На правах рукописи

Дудина Ирина Александровна

Поиск ошибок переполнения буфера в исходном коде программ с помощью символьного выполнения

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

Автореферат
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва — 2019

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова РАН и на кафедре системного программирования факультета вычислительной математики и кибернетики Федерального государственного бюджетного образовательного учреждения высшего образования «Московский государственный университет имени М.В. Ломоносова».

Научный руководитель: **Белеванцев Андрей Андреевич,**
доктор физико-математических наук

Официальные оппоненты: **Галатенко Владимир Антонович,**
доктор физико-математических наук,
заведующий сектором Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Волконский Владимир Юрьевич,
кандидат технических наук,
начальник отделения «Системы программирования» Публичного акционерного общества «Институт электронных управляющих машин им. И. С. Брука»

Ведущая организация: Федеральный исследовательский центр «Информатика и управление» Российской академии наук

Защита состоится 16 мая 2019 г. в 13 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В. П. Иванникова Российской академии наук» по адресу: 109004, Москва, ул. А. Солженицына, 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки «Институт системного программирования им. В. П. Иванникова Российской академии наук».

Автореферат разослан « » 2019 г.

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

Общая характеристика работы

Актуальность темы. В последнее время автоматический поиск ошибок в программном коде является неотъемлемой частью процесса разработки современного программного обеспечения. Чтобы повысить безопасность и надежность программ, нужно найти в них и устранить как можно большее число ошибок, желательно на ранних этапах разработки. Для поиска, как правило, используется набор методов анализа и тестирования, дополняющих друг друга — статический и динамический анализ программы, дедуктивная верификация, фаззинг, тестирование на проникновение. Одним из общепризнанных подходов является статический анализ исходного кода, не предполагающий запуск анализируемой программы и позволяющий найти ошибки даже на не покрытых при тестировании путях.

Промышленное использование статического анализатора в цикле разработки для больших программных систем определяет конкретные требования к анализу: проверка миллионов строк кода за несколько часов (время ночной сборки), учет всех конструкций поддерживаемого языка программирования, высокий уровень истинных срабатываний (не менее 50-70 %, в противном случае затраты на разбор ложных предупреждений нивелируют пользу автоматического поиска ошибок), пропуск по возможности небольшого числа реальных ошибок. Чтобы упростить необходимый ручной труд пользователя по разбору предупреждений, каждое выданное предупреждение должно сопровождаться достаточной аргументацией возникших подозрений о наличии ошибки.

Одним из наиболее распространенных типов ошибок в программах на языках C/C++ являются ошибки *доступа к буферу*. Они возникают в том случае, когда обращение к буферу происходит по индексу, выходящему за его границы, т.е. происходит доступ (чтение или запись) к памяти вне данного буфера. Такое поведение может привести к аварийному завершению программы, неправильной работе, иногда — к появлению эксплуатируемой уязвимости. В классификации CWE выделено более десятка типов ошибок доступа к буферу. Кроме того, обращение на запись за правой границей буфера входит в список 25 наиболее опасных ошибок CWE. В настоящей работе термин *переполнение буфера* будет использоваться как синоним ошибки доступа к буферу и подразумевать переполнение как правой, так и левой границы при обращении на запись или чтение.

Существующие методы поиска ошибок доступа к буферу принадлежат к одному из упомянутых выше видов анализа. Подходы на основе динамического анализа и фаззинга определяют входные данные программы в случае найденной ошибки, однако требуют значительных ресурсов и могут пропустить редкие пути выполнения. Методы верификации, как правило, имеют ограниченную масштабируемость (до десятков тысяч строк кода), нуждаются в ручном задании модели окружения программы, накладывают ограничения на используемые языковые конструкции. Одним из самых подходящих для применения к большим программам в ходе разработки является метод статического анализа.

В работах ряда ученых (Й. Кси, А. Айкен, К. Цифуэнтес, Т. Кременек, К. Йи, П. О'Хирн и др.) предложены модели программ и соответствующие алгоритмы статического анализа для поиска переполнений буфера. Однако эти алгоритмы либо не обеспечивают необходимой полноты анализа, основываясь лишь на поиске шаблонов на синтаксическом дереве, либо анализируют только одну функцию или файл программы, либо выдают большое количество ложных срабатываний, либо реализованы в коммерческих закрытых инструментах (анализатор Prevent компании Synopsis, анализатор HP Fortify, инструмент Klocwork компании RogueWave). Актуальным остается построение алгоритмов поиска переполнений буфера, сочетающих высокое качество и полноту анализа с масштабируемостью для программных систем размера, сравнимого с размером современных дистрибутивов мобильных операционных систем типа Android. Такие алгоритмы по необходимости должны выполнять межпроцедурный контекстно-чувствительный анализ, различающий пути выполнения для поиска ошибок, реализуемых лишь на некоторых путях, и настраиваться с помощью эвристик для достижения компромисса между полнотой, точностью и скоростью анализа. Инструмент Svace, разрабатываемый в Институте системного программирования им. В.П. Иванникова РАН, не содержит модуль поиска переполнений буфера, обеспечивающий чувствительность к путям, и поэтому пропускает значительное количество ошибок.

Целью данной работы является разработка методов поиска ошибок доступа к буферу в исходном коде программ на языках C/C++, которые могут быть применены в жизненном цикле разработки программных систем размером в миллионы строк кода, и их реализация в инструменте статического анализа. Разработанные методы должны обеспечивать высокое количество истинных срабатываний (не менее 65 %) и применять чувствительный к путям выполнения анализ.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. формализация ошибки переполнения буфера и разработка на её основе внутрипроцедурного метода построения достаточных условий возникновения переполнения, который учитывает отдельные пути выполнения программы;
2. разработка межпроцедурного метода поиска ошибок доступа к буферу, размер которого известен во время компиляции;
3. разработка методов поиска ошибок переполнения, пригодных для буферов произвольного размера и для строк языка C;
4. реализация семейства предложенных методов в статическом анализаторе и оценка их эффективности на больших программных системах.

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

1) Предложено формальное определение ошибки доступа к буферу и разработан внутрипроцедурный, различающий пути выполнения метод статического анализа на основе символьного выполнения с объединением состояний, который позволяет строить достаточные условия возникновения ошибки переполнения в некоторой точке функции для случаев доступа к буферу известного на момент компиляции размера. Сформулированы ограничения на анализируемые программы, в рамках которых доказаны корректность и точность разработанных методов анализа.
2) Разработан межпроцедурный контекстно-чувствительный алгоритм для поиска переполнений буфера, применяющий предложенный внутрипроцедурный метод для построения резюме функции и поддерживающий буферы с известным на момент компиляции размером. Доказана корректность разработанного алгоритма.
3) Разработан метод поиска переполнений для буферов произвольного размера, который применяет как предложенные методы анализа для буферов константного размера, так и метод перебора значений условий переходов непосредственно на основе разработанного формального определения ошибки.
4) Разработаны расширения предложенных методов для поиска переполнений при работе со строками языка C, а также при использовании недоверенных входных данных.

Теоретическая и практическая значимость. Теоретическая значимость заключается в разработанном семействе методов межпроцедурного статического анализа для поиска переполнений буфера, учитывающих влияние различных путей выполнения программы и применимых к анализу больших программных систем, в теоретическом обосновании корректности и точности этих методов. Кроме того, предложенные методы формализации ошибки и построения достаточных условий ошибки, составляющие основу подхода к поиску переполнений, могут быть использованы и для поиска других ошибок, в частности, проверки корректности работы с коллекциями языка C++.

Практическая значимость работы состоит в том, что предложенные методы поиска ошибок переполнения реализованы в статическом анализаторе Svace и в его составе внедрены в промышленный цикл разработки крупной коммерческой компании. Кроме того, описанные алгоритмы анализа могут использоваться при преподавании магистерских курсов по современным методам статического анализа в МГУ и МФТИ.

Методология и методы исследования. Для решения задач, поставленных в диссертации, использовались методы теории компиляции, теории решеток, анализа потока данных, абстрактной интерпретации, символьного выполнения.

Основные положения, выносимые на защиту.

1) Формальное определение ошибки доступа к буферу и внутрипроцедурный алгоритм для поиска таких ошибок с помощью построения достаточных

условий возникновения ошибки в заданной точке функции. Алгоритм основан на символьном выполнении с объединением состояний и пригоден для случаев доступа к буферу известного на момент компиляции размера.

