисление новых наборов внешних данных;
- оценка новых наборов внешних данных для выбора наиболее перспективного пути для анализа.

Процесс сбора трассы программы заключается в запуске программы на исполнение на определённом наборе внешних данных с записью выполненных операций в виде формул, в которых внешние данные программы представляются как свободные переменные. Такой метод позволяет после сбора ограничений на пути исполнения (англ. path constraints) в последствии добавить к ним утверждения, ограничивающие множество значений свободных переменных таким образом, чтобы вычислить новые значения

внешних данных для исполнения программы по альтернативному пути или для проверки возможности нарушения предикатов безопасности определённых операций.

2.2.1. Сбор трассы исполнения программы

В основе способов сбора трассы исполнения программы применяются два принципиально разных метода: инструментация кода программы, то есть добавление инструментирующего кода непосредственно в код программы, и инструментация среды исполнения программы. В обоих случаях инструментация производится с целью получения информации о состоянии программы во время исполнения. При проведении инструментации добавляемый код не меняет логику вычислений программы, но ухудшает производительность исполнения инструментированной программы по сравнению с неинструментированной в связи с исполнением дополнительного инструментирующего кода.

Статическая инструментация исходного кода

При статической инструментации исходного кода программы происходит преобразование исходного кода программы путём добавления операций по сбору информации о состоянии программы во время исполнения непосредственно в исходный код программы. Такой метод обладает определёнными преимуществами:

- предоставляет возможность проведения инструментации один раз перед преобразованием программы в исполняемый код, что снижает накладные расходы на инструментацию программы в процессе исполнения;
- позволяет проводить инструментацию программы на верхнем уровне представления программы с учётом семантики языка программирования.

С другой стороны, у данного метода есть недостатки:

- требуется обеспечение синтаксической и семантической корректности инструментирующего исходного кода, добавляемого в исходный код программы;
- если требуется сбор полной трассы исполнения программы, то наличие исходного кода всех библиотечных функций, использованных в программе, является необходимым условием для проведения корректной инструментации, что не всегда возможно.

Среди известных на данный момент инструментальных средств преобразования исходного кода программы можно отметить *CIL* [92] для программ на языке Си, *ROSE* [138] – компиляторная инфраструктура для языков Си, Си++ и Фортран, TXL [139].

Статическая инструментация программы в процессе преобразования исходного кода в исполняемый код

Для проведения подобного преобразования программы требуется инструментальная поддержка со стороны компилятора или интерпретатора исходного кода, так как при применении данного метода инструментирующий код встраивается средствами компилятора. Метод обладает следующими преимуществами:

- не требует генерации или вставки синтаксически корректного исходного кода программы;
- инструментация проводится один раз в процессе компиляции и исключает накладные расходы на инструментацию в процессе исполнения программы.

В то же время данному методу также присущи недостатки, связанные с невозможностью инструментации кода библиотечных функций, как и у метода инструментации исходного кода.

Современные компиляторы поддерживают программные интерфейсы модификации программ, такие как *GIMPLE*[124], реализованный в рамках проекта *GCC* [125] и *libTooling* из компиляторной инфраструктуры *Clang/LLVM* [140].

Статическая инструментация исполняемого кода программы

При применении данного метода производится вставка инструментирующего кода непосредственно в исполняемый код программы до его непосредственного исполнения на вычислительном устройстве. Применение данного метода позволяет инструментировать весь исполняемый код программы, включая код библиотечных и системных функций. При этом во время инструментации уже не может учитываться семантика исходного языка программирования и проводится инструментация на уровне машинных команд, где уже нет доступа к информации о высокоуровневых типах переменных и др.

Динамическая инструментация кода программы

Динамическая инструментация отличается от статической тем, что осуществляется в процессе исполнения программы. Код программы исполняется под контролем инструмента, который осуществляет его преобразование «на лету» путём добавления инструментирующего кода. Преимуществами данного метода являются:

- отсутствие необходимости предварительной инструментации кода программы и динамически подключаемых библиотек;
- возможность реализации единого инструментирующего кода для программ, скомпилированных под разные процессорные архитектуры.

Стоит отметить и недостаток данного метода инструментации: значительные накладные расходы на инструментацию программы в процессе её исполнения.

Среди средств динамической инструментации программ можно отметить среду динамической бинарной трансляции *Valgrind* [97], позволяющий проводить инструментацию программ для операционных систем *Linux*, *Solaris*, *Mac OS X*, *Android*, инструменты динамиеский инструментации *Pin* [141], поддерживающий инструментацию программ для операционных систем *Windows*, *Linux*, *Mac OS X*, и *DynamiRIO* [142],

поддерживающий инструментацию программ для операционных систем *Windows*, *Linux* и *Android*.

Встраивание в интерпретирующую среду

Одним из методов сбора информации о программе в процессе исполнения является модификация интерпретирующей среды, или встраивание в интерпретирующую среду. Такой метод может быть реализован для интерпретируемых языков программирования или для программ на языках программирования, компилируемых в код для виртуальных машин, таких как *Java Virtual Machine* (*JVM*) [143] и *Common Language Runtime* (*CLR*) [144]. При таком методе инструментирующий код встраивается непосредственно в исполняющую среду и позволяет осуществлять полный контроль над исполнением программы. Преимуществами данного метода являются:

- отсутствие необходимости модификации кода программы при сохранении возможности получения информации о состоянии программы во время исполнения;
- возможность обработки динамически загружаемых библиотек средствами самой исполняющей среды.

Недостаток данного метода заключается в неизбежности накладных расходов на исполнение инструментирующего кода в виртуальной машине.

В качестве примера применения данного метода можно указать систему построения инструментов анализа Java-программ *Javana* [145], инструмент *Pex* [129], который использует для инструментации программ, исполняющихся в среде *CLR*, программный интерфейс профилировки приложений *Microsoft .Net*, и метод анализа программ на основе модификации *JVM Dalvik* [146] и *JVM Avian* [147].

Полносистемная эмуляция

Полносистемная эмуляция аппаратуры – это метод разграничения вычислений, производимых в эмулируемом аппаратном окружении, и вычислений на устройстве, на котором запущен эмулятор. Такой метод позволяет получить максимальный уровень контроля за вычислительными

процессами в эмулируемом аппаратном обеспечении и, как следствие, получать поток исполненных инструкций непосредственно с процессора эмулируемого вычислительного устройства. Преимущества метода для получения трасс исполнения программ:

- отсутствие необходимости модификации кода программного обеспечения;
- возможность получения трасс инструкций, проходящих через код ядра операционной системы;
- возможность влияния на события аппаратуры для управляемого анализа реакции операционной системы на аппаратные события.

В качестве недостатка данного метода стоит отметить необходимость извлечения трассы исполнения анализируемой программы из всего потока инструкций процессора. Среди инструментов полносистемной эмуляции, примененных для анализа программ, стоит выделить *QEMU* [102], на базе которого построены такие инструменты, как S^2E [101] и *TrEx* [148].

2.2.2. Преобразование трассы в символьную формулу

Для реализации возможности вычисления новых внешних данных программы операции, выполняющиеся над данными программы, представляются в виде, пригодном для вычисления с помощью формул в теориях (англ. satisfiability modulo theories). Среди форматов описания формул в теориях стоит отметить формат описания формул инструмента *Cooperative Validity Checker* [149] и *SMT-LIB* [150], поддерживаемые современными решателями [98, 151, 152, 153, 154].

Для этого производится преобразование операций вычислительной системы в семантически эквивалентные операции формата описания формул. Например, решатели, работающие в логике битовых векторов, поддерживают операцию сложения двух битовых векторов *bvadd* в формате описания формул SMT-LIB v2.0, с помощью которой можно смоделировать семантику операции сложения `add` из набора инструкций процессора Intel x86. С другой стороны, чтобы смоделировать семантику операции сравнения `cmp` в формуле SMT-Lib

v2.0 необходимо провести ряд вычислений над аргументами операции сравнения, чтобы получить значение флага `ZF` (листинг 9):

Листинг 9. Пример описания семантики инструкции `cmp` в формате SMT-Lib v2.0

`cmp ax, bx`	`( declare-fun cmpRes () (_ BitVec 16) )`
	`( declare-fun zfRes () (_ BitVec 1) )`
	`( assert (= cmpRes (bvsub rAX rBX)) )`
	`( assert`
	`(= zfRes (ite`
	`(= cmpRes (_ bv0 16) (_ bv0 1) (_ bv1 1)))`
	`)`

где: первые две инструкции декларируют именованный переменные `cmpRes` длиной 16 и `zfRes` длиной 1 бит, третья инструкция вычисляет разность ранее определенных переменных `rAX` и `rBX`, соответствующих значениям регистров `ax` и `bx`, и четвертая инструкция вычисляет значение флага `ZF`.

Преобразование может производиться как непосредственно напрямую из некоторого представления операций в формулу, как сделано в модуле трассировщика инструмента *Avalanche* [96], так и при помощи специализированных библиотек, таких как *Triton* [155], которые предназначены для преобразования потока команд процессора в символьную формулу.

Полученная формула описывает исполнение программы по пройденному пути. Такая формула получила название предиката пути (англ. path predicate), или ограничений пути исполнения (англ. – path constraints). Имея предикат пути, можно вычислить новый набор внешних данных, который позволяет либо провести исполнение программы по новому пути, либо исполнить программу по тому же пути, но при этом получить новый набор внешних данных программы, например для проявления программной ошибки.

Для исполнения программы по новому пути производится разбиение формулы на несколько формул по количеству условных переходов. Все

операции после вычисления условия условного перехода отбрасываются, а условие, от которого зависит условный переход, инвертируется. В случае вычислимости полученной формулы решатель вычислит значения свободных переменных, которые соответствуют внешним данным программы. На вычисленном наборе внешних данных исполнение программы пройдёт по альтернативному пути.

Для проверки операции, потенциально приводящей к проявлению ошибки в программе, производится проверка нарушения условия безопасного исполнения операции. В статье [156] условие возникновения ошибки называется предикатом безопасности (англ. security predicate). Для проверки возможности проявления ошибки в программе к предикату пути до потенциально опасной операции добавляются условия нарушения безопасного исполнения операции. В случае совместности условий полученной формулы решатель вычислит значения свободных переменных, используя которые можно сформировать набор внешних данных, на которых проявится ошибка в программе.

2.2.3. Алгоритмы выбора трассы для анализа

Одним из способов обеспечения эффективности динамического символьного исполнения при наличии проблемы взрывного роста количества путей для анализа в процессе динамического символьного исполнения является применение алгоритмов выбора следующего пути для анализа. При выборе эвристики могут применяться различные алгоритмы.

Случайный выбор. Выбор следующего пути производится случайным образом.

Обход «в ширину». Первым для анализа выбирается путь, проходящий через первую непройденную альтернативную ветку каждого встретившегося на пути исполнения условного перехода.

Обход «в глубину». Первым для анализа выбирается путь, имеющий наибольшую длину.

Максимизация покрытия. Первым выбирается путь, при прохождении по которому наблюдается наибольший прирост проанализированных базовых блоков программы.

Сначала путь с ошибкой (англ. buggy-path first). Первым выбирается путь, на котором уже обнаружена ошибка, не приведшая к аварийному завершению программы. Эвристика описана в работе, посвящённой инструменту автоматической генераций эксплуатирующих внешних данных (эксплойтов) AEG [157], и основана на наблюдении, что если на пути обнаружена ошибка на единицу при работе с буфером, то далее на таком пути с большей вероятностью может быть найдена эксплуатируемая ошибка.

Исключение недостигающих путей. Выбираются только те пути, которые ведут к требуемой точке в программе. Метрика описана в работе [131], посвящённой построению тестового покрытия для изменившихся регионов программы, и в работе [132], посвящённой построению тестового покрытия для измененных регионов программы и обнаружения ошибок путём проверки нарушения утверждений (англ. asserts).

Направленный анализ на основе эвристики близости. Первым выбирается путь с наименьшим расстоянием в терминах количества инструкций или базовых блоков до заданной точки в коде программы. Эвристика наименьшего расстояния (англ. shortest-distance symbolic execution) описана в работе [134], посвящённой генерации внешних данных, на которых исполнение программы проходит по определённой строке исходного кода. В работе [133], посвящённой решению задачи построения внешних данных для воспроизведения ошибки по информации об аварийном завершении программы, содержащей стек вызовов (англ. crash dump with call stack), описывается эвристика близости (англ. proximity) до определенной точки в программе, которая используется для прохождения точек в программе из стека вызовов и адреса операции, на которой произошло аварийное завершение работы программы.

2.2.4. Методы регистрации ошибок в программе

Регистрация аварийного завершения программы.

Данный метод регистрирует событие аварийного завершения программы, и таким образом делается вывод об обнаружении ошибки. Такой метод позволяет регистрировать как состояния принудительного завершения работы программы путём вызова функций *abort* (сигнал SIGABRT), так и реализацию ошибок при нарушении прав доступа к памяти (сигнал SIGSEGV), ошибки деления на ноль (сигнал SIGFPE) и др.

Проверка нарушения предиката безопасности.

Данный метод использует методы символьного исполнения для проверки возможности нарушения корректного состояния программы при выполнении различных операций. Для каждого типа ошибок формулируется свой предикат безопасности.

1. ***Деление на ноль*** (CWE-369) [158] – проверяется невозможность получения внешних данных, при которых делитель станет равным нулю. При наличии возможности нарушения предиката в процессе исполнения программы в точке деления происходит попытка деления на ноль, что приводит к аварийному завершению программы.
2. ***Выход за границы буфера в памяти*** (CWE-119) [159] – проверяется нарушение границ буфера, выделенного в памяти, при обращении по указателю на запись или чтение. При нарушении границ буфера в памяти может произойти выход за пределы выделенной программе страницы памяти или нарушение прав доступа к памяти, выделенной программе. В этом случае произойдёт аварийное завершение программы. Если выхода за пределы памяти, выделенной программе, или нарушения прав доступа к сегменту памяти не происходит, то в этом случае возможны следующие вредоносные последствия:
   - при совершении операции записи:
     - порча данных при записи;
     - порча стека вызовов программы;

- передача управления на вредоносный код или случайный адрес в программе путём замены адреса возврата из функции;

- при совершении операции чтения:
  - несанкционированный доступ к данным в памяти программы.

3. ***Неверное завершение работы с ресурсом или освобождение ресурса*** (CWE-400, CWE-404) [160, 161] – проверяется корректность получения доступа к ресурсу и его освобождения. Проверяется возможность потери контроля над ресурсом путём потери доступа к адресу на куче или дескриптору ресурса операционной системы. В качестве ресурса может выступать любой объект операционной системы, доступ к которому может получить программа: блок динамической памяти программы, дескриптор файла, сокетов и других ресурсов. Если на пути исполнения программы производится получение доступа к ресурсу путём выделения блока памяти на куче, открытия файла или другого объекта операционной системы, но не производится освобождение блока памяти на куче или закрытие файла или другого ресурса операционной системы после окончания работы с ним, то при неоднократном прохождении исполнения по данному пути может произойти падение производительности работы операционной системы или ошибка отказа в выделении ресурса, которая может привести к аварийному завершению программы.
4. ***Использование памяти на куче после освобождения*** (CWE-416) [162] – проверяется возможность обращения по адресу в памяти на куче после освобождения блока, которому он принадлежит. Подобный дефект может привести к аварийному завершению программы, получению неожиданного значения данных программы или исполнению нежелательного кода.
5. ***Повторное освобождение блока памяти на куче*** (CWE-415) [163] – проверяется возможность повторного вызова функции освобождения блока памяти на куче со значением указателя на освобождённый блок памяти на пути исполнения программы. Повторное освобождение блока памяти на куче может повредить внутренние структуры менеджера памяти

и привести к аварийному завершению программы или получению указателя на один блок памяти при последующем выделении блока памяти на куче для разных вызовов функции выделения памяти, что может привести к последующему переполнению буфера или получению несанкционированного доступа к данным в памяти.

6. ***Разыменование нулевого указателя*** (CWE-476) [164] – проверяется возможность обращения к памяти на чтение или на запись по указателю, выставленному в нулевой адрес. Обращение к нулевому адресу на куче приведёт к аварийному завершению программы.
7. ***Использование неинициализированной переменной*** (CWE-467) [165] – проверяется возможность использования переменной до явного присвоения ей некоторого значения. Данный дефект может привести как к некорректной работе программы, так и к аварийному её завершению.
8. ***Работа с данными, полученными из ненадёжных источников*** (CWE-134, CWE-129, CWE-20) [166, 167, 168] – проверяется возможность использования значения из ненадёжного источника. Данный дефект может привести к некорректной работе программы, выполнению вредоносного кода и утечке защищенных данных.

Глава 3. Метод классификации предупреждений об ошибках в программах

3.1. Модель обнаружения ошибок в программе

3.1.1. Формализация понятия ошибки в программе

В основе предлагаемого в данной работе метода классификации предупреждений об ошибках в программах лежит модель обнаружения ошибок в процессе символьного исполнения программы и алгоритм комбинированного анализа программ.

Для того чтобы описать метод обнаружения ошибок в программе необходимо формализовать понятие ошибки.

Определим программу как четверку:

$$P = <F, S, s_I, S_T>, \quad (1)$$

где: S – множество состояний программы; s_I – начальное состояние программы; S_T – множество конечных состояний программы; F – множество операций, каждая из которых переводит программу из одного состояния в другое:

$$f: S \to S, f \in F, \quad (2)$$

Тогда исполнение программы можно определить как последовательность переходов между состояниями программы $s_i \in S$, осуществляемых операциями $f_i \in F$:

$$\{ f_1(s_1) \to s_1, f_1(s_1) \to s_2, \ldots , f_i(s_i) \to s_T \}, \quad (3)$$

где: $s_I \in S$ – начальное состояние программы; $s_T \in S_T$ – одно из конечных состояний программы; $f_i \in F$ – множеству операций программы, переводящих одно состояние программы в другое.

Определим состояние программы как,

$$s = <d, f> \mid d \in D, f \in F, \quad (4)$$

где d – подмножество множества данных, обрабатываемых программой; D – множество данных, обрабатываемых программой; f – следующая операция программы.

Тогда исполнение операции программы можно представить в виде

$$f_i(d_i) \rightarrow < d_j, f_j > \mid d_i,\ d_j \in D,\ f_i, f_j \in F. \quad (5)$$

Определение 1. *Ограничением пути исполнения в программе PC* будем называть множество ограничений на значения данных программы, полученное путём преобразования операций, выполненных над данными программы на пути исполнения, в элементы множества ограничений и однозначно описывающее исполнение программы по пути, достигающем состояния *s*.

Не ограничивая общности рассуждений, всё множество операций в программе можно разделить на три вида:

$$f_i(d_i) \rightarrow < d_j, f' > \left| \begin{cases} d_i \neq d_j,\ f_i \rightarrow f_j & \text{вычислительная операция} \\ d_i = d_j,\ f_i \rightarrow f_j & \text{безусловный переход} \\ d_i = d_j, \begin{cases} f_i \rightarrow f_j \mid p_i \\ f_i \rightarrow f_k, \mid \neg p_i \end{cases} & p_i \in PC \text{ условный переход} \end{cases} \right. , \quad (6)$$

где:

- вычислительная операция изменяет состояние программы путём изменения множества данных программы $d_i \rightarrow d_j$ и переводит исполнение программы на следующую операцию $f_i \rightarrow f_j$;
- безусловный переход изменяет состояние программы путём перевода исполнения на следующую операцию $f_i \rightarrow f_j$;
- условный переход изменяет состояние программы путём перевода исполнения на операцию f_j, если подмножество предусловий состояния, являющееся условием перехода, вычисляется в истину, и на операцию f_k, если подмножество предусловий состояния, являющееся условием перехода, вычисляется в ложь.

Разделим множество данных программы *D*, на котором определено ограничение пути исполнения в программе *PC*, на два множества: *V* – множество внутренних данных программы; *W* – множество внешних данных программы. Тогда предусловие состояния s_i можно описать как функцию:

$$p(w, v) \mid w \subseteq W, v \subseteq V \ . \quad (7)$$

Заменим элементы множества W на элементы множества переменных X, где позиции каждого элемента множества внешних данных в потоке внешних данных программы соответствует переменная $x \in X$.

Определение 2. *Множеством свободных переменных* будем называть множество переменных, значения которых соответствуют значениям элементов множества внешних данных программы.

Определение 3. *Множество зависимых переменных* формируется из переменных, являющихся результатом исполнения операций, множество аргументов которых содержит хотя бы одну свободную или хотя бы одну зависимую переменную.

Определение 4. *Символьное ограничение пути SPC*, или *символьное предусловие* состояния s в программе, является множеством ограничений на значения зависимых и свободных переменных и внутренних данных программы, полученных путём преобразования операций над ними на пути исполнения программы, предшествующем состоянию s.

Множество значений данных программы удовлетворяющих символьному предусловию пути для состояния s_i описывается функцией $\pi (x_i, v_i)$:

$$D_{\pi i} = \pi (x_i, v_i) \mid D_{\pi i} \subseteq D \ , \quad (8)$$

Определение 5. *Состояние ошибки* это такое состояние в программы $s_{err} \in S$, при котором дальнейшее исполнение программы ошибочно.

Определение 6. *Множество определения операции f* – это такое подмножество данных программы $D' \subseteq D$, на котором выполнение операции f не приводит к достижению состояния ошибки.

Утверждение 1. Для каждой операции f, входящей в множество операции программы F, существует множество определения данной операции D':

$$\forall f \in F \ \exists \ D' \subseteq D, \quad (9)$$

Утверждение 2. Если множество значений аргументов операции f не входит в множество определения операции D' и множество D' не пусто, то исполнение операции приводит программу в состояние ошибки

$$f(d) \to s_{err} \mid f \in F, \forall\, d \notin D' \,\&\, D' \neq \varnothing, \tag{10}$$

Если множество определения операции пусто, то выполнение операции не зависит от данных программы, что соответствует определению операции на всём множестве данных программы, и, следовательно, либо любое исполнение операции гарантированно приводит к достижению состояния ошибки в программе s_{err}, либо выполнение данной операции не может привести исполнение программы в состояние ошибки s_{err}.

Теорема 1. Если множество определения D' операции f_i не пусто и дополнение множества D' и множества значений данных программы $D_{\pi i}$, ограниченных функцией $\pi\ (x_i,\ v_i) = D_{\pi i}$, не пусто, то существуют такие значения внешних данных, исполнение программы на которых приведёт к достижению ошибочного состояния s_{err} в программе:

$$f_i(d_i) \to s_{err} \mid \forall\, d_i \in D_{err},\ D_{err} = D_{\pi i} \setminus D' \,\&\, D' \neq \varnothing\ . \tag{11}$$

Доказательство. Исходя из формулировки *леммы 2*, достижение состояния ошибки в программе возможно при выходе значения аргументов операции f_i за пределы множества области определения операции D'. Если символьные ограничения пути исполнения *SPC* наложенные на значения внешних данных программы описываемые функцией $\pi\ (x_i,\ v_i)$ формируют такое множество $D_{\pi i}$, что значения аргументов f_i попадают в множество допустимых значений данных программы $D_{\pi i}$, но не попадают в множество определения D' операции f_i, то, исходя из *леммы 2*, программа достигнет состояния ошибки s_{err}, что и требовалось доказать.

Следствие теоремы 1. Для того чтобы вычислить внешние данные программы, приводящие исполнение программы в состояние ошибки, необходимо и достаточно для каждого типа ошибки определить множество операций, исполнение которых может приводить к ошибке, и сформулировать

условие выхода значений аргументов операции за пределы множества определения этих операций и при этом входящих в множество значений данных программы.

Для иллюстрации покажем применимость *теоремы 1* для обнаружения нескольких видов ошибок.

3.1.2. Ошибка деления на ноль

Определение 7. *Операция целочисленного деления* – операция от двух аргументов, множество допустимых значений аргумента, соответствующего делимому, определено на всём множестве целых чисел, а множество допустимых значений аргумента, соответствующего делителю, определено на всём множестве целых чисел, за исключением элемента, соответствующего нулю:

$$f_{div}(d_{dividend}, d_{divisor}) \mid d_{dividend} \in D_{numbers}\ ,\ d_{divisor} \in D_{numbers} \setminus D_0, \qquad (12)$$

где:

$d_{dividend}$ – множество значений делимого;

$d_{divisor}$ – множество значений делителя;

$D_{numbers}$ – множество целых чисел;

D_0 – множество чисел, содержащее один элемент, равный нулю.

Теорема 2. Для определения ошибки деления на ноль для операции целочисленного деления необходимо и достаточно добавить в символьное предусловие операции f_{div} ограничение множества значений переменной, соответствующей делителю $x_{divisor}$, условие равенства нулю элементов этого множества:

$$f_{div}(d_{dividend}, x_{divisor}) \rightarrow s_{err} \mid d_{dividend} \in D_{numbers}\ ,\ (SPC\ \&\ (x_{divisor} = 0)) \neq \varnothing. \qquad (13)$$

Доказательство. Множество определения операции целочисленного деления в соответствии с *определением 7* совпадает с множеством чисел, за исключением нуля для делителя. Если символьное предусловие операции деления *SPC* позволяет вычисление значений множества внешних данных программы таких, что аргумент операции деления, соответствующий делителю, принимает значение ноль, то дополнение разности множеств ($D_{\pi i}$ \

D'_{div}) не пусто и включает ноль, следовательно, происходит нарушение множества определения операции деления в соответствии с *определением 7*, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки в программе s_{err}, что и требовалось доказать.

3.1.3. Ошибки работы с указателями на адрес в памяти

Здесь рассматривается модель обнаружения ошибок доступа к оперативной памяти в программе для современных операционных систем *Linux*, *FreeBSD*, *Microsoft Windows* и *Mac OS*. В общем случае правила работы с оперативной памятью зависят от реализации как аппаратной платформы, так и операционной системы. Например, в указанных выше операционных системах программе запрещён доступ к странице по нулевому адресу. Это трактуется как ошибка доступа к памяти, что в других операционных системах не обязательно будет трактоваться как ошибка.

Определение 8. *Указателем* будем называть переменную в программе, значение которой соответствует адресу в памяти.

Определение 9. Операцией *разыменования указателя* будем называть операцию f_{deref} взятия значения данных программы по адресу в памяти, заданного как аргумента операции разыменования. Областью определения операции разыменования D_{deref} являются адреса памяти, доступные программе.

Теорема 3. Для определения ошибки разыменования нулевого указателя необходимо и достаточно в символьное предусловие операции разыменования f_{deref} добавить условие равенства значения переменной указателя нулю:

$$f_{deref}(x) \to s_{err} \mid (SPC \ \& \ (x = 0)) \neq \varnothing. \qquad (14)$$

Доказательство. Исходим из того, что множество определения операции разыменования f_{deref} не включает в себя страницу памяти по нулевому адресу. Если символьное предусловие *SPC* операции разыменования позволяет вычисление значений множества внешних данных программы, таких, что аргумент операции разыменования принимает значение ноль, то дополнение разности множеств ($D_{\pi i} \setminus D'_{deref}$) не пусто и содержит ноль.

Происходит нарушение множества определения операции разыменования указателя, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки в программе s_{err}, что и требовалось доказать.

Определение 10. *Буфер в памяти* – область памяти, ограниченная адресом начала A_{begin} и адресом конца A_{end} блока данных программы в памяти.

Определение 11. Операцией *разыменования при доступе к буферу в памяти* будем называть операцию разыменования указателя, область определения которой D'_{buffer} ограничена адресом начала A_{begin} и адресом конца A_{end} данных программы в памяти.

Исходя из того, что подсистема управления динамической памятью операционной системы при выделении блока оперативной памяти программе производит выделение нового блока таким образом, что каждый новый выделенный блок памяти не пересекается по интервалу адресов начала и конца блока $<A_{begin}, A_{end}>$ с другими выделенными блоками памяти на момент его выделения. Также менеджер может выделить блок памяти, включающий адреса, попадающие в интервал адресов ранее освобождённых блоков.

Для контроля доступа к выделенным и освобождённым блокам памяти заведём два множества: множество выделенных блоков памяти D_{alloc} и множество освобождённых блоков памяти D_{freed}, каждое из которых в качестве элементов содержит описание блоков памяти в виде интервала адресов $<A_{begin}, A_{end}>$, где A_{begin} – адрес начала блока памяти, A_{end} – адрес конца блока памяти. При выполнении операции выделения блока в памяти интервал адресов $<A_{begin}, A_{end}>$ добавляется в множество выделенных блоков памяти D_{alloc}. При выполнении операции освобождения блока в памяти интервал адресов $<A_{begin}, A_{end}>$ удаляется из множества выделенных блоков памяти D_{alloc} и добавляется в множество освобожденных блоков памяти D_{freed}. Если возникает пересечение адресов нового выделенного блока с уже освобожденными блоками в множестве D_{freed}, то в множестве D_{freed} происходит преобразование интервалов адресов, которые пересекаются с выделенным блоком таким образом, чтобы

интервал адресов выделенного блока не пересекался ни с одним из интервалов в множестве освобождённых блоков памяти D_{freed}.