2) Межпроцедурный контекстно-чувствительный алгоритм для поиска переполнений буфера, применяющий разработанный внутрипроцедурный метод построения достаточных условий ошибки для построения резюме функции.
3) Метод поиска ошибок доступа к буферам произвольного размера, применяющий как описанные методы анализа буферов константного размера, так и непосредственно использующий сформулированное определение ошибки.
4) Расширения разработанных методов, которые делают их применимыми для поиска переполнений при работе со строками и с недоверенными входными данными.

Апробация работы. Основные результаты работы докладывались на Открытой конференции ИСП РАН в 2017 и 2018 гг.; конференциях «Ломоновские чтения» в 2017 и 2018 гг.; VII Конгрессе молодых ученых, г. Санкт-Петербург; конференции молодых ученых по программной инженерии «SYRCoSE-2018», г. Великий Новгород; на международном симпозиуме Ivannikov Memorial Workshop-2018, Ереван, Армения.

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

Публикации. Основные результаты по теме диссертации изложены в 8 печатных работах [1—8], 6 из которых [1—6] изданы в журналах, входящих в перечень рецензируемых научных изданий ВАК при Минобрнауки РФ, 2 — в тезисах докладов. Работы [1; 4] индексированы системой Web of Science. В работах [1; 3] автору принадлежит описание внутрипроцедурного и межпроцедурного алгоритмов поиска переполнений, в работе [5] — описание метода поиска переполнений в строках, в работе [4] — описание чувствительного к путям анализа в инструменте Svace. В работах [7; 8] автором была выполнена разработка и реализация рассмотренных алгоритмов анализа.

Содержание работы

Во **введении** обосновывается актуальность выбранного направления исследования, формулируются его цель и задачи; излагается научная новизна, теоретическая и практическая значимость работы, приводятся основные положения, выносимые на защиту.

В первой главе приводится обзор существующих методов поиска переполнения буфера и выделяются основные требования к современному промышленному детектору ошибок доступа к буферу.

Переполнением буфера называют ситуацию, возникающую во время выполнения программы вследствие обращения к некоторому массиву по индексу,

имеющему отрицательное значение, либо выходящему за размеры самого буфера. Особенно критичной такая ошибка является в языках C и C++. По данным национальной базы уязвимостей США (NVD), ошибки подобного рода являются причиной 9,49 % всех уязвимостей, занесённых в базу CVE в 2018 г.

В обзоре методов (раздел 1.1) приводится описание и характеристика с точки зрения рассматриваемой задачи таких подходов, как динамический и статический анализ кода, формальная верификация. К подходам динамического анализа можно отнести динамическую (Valgrind, DynamoRIO, Pin) и статическую (AddressSanitizer) инструментацию, динамическое символьное выполнение, фаззинг. Преимуществами данных методов является отсутствие ложных срабатываний, предоставление пользователю входных данных, на которых происходит ошибка. Кроме того, для анализа не требуется исходный код программы. Однако для обнаружения ошибок необходимо решить задачу генерации набора входных данных, позволяющего наиболее полно покрыть все потенциально ошибочные ситуации программы. Таким образом, затруднительно обнаружение ошибок на редко исполняемых путях и «далеко» от начала программы.

Исследование всех путей в программе возможно с помощью статического анализа (раздел 1.1.2). Данный подход предполагает анализ кода программы без её запуска. К преимуществам статического анализа можно также отнести возможность анализа неполной программы. К сожалению, следствием теоремы Райса является алгоритмическая неразрешимость задачи поиска критических ошибок в программах. В частности, задача обнаружения переполнения буфера тривиально сводится к задаче останова, которая является алгоритмически неразрешимой. Таким образом, анализатор, обнаруживающий в любой программе абсолютно все ошибки и при этом не выдающий ложных срабатываний, построить невозможно. Все существующие подходы реализуют некоторый компромисс между полнотой анализа (процентом обнаруживаемых реальных ошибок), точностью анализа (процентом истинных предупреждений) и масштабируемостью. В случае, когда необходимо проанализировать весь код большого проекта (до десятков миллионов строк), требуется искать эвристики, ограничивающие сложность методов статического анализа.

Методы формальной верификации программ могут подтвердить отсутствие ошибок в анализируемой программе, однако могут вводить ограничения на множество используемых конструкций языка; кроме того, используемые тяжеловесные подходы плохо масштабируются на программы большого размера. Эти инструменты не пропускают ошибок, однако могут допускать ложные срабатывания. Среди данных подходов стоит отметить ограничиваемую проверку моделей (Bounded Model Checking, BMC) и уточнение абстракций по контрпримерам (CounterExample-Guided Abstraction Refinement, CEGAR).

Примером инструмента, использующего методы верификации для полностью автоматического анализа крупных программ, является Facebook Infer. Главной особенностью этого инструмента является метод моделирования памяти, основанный на сепарационной логике (separation logic) и биабдукции

(bi-abduction). При анализе функции выносятся суждения о свойствах памяти программы, выводится предусловие, необходимое для корректности этих суждений, и создаются спецификации функций в виде компактных переиспользуемых резюме для организации масштабируемого межпроцедурного анализа. Переполнение буфера в Infer ищется детектором InferBO, который использует символьные интервалы для отслеживания границ буфера и инфраструктуру анализатора для записи резюме функций.

Из простых инструментов анализа, использующих лексический и синтаксический анализы для поиска ошибок, можно назвать ITS4, Flawfinder, RATS. Инструмент BOON обходит дерево разбора, чтобы создать систему целочисленных неравенств, описывающую условия безопасного доступа к строкам. Анализатор Splint опирается в поиске ошибок на созданные аналитиками аннотации функций, выполняя легковесный внутрипроцедурный анализ. Анализатор CSSV выполняет межпроцедурный анализ также с помощью пользовательских аннотаций. Эти анализаторы представляют в основном исторический интерес.

Межпроцедурный анализ потока данных, основанный на абстрактной интерпретации, обычно используется для построения консервативных анализаторов, которые ставят себе целью не пропустить реальных ошибок. Как правило, такие инструменты не могут анализировать большие программы с высоким уровнем истинных срабатываний либо накладывают ограничения на конструкции анализируемого языка.

Ошибка переполнения буфера может проявляться не при любом запуске, а только при прохождении потока управления через некоторую последовательность инструкций (т.е. зависеть от пути в программе). Чтобы обнаруживать такие ошибки, при этом не превышая допустимое количество ложных срабатываний,необходимо реализовать *чувствительный к путям* анализ, т.е. анализ, который различает состояния программы на разных путях и выдаёт ошибки только для ошибочных путей, являющихся потенциально выполнимыми. Примерами таких инструментов являются Marple, S-Looper, Parfait. Инструмент ARCHER, реализованный на OCaml создателями известного инструмента статического анализа Saturn, осуществляет символьное выполнение с ограничением на количество рассмотренных путей и время анализа одной функции. Поиск межпроцедурных срабатываний организован с помощью метода резюме.

Ряд современных промышленных анализаторов, таких как Coverity Prevent, Klocwork, Polyspace, IAR Static Analyzer и некоторые другие, включают в себя детекторы переполнения буфера, однако используемые ими алгоритмы закрыты, что затрудняет суждения о качестве этих детекторов.

Существенная часть ошибок переполнения буфера возникает из-за нарушения контракта функции, например, когда в библиотечную функцию передаются значения, не соответствующие предусловию функции. Чтобы их обнаруживать, статический анализ должен быть *межпроцедурным*, т.е. при анализе одной функции учитывать информацию о других функциях. Для этого можно проводить анализ всей программы целиком как единого графа потока управления с разной

степенью *контекстной чувствительности*. Полностью нечувствительный анализ проверяет все контексты вызова совместно, что снижает точность анализа, но хорошо масштабируется. Если же в каждой точке вызова код вызываемой функции анализируется независимо, анализ является контекстно-чувствительным, что обеспечивает хорошую точность, но для масштабируемости требует ограничивать степень чувствительности, выбирая группу контекстов, для которых функция будет проанализирована единожды. Разумным компромиссом может быть сочетание этих двух подходов.

В качестве альтернативного подхода можно при анализе инструкции вызова функции учитывать некоторое полученное извне приближение контракта функции. Полностью автоматические инструменты пытаются самостоятельно получить в ходе анализа описание контракта функции и сохранить его в резюме, которое используется для анализа вызывающих функций. Параметризация резюме относительно контекста вызова позволяет построить масштабируемый контекстно-чувствительный анализ, посещающий каждую функцию один раз.

В разделе 1.2 производится анализ известных тестовых наборов программ, содержащих переполнения буфера, и уязвимостей из реальных проектов, на основе чего формулируются требования к алгоритмам анализа для поиска переполнений. Раздел 1.2.1 содержит описание двух популярных тестовых наборов Juliet Test Suite C/C++ и Toyota ITC Benchmark, используемых для оценки качества статических анализаторов. Выделяются основные характеристики ошибочных ситуаций, проверяемых тестовыми наборами, с точки зрения ошибки переполнения буфера. Лучше всего структурированным и наиболее богатым по содержанию набором является Juliet, содержащий более 10 тыс. тестов, относящихся к переполнению. Приведённые тесты разбиты на группы по классификации CWE, по виду потока управления и данных, по типу ошибочной конструкции, что позволяет выносить достаточно обоснованные суждения о поддерживаемых анализатором видах анализа.

Для более ясного понимания того, какие именно возможности анализатора особенно важны для обнаружения потенциальных уязвимостей, было проведено исследование набора уязвимостей из реальных проектов, связанных с переполнением буфера. Следует отметить, что подобная выборка заведомо оказывается смещённой в сторону эксплуатируемых ошибок и к тому же не включает ошибки, исправленные на стадии разработки.

Для исследования было выбрано 132 случайных записи из категории «overflow» из базы уязвимостей CVE. Для 33 из них удалось найти исходный код, для ещё 14 некоторые данные были извлечены из описания уязвимости. В результате исследования к наиболее важным возможностям статического анализатора были отнесены межпроцедурный анализ, чувствительность к путям и контексту, а также базовая поддержка циклов. Кроме того, полезными оказываются отслеживание аффинных отношений между переменными и моделирование строк как важного случая использования массивов.

Во второй главе формулируется определение ошибки доступа к буферу в функции, учитывающее возможность наличия неизвестных предусловий этой функции.

Чтобы исследовать корректность функции во всех потенциальных контекстах вызова (а не только в реально существующих в проекте на данный момент), будем считать анализируемую функцию точкой входа в программу и рассматривать вместе со всеми вызываемыми ею, в том числе транзитивно. Будем называть *неизвестными переменными* вычисляемые вне анализируемой функции значения, параметризующие выполнение данной функции. Такими неизвестными переменными будут являться 1) входные параметры, 2) состояние памяти на входе в функцию, 3) результаты вызова неизвестных (не анализируемых) функций, 4) значения в памяти после вызова неизвестной функции, которые могли быть перезаписаны в результате этого вызова. Набор значений неизвестных переменных однозначно определяет *конкретное выполнение* функции, то есть путь на графе потока управления (ГПУ) и значения всех переменных, состояние памяти на каждом его ребре. Совокупность ограничений на множество значений неизвестных переменных будем называть *контрактом*.

Предлагается ввести априорное предположение о свойствах контрактов функций, которое определит множество рассматриваемых контрактов и, как следствие, позволит отделить потенциально возможные ошибочные сценарии исполнения функции от предположительно запрещенных контрактом. В рамках данной работы будем считать подозрительными такие ситуации, в которых наличие ошибки доступа к буферу следует из свойств ГПУ программы, но не зависит напрямую от множества допустимых значений входных параметров функции.

Пусть G^* — подграф межпроцедурного потока управления программы, содержащий только анализируемую функцию и всех её потомков в графе вызовов вплоть до листьев. Пусть G^*_k — граф после развёртки каждого его цикла на k итераций (k — параметр алгоритма). Рассмотрим множество P всех путей графа G^*_k.

Будем говорить, что некоторый путь $p \in P$ *выполним*, если существует хотя бы одно соответствующее ему конкретное выполнение. Пусть P^f — подмножество путей графа, состоящее только из тех путей, которые выполнимы при условии, что набор неизвестных переменных может принимать абсолютно любые сочетания значений. Обозначим как P^c подмножество путей графа потока управления программы, состоящее только из тех путей, которые выполнимы, если набор неизвестных переменных удовлетворяет подразумеваемому программистом контракту. Очевидно, что $P^c \subseteq P^f \subseteq P$. Наше предположение о контрактах будет заключаться в том, что допустимые контракты не сужают множество выполнимых путей, т.е. $P^c = P^f$.

Введение такого предположения приводит нас к следующему определению ошибочной функции:

Определение 1. Будем говорить, что функция содержит ошибку в инструкции обращения к буферу размера `S` по индексу `i`, если в графе G^*_k существует путь через эту инструкцию, удовлетворяющий следующим условиям:

1. на любом соответствующем конкретном выполнении на входе в инструкцию доступа верно $(0 > \mathtt{i}) \vee (\mathtt{i} \geqslant \mathtt{S})$,
2. данному пути соответствует хотя бы одно конкретное выполнение (в предположении, что набор неизвестных переменных может принимать любые комбинации значений).

***Теорема* 1.** *Если для любого контракта некоторой функции верно $P^c = P^f$, то она содержит ошибку в инструкции обращения к буферу размера $\mathtt{S}$ по индексу $\mathtt{i}$ по определению 1 тогда и только тогда, когда для неё существует путь через эту инструкцию, проходящий не более k^n раз по каждому обратному ребру цикла вложенности n, и для произвольного контракта (i) этот путь выполним, (ii) каждое его конкретное выполнение в рамках этого контракта приводит к некорректному доступу к буферу.*

Наличие ошибки, удовлетворяющей этому определению, следует из свойств графа потока управления и не зависит напрямую от множества допустимых значений неизвестных переменных.

В отдельный класс ошибок, обнаружение которых требует привлечения специализированных подходов, можно выделить межпроцедурные ошибки (раздел 2.2), т.е. ситуации, когда корректность работы некоторой функции зависит от выполнения некоторого условия над значениями, получаемыми из других функций (значения, возвращаемые или изменяемые вызываемыми функциями, либо передаваемые в качестве параметров из вызывающей функции), и это условие нарушается на некотором исполнении программы, что приводит к ошибке.

Третья глава посвящена рассмотрению алгоритма поиска переполнения буфера в рамках одной функции.

Изложение алгоритма будет проводиться для программ на модельном языке (раздел 3.1). Программа оперирует значениями битовых векторов различной длины из множества $B \subset \mathbb{N}$. Множество всех значений битового вектора длины $b \in B$ обозначим C_b. Эти значения хранятся в переменных из соответствующего множества V_b, в ячейках памяти из множества M_b либо в массивах из множества A_b. Адреса ячеек памяти из множества M_b и массивов из множества A_b хранятся в переменных-указателях из множества P_b.

Конкретное состояние программы описывается тремя отображениями:

- $\mathbb{X} : V \cup M \cup (A \times C_r) \to C$ — задаёт значения переменных и значения в ячейках памяти,
- $\mathbb{P} : P \cup M^p \to M \cup A$ — задаёт значения переменных-указателей и значения в ячейках памяти, хранящих адреса,
- $\mathbb{B} : A \to C_r$ — задаёт размеры выделенных массивов.

В разделе 3.2 приводится внутрипроцедурный, различающий пути выполнения метод статического анализа на основе символьного выполнения с объединением состояний, который позволяет строить достаточные условия возникновения ошибки переполнения в некоторой точке функции для случаев

доступа к буферу известного на момент компиляции размера, а также сформулированы ограничения на анализируемые программы, в рамках которых доказаны корректность и точность разработанных методов анализа.

Для проведения анализа текст функции преобразуется в граф потока управления. Вершинами в этом графе являются инструкции исходной программы. При этом при генерации ветвления с условием c на соответствующих выходных рёбрах вставляются вспомогательные инструкции `assume`(c) для перехода по истинной ветке и `assume`$(\neg c)$ для перехода по ложной ветке.

Анализ производится над развёрткой ГПУ функции на k итераций, где $k \in \mathbb{N}$ — параметр алгоритма. Развёртка представляет собой ациклический граф потока управления, полученный из исходного графа разворачиванием (копированием) заголовка и тела каждого цикла k раз, при этом обратные рёбра из итерации $i \in [1,k-1]$ ведут в заголовок $(i+1)$-ой итерации, а обратные рёбра из k-ой итерации удаляются. Итоговое множество инструкций, полученное после добавления к исходным инструкциям программы инструкций `assume` и копирования при развёртке, обозначим *Instr*.

Полученный ориентированный граф G_k не содержит циклов, поэтому его вершины (соответствующие инструкциям из *Instr*) могут быть упорядочены с помощью топологической сортировки. Теперь при использовании данного порядка при обходе инструкций программы в процессе анализа гарантируется, что при обработке любой инструкции все её предки уже были проанализированы.

Анализ осуществляется путём продвижения абстрактного состояния программы от входного ребра в функцию до рёбер из инструкций возврата по ГПУ с помощью передаточных функций (по аналогии с прямым анализом потока данных). В то же время вид абстрактного состояния определяет сходство данного подхода с символьным выполнением.