Теорема 4. Для определения ошибки доступа к буферу в памяти по указателю f_{dbuf} необходимо и достаточно в символьное предусловие операции добавить условие проверки значения переменной указателя меньше адреса начала буфера A_{begin} и больше адреса конца буфера A_{end}:

$$f_{dbuf}(d) \rightarrow s_{err} \mid (SPC \;\&\; (d < A_{begin} \;||\; d > A_{end})) \neq \varnothing. \qquad (15)$$

Доказательство. Множество определения операции доступа к буферу f_{dbuf} по *определению 11* ограничено адресом начала A_{begin} и адресом конца буфера A_{end}. Если символьное предусловие *SPC* операции разыменования f_{dbuf} позволяет вычисление значений множества внешних данных программы, таких, что аргумент операции разыменования принимает значение меньше адреса начала буфера A_{begin} или больше адреса конца буфера A_{end}, то дополнение разности множеств ($D_{\pi i} \setminus D'_{buffer}$) не пусто и содержит адрес, выходящий за границы буфера. В соответствии с *определением 11* происходит нарушение множества определения операции разыменования указателя при доступе к буферу, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки в программе s_{err}, что и требовалось доказать.

Теорема 5. Для определения ошибки доступа к блоку в памяти по указателю f_{deref} после освобождения блока необходимо и достаточно к символьному предусловию *SPC* операции f_{deref} добавить условие проверки, что в множестве освобождённых блоков D_{freed} существует интервал адресов $<A_{begin}, A_{end}>$ такой, что значение указателя больше либо равно адресу начала интервала A_{begin} и меньше либо равно адресу конца интервала A_{end}, а в множестве выделенных блоков памяти D_{alloc} нет интервала такого, что значение указателя больше либо равно адресу начала интервала A'_{begin} и меньше либо равно адресу конца интервала A'_{end}:

$$f_{deref}(d) \rightarrow s_{err} \left| \begin{array}{c} SPC \\ \& \\ \exists < A_{begin}, A_{end} > \in D_{freed} \mid (d > A_{begin} \,\&\, d < A_{end}) \\ \& \\ \nexists < A_{begin}, A_{end} > \notin D_{alloc} \mid \left(d > A_{begin} \,\&\, d < A_{end}\right) \end{array} \right. . \quad (16)$$

Доказательство. Доступ к блоку памяти сводится к доступу к буферу в памяти. Если символьное предусловие *SPC* операции разыменования f_{deref} позволяет вычисление значений множества внешних данных программы таких, что аргумент операции разыменования принимает значение не попадающее ни в один из интервалов адресов выделенных блоков памяти D_{alloc}, то в следствие *теоремы 4* происходит нарушение множества определения операции разыменования указателя при доступе к буферу, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки в программе s_{err}. При этом если аргумент операции разыменования принимает такое значение, что оно попадает хотя бы в один из интервалов адресов множества освобождённых блоков памяти D_{freed}, то происходит обращение к буферу памяти, который был освобождён ранее. Что и требовалось доказать.

Следствие теоремы 5. Операция повторного освобождения блока памяти сводится к операции доступа к освобождённому блоку памяти в функции освобождения блока динамической памяти, и для обнаружения ошибок повторного освобождения памяти может быть применена *теорема 5* с наложением дополнительного ограничения на множество операций, являющихся вызовом функций операционной системы, предназначенных для освобождения блока или множества блоков динамической памяти.

3.1.4. Ошибки использования неинициализированных переменных

Стандарты языков программирования Си и Си++ не гарантируют инициализацию переменных в программе. Язык Си++ предоставляет возможность определения начального значения переменной при её объявлении, но не обязывает это делать. Использование неинициализированных переменных приводит к неопределённому поведению

программы. Соответственно, если алгоритм программы не предусматривает возможность использования неинициализированной переменной со случайным значением, то его исполнение может привести к достижению ошибочного состояния s_{err} в программе.

Определим множество X_{init}, элементами которого являются переменные, которым было присвоено значение адреса памяти в процессе исполнения программы. Когда в программе встречается операция инициализации переменных, в множество X_{init} добавляется элемент, соответствующий инициализированной переменной. Когда область памяти, соответствующая переменной, становится недоступной для вычислений, например при выходе из подпрограммы для переменных, размещённых на стеке, из множества X_{init} удаляется элемент, соответствующий этой переменной. Если ограничения пути позволяют достигнуть состояния использования в операции $f(x)$ значения переменной, не входящей в множество инициализированных переменных, то в общем случае поведение программы не определено и может достичь состояния ошибки s_{err}.

Определение 12. Операцией получения значения переменной $f_{read}(x)$, где $f_{read} \in F$, а $x \in X$ – множеству данных программы, будем называть операцию получения значения данных программы, содержащихся по адресу переменной x. Множеством определения операции получения значения является множество адресов, соответствующих переменным $x \in X_{init}$, которым ранее было присвоено значение данных программы.

Теорема 6. Для обнаружения ошибки использования неинициализированных переменных необходимо и достаточно в предусловие операции, использующей переменную, добавить условие наличия адреса переменной в множестве переменных, которым присвоено значение:

$$f_{read}(x) \rightarrow s_{err} \mid SPC \;\&\; (x \notin X_{init}). \qquad (17)$$

Доказательство. Если символьное предусловие операции SPC получения значения переменной программы $f_{read}(d)$ позволяет вычислить такой набор внешних данных, что на пути исполнения нет операции

присвоения данных программы, и, следовательно, переменная d не входит в множество переменных, которым ранее было присвоено значение, то происходит нарушение области определения операции взятия значения переменной, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки s_{err} в программе. Что и требовалось доказать.

3.1.4. Ошибки неправильного завершения работы с ресурсом

Как отмечалось в 2.2.4, неправильное завершение работы с ресурсом или неправильное его освобождение могут значительно повлиять на возможность дальнейшего предоставления данного ресурса в процессе работы программы, на производительность работы программы и операционной системы в целом. Основные причины ошибок работы с ресурсами заключаются в потере значения определителя ресурса до вызова функции операционной системы, освобождающей ресурс. Таким образом, для обнаружения ошибок завершения работы и освобождения ресурса необходимо обнаруживать нарушения в последовательности операции присвоения значения или удаления переменных, содержащих значение определителя ресурсов, и нарушения в последовательности вызова функций получения и освобождения ресурса. В качестве примера ошибки типа «утечка ресурса» можно привести перезапись или уничтожение на стеке указателя на блок памяти на куче до его освобождения при выходе из подпрограммы, присвоение его значения глобальной переменной или возвращаемому значению функции. К этому же классу ошибок относится отсутствие операции закрытия файла, сетевого сокета или другого ресурса операционной системы до потери значения определителя ресурса.

Определение 13. Операция присвоения значения переменной $f_{write}(x, d)$ производит запись значения данных программы d по адресу переменной x. Зададим множество выделенных ресурсов операционной системы как D_{ralloc}, а множество переменных, содержащих данные, соответствующие определителям ресурсов, как X_{descr}. Область определения операции присвоения $f_{write}(x, d)$ задаётся формулой:

$$x \in X_{descr} \,\&\, f_{read}(x) \in D_{ralloc} \,\&\, \exists\, x' \in X_{descr} \mid x \neq x' \,\&\, f_{read}(x) = f_{read}(x').\ (18)$$

Определение 14: Операция удаления переменной $f_{destroy}(x)$ производит исключение значения данных программы d по адресу переменной x из множества данных программы D. Зададим множество выделенных ресурсов операционной системы как D_{ralloc}, а множество переменных, содержащих данные, соответствующие определителям ресурсов, как X_{descr}. Область определения операции удаления переменной $f_{destroy}(x)$ задаётся формулой:

$$x \in X_{descr} \,\&\, f_{read}(x) \in D_{ralloc} \,\&\, \exists\, x' \in X_{descr} \mid x \neq x' \,\&\, f_{read}(x) = f_{read}(x').\ (19)$$

Теорема 7. Для обнаружения ошибок освобождения ресурса операционной системы необходимо и достаточно к символьному ограничению пути SPC для операций записи f_{write} или удаления $f_{destroy}$ переменной x добавить условие проверки наличия значения переменной $f_{read}(x)$ во множестве выделенных ресурсов операционной системы D_{ralloc} и отсутствия ещё хотя бы одной переменной x', принадлежащей множеству переменных, хранящих значение определителя ресурса операционной системы X_{descr}, равное значению записываемой переменной:

$$f_{write}(x) \rightarrow s_{err} \mid SPC \,\&\, (f_{read}(x) \in D_{ralloc}) \,\&\, \nexists\, x' \in X_{descr} \mid f_{read}(x') = f_{read}(x).\ (20)$$

Доказательство. Если символьное предусловие выполнения операции перезаписи или уничтожения переменной SPC позволяет вычислить такой набор внешних данных программы $d \in D$, что выполнение происходит по пути, содержащему операцию записи $f_{write}(x)$ или уничтожения переменной $f_{destroy}(x)$, со значением определителя ресурса $f_{read}(x) = d \in D_{ralloc}$, и нет такой переменной $x' \in X_{descr}$, значение которой равно значению определителя ресурса, то происходит нарушение множества определения операции записи, что, согласно *теореме 1*, приводит к достижению состояния ошибки s_{err} в программе, что и требовалось доказать.

3.2. Алгоритм комбинированного анализа

3.2.1. Общий алгоритм комбинированного анализа

Для реализации метода классификации предупреждений об ошибках решены несколько задач, и для их решения разработан набор алгоритмов, позволяющих достичь цели классификации предупреждений о программных ошибках. Результаты разработки алгоритмов комбинированного анализа изложены в публикациях автора данной диссертации [169, 170, 171].

Общий алгоритм, лежащий в основе предлагаемого метода, состоит из нескольких этапов.

Алгоритм 1. Общий алгоритм комбинированного анализа:

1. Обнаружение потенциальных ошибок в программе методами статического анализа.
2. Статический анализ исполняемого кода программы для получения сокращённого графа потока программы с целью вычисления путей, содержащих трассу событий для каждой найденной с помощью статического анализа ошибки.
3. Вычисление расстояния от каждого базового блока программы, лежащего на путях, вошедших в сокращённый граф потока программы, до точки на трассе событий для каждой ошибки, найденной с помощью статического анализа.
4. Реализация направленного динамического символьного исполнения программы с целью генерации внешних данных, приводящих к исполнению программы по путям, достигающим трассу ошибки.
5. Генерация внешних данных программы, нарушающих предикат безопасности в точке реализации ошибки.
6. Проверка проявления ошибки в процессе исполнения программы на внешних данных, сгенерированных на этапе 5.

Реализация методов статического анализа программ выходит за рамки данной работы, в связи с чем информация об ошибках, найденных методами статического анализа, будет использоваться как исходные данные для описанного метода.

3.2.2. Алгоритм построения путей, достигающих точки проявления ошибки

В случае, если трасса ошибки получена от инструмента статического анализа исходного кода программ, требуется решить задачу сопоставления трассы исполнения программы в исходном тексте на языке высокого уровня и в машинном коде.

Алгоритм 2. Алгоритм построения сокращённого межпроцедурного графа потока программы:

1. Сначала строится *сокращённый граф вызовов подпрограммы*. Построение начинается с подпрограмм, в которых находятся точки проявления ошибки в программе. Далее граф дополняется подпрограммами, вызывающими подпрограммы, содержащие точки проявления ошибки в программе. Построение графа продолжается до тех пор, пока не будет достигнута подпрограмма, содержащая точку входа в программу или главную функцию вычислительного потока. Таким образом из машинного кода программы извлекается *сокращённый граф вызовов*, содержащий только те подпрограммы, исполнение которых необходимо для проявления ошибок, предупреждения о которых получены от инструмента статического анализа программ.
2. Далее для подпрограмм, вошедших в *сокращённый граф вызовов*, производится статическое извлечение *сокращённого межпроцедурного графа потока*, содержащего только те пути исполнения, которые включают трассу предупреждения об ошибке, полученной от инструмента статического анализа программ.
3. После получения *сокращённого межпроцедурного графа потока программы* производится вычисление числовой метрики расстояния, выражающейся в количестве условных переходов от каждого базового блока программы до каждой точки трассы событий предупреждения об ошибке в программе. В дальнейшем расстояние

от базового блока до точки в программе используется как численная оценка при выборе следующего пути исполнения для анализа: сначала выбираются пути, для которых эта оценка меньше.

По результатам выполнения алгоритма строятся структуры данных, необходимые для проведения направленного анализа в процессе динамического символьного исполнения программы.

3.2.3. Алгоритм направленного динамического символьного исполнения программы

Для вычисления набора внешних данных программы, приводящих к исполнению по пути, содержащему трассу предупреждения об ошибке, полученную от инструмента статического анализа, используется метод направленного итеративного динамического символьного исполнения программы.

Алгоритм 3. Алгоритм направленного динамического символьного исполнения:

1. Производится топологическая сортировка точек событий на трассе предупреждения об ошибке на графе потока программы.
2. На первой итерации анализа программа запускается на выполнение с произвольным набором внешних данных.
3. Производится сбор трассы исполнения программы и преобразование её в символьную формулу, в которой внешние данные программы представлены свободными (символьными) переменными.
4. Если путь исполнения в программе не прошёл по трассе событий предупреждения об ошибке в программе, то производится инвертирование условия для условного перехода, находящегося в базовом блоке, с наименьшей оценкой расстояния до следующей точки на трассе событий предупреждения об ошибке в программе.
5. Вычисляется новый набор внешних данных программы с целью проведения исполнения программы по новому пути.

6. Производится запуск программы на новом наборе внешних данных и выполнение алгоритма продолжается с этапа 3 до тех пор, пока следующая точка на трассе событий предупреждения об ошибке не будет достигнута или не останется путей, достигающих следующую точку на трассе предупреждения об ошибке.
7. Работа алгоритма завершается при достижении последней точки трассы событий предупреждения об ошибке – точки проявления ошибки.

Рис. 3 иллюстрирует работу алгоритма. Серым цветом обозначены базовые блоки на пути исполнения программы на текущей итерации. Чёрным цветом обозначен базовый блок, содержащий операцию на трассе событий предупреждения об ошибке в программе. От блока *A* расстояние до

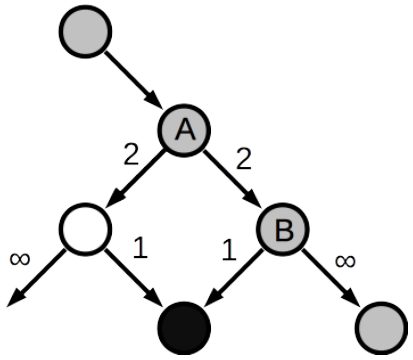


Рисунок 3. Схема алгоритма направленного анализа

интересующего базового блока равно двум переходам. От блока *B* – одному переходу. Соответственно сначала для инвертирования будет выбран условный переход в базовом блоке *B*, а затем – в базовом блоке *A*.

Использование алгортима направленного динамического символьного исполнения позволяет значительно уменьшить влияние проблемы экспоненциального роста количества путей для анализа через ограничение количества исследуемых путей исполнения в программе набором путей, приводящих исполнение в точку проявления ошибки.