Инструкции обрабатываются по очереди в топологическом порядке. Для каждой инструкции сначала вычисляется входное состояние из состояний на входящих рёбрах (если их больше одного, то происходит слияние состояний). Далее, если инструкция осуществляет доступ к буферу на запись или чтение, то проверяется отсутствие ошибки. Затем с учётом семантики инструкции происходит вычисление выходного состояния с помощью передаточных функций, которое будет считаться состоянием на всех выходных рёбрах из данной инструкции.

Для определения абстрактного состояния введём следующие обозначения:

- AM_b — множество абстрактных ячеек памяти размера $b \in B$, μ — специальное значение, обозначающее неизвестную ячейку, $AM \doteq \bigcup_{b \in B} AM_b \cup \{\mu\}$.
- $AArr_b$ — множество абстрактных массивов из элементов размера $b \in B$, $AArr \doteq \bigcup_{b \in B} AArr_b$.
- $AM^p \subseteq AM_r$ — множество абстрактных ячеек памяти, хранящих значения указателей.
- S_b — множество символьных переменных размера $b \in B$, $S \doteq \bigcup_{b \in B} S_b$.

– SE_b — множество символьных выражений размера $b \in B$. Определяется рекурсивно с помощью классических операций логики битовых векторов. $SE \doteq \bigcup_{b \in B} SE_b$.
– 2^T — множество всех подмножеств произвольного множества T.
– $\mathcal{C}$ — множество формул логики первого порядка без кванторов, где в качестве термов выступают элементы множества SE, в качестве предикатов используются отношения $<_s, <_u, >_s, >_u, \leqslant_s, \leqslant_u, \geqslant_s, \geqslant_u, =, \neq$. Условия из множества $\mathcal{C}$ представляют собой свободные от кванторов формулы логики битовых векторов (Quantifier-Free Bit-Vector Logic). Тождественную истину и ложь будем обозначать $\top$ и $\bot$ соответственно.

Абстрактное состояние программы $\mathcal{A}$ представляет собой набор:

$$
\begin{aligned}
\mathcal{A} = &\langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle, \text{ где} \\
&\pi \in \mathcal{C}, \\
&\sigma : V \cup AM \to SE, \\
&\mathcal{P} : P \cup AM^p \to 2^{(AM \cup AArr) \times \mathcal{C}}, \\
&\mathcal{B} : AArr \to C_r, \\
&\mathcal{V} : SE \to Summary.
\end{aligned}
$$

Значения, хранящиеся в переменных и ячейках памяти в абстрактном состоянии, представлены символьными выражениями, как при символьном выполнении. Это соответствие определяется отображением σ.

Формула $\pi \in \mathcal{C}$, определённая для некоторой точки в функции, является необходимым условием достижимости данной точки. Это условие аналогично предикату пути в символьном выполнении, однако в данном случае π определяется не для пути, а для точки, что соответствует символьному выполнению с объединением состояний.

Отображение $\mathcal{P}$ для каждого указателя $p \in P_b \cup AM^p$ определяет множество пар $\mathcal{P}(p) = \{\langle a_i, \gamma_i \rangle \mid a_i \in (AM_b \cup AArr_b),\ \gamma_i \in \mathcal{C}\}$, означающих факты «если выполнено γ_i, то p_b указывает на a_i». При этом гарантируется однозначность выбора значения, т.е. $\forall\, i \neq j \ \Rightarrow\ \gamma_i \wedge \gamma_j = \bot$. Таким образом определяется чувствительный к путям аналог классического отношения «указывает на».

Частичное отображение $\mathcal{B}$ определяет константный размер массива из $AArr$, представленный в дополнительном коде в битовом векторе размера r. Если значение $\mathcal{B}$ определено для m, то этот факт будем записывать как $m \in \mathcal{B}$.

Введём понятие функции конкретизации абстрактного состояния ψ. Пусть отображение ψ определено для символьных переменных $\psi : S_b \to C_b$, множества абстрактных ячеек памяти $\psi : AM_b \to M_b$ и абстрактных массивов $\psi : AArr_b \to A_b$. Тогда естественным образом данное отображение можно рекурсивно доопределить для символьных выражений $\psi : SE_b \to C_b$.

Определение 2. Будем называть функцию конкретизации ψ абстрактного состояния $\mathcal{A} = \langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle$ *эквивалентной* конкретному состоянию $\langle \mathbb{X}, \mathbb{P}, \mathbb{B} \rangle$, если выполнены следующие условия:

1. $\forall \mathtt{x} \in V : \psi(\sigma(\mathtt{x})) = \mathbb{X}(\mathtt{x})$ (эквивалентность значений переменных),
2. $\forall a \in AM : \psi(\sigma(a)) = \mathbb{X}(\psi(a))$ (эквивалентность значений в памяти),
3. $\forall \mathtt{p} \in P : \exists \langle d, \gamma \rangle \in \mathcal{P}(\mathtt{p}) : \psi(\gamma) = \top \wedge (\psi(d) = \mathbb{P}(\mathtt{p}))$ (эквивалентность значений указателей),
4. $\forall a \in AM^p : \exists \langle d, \gamma \rangle \in \mathcal{P}(a) : \psi(\gamma) = \top \wedge (\psi(d) = \mathbb{P}(\psi(a)) \vee d = \mu)$ (эквивалентность значений адресов в памяти),
5. $\forall m \in AArr : m \in \mathcal{B} \Rightarrow \mathcal{B}(m) = \mathbb{B}(\psi(m))$ (эквивалентность размеров массивов).

Теперь с помощью данной функции можно определить свойства корректности и точности абстрактного состояния.

Определение 3. Абстрактное состояние $\mathcal{A} = \langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle$ *корректно* описывает рассматриваемое анализом конкретное состояние, если существует такая эквивалентная ему функция конкретизации ψ, что $\psi(\pi) = \top$, т.е. условие достижимости корректно.

Определение 4. Абстрактное состояние $\mathcal{A} = \langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle$ *точно* описывает множество рассматриваемых анализом конкретных состояний данной точки, если для любой функции конкретизации ψ, для которой $\psi(\pi) = \top$, найдётся эквивалентное ей конкретное состояние.

Раздел 3.2.2 посвящен общей схеме построения достаточных условий ошибки. Пусть *Aux* — множество вспомогательных переменных анализа, не зависящих от значений $\vec{t}$ и описывающих свойства всей функции в целом. Значениями этих переменных могут являться битовые вектора произвольной ширины $b \in B$. Множество символьных выражений над такими переменными обозначим SE_b^{Aux}, пусть также $SE^{Aux} \doteq \bigcup_{b \in B} SE_b^{Aux}$. Отметим, что так как размеры всех буферов константны и не зависят от входных значений, то они принадлежат SE_r^{Aux}.

Искомое условие наличия ошибки переполнения может быть составлено как дизъюнкция условий ошибки для двух случаев: когда индекс отрицательный либо переполняет правую границу буфера. Т.к. определение 1 формулирует наличие ошибки через существование пути на графе развёртки, обладающего определёнными свойствами, то для построения достаточных условий ошибки по этому определению не получится рассматривать лишь отдельные состояния программы, но требуется анализировать свойства путей. Каждый путь на графе развёртки в результате анализа может быть представлен последовательностью абстрактных состояний, которая может использоваться для установления свойств этого пути. Рассмотрим некоторый путь $p = \{q_i\},\ q_i \in \mathit{Instr}$ на графе развёрки от входа в функцию до точки q_n и соответствующую ему последовательность абстрактных состояний $\{\mathcal{A}_i\}$. Обозначим $\pi_i = \pi(q_i)$.

Определение 5. Пусть для символьных выражений $v \in SE_b$, $x \in SE_b^{Aux}$ в точке программы q_n определено условие в виде формулы $NotLess(q_n, v, x) \in \mathcal{C}$, обладающее следующим свойством. Если найдётся такая последовательность функций конкретизации $\{\psi_i\}$ для $\{\mathcal{A}_i\}$, что $\forall\, i : \psi_i(\pi_i) = \top$, абстрактное состояние $\mathcal{A}_n$ точно и выполнено $\psi_n(NotLess(q_n, v, x)) = \top$, то гарантируется, что для любой последовательности функций конкретизации $\{\widetilde{\psi}_i\} : \forall i : \widetilde{\psi}_i(\pi_i) = \top$ заведомо выполнено $\widetilde{\psi}_n(v) \geqslant_s \widetilde{\psi}_n(x)$. Аналогичная формула $NotGreater(q_n, v, x) \in \mathcal{C}$ известна для отношения $\leqslant_s$.