3.2.4. Проверка нарушения предиката безопасности

После того как построен набор внешних данных, на котором происходит исполнение программы по пути, содержащему трассу событий

предупреждения об ошибке, производится дополнение предиката пути условием нарушения предиката безопасности операции. Если условия в предикате пути и инвертированном предикате безопасности совместны, то вычисляются внешние данные программы, которые приводят к нарушению предиката безопасности операции и, следовательно, к проявлению ошибки. Если внешние данные программы, на которых проявляется ошибка в программе, построены, то это означает подтверждение ошибки в программе.

3.3. Классы предупреждений об ошибках

В настоящем исследовании предлагается метод классификации предупреждений об ошибках инструмента статического анализа. Метод классификации построен на базе модели обнаружения ошибок в программе и алгоритмах направленного динамического символьного исполнения, позволяющих вычислить внешние данные программы, на которых происходит проявление ошибки в программе. Метод классификации предупреждений об ошибках инструмента статического анализа предназначен для разделения всех предупреждений об ошибках на три непересекающихся класса:

1. Подтверждённые предупреждения: для их подтверждения удалось построить внешние данные.
2. Неподтверждённые предупреждения: для них удалось показать несовместность условий на трассе событий.
3. Предупреждения, для которых не удалось подобрать внешние данные программы, проводящие исполнение по трассе предупреждения об ошибке, и не удалось доказать несовместность условий на трассе событий.

Для предупреждений об ошибках, попавших в первый класс, вычислен набор внешних данных программы, при подаче которых на вход программе удаётся проявить ошибку в программе. Вычисленный набор внешних данных можно использовать для отладки программы и для обучения инструментов классификации и ранжирования, основанных на алгоритмах машинного

обучения. Программные ошибки, предупреждения о которых попали в первый класс, требуют исправления.

Предупреждения об ошибках второго класса, для которых удалось доказать методами решения формул в теориях несовместность условий на трассе событий в программе от точки инициализации ошибочной ситуации до точки проявления ошибки, являются явными ошибками первого рода (ложноположительными утверждениями об ошибке). Такие предупреждения могут быть использованы для отладки статического анализатора и как выборка для обучения инструментов классификации и ранжирования, основанных на алгоритмах машинного обучения.

Для предупреждений об ошибках, попавших в третий класс, требуется инспекция предупреждений специалистом.

Данная классификация позволяет ограничить множество предупреждений об ошибках статического анализатора, которое необходимо исследовать специалисту, и, следовательно, снижает трудозатраты высококвалифицированного специалиста. Метод классификации изложен в работе автора данной диссертации [172].

Предложенные алгоритмы и метод не зависят от реализации методов статического анализа программ, что позволяет применять их совместно с инструментами статического анализа программ как для исходного текста программы, так и для исполняемого кода программ.

Глава 4. Реализация и оценка применения комбинированного анализа программ

4.1. Модуль сопоставления трассы событий предупреждения об ошибке

Реализация и оценка предложенного метода производилась на базе инструментов статического анализа исходного текста программ *Svace* и инструменте динамического символьного исполнения *Anxiety* [173], разрабатываемых в Институте системного программирования им. В. П. Иванникова Российской академии наук.

Инструмент статического анализа исходного кода программ *Svace* представляет собой ядро анализа программ, реализующее:

- алгоритмы анализа абстрактного синтаксического дерева для легковесного анализа исходного кода программ на языках Си и Си++;
- алгоритмы анализа потока программы, чувствительного к путям и контексту при проведении межпроцедурного анализа;
- набор анализаторов, предназначенных для обнаружения ошибок различных типов в исходном коде программ.

В данной работе не рассматриваются алгоритмы статического анализа исходного кода программ и результат работы статического анализатора используется как исходные данные для работы алгоритма направленного анализа.

Для реализации алгоритмов направленного анализа в инструмент статического анализа Svace был добавлен модуль сопоставления операндов выражений условных переходов с целью сокращения количества путей, которые необходимо исследовать методом динамического символьного исполнения. Модуль построен как независимый подключаемый модуль (англ. plugin) для компилятора *Clang* [174] и использует абстрактное синтаксическое дерево для анализируемого компилируемого модуля программы с целью получения позиций операндов условных выражений в операторах условного перехода и циклах. Далее в процессе проведения направленного динамического символьного исполнения данная информация используется

для фиксации конкретного операнда условного выражения, чтобы предотвратить инвертирование данного условия на уровне машинного кода и зафиксировать путь исполнения через это условие в направлении, указанном в трассе событий предупреждения об ошибке.

Для решения задачи сопоставления трассы событий предупреждения об ошибке в исходном тексте программы и машинном коде программы производится дизассемблирование программы. Далее из ассемблерного кода выделяются подпрограммы, которые в свою очередь делятся на базовые блоки, то есть последовательности инструкций программы, имеющие одну точку входа с определённым адресом и одну точку выхода и не содержащие операции переходов. Информация о программе сохраняется в виде графа вызовов. Граф вызовов представляет собой ориентированный граф, в котором вершинам соответствуют подпрограммы, а рёбрам – операции вызовов подпрограмм. Каждая подпрограмма идентифицируется адресом точки входа в подпрограмму. Далее код каждой подпрограммы сохраняется в виде ориентированного графа, в котором вершинами являются базовые блоки, а рёбрам соответствуют переходы между базовыми блоками. Для каждого базового блока сохраняется адрес первой инструкции блока и его длина. Данная структура, при наличии информации о соответствии строк и адресов, позволяет сопоставить трассу исполнения в исходном коде программы трассе исполнения в машинном коде.

Прежде всего стоит учесть, что количество путей в машинном коде больше, чем в исполняемом коде. Это связано с тем, что сложные условные выражения порождают дополнительные пути в машинном коде, связанные с необходимостью вычисления подвыражений сложных условных выражений.

Пример, иллюстрирующий описанную ситуацию, приведён на листинге 10. В машинном коде вычисление сложного условия будет разбито на несколько базовых блоков. Один – для вычисления условия *A*, другой – для вычисления условия *B*. Соответственно, для приведенного примера двум

путям исполнения в исходном тексте программы будут соответствовать три пути исполнения в машинном коде.

Листинг 10. Программа, для которой в исполняемом коде будет больше путей для анализа, чем в исходном тексте программы

```
if (A || B)
{ …
} else
{ …
}
```

```
  4004f6:   cmp   $0x0,-0x4(%rbp)
  4004fa:   jne   400502           ; A
  4004fc:   cmp   $0x0,-0x3(%rbp)
  400500:   je    400515           ; B
  400502:   ; then-branch
  ...
  400515:   ; else-branch
```

К тому же возможны разные варианты переходов в машинном коде в зависимости от условного выражения и операций в программе, содержащихся внутри `then-` и `else-` веток (листинг 11).

Листинг 11. Варианты генерации кода для условных переходов

```
if(A)
{
    // then-ветка
}
else
{
    // else-ветка
}
```

```
; Вариант 1: переход
; на then-ветку

4004fc: cmp $0x0, -
0x3(%rbp)
400500: jne 400515
…
400515: ; then-ветка
```

```
; Вариант 2: переход
; на else-ветку

4004fc: cmp $0x0, -
0x3(%rbp)
400500: je 400515
…
400515: ; else-ветка
```

Правила генерации кода для сложных условий с конъюнкциями, дизъюнкциями и отрицаниями для версий компилятора *GCC 4.6.3* и *4.9.2* для платформы x86-64 приведены в таб. 3.

Таблица 3. Варианты генерации условных переходов

Операция	Вариант генерации исполняемого кода
A && B	• переход на then-ветку: вариант 2 для A и вариант 1 для B; • переход на else-ветку: вариант 2 для A и вариант 2 для B
A || B	• переход на then-ветку: вариант 1 для A и вариант 1 для B; • переход на else-ветку: вариант 1 для A и вариант 2 для B
!A	• переход на then-ветку: вариант 2 для A; • переход на else-ветку: вариант 1 для A.

Для каждого предупреждения об ошибке производится разбор трассы событий. Для каждой точки на трассе событий, которая выражена в виде позиции в исходном тексте программы, определяется базовый блок, в который попадает событие из трассы. Для этого используется информация о сопоставлении исполняемого кода и исходного текста программы: на операционной системе *Linux* используется секция объектного файла *ELF* (англ. Executable and Linkable Format – формат для исполнения и связывания) с информацией о разметке кода *DWARF* (англ. Debugging With Attributed Record Format – формат для отладки с атрибутированными записями), на операционной системе *Windows* используется информация из файлов с разметкой в формате *PDB* (англ. Program Database – база данных программы). Далее производится определение, каким подпрограммам соответствует каждый базовый блок, содержащий точку события из трассы событий предупреждения об ошибке. После чего производится топологическая сортировка подпрограмм на графе вызовов с целью построения путей на графе вызовов от точки входа в программу или в процедуру вычислительного потока (англ. thread) до первой подпрограммы, в которой находится первый базовый блок, содержащий событие трассы предупреждения об ошибке. Далее строятся пути на графе вызовов до всех подпрограмм, содержащих базовые блоки, в которых расположены точки событий трассы предупреждения об ошибке. Таким образом строится сокращённый граф вызовов для предупреждения об ошибке.

Далее для каждой подпрограммы производится вычисление путей, проводящих исполнение через базовые блоки событий трассы предупреждения об ошибке и через базовые блоки, содержащие операции вызова подпрограмм, вошедших в сокращённый граф вызовов. Таким образом строится сокращённый межпроцедурный граф потока программы.

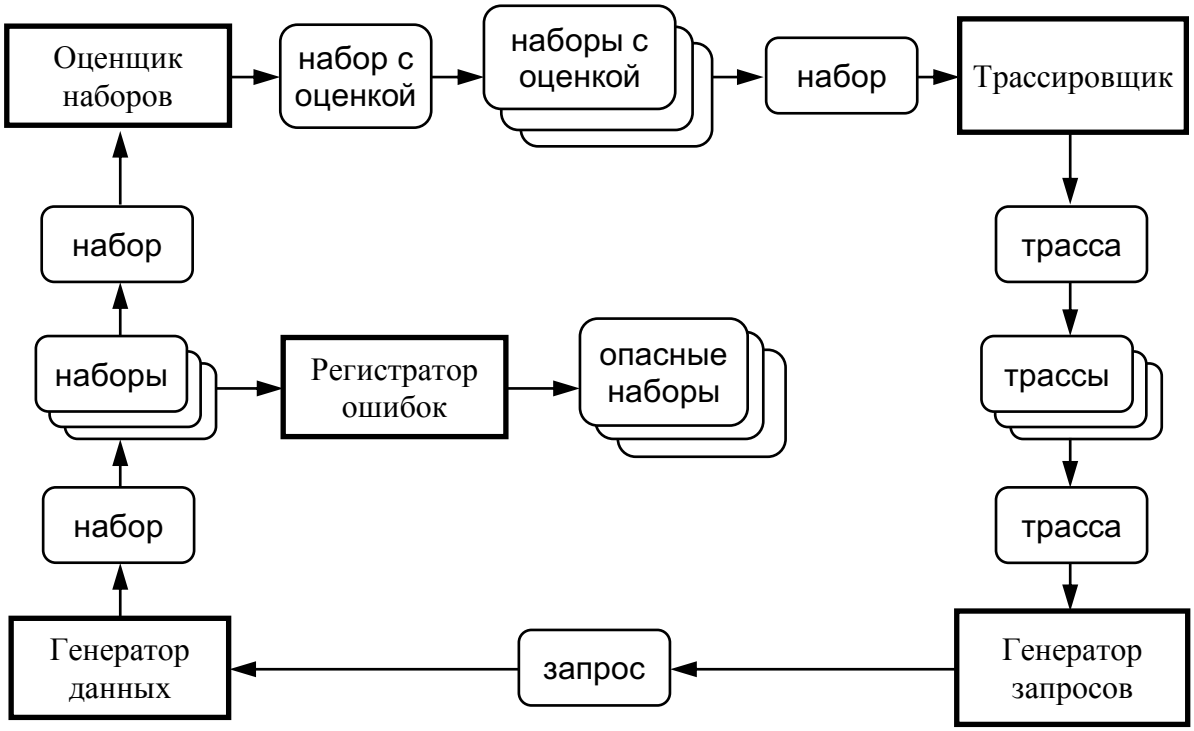
После построения сокращённого графа потока программы для каждого базового блока, соответствующего вершине графа, производится вычисление и сохранение расстояния в виде количества переходов от базового блока до каждого из базовых блоков, содержащих точку события из трассы предупреждения об ошибке. Информация о расстоянии используется в процессе динамического символьного исполнения программы для выбора следующего пути для анализа, и, таким образом, реализуется алгоритм направленного динамического символьного исполнения.

4.2. Инструмент динамического символьного исполнения программ

4.2.1 Архитектура инструмента Anxiety

Инструмент динамического символьного исполнения программ *Anxiety*, разработанный под руководством автора данной диссертационной работы в

Рисунок 4. Модульная схема инструмента *Anxiety*



Институте системного программирования им. В.П. Иванникова Российской академии наук, представляет собой систему анализа приложений пользовательского режима для операционных систем семейств *Linux* и *Windows*, реализованную в виде модульной архитектуры и построенную на базе динамической инструментации программ, символьных вычислений для представления семантики операций и решателей формул в теориях логики первого порядка для целочисленной арифметики, битовых векторов, массивов и др. Описание инструмента *Anxiety* приведено в публикации автора [173].

Модульная архитектура инструмента *Anxiety* рис. 3 позволяет решить две системных задачи:

1. Абстракция входных и выходных интерфейсов каждого модуля позволяет решить задачу замены реализации модулей, не меняя общий алгоритм работы инструмента:
   - для решения задачи сбора трассы исполнения программ, исполняющихся на различных операционных системах и аппаратных платформах, достаточно реализовать специализацию *модуля трассировщика* без изменения остальных модулей инструмента;
   - для подключения нового инструмента решения формул в теориях достаточно реализовать специализацию *модуля генерации набора данных* для каждого поддерживаемого инструмента решения формул в теориях без необходимости изменения алгоритмов работы других модулей;
   - для реализации возможности использования альтернативных алгоритмов выбора следующего пути для анализа достаточно реализовать специализации *модуля оценки набора данных*;
   - для реализации алгоритмов обнаружения ошибок достаточно реализовать специализацию модуля *детектора ошибок*.
2. Независимость алгоритмов работы модулей друг от друга позволяет реализовать работу модулей в параллельном режиме и использовать

возможности распределённых вычислений путем выделения отдельных вычислительных потоков или узлов в облаке для каждого модуля.

Разработанное архитектурное решение для инструмента динамического символьного исполнения позволяет минимальными усилиями производить поддержку новых аппаратных платформ и операционных систем, а также использовать возможности современных параллельных и распределённых вычислительных инфраструктур.

4.2.2. Модуль трассировки исполнения программы.

Модуль предназначен для сбора трассы исполнения программы в виде выполненных операций. Может быть построен на основе сред динамической бинарной трансляции или динамической инструментации. В базовой реализации трассировщик использует среду динамической трансляции программ *Valgrind* [97] для поддержки операционных систем на базе ядра *Linux 2.4* и систему динамической инструментации программ *DynamoRIO* [142] для современных версий операционных систем семейства *Windows* и *Linux*.