***Теорема* 2.** *Пусть для инструкции q_{ac} доступа к буферу* `b` *по индексу* `i` *гарантируется, что абстрактное состояние в этой точке $\mathcal{A}_{q_{ac}} = \langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle$ точно, и в любой точке на любом пути от входа в функцию до q_{ac} соответствующее абстрактное состояние корректно. Тогда достаточным условием наличия ошибки в q_{ac} является формула:*

$$\pi(q_{ac}) \wedge \bigvee_{\substack{\langle m, \gamma \rangle \in \mathcal{P}(\mathtt{p}),\\ m \in \mathcal{B}}} \gamma \wedge (NotLess(q_{ac}, \sigma(\mathtt{i}), \mathcal{B}(m)) \vee NotGreater(q_{ac}, \sigma(\mathtt{i}), -1)) \tag{1}$$

Таким образом, задача поиска ошибок переполнения описанного типа сводится к вычислению как можно более слабых условий $NotLess(q, v, x)$ и $NotGreater(q, v, x)$. Для её решения для символьного выражения v в точке q определяется значение $\mathcal{V}(v) \in Summary$, суммирующее информацию о значениях v по всем путям, заканчивающимся в q. Искомые условия $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ будут вычисляться с помощью $\mathcal{V}(v)$. Далее значение v в точке q будем обозначать как $\mathcal{V}(q, v)$.

Предлагается организовать поиск ошибок доступа к буферу в три этапа:

1. В ходе анализа для выражений $v \in SE$ в каждой точке программы $q \in Instr$ построить отображение $\mathcal{V} : SE \to Summary$.
2. При обработке инструкции q_{ac} доступа к буферу `b` по индексу `i` на основе значения $\mathcal{V}(ac, \sigma(\mathtt{i}))$ составляется формула (1) и проверяется на выполнимость.
3. В случае, если формула выполнима, т.е. подобраны значения символьных переменных S, приводящие к переполнению, из $\mathcal{V}(ac, \sigma(\mathtt{i}))$ путём подстановки конкретных значений переменных извлекается конкретное выполнение, приводящее к ошибке, и выдается предупреждение, указывающее на соответствующий этому выполнению путь.

Описание отображения $\mathcal{V}$, его построения и алгоритма вычисления условия $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ с его помощью приведено в разделе 3.2.3. Значения множества *Summary* принадлежат одному из шести типов: C — константы; *Assume* — отношение над символьными выражениями, вытекающее из условия перехода, доминирующего данную точку; *Arithm* — результат арифметической операции; *Cast* — результат инструкции приведения типа; *Join* —

совокупность нескольких значений *Summary*, снабженных условиями. Элемент *Summary* создаётся для символьного выражения в соответствующей инструкций, если для её операндов определено отображение $\mathcal{V}$. Элементы *Summary* имеют вид ориентированного ациклического графа. Построение формул $\mathit{NotLess}(q, v, x)$ и $\mathit{NotGreater}(q, v, x)$, восстановление трасс срабатываний и миграция в контекст вызова при межпроцедурном анализе происходит рекурсивно, в каждой вершине графа процедура шага рекурсии определяется типом *Summary*. Для каждого из типов *Summary* приведены условия $\mathit{NotLess}(q, v, x)$ и $\mathit{NotGreater}(q, v, x)$ и доказана лемма о том, что они удовлетворяют определению 5.

В разделе 3.3 приводятся передаточные функции для инструкций модельного языка (кроме инструкции вызова) и для каждой инструкции приводится обоснование сохранения корректности и точности абстрактного состояния (за исключением инструкции чтения из буфера, для которой гарантируется только корректность). Для передаточных функций, модифицирующих отображение $\mathcal{P}$, показано сохранение однозначности выбора значения. К примеру, для инструкции `x = load(p)`, $\texttt{x} \in V_b, \texttt{p} \in P_b$, которая загружает значение из ячейки памяти по адресу `p` в переменную `x`, для случая, когда `p` может указывать более чем на одну ячейку, передаточная функция выглядит следующим образом:

$$\text{LOADV-2}\ \frac{\begin{array}{cc} \mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle & q \vdash \texttt{x = load(p)} \\ \mathcal{P} \vdash \texttt{p} \to \{\langle a_i, \gamma_i \rangle\} & \forall a_i : \sigma \vdash a_i \to v_i \\ v = \mathit{ite}(\gamma_1, v_1, \mathit{ite}(\gamma_2, v_2, \ldots)) & s_v = \mathit{joinVals}(v,\ \{\langle \mathcal{V}(v_i), \gamma_i \rangle\}) \end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\{v \mapsto s_v\} \rangle}$$

Раздел 3.3.7 посвящён процедуре объединения состояний, для неё также доказано сохранение корректности и точности абстрактного состояния. В разделе 3.4 описано изменение семантики инструкций при анализе циклов, не нарушающее корректности абстрактного состояния.

Раздел 3.5 посвящён доказательству корректности анализа.

***Теорема* 3.** *Для любой функции на модельном языке на любом ребре графа развёртки при выполнении следующих условий:*

1. *среди параметров функции нет алиасов,*
2. *ни на одном пути от входа до данной точки нет инструкций вызова,*

абстрактное состояние анализа корректно в данной точке.

***Теорема* 4.** *Пусть для произвольной функции на модельном языке на любом ребре графа развёртки верны условия теоремы 3 и дополнительно выполнено:*

1. *ни один из путей до данной точки не проходит через k-ую итерацию цикла,*
2. *ни на одном пути от входа до данной точки нет инструкций чтения значения из массива.*

Тогда абстрактное состояние анализа точно в данной точке.

Таким образом, если на любом ребре от входа в функцию до инструкции доступа выполнены условия теорем 3 и 4, то выполнены условия теоремы 2 и достаточным условием наличия ошибки будет выполнимость формулы (1).

В **четвёртой главе** приведено описание межпроцедурного контекстно-чувствительного алгоритма поиска переполнений буфера, применяющего предложенный внутрипроцедурный метод для построения резюме функции и поддерживающего буферы с константным размером.

Межпроцедурный анализ реализуется с помощью резюме: все функции анализируются единожды в порядке от листьев к корню в графе вызовов программы, приведённом к ациклическому виду разрывом некоторых рёбер. На основе результата внутрипроцедурного анализа функции формируется и сохраняется её резюме, т.е. краткое описание эффекта от её исполнения. При обработке инструкции вызова известной функции происходит применение её резюме, которое обязательно уже сформировано в силу порядка обхода функций (кроме случая рекурсивных и косвенно-рекурсивных функции). К преимуществам данного подхода можно отнести однократный анализ каждой функции и возможность построить контекстно-чувствительный анализ.

Будем считать, что в резюме функции записывается объединённое абстрактное состояние из состояний на выходе из инструкций `return`.

Алгоритм применения резюме заключается в следующем:

1. В ходе внутрипроцедурного анализа при инициализации ячеек памяти строится *граф памяти* на входе в функцию — ориентированный граф, вершинами которого являются формальные параметры типов `ptr`, `arr`, абстрактные ячейки памяти и абстрактные массивы, а дуги определяются отношением «указывает на». Кроме этого, для параметров и абстрактных ячеек памяти, хранящих битовые векторы, запоминаются созданные символьные переменные.
2. При применении резюме с помощью этого графа строится функция $\mathfrak{M}$, определяющая соответствие его вершин абстрактным ячейкам памяти и массивам в контексте вызова.
3. Также на основе информации из п. 1 символьным переменным ставятся в соответствие символьные выражения, содержащиеся в точке вызова в соответствующих фактических аргументах и ячейках памяти.
4. Символьные выражения «мигрируют» в контекст вызываемой функции рекурсивно, базой рекурсии является соответствие символьных переменных и символьных выражений, построенное в предыдущем пункте. Полученное отображение из символьных выражений вызываемой функции в символьные выражения вызывающей функции обозначим $\mathfrak{E}$. По $\mathfrak{E}$ определяется соответствие условий $\mathfrak{C}$.
5. Абстрактное состояние, помещённое в резюме, используется в качестве передаточной функции для инструкции вызова. С помощью отображения $\mathfrak{M}$ обновляются отображения $\sigma, \mathcal{P}$ в контексте вызова в соответствии с их

значениями в резюме. Переменной, в которую в точке вызова сохраняется результат функции, ставится в соответствие $\mathfrak{E}(\sigma(f_{ret}))$.

6. При сопоставлении символьных выражений мигрируют также значения $\mathcal{V}$ для этих выражений.

   Миграция значений $\mathcal{V}$ описана в разделе 4.2.

Рассмотрим ошибку доступа к буферу, возникающую в инструкции `p[i] = h`. В данной инструкции используются две переменные — адрес буфера `p` и индекс `i`, значение каждой из которых может определяться в рамках текущей функции либо вычисляться с помощью значений, вычисляемых в вызываемых или вызывающих функциях. В рамках текущей работы для переменной, содержащей адрес буфера, будем рассматривать два варианта: адрес известен в самой функции (явно используется определенный в данной функции или глобальный массив), адрес передан в качестве параметра-указателя.

Первый случай рассмотрен в разделе 4.2. Для вычисления достаточного условия ошибки в вызывающей функции необходимо поместить в резюме значение $\mathcal{V}$ для возвращаемого значения. При применении резюме оно будет транслировано в контекст вызывающей функции, где может быть использовано для проверки инструкций доступа или создания других значений.

Так как анализ функций производится «снизу-вверх», то при анализе функции ничего не известно о возможных значениях её параметров. Для того, чтобы начать отслеживать такие межпроцедурные зависимости между значениями, было введено ещё два класса значений атрибута $\mathit{FParam} \cup \mathit{AParam} \subset \mathit{Summary}$. Значение атрибута типа $\mathit{FParam} = \{v \mid v \in S\}$ сопоставляется каждому формальному аргументу функции (и значению памяти, определённому в вызывающем контексте) и содержит его символьную переменную. Далее производится обычный анализ, и его результаты сохраняются в резюме.

При применении резюме, если в контексте вызывающей функции символьное выражение $v \in \mathit{SE}$, передаваемое в качестве фактического аргумента, имело некоторое непустое значение $\mathcal{V}$ в точке вызова $\mathcal{V}(p_{call}, v) = s^{actual}$, то для всех мигрирующих из резюме значений атрибутов происходит подстановка на место листа-формального параметра нового значения $s_v = \langle \mathfrak{E}(v), s^{actual} \mid v \in \mathit{SE}, s^{actual} \in \mathit{Summary} \rangle \in \mathit{AParam}$.

Миграция значений *Summary* выполняется с помощью отображения $\mathfrak{V}$. Константы не нуждаются в миграции, остальные значения *Summary* мигрируют рекурсивно с помощью отображения $\mathfrak{E}$. Единственная нетривиальная процедура выполняется при миграции значения *FParam* — она описана выше.

***Теорема* 5.** *Значение $\mathcal{V}$, полученное в результате межпроцедурного анализа с применением резюме функции, сохраняет корректность, т.е. для него по-прежнему остаётся верна теорема 2.*

В контексте рассмотрения локальных массивов, кроме этого, необходимо поддержать случаи, когда размер буфера или значение индекса определяется в одной из функций-предков (с точки зрения графа вызовов) по отношению

к инструкции доступа к буферу (см. раздел 4.3). Для этого было введено два типа факта доступа к буферу внутри функции, сохраняемых в резюме для последующего анализа в вызывающей функции: факт доступа к буферу известного размера, хранящий значение $\mathcal{V}$ для индекса, условие доступа и размер буфера; и факт доступа к буферу неизвестного размера, который вместо размера хранит возможные значения абстрактного массива с условиями. В данном разделе интерес представляет только ошибка выхода за правую границу буфера.

Рассмотрим некоторую инструкцию доступа к буферу p_{access} : `p[i] = h`. Если для символьного выражения индекса в данной точке значение атрибута $\mathcal{V}$ не определено, то информация об его возможных значениях отсутствует как в данной функции, так и во всех вызывающих (т.к. для переменных и значений памяти, определённых в вызывающей функции, создаются значения *FParam*), поэтому проверить такой доступ невозможно. Предположим, что значение атрибута $\mathcal{V}$ для индекса определено и равно s_i. Пометим в соответствующем ему графе все листовые вершины типа *FParam*, далее рекурсивно пометим все вершины, у которых хотя бы один из потомков помечен (кроме вершин типа *Join*, которые помечаются, если все потомки помечены). Непомеченные вершины соответствуют значениям, полностью определённым в данной функции. Помеченные вершины соответствуют значениям, которые могут быть полностью определены только в функции «выше» по графу вызовов.

Если в s_i есть непомеченные вершины, то необходимо проверить переполнение в данной точке в ходе анализа текущей функции с помощью формулы (1). Если ошибка была найдена, то проверка этой инструкции доступа заканчивается.

Если ошибка в данной функции не была обнаружена, значение s_i содержит помеченные вершины и размер хотя бы одного массива, на который может указывать `p`, известен, то в резюме записывается факт доступа к буферу известного размера.

Если для некоторых массивов, к которым может происходить обращение в данной точке, размер известен только в вызывающей функции (адрес буфера передан в качестве параметра), то в резюме записывается факт доступа к буферу неизвестного размера.

Теперь рассмотрим алгоритм применения резюме в точке p_{call}. Предположим, что в нём содержится факт доступа к буферу внутри вызываемой функции. Значение s_i при применении резюме трансформируется в значение $s^{actual} = \mathfrak{V}(s_i)$ по обычным правилам, описанным в разделе 4.2.

Рассмотрим случай, когда в резюме записан факт доступа к буферу известного размера. Тогда, если в s^{actual} есть непомеченные вершины, то необходимо проверить наличие ошибки в данной точке с помощью факта доступа. Если ошибка была обнаружена, то проверка этого факта доступа заканчивается. В противном случае, если s^{actual} содержит помеченные вершины, то в резюме записывается факт доступа к буферу известного размера.

Если в резюме записан факт доступа к буферу неизвестного размера и в s^{actual} есть непомеченные вершины, то проверяется условие ошибки для этого факта доступа для массивов, размер которых стал известен в этой функции.

Если ошибка не была обнаружена и s^{actual} содержит помеченные вершины, то в резюме записывается в общем случае два факта доступа: один для доступа к массивам, размер который стал известен в этой функции, и второй для остальных массивов.

***Теорема* 6.** *Пусть функция f содержит инструкцию $p_{callG} \in Instr$ вызова функции g. Если*

1. *в резюме функции g содержится факт доступа к буферу,*
2. *выполнимо условие ошибки, соответствующее виду факта доступа,*
3. *в любой точке на любом пути от входа в функцию f до инструкции доступа соответствующее абстрактное состояние корректно,*
4. *во всех точках вызова на любом пути от p_{callG} до p_{access} абстрактное состояние точно,*

то в точке p_{callG} функции f имеется ошибка переполнения.

***Теорема* 7.** *Рассмотренный алгоритм анализа программы для поиска переполнения буфера завершается для любой программы.*

В **пятой главе** рассматриваются расширения предложенных методов для поиска переполнений при работе со строками языка C, при использовании недоверенных входных данных, а также в циклах.

Первым из рассмотренных случаев является переполнение при работе со строками (раздел 5.1). Особенностью таких ошибок переполнения является тот факт, что работа со строками в языке C преимущественно осуществляется с помощью библиотечных функций. При этом, как правило, происходит доступ к элементам массива по различным индексам, наибольший из которых равен длине строки. Такое поведение само по себе небезопасно в случае, когда невозможно гарантировать, что длина строки заведомо меньше размера отведённого под неё массива. Поэтому в таких случаях используют «безопасные» версии функций, принимающие в качестве параметра число, с помощью которого ограничивается максимальный индекс доступа к строке. Таким образом, при анализе программы, использующей строки, необходимо по крайней мере моделировать длины строк для проверки безопасности доступа к соответствующим массивам.

Для описания алгоритмов поиска переполнения при работе со строками расширим модельный язык. Для краткости изложения при формальном описании алгоритмов будем считать, что один символ кодируется одним байтом. Строками будем считать массивы из однобайтовых элементов и строковые литералы. Последние можно использовать только в качестве аргументов функции.

Поддержка строковых операций будет заключаться в 1) расширении абстрактного состояния отображением, задающим длину каждой строки; 2) расширении отображения $\mathcal{V}$ для обнаружения ошибок при работе со строками.

Содержимое каждой строки в абстрактном состоянии будет представлено одним символьным выражением, означающем длину данной строки. Выбор такой абстракции, с одной стороны, позволит получить более точное абстрактное состояние, что в свою очередь поможет найти больше ошибок и не выдавать ложных предупреждений на невыполнимых путях, предикаты которых содержат условия на длины строк. С другой стороны, добавление всего лишь одного символьного выражения для каждой строки не приведёт к значительному увеличению размера абстрактного состояния.