Модуль состоит из двух компонентов:

- компонент динамической инструментации исполняемого кода производит декодирование машинных команд, отправляемых на исполнение, и позволяет получить состояние программы до и после исполнения команды;
- компонент пометки внешних данных программы и распространения пометки по потоку программы.

Компонент динамической инструментации исполняемого кода проводит «тяжеловесную» инструментацию всех операций программы и преобразование их в формат среды динамического анализа *Triton* [155] для сбора трассы исполнения в виде символьной формулы. При обработке операций получения внешних данных программы компонент динамической инструментации исполняемого кода производит замену потока внешних

данных, полученных при инициализации анализа в качестве начального набора данных или вычисленных по результатам одной из предыдущих итераций анализа, с целью проведения анализа по новому пути.

В качестве источников внешних данных для программы инструментом *Anxiety* поддерживаются:

- стандартный поток ввода программы;
- переменные окружения;
- параметры командной строки;
- файлы;
- сокеты.

При отслеживании потока помеченных (англ. tainted) данных программ используется компонент пометки внешних данных и распространения помеченности данных по потоку программы. Для решения проблемы избыточной помеченности (англ. overtainting – перепомеченность) помечаются и, соответственно, попадают в символьную формулу только те операции, аргументы которых либо являются внешними данными программы, либо являются результатом операций над внешними данными программы. Такой подход позволяет контролировать поток помеченных данных программы и снизить сложность символьной формулы путём исключения потока данных программы, не зависящего от внешних данных программы. С другой стороны, данный подход приводит к усложнению алгоритмов распространения помеченности по потоку данных программы и к возникновению проблемы недостаточной помеченности потока данных (англ. undertaintedness – недопомеченность), что, в итоге, приводит к обрыву потока помеченных данных, связанному с наличием в программе косвенных зависимостей потока помеченных данных по управлению и по данным.

Пример косвенных зависимостей по управлению приведён на листинге 12.

Листинг 12. Пример косвенной зависимости по управлению для помеченного потока данных

```
1: char * dbz_error(size_t n)  // Значение переменной dev
2: {                           // косвенно зависит от внешних
3:     int dev = 1;            // данных функций dbz_error и
4:     if(n > 0)               // получит нулевое значение, если
5:         dev = dev - 1;      // значение аргумента n будет
6:     return n/dev;           // больше нуля, что приведёт
7: }                           // к ошибке деления на ноль
```

Пример косвенных зависимостей по данным в виде табличных преобразований приведён на листинге 13 .

Листинг 13. Пример косвенной зависимости по данным для помеченного потока данных

```
1: #include <stdbool.h>          // Возвращаемое значение функции
2: bool check_even(int n)        // check_even косвенно зависит
3: {                             // от таблицы значений tab
4:     bool tab[] =              // обращение к таблице
5:        { true, false }        // происходит по помеченному
6:     return tab[n % 2];        // индексу
7: }
```

Разработка методов, учитывающих косвенные зависимости по управлению и по данным, является перспективным направлением исследований в области динамического символьного исполнения.

К тому же нередко в программах используются возможности косвенного вызова функций (использование виртуальных методов, использование вызова функций по указателю), а оптимизирующие компиляторы могут реализовывать оператор выбора как передачу управления с использованием таблицы адресов. В этом случае возникают две проблемы:

- неявная зависимость потока программы от помеченных данных. Например, в программе может инициализироваться указатель на функцию обработки внешних данных в зависимости от значения аргумента командной строки или от значения в заголовке формата;

- для программ, использующих переходы по вычисляемым адресам, статическое построение графа вызовов и потока программы становится затруднительным.

Необходимость решения данных проблем динамического символьного исполнения программ описывалось в работах [115, 91] и является первоочередным направлением для дальнейших исследований в области символьного исполнения программ.

4.2.3. Модуль генерации запросов

В процессе обработки операций условных переходов в символьную трассу, соответствующую исполнению программы, вставляются специальные маркеры для условий, которые можно инвертировать, и, таким образом, получить формулу для вычисления внешних данных программы, которые, в случае вычислимости позволят исполнить программу по альтернативному пути для данного условного перехода.

Модуль разделения трассы предназначен для преобразования полученной трассы в виде символьной формулы в несколько трасс с целью получения внешних данных, соответствующих альтернативному пути исполнения на условных переходах, встретившихся в процессе исполнения программы по анализируемому пути исполнения. Например, трасса:

$op_1 \to op_2 \to j_1 \to op_3 \to op_4 \to j_2 \to ... \to op_i \to op_{i+1} \to j_k \to \ldots$

преобразуется в несколько трасс:

1. $op_1 \to op_2 \to \neg j_1$;

2. $op_1 \to op_2 \to j_1 \to op_3 \to op_4 \to \neg j_2$;

...

k. $op_1 \to op_2 \to j_1 \to op_3 \to op_4 \to j_2 \to ... \to op_i \to op_{i+1} \to \neg j_k$.

Таким образом, на каждой итерации анализа можно сгенерировать k трасс, соответствующих альтернативным путям исполнения программы.

В работе разделителя трасс учитывается информация о том, начиная с какого условного перехода необходимо проводить разделение трассы, чтобы избежать повторной обработки трасс, проанализированных на предыдущих

итерациях. Также учитывается, сколько из далее встретившихся на пути исполнения условных переходов необходимо инвертировать с целью контроля распределения нагрузки между модулями инструмента на каждой итерации анализа и производительности анализа в целом. Каждая полученная трасса преобразуется в запрос к решателю ограничений для вычисления внешних данных программы, на которых исполнение пойдёт по новому пути, в случае совместности ограничений.

4.2.4. Модуль генерации данных

Модуль генерации данных предназначен для вычисления внешних данных программы в соответствии с формулами, полученными от генератора запросов. Вычисление внешних данных производится при помощи решателей формул в теориях для вычисления конкретных значений свободных переменных, удовлетворяющих ограничениям в символьных формулах. Модуль поддерживает подключение различных решателей формул в теориях, поддерживающих формулы в формате *CVC* [149] и *SMT-LIB* [150]. В базовой реализации используется решатель *Z3* [151]. После вычисления значений свободных переменных в формулах формируется новый набор внешних данных программы, который в дальнейшем проходит оценку в модуле оценки путей исполнения с целью использования оценки набора для выбора пути на следующих итерациях анализа и для использования в модуле проверки и регистрации ошибок.

4.2.5. Модуль оценки путей для анализа

Модуль используется для расчёта оценки пути исполнения, соответствующего набору данных программы. Поддерживается расчёт таких оценок, как прирост покрытия по базовым блокам, длина пути исполнения и др.

Для оценки пути делается легковесная инструментация, которая собирает информацию о базовых блоках, пройденных в процессе исполнения по пути, соответствующему оцениваемому набору данных, и вычисляет оценку пути в зависимости от алгоритма, заданного при запуске инструмента

Anxiety. Для расчёта оценки путь исполнения оценивается по следующим алгоритмам:

- максимальный прирост количества базовых блоков на пути исполнения, через которые не происходило исполнение программы на предыдущих итерациях анализа;
- направленный анализ, использующий информацию об адресах в программе, через которые необходимо построить путь исполнения программы.

Для решения задачи направленного динамического символьного исполнения используется информация, построенная на этапе статического анализа машинного кода программы, с целью проведения исполнения программы через базовые блоки, содержащие события трассы предупреждения об ошибке, полученной от инструмента статического анализа. В процессе исполнения программы производится оценка расстояния от базовых блоков, лежащих на пути исполнения текущей итерации анализа до следующей точки на пути исполнения, приводящего к исполнению программы через точку следующего события на трассе предупреждения об ошибке. Для инвертирования условного перехода на пути выбирается та операция условного перехода, для которой расстояние от базового блока, предшествующего ей, минимально. Таким образом реализуется алгоритм наискорейшего достижения точки реализации дефекта. При этом в процессе динамического символьного исполнения фиксируются направления переходов, заданные событиями на трассе предупреждения об ошибке.

4.2.6. Модуль регистрации ошибок

Модуль предназначен для регистрации и проверки ошибок в программе и сохранения внешних данных программы, на которых произошла регистрация ошибки. Для регистрации ошибок используются два подхода:

1. *Регистрация аварийного завершения программы*. Если в процессе исполнения программы системой регистрируется выполнение операции, которая программе не разрешена, например доступ к

странице памяти, которая программе не принадлежит (SIGSEGV), или аргумент операции не попадает в область определения операции, например ноль в делителе операции деления (SIGFPE), или в программе вызывается операция немедленного завершения работы программы (SIGABRT), то система извещает программу о том, что произошла исключительная ситуация системного уровня. Если в программе не установлены обработчики системных исключительных ситуаций, то происходит аварийное завершение работы программы, что является сигналом о наличии ошибки в программе.

2. *Проверка нарушения предиката безопасности*. Некоторые виды ошибок не нарушают системные правила, которые приводят к появлению исключительной ситуации системного уровня. В таком случае требуется проверка, что операция в программе приводит к нарушению области определения операции. Например, осуществляется выход за границы буфера, выделенного на куче или на стеке. В таком случае требуется проверка определённых условий, например попадания адреса, по которому программа пытается считать или записать значение в множество разрешённых адресов.

Если модуль определяет, что при запуске программы с набором внешних данных происходит регистрация ошибки одним из двух доступных способов, то такой набор сохраняется для демонстрации наличия ошибки в программе и отладки программы. Если в процессе направленного исполнения программы обнаруживается ошибка времени исполнения, приводящая к аварийному завершению программы, но не в месте, указанном инструментом статического анализа программ, то такие внешние данные также сохраняются для дальнейшего исследования причины аварийного завершения программы.

4.2.7. Ограничения реализации

Несмотря на подтверждённое успешное применение инструмента *Anxiety* для поиска ошибок в программах, у инструмента есть известные

ограничения, присущие инструментам, реализующим динамическое символьное исполнение программ. Данные ограничения приводят к ожидаемым недостаткам анализа программ.

Недостаточная помеченность потока данных программы. Как отмечалось ранее, косвенные зависимости по управлению и по данным не позволяют распространять поток помеченных данных на операции, косвенно зависящие от внешних данных программы, что отрицательно сказывается на точности результатов работы инструмента.

Ограниченная поддержка переходов по вычисляемому адресу. Наличие в коде программы переходов по вычисляемым адресам, например при использовании виртуальных функций, вносит неточности в построенный граф вызовов программы, что, в свою очередь, отрицательно влияет на достижимость трассы событий предупреждения об ошибке.

Ограниченная поддержка проверки условий проявления ошибок. На данный момент в инструменте реализована поддержка проверки безопасности операций деления и экспериментальная реализации нарушения границ буферов в памяти. Есть возможность целенаправленно вычислять значение делителя в ноль, если делитель зависит от помеченного потока данных программы, и целенаправленно вычислять внешние данные программы, на которых происходит нарушение границ буферов в памяти для операций доступа к буферам в памяти для ограниченного набора операций. Разработка методов целенаправленной проверки предикатов безопасности для различных видов ошибок в программах также является перспективным направлением развития возможностей инструмента.

Отсутствие механизма выделение под-трассы, соответствующей пути от точки инициализации ошибочной ситуации до точки проявления ошибки. На данный момент не реализован алгоритм выделения трассы для проверки несовместности условий на пути от точки инициализации ошибочной ситуации до точки проявления ошибки.

Отсутствие реализации данного алгоритма не позволяет произвести классификацию предупреждений об ошибке во второй класс – класс ложноположительных предупреждений об ошибке.

Несмотря на наличие ограничений, описанных выше, инструмент успешно применяется для обнаружения ошибок в программах в процессе сертификации программного обеспечения для использования на территории Российской Федерации. Инструментом *Anxiety* найдено и зарегистрировано в базе ФСТЭК более тридцати уникальных ошибок в программах для операционной системы *Astra Linux*.

4.3. Методы оценки инструментов анализа программ

Среди методов оценки качества инструментов анализа программ стоит отметить следующие.

Применение тестовых наборов, созданных разработчиками инструмента, которые применяются для контроля качества анализа на стадии разработки инструмента. Как правило, представляют собой набор приёмочных тестов, показывающих работоспособность различных функциональных возможностей инструмента.

Применение сертифицированных тестовых наборов, разрабатываемых сторонними организациями. Данные наборы представляют собой наборы тестовых примеров, рассчитанных на проверку различных характеристик анализатора и на проверку возможности обнаружения инструментом различных типов ошибок в программном обеспечении. Они могут использоваться для оценки качества инструмента при приёмочном тестировании в организациях, планирующих использовать тестируемый инструмент, и для оценки качества реализации инструмента разработчиками. Среди таких наборов стоит отметить упоминавшийся в первой главе набор тестов для инструментов статического анализа из коллекции *Juliet Test Suite* проекта *SAMATE* [175], разработанный в Национальном институте стандартов и технологий США, набор тестов Проекта сравнения инструментов

символьного исполнения программ для исполнения в средах *JVM* и *.Net* – *SETTE* [176] и набор программ *LAVA-M* со специально внедренными ошибками при помощи инструмента *LAVA* [177].

Проверка результатов анализа программ с открытым исходным кодом. Данный метод позволяет одновременно проверить возможности инструментов для обнаружения ошибок в реальных программах и проверить программы, доступные для использования, на наличие ошибок и уязвимостей.

На данный момент нет общепринятого международного стандарта для проверки качества инструментов анализа. В связи с этим разработчики инструментов создают свои и используют доступные тестовые наборы для инструментов, а также стандартной практикой оценки стала проверка разработанных инструментов анализа на наборе программ, доступных для использования.

4.4. Оценка предложенного метода

Оценка предложенного метода осуществлялась на наборе утилит командной строки из набора программ с открытым исходным кодом для операционной системы *Debian Linux*:

cabextract – инструменты распаковки *Microsoft* cabinet-файлов;
cpio – утилита сжатия и извлечения архивов;
elfutils – набор утилит для работы с файлами в формате *ELF*;
epstool – утилита для создания и извлечения изображений из файлов формата *EPS*;
expat – парсер файлов в формате *XML*;
faad – программа перекодирования файлов *MPEG2, MPEG4*;
jasper – утилита конвертации файлов изображений;
orcc – компилятор программ обработки потоков данных;
pngtools – утилиты работы с файлами в формате *PNG*;
unmo3 – распаковщик звуковых файлов в формате *MO3*;
usepackage – утилита управления установленными пакетами *Linux*.

В процессе анализа результатов проверки стоит учесть, что проверялась возможность построения внешних данных для прохождения по трассе предупреждения ошибки в связи с тем, что для большинства ошибок,

найденных на проектах с открытым исходным кодом, ещё не реализованы алгоритмы проверки условия проявления ошибки. В процессе проверки использовалась экспериментальная реализация проверки нарушения предиката безопасности для операций обращения к буферам на стеке. Таким образом, оценивалась возможность реализации алгоритма комбинированного анализа и возможности реализации алгоритма направленного анализа и достижения точек событий на трассе предупреждения об ошибке в программе.

Результаты обнаружения инструментом статического анализа и построения внешних данных программы инструментом динамического символьного исполнения для различных типов предупреждений об ошибках переполнения буферов в памяти представлены в таб. 4. Колонка с заголовком *CA* отражает количество предупреждений инструмента статического анализа, колонка с заголовком *ДА* отражает количество построенных наборов внешних данных, приводящих к исполнению программы по пути, содержащему трассу предупреждения об ошибке.

Таблица 4. Результаты анализа на переполнение буфера в памяти

Предупреждение об ошибке	CA	ДА
BUFFER_OVERFLOW.ARGV	95	7
BUFFER_OVERFLOW.EX	12	0
OVERFLOW_AFTER_CHECK.EX	1	0
OVERFLOW_UNDER_CHECK	55	0
STRING_OVERFLOW	15	2

Группа предупреждений об ошибках переполнения буфера позволяет выявить подозрительные места в коде, где возможен выход за границы при доступе к буферу в памяти. Для 9 из 178 предупреждений удалось построить внешние данные программы, которые приводят к исполнению программы по пути исполнения программы, содержащему трассу предупреждения о дефекте. Для 2 предупреждений на основе прототипной реализации проверки нарушения предиката безопасности удалось подобрать внешние данные программы, на которых происходит переполнение буфера. Переполнение

было зарегистрировано для строковых буферов, размещённых на стеке значением, соответствующим аргументу командной строки.

Результаты обнаружения инструментом статического анализа и построения внешних данных программы инструментом динамического символьного исполнения для предупреждений об ошибках работы с указателями на освобождённый блок памяти представлены в таб. 5.

Группа предупреждений об ошибках работы с указателями на освобождённый блок в памяти позволяет выявить подозрительные места в коде, где возможен доступ к освобождённым блокам памяти.

Таблица 5. Результаты анализа работы с указателями на освобожденный блок памяти

Предупреждение об ошибке	СА	ДА
DANGLING_POINTER.STAT	9	0
DANGLING_POINTER.STRICT	512	25
USE_AFTER_FREE	5	1
USE_AFTER_RELEASE	1	1
DEREF_AFTER_FREE	11	0
DOUBLE_CLOSE.PROC	51	0
PASSED_TO_PROC_AFTER_FREE.EX	8	0
DOUBLE_FREE.EX	18	0

В эту же группу включён дефект обращения к дескриптору ресурса, работа с которым была завершена, и ресурс был освобождён. Для 27 из 615 обнаруженных ошибок удалось построить внешние данные программы, проявляющие ошибку в программе. Все предупреждения об ошибке *DANGLING_POINTER.STRICT*, для которых удалось построить внешние данные программы для проведения исполнения программы по пути, содержащему трассу предупреждения об ошибке, оказались ложными в связи с тем, что во всех 25 случаях после освобождения указателя, принадлежащего структуре, происходило освобождение структуры, либо переменная указателя выходила за область видимости, либо не использовалась далее в вычислениях. Для более точной классификации подобных дефектов требуется реализация проверки нарушения предикатов безопасности, построенная на применении

теневой памяти с целью более точного контроля использования «повисших» указателей.

Для предупреждения об ошибке типа *USE_AFTER_FREE* построены внешние данные программы, приводящих к исполнению программы по пути, содержащем операцию освобождения блока памяти на куче, на который указывает указатель, после чего происходит присваивание указателя переменной, принадлежащей структуре. В общем случае для данного типа ошибок требуется проверка использования присвоенного значения адреса в процессе вычислений в программе. Однако наличие подобной трассы без присваивания указателю значения *NULL* может привести к сложностям в процессе отладки программ в случае, если значение указателя будет использовано для доступа к освобождённому блоку памяти. Наличие внешних данных программы, проявляющих указанную проблему, значительно облегчает отладку и исправление программы.

Для предупреждения об ошибке типа *USE_AFTER_RELEASE* построены внешние данные, на которых проявляется ошибка повторного закрытия файла. В общем случае поведение программы при вызове функции закрытия файла, который уже был закрыт, не определено, но может привести к аварийному завершению программы. Наличие внешних данных явно проявляет ошибку в программе и значительно облегчает её отладку.

Результаты обнаружения инструментом статического анализа и построения внешних данных программы инструментом динамического символьного исполнения для предупреждений об ошибках работы с указателями на освобожденный блок памяти представлены в *таблице 6*.

Таблица 6. Предупреждения об ошибках разыменования указателя без проверки на NULL после проверки на NULL на одном из путей, достигающих операции разыменования

Предупреждение об ошибке	СА	ДА
DEREF_AFTER_NULL	2	0
DEREF_AFTER_NULL.EX	116	13
DEREF_AFTER_NULL.MIGHT	52	0
DEREF_AFTER_NULL.RET	4	0

Группа предупреждений об ошибках использования указателя без проверки его значения на *NULL* при условии, что ранее на одном из путей проводилась проверка возможности равенства указателя значению *NULL*, позволяет выявить подозрительные места в коде, где возможно разыменование нулевого указателя, которое однозначно приведёт к исключительной ситуации доступа к памяти в программе. Для 13 из 174 предупреждений об ошибке удалось построить внешние данные программы, проявляющие ошибку. При ручной проверке во всех случаях, для которых удалось построить внешние данные, возможность разыменования нулевого указателя подтвердилась, но для автоматического построения внешних данных программы, явно проявляющих ошибку, требуется применение техники анализа алгоритмов циклических вычислений и поддержки неявных зависимостей по управлению и по данным.

Результаты обнаружения инструментом статического анализа и построения внешних данных программы, на которых проявляется ошибка, инструментом динамического символьного исполнения для различных предупреждений об ошибках разыменования нулевого указателя *DEREF_OF_NULL* представлены в таб. 7.

Таблица 7. Предупреждения об ошибках разыменования нулевого указателя.

Предупреждение об ошибке	СА	ДА
DEREF_OF_NULL	1	0
DEREF_OF_NULL.ASSIGN	4	0
DEREF_OF_NULL.EX	44	4
DEREF_OF_NULL.RET.PROC.STAT	18	0
DEREF_OF_NULL.RET.STAT	29	2
NULL_AFTER_DEREF	15	0

Группа предупреждений об ошибках использования нулевого указателя позволяет выявить подозрительные места в программе, где возможно разыменование нулевого указателя, которое однозначно приведёт к исключительной ситуации доступа к памяти в программе. Для 6 из 111 дефектов удалось построить внешние данные программы, на которых исполнение программы происходит по путям, содержащим трассу

предупреждения об ошибке. Четыре предупреждения об ошибках, для которых удалось построить внешние данные, оказались ложноположительными. Например, для одной из функций, которая возвращает тип *int*, был обнаружен дефект статистического обнаружения разыменования *NULL*, хотя функция не возвращает значений, указывающих на адрес в памяти. Для уточнения возможности разыменования нулевого указателя методом динамического символьного исполнения необходимо проверять нарушение предиката безопасности для операций доступа к памяти через разыменование нулевого указателя.

Результаты обнаружения инструментом статического анализа и построения внешних данных программы инструментом динамического символьного исполнения для предупреждений об ошибках утечки ресурсов представлены в таб. 8.

Таблица 8. Предупреждения об ошибках утечки ресурсов

Предупреждение об ошибке	CA	ДА
HANDLE_LEAK	33	1
HANDLE_LEAK.EX	33	1
HANDLE_LEAK.STRICT	2	0
MEMORY_LEAK	27	3
MEMORY_LEAK.EX	67	3
MEMORY_LEAK.STRUCT	32	3

Группа предупреждений об ошибках утечки ресурсов позволяет выявить подозрительные места в программе, где возможна потеря контроля над ресурсом операционной системы, таких, как указатель на буфер памяти, выделенный на куче, или дескрипторов ресурсов, таких, как файл, сокет и др. Для 11 из 194 предупреждений об ошибке были построены внешние данные, на которых исполнение программы происходит по пути, содержащему трассу предупреждения об ошибке. Предупреждение об ошибке утечки ресурса *HANDLE_LEAK.EX*, для которого удалось построить внешние данные, оказалось ложным в связи с тем, что ресурс, соответствующий дескриптору, на пути исполнения освобождался, но оставалась копия дескриптора ресурса, которая ложно воспринималась как содержащая дескриптор открытого

ресурса. Для подтверждения данного класса предупреждений об ошибках требуется реализация проверки нарушения предиката безопасности, построенная на контроле множеств выделенных и освобождённых ресурсов. Для остальных дефектов из группы на построенных внешних данных была подтверждена утечка ресурса.

В процессе построения входных данных для прохождения по трассе предупреждений об ошибках инструментом *Anxiety* сгенерировано 26597 уникальных наборов входных данных, которые были обнаружены в процессе работы алгоритма достижения трассы предупреждений об ошибках и привели к аварийному завершению анализируемых программ по 24 уникальным адресам инструкций в программах и системных библиотеках. Из них 2 адреса соответствовали вызову функции принудительной остановки программы *abort()* (SIGABRT), а по 22 адресам произошла ошибка сегментации (SIGSEGV), которая указывает на ошибки работы с памятью. Для 13 из 24 адресов удалось восстановить строку в исходном тексте программы и ни для одной из строк не было обнаружено предупреждений об ошибке от инструмента статического анализа *Svace*. Наличие данных ошибок в программе является одной из причин невозможности достижения трассы предупреждений об ошибках в связи с аварийным завершением программы до достижения точки проявления ошибки.

Несмотря на известные ограничения инструмента динамического символьного исполнения, удалось построить внешние данные программы для прохождения по трассе событий предупреждения об ошибке для нескольких типов ошибок, что подтверждает применимость разработанных алгоритмов для классификации предупреждений об ошибках предложенным методом, повышает точность результатов статического анализа программ и позволяет использовать построенные внешние данные программы для её отладки. Также удалось обнаружить несколько критических ошибок времени исполнения программы, не обнаруженных методами статического анализа программ.

Заключение

В работе приведены результаты исследования и разработки метода комбинированного анализа программ для классификации предупреждений об ошибках в программе.

Основные результаты работы:

- разработана модель обнаружения ошибок в программе в процессе символьного исполнения программы;
- разработаны алгоритмы комбинированного анализа программ на основе направленного динамического символьного исполнения программ, использующего предупреждение об ошибке, полученное методами статического анализа исходного кода программ, для вычисления внешних данных программы, приводящих к проявлению программной ошибки в процессе исполнения программы;
- разработан метод классификации предупреждений об ошибках в программе, основанный на применении разработанных модели и алгоритмов комбинированного анализа программ.

Предложенный метод имеет следующие отличия от аналогичных методов реализации комбинированного анализа:

1. Для проведения направленного динамического символьного исполнения используется трасса событий предупреждения об ошибках, полученная от инструмента статического анализа программ.
2. Алгоритм направленного анализа позволяет значительно сократить количество требующих анализа путей исполнения в программе в процессе динамического символьного исполнения.
3. Метод может быть применён как к результатам статического анализа исходного кода программ, так и к результатам статического анализа машинного кода программ.
4. Метод позволяет повысить точность результатов статического анализа программ.

5. Метод позволяет обнаруживать ошибки в программах, которые не обнаруживаются при помощи статического анализа программ, примененного отдельно.

Для развития метода предлагаются направления исследований, основанные на известных ограничениях метода:

1. Разработка алгоритмов достаточной и не избыточной помеченности потока данных программы для увеличения точности динамического символьного исполнения и расширения класса программ, доступных для анализа.
2. Разработка методов символьного исполнения для программ, использующих косвенные вызовы подпрограмм и переходы по вычисляемым адресам.
3. Разработка методов символьного исполнения программ с параллельными вычислениями.
4. Разработка методов символьного исполнения интерактивных программ с графическим пользовательским интерфейсом;
5. Разработка алгоритмов вычисления условий проявления ошибок для широкого класса ошибок.

Список публикаций автора по теме диссертации

1. Герасимов А. Ю., Круглов Л. В. Вычисление входных данных для достижения определённой функции в программе методом итеративного динамического анализа / А. Ю. Герасимов, Л. В. Круглов // Труды ИСП РАН. 2016. Т. 28, вып. 5. С. 159-174.
2. Герасимов А. Ю. Обзор подходов к улучшению качества результатов статического анализа программ // Труды ИСП РАН. 2017. Т. 29, вып. 3. С. 75-98.
3. Герасимов А. Ю. Подход к определению достижимости программных дефектов, обнаруженных методом статического анализа, при помощи динамического символьного исполнения / А. Ю. Герасимов, Л. В. Круглов, М. К. Ермаков, С. П. Вартанов // Труды ИСП РАН. 2017. Т.29, вып. 5. С. 111-134.
4. Gerasimov A. Reachability confirmation of statically detected defects using dynamic analysis / A. Gerasimov, L. Kruglov // Proceedings of the 11th International Conference on Computer Science and Information Technologies (CIST). Yerevan, 2017.
5. Герасимов А. Ю. Об ограничениях классификации дефектов в программах, найденных методами статического анализа программ при помощи динамического символьного исполнения // 60-я конференция МФТИ. Долгопрудный, 2017. С. 101-103.
6. Gerasimov A. Anxiety: a dynamic symbolic execution framework / A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Astyan // Ivannikov ISPRAS Open Conference - 2017. Moscow, 2017.
7. Gerasimov A. Reachability confirmation of statically detected defects using dynamic analysis / A. Gerasimov, L. Kruglov // Proceedings of the Ivannikov Memorial Workshop. Yerevan, Armenia, 3-4 May, 2018.
8. Герасимов А. Ю. Направленное динамическое символьное исполнение программ для подтверждения ошибок в программах // Программирование. 2018. №5. С. 316-323.

9. Gerasimov A. Y. An approach to Reachability Determination for Static Analysis Defects with the Help of Dynamic Symbolic Execution / A. Y. Gerasimov, L. V. Kruglov, M. K. Ermakov, S. P. Vartanov // Programming and Computer Software. 2018. Vol. 44. No. 6. P. 443-451.
10. Свидетельство о государственной регистрации программы для ЭВМ 2016660242. Российская Федерация. Инструмент итеративного динамического анализа программ / А. Ю. Герасимов. - заявл. 12.07.2016; зарегистрир. 08.09.2016; опубл. 20.10.2016.
11. Свидетельство о государственной регистрации программы для ЭВМ 2016660244. Российская Федерация. Инструмент вычисления наборов входных данных для достижения определенной инструкции в программе / А. Ю. Герасимов. - заявл. (12.07.2016); зарегистрир. 09.09.2016; опубл. (20.10.2016).
12. Свидетельство о государственной регистрации программы для ЭВМ 2017660034. Российская Федерация. Anxiety: модуль направленного анализа для инструмента итеративного динамического символьного исполнения программ / А. Ю. Герасимов. - заявл. (17.07.2017); зарегистрир. 13.09.2017; опубл. (13.09.2017).
13. Свидетельство о государственной регистрации программы для ЭВМ 2017660037. Российская Федерация. Anxiety: модульный инструмент итеративного динамического символьного исполнения программ / А. Ю. Герасимов. - заявл. (17.07.2017); зарегистрир. 13.09.2017; опубл. (13.09.2017).
14. Свидетельство о государственной регистрации программы для ЭВМ 2017660154. Российская Федерация. Anxiety: модуль параллельных вычисления для инструмента динамического символьного исполнения / А. Ю. Герасимов. - заявл. (17.07.2017); зарегистрир. 18.09.2017; опубл. (18.09.2017).