Итак, добавим в абстрактное состояние $\mathcal{A}$ новое отображение $\mathcal{S} : \mathit{String} \to \mathit{SE}_r$. С учётом нового отображения дополним определение эквивалентности функции конкретизации ψ абстрактного состояния $\langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \,\rangle$ конкретному состоянию $\langle \mathbb{X}, \mathbb{P}, \mathbb{B} \rangle$. К пяти условиям, перечисленным в определении 2, добавим также условие эквивалентности длин строк.

Для обнаружения переполнения буфера при вызове библиотечных функций работы со строками доопределим отображение $\mathcal{V}$ для символьных выражений, соответствующих длинам строк. Т.к. длина любого строкового литерала является константой, а на множестве констант отображение $\mathcal{V}$ тождественно, то расширение $\mathcal{V}$ для строковых литералов тривиально. Для моделирования функций простого копирования и конкатенации (без ограничения на длину), функций доступа к массиву символов можно воспользоваться уже введёнными типами из множества *Summary*. Однако при обработке «безопасных функций» (`strncpy`, `strncat`) для длин изменяющихся строк использован новый тип элементов $\mathit{Min} \subset \mathit{Summary}$, его свойства описаны в разделе 5.1.2.

Модификации передаточных функций анализа для функций работы со строками рассмотрены в разделе 5.1.3, где, как и в разделе 3.3, для них показывается сохранение корректности и точности абстрактного состояния.

Для организации межпроцедурного анализа объединение отображений $\mathcal{S}$ для точек выхода из функции помещается в резюме функции. При применении резюме в точках вызова значения $\mathcal{S}$ из резюме транслируются в контекст вызывающей функции и сопоставляются соответствующим переменным из множества *String* по тому же принципу, как происходит трансляция отображения σ. Значения типа *Min* транслируются естественным образом: значение в контексте вызова составляется из соответствующих оттранслированных значений аргументов. Аналогично реализуется поддержка широких строк (раздел 5.1.4).

Одним из наиболее серьезных источников уязвимостей, связанных с переполнением буфера, является обращение к буферу по индексу, вычисленному на основе данных, полученных из недоверенного источника. Описанный базовый алгоритм был расширен (раздел 5.2) для организации поиска подобных ошибок. Достаточно предъявить один-единственный набор входных данных, приводящий к переполнению, чтобы утверждать о наличии ошибки в программе, поэтому требование независимости ошибки от входных данных снимается.

Для обнаружения дефектов рассматриваемого типа анализ циклов особенно важен, т.к. обработка значений буферов обычно реализуется с помощью

циклов, и ошибки зачастую возникают на последней итерации, а также после цикла при использовании вычисленных в цикле значений. Для поиска подобных дефектов используется информация об индуктивных переменных, полученная с помощью консервативного анализа (раздел 5.3).

Шестая глава посвящена описанию метода поиска переполнений для буферов произвольного размера, который применяет как предложенные методы анализа для буферов константного размера (раздел 6.1), так и метод перебора значений условий переходов непосредственно на основе разработанного формального определения ошибки (раздел 6.2).

В первую очередь расширим модельный язык инструкцией выделения нового массива, в которой количество элементов нового размера задаётся значением некоторой переменной (а не только константой, как было ранее). Расширим также отображение $\mathcal{B}$ — теперь размер массива будет моделироваться произвольным символьным выражением ширины r: $\mathcal{B} : A \to SE_r$.

Первый подход описан в разделе 6.1 и заключается в использовании тех же условий *NotLess* и *NotGreater* не только для анализа значения индекса массива, но и для анализа размера выделенного буфера. Если для инструкции доступа `ac: x = m[i]` найдётся такая константа, что для некоторого выполнимого пути размер выделенного буфера всегда будет не больше этой константы, а индекс — не меньше, то такой путь будет удовлетворять условию ошибки. Формально это условие можно записать в виде следующей теоремы.

***Теорема* 8.** *Пусть для инструкции доступа к буферу q_{ac} `: x = p[i]` $\in$ Instr гарантируется, что абстрактное состояние в этой точке $\mathcal{A}_{q_{ac}} = \langle\, \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \,\rangle$ точно, и в любой точке на любом пути от входа в функцию до q_{ac} соответствующее абстрактное состояние корректно. Тогда достаточным условием наличия ошибки переполнения буфера в этой точке будет являться выполнимость формулы:*

$$\exists \chi \;:\; \pi(ac) \bigvee_{\langle m,\gamma\rangle \in \mathcal{P}(p), m \in \mathcal{B}} \gamma \wedge \mathit{NotLess}(ac, \sigma(i), \chi) \wedge \mathit{NotGreater}(ac, \mathcal{B}(m), \chi).$$

Данный подход может быть применён и для межпроцедурного случая, если вместо константного размера буфера использовать $\mathcal{V}(h)$ для аргумента инструкции `p = alloca(b, h)`.

Второй подход описан в разделе 6.2 и заключается в использовании достаточного условия ошибки с кванторами всеобщности. Перенумеруем все элементарные условия инструкций `assume`, соответствующие переходу по `if`-ветке, и обозначим полученную последовательность $\{W_i \mid W_i \in SimpleCond\}$. Каждое конкретное значение этой последовательности определяет путь (возможно, невыполнимый) на графе развёртки. В определении ошибки будем рассматривать не пути на графе, а конкретные наборы значений $\{w_i \mid w_i \in \{\top, \bot\}\}$ последовательности $\{W_i\}$.

Данный подход позволяет существенно сократить размер формулы за счёт единого предиката достижимости точки для всех путей, но по-прежнему требует

проверки на разрешимость формулы с кванторами всеобщности, что ограничивает его масштабируемость. Кроме того, никак не учитывается тот факт, что для произвольного пути выполнения некоторые значения $\{w_i\}$ могут остаться невычисленными и это может стать причиной выдачи ложного предупреждения.

Пусть $V \doteq \{V_i = \sigma_{q_i}(\mathtt{h}) \diamond \sigma_{q_i}(\mathtt{g}) \mid q_i : \texttt{ assume h} \diamond \mathtt{g} \in \mathit{Instr}\} \subset \mathcal{C}$ — множество элементарных условий над символьными переменными, созданных при интерпретации инструкций `assume`. Это множество будет использоваться в качестве аналога W для абстрактного состояния. Пусть $\pi_q^{\{v_i\}} \doteq \pi \wedge \bigwedge_{V_i} V_i = v_i$, где $\forall i : v_i \in \{\top, \bot\}$, $E \doteq \bigvee_{\langle m,\gamma\rangle \in \mathcal{P}(p), m \in \mathcal{B}} \gamma \wedge (\sigma(i) \geqslant_s \sigma(\mathcal{B}(m)))$. Тогда искомая формула имеет вид $\exists\{v_i\} : \quad \exists\vec{s} : \pi_q^{\{v_i\}} \wedge \forall\vec{s} : \pi_q^{\{v_i\}} \Rightarrow E$.

Седьмая глава посвящена реализации предложенных алгоритмов в рамках инструмента статического анализа Svace и обсуждению полученных результатов. Svace — статический анализатор исходного кода, разрабатываемый в ИСП РАН. Его ключевыми особенностями являются хорошая масштабируемость, высокая доля истинных срабатываний (60-80 %) и большое количество типов обнаруживаемых ошибок. Поддерживается анализ языков C, C++, Java.

Анализатор представляет собой совокупность ядра и детекторов, отвечающих за поиск конкретных типов дефектов. Рассмотренный подход был реализован в виде нескольких детекторов анализатора Svace.

В общих чертах поиск ошибок осуществляется в соответствии с описанным алгоритмом. При проверке инструкций доступа и вызова функций для выдачи предупреждения сначала производится быстрая проверка с помощью интервалов значений индекса. Если отсутствие ошибки доказано не было, то рекурсивно строятся достаточные условия ошибки. Полученные условия формулируются в виде запроса на языке SMTLib2 и передаются решателю Z3 для проверки на выполнимость. Если формула оказалась выполнима, то из решателя получается модель условия, по которой строится конкретный ошибочный путь. Данный путь используется для генерации сообщения об ошибке. Сообщение содержит общее описание ошибки и для каждого узла из $\mathcal{V}$ поясняющую строку о причинах конкретного срабатывания.