Список литературы

1. Sketch of The Analytical Engine Invented by Charles Babbage by L. F. Menabea. *Bibliotheque Universalle de Geneve, October, 1942, No. 82. With notes upon the Memoir by the translator Ada Augusta, countess of Lovelace.* Доступ к документу 05.05.2017: https://www.fourmilab.ch/babbage/sketch.html
2. ГОСТ Р 56939-2016. Защита информации. Разработка безопасного программного обеспечения. Общие требования. М: Стандартинформ, 2016
3. IEEE Standard Classification for Software Anomalies. *IEEE Computer Society, IEEE, 3 Park Avenue, New York, NY 10016-5997, USA, 2010*
4. W. W. Peng, Dolores R. Walles. Software Error Analysis. *Systems and Software Technology Division, Computer System Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA, 1993*
5. W. A. Florac. Software Quality Measurement: A Framework for Counting Problems and Defects. *Teschnical Report CMU/SEI-92-TR-022, Software Engineering Institute, 1992*
6. ANSIAEEE Std 610. 12, IEEE Standard Glossary of Software Engineering Terminology. *The Institute of Electrical and Electronics Engineers, February, 1991*
7. J. M. Juran (ed.), Juran's Quality Control Handbook, 4th ed., *McGraw-Hill, Inc., New York, 1988*
8. M. L. Shooman, A Class of Exponential Software Reliability Models. *Workshop on Software Reliability, IEEE Computer Society Technical Committee on Software Reliability Engineering, Washington, DC, April 13, 1990.*
9. D. R. Wallace, L. M. Ippolito, D. R. Kuhn, NIST SPEC PUB 500-204, High Integrity Software Standards and Guidelines, *U.S. Department of Commerce/National Institute of Standards and Technology, 1992.*

10. А. А. Белеванцев. Многоуровневый статический анализ исходного кода для обеспечения качества программ. *Диссертация на соискание учёной степени доктора физико-математических наук, Москва, 2017*
11. Common Vulnerabilities Enumeration Details. Доступ к документу 23.09.2018: http://cvedetails.com
12. IEEE Standard Classification for Software Anomalies. *IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, 7 January 2010*
13. Common Weakness Enumeration. Доступ к документу 23.09.2018 http://cwe.mitre.org/index.html
14. Software Fault Patterns. Доступ к документу 23.09.2018: https://samate.nist.gov/BF/Enlightenment/SFP.html
15. Open Web Application Security Project. Доступ к документу 23.09.2018: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
16. K. Tsiopenyuk, B. Chess, G. McGraw. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *Journal IEEE Security and Privacy, vol. 3, issue 6, pp. 81-84, November 2005*
17. E. E. Ogheneovo. Software Dysfunction: Why Do Software Fail? *Journal of Computer and Communications, 2014, 2, pp. 25-35, ISSN Print: 2327-5219, ISSN Online: 2327-5227.*
18. Why Does Software Have Bugs? Доступ к документу 23.09.2018: http://www.softwaretestinghelp.com/why-does-software-have-bugs/
19. S. H. Kan. Metrics and Models in Software Quality Engineering. *Addison-Wesley Professional, 2002, ISBN-13:978-0-201-72915-3, pp 88-96*
20. B. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes, vol. 11, issue 4, pp. 14-24, August 1986*
21. В. В. Кулямин. Методы верификации программного обеспечения. 2008.
22. G. J. Myers. The Art of Software Testing. Second Edition. *John Wilson & Sons Inc. Hoboken, New Jersey, 2004, ISBN: 0-471-46912-2*

23. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering, 6(3):236-246. May 1980.*
24. E. W. Dijkstra. On the reliability of the programs. Доступ к документу 03.05.2017: https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF
25. P. Runeson, C. Andersson, T. Thelin, A. Andrews, T. Berling. What Do We Know about Defect Detection Methods? *IEEE Software May/June 2006, pp. 82-90*
26. D. M. Ritchie. The development of the C language. *Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages, pp. 201-208 Cambridge, MA, USA – April 20-23, 1993*
27. S. C. Johnson. A Portable Compiler: Theory and Practice. *Proceedings of 5th ACM POPL Symposium, January 1978*
28. S. C. Johnson. Lint, a Program Checker. *Unix Programmer's manual, Seventh Edition, Vol. 2B, M.D. McIlroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.*
29. B. Chelf, A. Chou. The next generation of Static Analysis. *Coverity, March 18, 2008. Доступ к документу 26.04.2018: https://www.scribd.com/document/60844660/Coverity-White-Paper-SAT-Next-Generation-Static-Analysis-0*
30. D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT), Volume 9 Issue 5, October 2007, pp. 505 - 525*
31. S. Chaki, E. Clarke, A. Groce, S. Jha, H. Veith. Modular verification of software components in C. *ICSE '03 Proceedings of the 25th International Conference on Software Engineering, pp. 385-395, Portland, Oregon — May 03 - 10, 2003*

32. Rough Auditing Tool for Security. Доступ к документу 02.06.2018: https://code.google.com/archive/p/rough-auditing-tool-for-security/downloads
33. Flawfinder. Доступ к документу 02.06.2018: https://www.dwheeler.com/flawfinder/
34. J. Viega, J. T. Bloch, Y. Kohno, G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. *Proceeding ACSAC '00 Proceedings of the 16th Annual Computer Security Applications Conference, p. 257, New Orleans, Louisiana, USA, December 11 - 15, 2000*
35. D. Larochelle, D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. *Proceedings of the 10th Usenix Security Symp. (Usenix 01), Usenix Assoc., 2001, pp. 177–189.*
36. G. J. Holzmann. UNO: Static Source Code Checking for User-Defined Properties. *IDPT ’02 Proceedings of the 6th World Conference on Integrated Design and Process Technology, Pasadena, California, USA, June 23-28, 2002*
37. G. Díaz, J. R. Bermejo. Static analysis of source code security: assessment of tools against SAMATE tests. *Journal Information and Software Technology archive Volume 55 Issue 8, August 2013, pp. 1462-1476*
38. M. Bishop, M. Dilger. Checking for Race Conditions in File Accesses, *Computing Systems, vol. 9, no. 2, pp. 131–152, 1996*
39. D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS 00), Internet Soc. , pp. 3–17, 2000*
40. X. Yichen, A. Chou, D. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. *Proceeding ESEC/FSE-11 Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 327-336, Helsinki, Finland — September 01 - 05, 2003*

41. J. Foster, T. Terauchi, A. Aiken, Flow-Sensitive Type Qualifiers, *Proceedings of the ACM Conference Programming Language Design and Implementation (PLDI 02), ACM Press, pp. 1–12, 2002*
42. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection using Boolean Satisfiability. *Journal ACM Transactions on Programming Languages and Systems (TOPLAS) - Special issue on POPL 2005, Volume 29 Issue 3, May 2007, Article No. 16*
43. Klocwork Static Code Analysis. Доступ к документу 02.06.2018: https://www.roguewave.com/products-services/klocwork/static-code-analysis
44. В. П. Иванников, А. А. Белеванцев, А. Е. Бородин, В. Н. Игнатьев, Д. М. Журихин, А. И. Аветисян, М. И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. *Труды Института системного программирования РАН, том 26, вып. 1, 2014, стр. 231-250.*
45. P. Emanuelsson, U. Nilsson, A Comparative Study of Industrial Static Analysis Tools. *Technical report. Department of Computer and Information Science, Linköping University. Linköping, Sweden, 2008*
46. D. Engler, B. Chelf, A. Chou, S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *OSDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1. San Diego, California – October 22-25, 2000*
47. B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge. Why don't software developers use static analysis tools to find bugs?. *ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USE, May 18-26, 2013*
48. M. Christakis, C. Bird. What developers want and need from program analysis: An empirical study. *ASE'16 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 332-343 Singapore, Singapore, September 03 - 07, 2016*

49. J. Franco, J. Martin. A history of Satisfiability. Handbook of Satisfiability. *IOS Press, 2009 doi:10.3233/978-1-58603- 929-5-3*
50. I. Gomes, P. Morgado, T. Gomes, R. Moreira. An overview on the Static Code Analysis approach in Software Development. Technical report, Faculdade de Engenharia da Universidade do Porto (2009), Доступ к документу 02.06.2018: https://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf
51. StyleCop project. Доступ к документу 23.09.2018: https://github.com/StyleCop/StyleCop
52. FxCop. Доступ к документу 23.09.2018: https://msdn.microsoft.com/ru-ru/library/bb429476(v=vs.80).aspx
53. D. Hovemeyer, W. Pugh. Finding bugs is easy. *Newsletter ACM SIGPLAN Notices, Vol. 39 Issue 12, pp. 92-106, December 2004*
54. Jlint. Доступ к документу 02.06.2018: http://jlint.sourceforge.net
55. C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. *PLDI '02 Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation, pp. 234-245, Berlin, Germany — June 17 - 19, 2002*
56. J. W. Duran, S. Ntafos. A Report on Random Testing. *ICSE '81 Proceedings of the 5th international conference on Software engineering, pp. 179-183, San Diego, California, USA, March 09 - 12, 1981*
57. Joe W. Duran, S. Ntafos. An Evaluation of Random testing. *Journal IEEE Transactions on Software Engineering archive, Volume 10 Issue 4, July 1984, Page 438-444 , IEEE Press Piscataway, NJ, USA*
58. G. Weinberg. Fuzz Testing and Fuzz History. Доступ к документу 22.04.2018: http://secretsofconsulting.blogspot.ru/2017/02/fuzz-testing-and-fuzz-history.html
59. K. V. Hanford. Automatic Generation of Test Cases. I*BM Systems Journal archive, Vol. 9, Issue 4, pp. 242-257, December 1970*

60. A. Takanen, J. D. Demott, C. Miller. Fuzzing for Software Security Testing and Quality Assurance. *Artech House Inc., 685 Canton Street, Norwood, MA, USA, 2008, ISBN: 978-1-59693-214-2*
61. B. Miller. Projects List. *Computer Sciences Department University of Wisconsin-Madison, CS736, Fall 1988.*
62. B. P. Miller, L. Fredriksen, B. So. An empirical study of the reliability of UNIX utilities. *Magazine Communications of the ACM, Volume 33 Issue 12, pp. 32-44, , Dec. 1990*
63. B. P. Miller, D. Koski, C. Ph. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. *Technical Report, 1995*
64. D. Aitel. An Introduction to Fuzzing: Using fuzzers (SPIKE) to find vulnerabilities. 2002. Доступ к документу 23.04.2018: http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitelspike.ppt
65. J. Röning, M. Laakso, A. Takanen, R Kaksonen. PROTOS - systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research, Seattle, USA. May 6, 2002,* Доступ к документу 23.04.2018: https://www.ee.oulu.fi/roles/ouspg/PROTOS_MSR2002-protos
66. B. Beizer. Black-box Testing: Techniques for Functional Testing of Software and Systems. *John Wiley & Sons Inc., 1995, ISBN:0-471-12094-4*
67. R. Kaksonen. "A Functional Method for Assessing Protocol Implementation Security", *Licentiate Thesis. VTT Publications 447. ISBN 951-38-5873-1, 2001*
68. D. Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. *Immunity Inc., February 2002*, Доступ к документу 25.04.2018: http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.html
69. M. Sutton, A. Greene, P .Amini. Fuzzing: Brute Force Vulnerability Discovery. *Addison-Wesley, 2007, ISBN:0-321-44611-9*
70. M. Zalewski. HTML Maglizer, Доступ к документу 25.04.2018: https://github.com/adobe/webkit/tree/master/Tools/mangleme

71. H. D. Moore, M. Murphy, A. Raff, T. Zoller. CSSDIE. Доступ к документу 25.04.2018: https://hdm.io/tools/see-ess-ess-die/cssdie.html
72. American Fuzzy Lop project historical notes. Доступ к документу 25.04.2018: http://lcamtuf.coredump.cx/afl/historical_notes.txt
73. gcov – a Test Coverage Program. Доступ к документу 26.04.2018: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
74. Bunny-the-fuzzer project. Доступ к документу 26.04.2018: https://code.google.com/archive/p/bunny-the-fuzzer/wikis/BunnyDoc.wiki
75. W. Drewry, T. Ormandy. Flayer: exposing application internals. *WOOT '07 Proceedings of the first USENIX workshop on Offensive Technologies, Article No. 1, Boston, MA, August 06 - 10, 2007*
76. Fuzzgrind project. Доступ к документу 26.04.2018: http://esec-lab.sogeti.com/pages/fuzzgrind.html
77. P. Godefroid, M. Y. Levin, D. Molnar, Automated Whitebox Fuzz Testing. *NDSS'2008 Proceedings of the Network and Distributed Systems Security, pp. 151-166, San Diego, February 8 - 11, 2008*
78. M. Zalewski. Symbolic execution in vuln research. Доступ к документу 12.08.2018: https://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html
79. J. D. DeMott. Enhancing Automated Fault Discovery and Analysis. *A dissertation submitted to Michigan State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy Computer Science, 2012*
80. P. Amini, A. Portnoy. Sulley Fuzzing Framework. 2010. Доступ к документу 26.04.2018: http://www.fuzzing.org/wp-content/SulleyManual.pdf
81. Peach. 2011. Доступ к документу 26.04.2018? https://www.peach.tech
82. G. van Rossum. An Introduction to Python for UNIX/C Programmers. *Proceedings of the NLUUG najaarsconferentie (Dutch UNIX users group), 1993*

83. A. Kuchling. XML, the eXtensible Markup Language. *Journal Linux Journal archive, Volume 1998, Issue 55es, Nov. 1998 Article No. 8 Belltown Media Houston, TX*
84. R.S. Boyer, B. Elspas, K.N. Levitt. SELECT – F Formal System for Testing and Debugging Programs by Symbolic Execution. *Proceedings of the Internations Conference on Reliable software. Los Angeles, Califirnia, USA, Aprul 21-23, 1975, pp. 234-245*
85. W. E. Howden. Experiments with a symbolic evaluation system. *AFIPS'76 Proceedings of the June 07-10, 1976, National Computer Conference and Exposition. pp.899-908*
86. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM, vol. 19, issue7,July 1976, pp. 385-394*
87. E. Albert, P. Arenas, M. Gómez-Zamalloa, J. M. Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency. *Advanced Lectures of the 14th International School on Formal Methods for Executable Software Models - Volume 8483, pp. 263-309, June 16 - 20, 2014*
88. K. Sen, D. Marinov, G. Agha. CUTE: A Concolic Unit Testing Engine for C. *Proceeding ESEC/FSE-13 Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT Iinternational Symposium on Foundations of Software Engineering. pp. 263-272, Lisbon, Portugal, September 05 - 09, 2005*
89. P. Godefoid, N. Klarlund, K. Sen. DART: directed automated random testing. *PLDI '05 Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213-223, Chicago, IL, USA — June 12 - 15, 2005*
90. C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. *SPIN'05 Proceedings of the 12th international conference on Model Checking Software, pp. 2-23, San Francisco, CA, August 22 - 24, 2005*

91. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler. EXE: automatically generating inputs of death. *CCS '06 Proceedings of the 13th ACM conference on Computer and communications security, pp. 322-335, Alexandria, Virginia, USA — October 30 - November 03, 2006*
92. G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *CC '02 Proceedings of the 11th International Conference on Compiler Construction, pp. 213-228, Grenoble, France, April 08 - 12, 2002*
93. S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau. Framework for instruction-level tracing and analysis of program executions. *VEE '06 Proceedings of the 2nd international conference on Virtual execution environments, pp. 154-163, Ottawa, Ontario, Canada, June 14 - 16, 2006*
94. Y. Hamadi. Disolver : A Distributed Constraint Solver. *Technical Report MSR-TR-2003-91, Microsoft Research, December 2003. Электронный ресурс URL:https://www.microsoft.com/en-us/research/publication/disolver-a-distributed-constraint-solver/*
95. P. Godefroid. Compositional dynamic test generation. *POPL '07 Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 47-54, Nice, France, January 17 - 19, 2007*
96. И. К. Исаев, Д. В. Сидоров. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. *Программирование, 2010, №4, с. 1-16.*
97. N. Nethercote, J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI '07 Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89-100, San Diego, California, USA, June 10 - 13, 2007*
98. V. Ganesh, D. L. Dill. A decision procedure for bit-vectors and arrays. *CAV'07 Proceedings of the 19th international conference on Computer aided verification, pp. 519-531, Berlin, Germany, July 03 - 07, 2007*

99. D. A. Molnar and D.Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. *Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, February 4, 2007.*
100. J. Newsome, D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. *NDSS'05 The 12th Annual Network and Distributed System Security Symposium, San Diego, California, 2005, 3-4 February*
101. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *ASPLOS XVI Proceedings of the 16th international conference on Architectural support for programming languages and operating systems, pp. 265-278, Newport Beach, California, USA, March 05 - 11, 2011*
102. F. Bellard. QEMU, a fast and portable dynamic translator. *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 41-46, Anaheim, CA, USA, April 10 - 15, 2005*
103. C. Cadar, D. Dunbar, D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI'08 Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209-224, San Diego, California, USA, December 08 - 10, 2008*
104. C. Lattner, V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *CGO '04 Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization, pp. 75-87, Palo Alto, California, USA, March 20 - 24, 2004*
105. P. Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk, The Journal of Defense Software Engineering, vol. 21, No. 6, pp. 18-21, 2008*
106. А. Е. Бородин. Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на

языках Си и Си++. *Диссертация на соискание учёной степени кандидата физико-математических наук. Москва, 2016*

107. A. Turing. On Computable Numbers With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Volume s2-42, Issue 1, pp. 230–265, 1 January 1937*
108. SEI CERT Coding Standards. Доступ к документу 13.08.2018: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
109. Motor Industry Software Reliability Association. Publications. Доступ к документу 13.08.2018: https://www.misra.org.uk/Publications/tabid/57/Default.aspx
110. Joint strike fighter air vehicle C++ coding standards for the system development and demonstration, *Copyright by Lockheed Martin Corporation, 2005*.
111. JPL Institutional Coding Standard for the C Programming Language, *Ject Propulsion Laboratory, California Institute of Technology, 2009*
112. A. Linn. Microsoft previews project Springfield, a cloud-based bug detector. Доступ к документу 12.08.2018: https://blogs.microsoft.com/ai/microsoft-previews-project-springfield-cloud-based-bug-detector/#sm.000z732l311kbezpugo1yumm6dcd4
113. Ch. Chen, B. Cui, J. Ma, R. Wu, J. Guo, W. Liu. A systematic review of fuzzing techniques. *Copmuters & Security, Vol. 75, pp. 118-137, 2018*
114. J. Li, B. Zhao, C. Zhang. Fuzzing: a survey. *Cybersecurity, 2018. doi: https://doi.org/10.1186/s42400-018-0002-y*
115. E. J. Schwartz, Th. Avgerinos, D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). *SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 317-331, Oakland, CA, USA, May 16 - 19, 2010*
116. T. Wang, T. Wei, G. Gu, W. Zou. TaintScope: a Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. *SP'10*

Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 497-512, Oakland, CA, USA, May 16 - 19, 2010

117. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna: Driller: augmenting fuzzing through selective symbolic execution. *NDSS'2016 Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 21-24 February 2016*
118. S. Ognawala, Th. Hutzelmann, E. Psallida, A. Pretschner. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. *SAC '18 Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pages 1475-1482, Pau, France — April 09 - 13, 2018*
119. C. Csallner, Y. Smaragdakis. Check'N'Crash: combining static checking and testing. *ICSE'05 Proceedings of the 27th international conference on software engineering, pp. 422–431, St. Louis, MO, USA, May 15-21, 2005*
120. C. Csallner, Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software|Practice & Experience, 34(11):1025-1050, Sept. 2004*
121. C . Csallner, Y.Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. *IISTA'06 Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 245–254, Portland, Maine, USA, July 17–20, 2006*
122. A. Hanna, H. Z. Ling, X. Yang, M. Debbabi. A synergy between static and dynamic analysis or the detection of software security vulnerabilities. *OTM'09 Proceedings of the Confederated International Congress, CoopIS, DOA, IS and ADBASE 2009 on the Move to Meaningful Internet Systems: part II, pp. 815–832, Vilamoura, Portugal, November 01-06, 2009*
123. D. Novillo. Tree ssa: A new optimization infrastructure for gcc. *Proceedings of the GCC Developers Summit, pp. 181–193, 2003*
124. J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. *Proceedings of the GCC Developers Summit, Ottawa, Ontario Canada, May 25–27, 2003*

125. GNU Project. GCC, the GNU Compiler Collection, Доступ к документу 20.08.2018 http://gcc.gnu.org/
126. S. Schwoon. Mode-Checking Pushdown Systems. *Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation, 2002*
127. X. Ge, K. Taneja, T. Xie, N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. *ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, pp. 992–994, Waikiki, Honolulu, HI, USA, May 21–28, 2011*
128. F. Logozzo. Practical Verification for the Working Programmer with Code Contracts and Abstract Interpretation (Invited Talk). *VMCAI'11 Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. pp. 19-22 Austin, TX, USA — January 23 - 25, 2011*
129. N. Tillmann and J. de Halleux. Pex: White Box Test Generation for. NET. *TAP'08 Proceedings of the 2nd international conference on Tests and proofs. pp. 134-153, Prato, Italy — April 09 - 11, 2008*
130. J. Feist, L. Mounier, S. Bardin, R. David, M.-L. Potet. Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free. *SSPREW '16 Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, Article No. 2, Los Angeles, California, USA — December 05 - 06, 2016*
131. K. Taneja, T. Xie, N. Tillmann, J. de Halleux. eXpress: guided path exploration for efficient regression test generation. *ISSTA '11 Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 1-11, Toronto, Ontario, Canada — July 17 - 21, 2011*
132. S. Person, G. Yang, N. Rungta, S. Khurshid. Directed incremental symbolic execution. *PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 504-515, San Jose, California, USA — June 04 - 08, 2011*

133. C. Zamfir, G. Candea. Execution synthesis: a technique for automated software debugging. *EuroSys '10 Proceedings of the 5th European conference on Computer systems, pp. 321-334, Paris, France — April 13 - 16, 2010*
134. K.-K. Ma, K. Y. Phang, J. S. Foster, M. Hicks. Directed symbolic execution. *SAS'11 Proceedings of the 18th international conference on Static analysis, pp. 95-111, Venice, Italy — September 14 - 16, 2011*
135. А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман. Компиляторы: Принципы, технологии и инструментарий. 2-е издание. *Москва, ООО «И.Д. Вильямс», 2008*
136. С. В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST. *Труды Института системного программирования РАН, том 20, стр. 51-68, 2011*
137. А. Е. Бородин, А. А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. *Труды Института системного программирования РАН, том 27, вып. 6, стр. 111-133, 2015*
138. ROSE Making compiler technology accessible. Publications Доступ к документу 11.09.2018: http://rosecompiler.org/?page_id=182
139. J. R. Cordy. Source Transformation, Analysis and Generation in TXL. *PEPM '06 Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 1 - 11, Charleston, South Carolina — January 09 - 10, 2006*
140. Clang 8 documentation. LibTooling. Доступ к документу 05.09.2018: https://clang.llvm.org/docs/LibTooling.html
141. V. J. Reddi, A. Settle, D. A. Connors, R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. WCAE '04 Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture, Article No. 22, Munich, Germany, 2004
142. D. Bruening, T. Garnett, S. Amarasinghe. An infrastructure for adaptive dynamic optimization. *CGO '03 Proceedings of the international symposium*

on Code generation and optimization: feedback-directed and runtime optimization, pp. 265-275, San Francisco, California, USA — March 23 - 26, 2003

143. T. Lindholm , F. Yellin, Java Virtual Machine Specification, *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999*
144. Visual Studio debugger extensibility. Доступ к документу 23.09.2018:https://docs.microsoft.com/en-us/visualstudio/extensibility/debugger/visual-studio-debugger-extensibility?view=vs-2017
145. D. Buytaert, J. Maebe, L. Eeckhout, K. De Bosschere. Building Java program analysis tools using Javana. *OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 653-654, Portland, Oregon, USA — October 22 - 26, 2006*
146. С. П. Вартанов, М. К. Ермаков, А. Ю. Герасимов. Прикладное применение динамического анализа программ, исполняющихся в интерпретирующих средах. *Труды Института Системного Программирования, том 29, вып. 1, стр. 135-148, 2017 г.*
147. S. Vartanov. Dynamic Symbolic Execution of Java Programs Using JNI. *Proceedings of the 11th International Conference on Computer Science and Information Technologies, Yerevan, Armenia, September 25 - 29, 2017*
148. В. А. Падарян, А. И. Гетьман, М. А. Соловьев. Программная среда для динамического анализа бинарного кода. *Труды Института Системного Программирования, том: 16, стр. 51-72, 2009*
149. A. Stump, C. W. Barrett, D. L. Dill. CVC: A cooperating validity checker. *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02), volume 2404 of Lecture Notes in Computer Science, pages 500–504. Springer-Verlag, Copenhagen, Denmark, July 2002.*

150. C. Barrett, A. Stump, C. Tinelli. The SMT-LIB standard: version 2.0. *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, England, 2010*
151. L. De Moura, N. Bjørner. Z3: an efficient SMT solver. *TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, pp. 337-340, Budapest, Hungary, March 29 - April 06, 2008*
152. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio. The MathSAT 4 SMT Solver. *CAV '08 Proceedings of the 20th international conference on Computer Aided Verification, pp. 299 - 303, Princeton, NJ, USA, July 07 - 14, 2008*
153. B. Dutertre. Yices 2.2. *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559, pp. 737-74, July 18 - 22, 2014*
154. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli. CVC4. *CAV'11 Proceedings of the 23rd international conference on Computer aided verification, pp. 171-177, Snowbird, UT, USA, July 14 - 20, 2011*
155. J. Salwan and F. Saudel. Triton: A concolic execution framework for x86-64 binaries. *In Symposium sur la securite des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015, pages 31--54. SSTIC, 2015*
156. А. Н. Федотов, В. В. Каушан, С. С. Гайсарян, Ш. Ф. Курмангалеев. Построение предикатов безопасности для некоторых типов программных дефектов. *Труды ИСП РАН, том 29, вып. 6, стр. 151-162, 2017*
157. Th. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley. Automatic exploit generation. *Communications of the ACM CACM Homepage archive, Vol. 57, Issue 2, pp. 74-84, February, 2014*
158. CWE-369: Divide By Zero. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/369.html

159. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/119.html
160. CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion'). Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/400.html
161. CWE-404: Improper Resource Shutdown or Release. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/404.html
162. CWE-416: Use After Free. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/416.html
163. CWE-415: CWE-415: Double Free. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/415.html
164. CWE-476: NULL Pointer Dereference. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/476.html
165. CWE-467: Use of sizeof() on a Pointer Type. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/467.html
166. CWE-134: Use of Externally-Controlled Format String. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/134.html
167. CWE-129: Improper Validation of Array Index. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/129.html
168. CWE-20: Improper Input Validation. Доступ к документу 23.09.2018: https://cwe.mitre.org/data/definitions/20.html
169. А. Ю. Герасимов, Л. В. Круглов. Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа. *Труды ИСП РАН. 2016. Т. 28. В. 5. С. 159-174*
170. А. Ю. Герасимов, Л. В. Круглов, М. К. Ермаков, С. П. Вартанов. Подход к определению достижимости программных дефектов, обнаруженных методом статического анализа, при помощи динамического символьного исполнения. *Труды ИСП РАН. 2017. Т.29 В. 5. С. 111-134*

171. A. Gerasimov, L. Kruglov Reachability confirmation of statically detected defects using dynamic analysis. *CSIT'2017 Proceedings of the 11th International Conference on Computer Science and Information Technologies (CIST), Yerevan, 2017*
172. А. Ю. Герасимов. Об ограничениях классификации дефектов в программах, найденных методами статического анализа программ при помощи динамического символьного исполнения. *60-я конференция МФТИ. Долгопрудный: 2017. С. 101-103*
173. A. Gerasimov, S.Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Astyan. Anxiety: a dynamic symbolic execution framework. *2017 Ivannikov ISPRAS Open Conference, Moscow, 2017*
174. C. Lattener. "LLVM and Clang: Next Generation Compiler Technology", *The BSD Conference, Ottawa, Canada, May, 2008*
175. NIST Software Assurance Reference Dataset Project Test Suites. Доступ к документу 28.09.2018: https://samate.nist.gov/SARD/testsuite.php
176. L. Cseppento, Z. Mikei. Evaluating Symbolc Execution-based Test Tools. *ICST'2017 Proceedings of the IEEE 8th Conference on Software Testing, Verification and Validation, Graz, Austria, April 13-17, 2015*
177. B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mamabertti, W. Robertson, F. Ulrich, R. Whelan. LAVA: Large-scale automated vulnerability addition. *2016 IEEE Symposium on Security and Privacy (SP), pp. 110-121, Sna Jose, CA, USA, May 23-25, 2016*