Для удобства пользователя и вследствие использования различных подходов к обнаружению переполнения все предупреждения об ошибке разделены на несколько типов:

- `BUFFER_OVERFLOW.EX` — переполнение буфера произвольного типа и константного размера, происходящее при разыменовании некорректного указателя (инструкции индексации);
- `BUFFER_OVERFLOW.LIB.EX` — переполнение буфера константного размера при использовании библиотечных функций, осуществляющих доступ к переданному в качестве аргумента буферу (например, `memcpy`);
- `OVERFLOW_AFTER_CHECK.EX` — доступ к буферу после подозрительного сравнения индекса с некоторым числом (неправильной проверки), в том числе предупреждения детектора, разработанного на основе эвристик для анализа циклов, рассмотренных в разделе 5.3;

– BUFFER_UNDERFLOW — переполнение левой границы буфера (обращение по отрицательному индексу);
– TAINTED_ARRAY_INDEX.EX — доступ к буферу по индексу, вычисленному из недоверенных данных;
– BUFFER_OVERFLOW.STRING — переполнение буфера при работе со строками;
– DYNAMIC_OVERFLOW.EX — переполнение буфера произвольного размера.

Для снижения количества ложных срабатываний был выделен ряд типичных ситуаций, в которых нарушается предположение о контрактах; для этих ситуаций были разработаны подавляющие ложные срабатывания эвристики.

Для оценки эффективности разработанных методов было произведено сравнение со статическим анализатором Infer (см. раздел 1.1.2). Для поиска ошибок переполнения буфера в Infer имеется экспериментальный детектор InferBO.

Для тестирования использовался набор синтетических тестов Juliet Test Suite C/C++ 1.3. Для задач настоящего исследования интерес представляли группы CWE 121, CWE 122, CWE 126 (переполнение правой границы массива) и CWE 124, CWE 127 (переполнение левой границы). Кроме описанных выше типов предупреждений, в рассмотрение также включены результаты нечувствительного к путям детектора STRING_MISMATCH_WIDE_NARROW — использование функций для обычных строк при работе с широкими строками (CWE 135).

Ложные срабатывания детекторов ошибки переполнения буфера у инструмента Svace на выбранных группах тестов составили 0,02 % от числа корректных функций и все относились к переполнению левой границы. Общее число обнаруживаемых Svace ошибок составляет 53,8 % для переполнения правой границы и 47,6 % в общем. Для Infer аналогичные показатели составляют: 9,4 % всего и 9,2 % для переполнения правой границы. Ложных срабатываний выдано не было.

Результаты работы детекторов на проекте Android 5.0.2 (в части, написанной на C/C++) приведены в таблице 1. Для каждого из типов выданных предупреждений приведено общее число срабатываний и результаты разметки. В категорию истинных срабатываний (true positive, TP) при разметке попадают предупреждения, анализ которых привёл к выводу о наличии ошибки в коде. К категории ложных срабатываний (false positive, FP) попадают предупреждения об ошибках, которые в реальном коде никогда не происходят («ошибки» анализа, вызванные неточностью модели программы или недостатками реализации). К категории «не ошибка» (won't fix, WF) относятся срабатывания, верно сообщающие о формально ошибочных, но намеренно созданных или не нуждающихся в исправлении (с точки зрения разработчика) ситуациях.

Как следует из таблицы, количество ложных срабатываний для проекта Android находится в заданных ограничениях (в среднем не превосходит 35 %). На проекте Tizen 5.0 было выдано 200 предупреждений о переполнении правой границы, среди них 33 % ложных. Основной причиной ложных срабатываний является неточность построенных условий, которая, в свою очередь, вызвана

Таблица 1 — Результаты тестирования на проекте Android 5

Тип срабатывания	Кол-во	TP		WF		FP	
		n	%	n	%	n	%
BUFFER_OVERFLOW.EX	30	15	50 %	7	23 %	8	27 %
BUFFER_OVERFLOW.LIB.EX	20	5	25 %	1	5 %	14	70 %
OVERFLOW_AFTER_CHECK.EX	62	40	65 %	14	22 %	8	13 %
BUFFER_OVERFLOW.STRING	5	4	80 %	1	20 %	0	0 %
DYNAMIC_OVERFLOW.EX	1	0	0 %	1	100 %	0	0 %
BUFFER_UNDERFLOW	28	4	14 %	10	36 %	14	50 %
Всего	**146**	**68**	**47 %**	**34**	**23 %**	**44**	**30 %**

неточным межпроцедурным анализом, неточным моделированием памяти (в т.ч. содержимого массивов) и т.п. Другой причиной ложных срабатываний является нарушение предположения о контрактах.

Анализ производился в 8 потоков на сервере с Intel® Core™ i7-6700 с 32GB RAM. Общий анализ проекта Android 5.0.2 занял 449 минут. Суммарно операции, реализованные в детекторах переполнения буфера (все, связанные с $\mathcal{V}$, и проверки доступа и фактов доступа), заняли 72 минуты процессорного времени, что вполне приемлемо на фоне общего времени анализа с учётом важности этого типа ошибок. Дополнительно потребовалось 11 минут для вычисления абстракции для длин строк.

В целом можно заключить, что обработка условий является главным узким местом данного анализа с точки зрения производительности. В этой связи перспективными направлениями исследования являются методы упрощения формул (возможно, неэквивалентного), определения несовместных формул на ранних этапах, использование инкрементального режима решателя, хранение только наиболее «полезных» в дальнейшем фактов анализа (разработка метода спекулятивного определения потенциальной «полезности»).

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

1. Предложено формальное определение ошибки доступа к буферу, и для него разработан внутрипроцедурный, различающий пути выполнения метод статического анализа на основе символьного выполнения с объединением состояний, который позволяет строить достаточные условия возникновения ошибки переполнения в некоторой точке функции для случаев доступа к буферу известного на момент компиляции размера. Сформулированы ограничения на анализируемые программы, в рамках которых доказаны корректность и точность предложенных методов анализа.
2. Разработан межпроцедурный контекстно-чувствительный алгоритм для поиска переполнений буфера, применяющий предложенный внутрипроцедурный метод для построения резюме функции и поддерживающий буферы с известным на момент компиляции размером. Доказана корректность разработанного алгоритма.

3. Разработан метод поиска переполнений для буферов произвольного размера, который применяет как предложенные методы анализа для буферов константного размера, так и метод перебора значений условий переходов непосредственно на основе разработанного формального определения ошибки.
4. Разработаны расширения предложенных методов для поиска переполнений при работе со строками языка C, а также при использовании недоверенных входных данных.
5. Предложенные методы реализованы в статическом анализаторе Svace для программ на языках C/C++ и показали масштабируемость до миллионов строк исходного кода, покрытие тестов на переполнение правой границы буфера из набора Juliet Test Suite в 54 % полностью без ложных срабатываний и точность анализа в 70 % истинных срабатываний для исходного кода операционной системы Android версии 5.0.2.

Публикации автора по теме диссертации

1. *Дудина И. А.*, *Белеванцев А. А.* Применение статического символьного выполнения для поиска ошибок доступа к буферу // Программирование. — 2017. — № 5. — С. 3—17.
2. *Дудина И. А.* Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа // Труды Института системного программирования РАН. — 2016. — Т. 28, № 5. — С. 119—134.
3. *Дудина И. А.*, *Кошелев В. К.*, *Бородин А. Е.* Поиск ошибок доступа к буферу в программах на языке C/C++ // Труды Института системного программирования РАН. — 2016. — Т. 28, № 4. — С. 149—168.
4. Design and Development of Svace Static Analyzers / A. Belevantsev [et al.] // 2018 Ivannikov Memorial Workshop (IVMEM). — 05/2018. — P. 3—9.
5. *Дудина И. А.*, *Малышев Н. Е.* Об одном подходе к анализу строк в языке Си для поиска переполнения буфера // Труды Института системного программирования РАН. — 2018. — Т. 30, № 5. — С. 55—74.
6. *Dudina I. A.* Buffer Overflow Detection via Static Analysis: Expectations vs. Reality // Proceedings of ISP RAS. — 2018. — Vol. 30, no. 3. — P. 21—30.
7. *Дудина И. А.*, *Белеванцев А. А.* К вопросу о преодолении ограничений статического анализа при поиске дефектов переполнения буфера // Ломоносовские чтения: Научная конференция, Москва, факультет ВМК МГУ имени М.В.Ломоносова, 17-26 апреля 2017 г. Тезисы докладов. — МАКС Пресс Москва, 2017. — С. 35—36.
8. *Дудина И. А.*, *Белеванцев А. А.* Методы организации межпроцедурного анализа для поиска ошибок переполнения буфера // Ломоносовские чтения 2018 ф-т ВМК МГУ. — МАКС Пресс Москва, 2018. — С. 33—34.