Федеральное государственное бюджетное учреждение науки
Институт системного программирования имени В. П. Иванникова
Российской академии наук
Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

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»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук
Белеванцев Андрей Андреевич

Москва — 2019

Оглавление

Стр.

Стр.

Стр.

Введение

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.

Объем и структура работы. Диссертация состоит из введения, семи глав и заключения, а также одного приложения. Полный объём диссертации составляет 145 страниц, включая 20 рисунков и 6 таблиц. Список литературы содержит 76 наименований.

Глава 1. Ошибка переполнения буфера и подходы к её обнаружению

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++: неприятная особенность ошибок при работе с памятью в этих языках заключается в том, что возникновение такой ошибки зачастую не приводит к мгновенным последствиям (таким как, например, аварийное завершение программы или генерация исключения). С одной стороны, это затрудняет тестирование и отладку программ, т.к. наблюдаемые аномалии в поведении программы могут проявиться гораздо позже реального возникновения ошибки (или не проявиться вовсе). С другой стороны, наличие такого дефекта может привести к появлению в программе эксплуатируемой уязвимости, позволяющей злоумышленнику спровоцировать отказ в обслуживании, получить доступ к чувствительным данным и даже захватить контроль над атакуемым устройством [9; 10]. По данным национальной базы уязвимостей США (NVD) ошибки подобного рода являются причиной 9,49 % всех уязвимостей, занесённых в базу CVE в 2018 г. [11]. Этот тип стал одним из двух наиболее распространённых типов уязвимостей в 2018 г., наряду с XSS (Cross-Site Scripting), доля которого составила 10,19 %.

В других языках программирования также возможна ситуация переполнения буфера, которая может привести к аварийному завершению программы. Например, в программе на языке Java в таком случае будет сгенерировано исключение `IndexOutOfBoundsException`, в Python — `IndexError`. Настоящая работа посвящена анализу переполнения для языков C и C++, однако рассматриваемые подходы могут быть применены и к программам на других языках.

1.1 Обзор существующих подходов к поиску переполнения буфера

В связи с тем, что наличие ошибок переполнения буфера в готовой программе может привести к значительному ущербу, масштаб которого зависит от сферы

назначения ПО, во многих случаях рационально затратить дополнительные усилия на превентивное обнаружение и исправление подобных ошибок.

Если для небольших программ это может быть сделано вручную специалистом по аудиту исходного кода, то исследование больших программных систем невозможно без применения автоматических, или по крайней мере автоматизированных подходов. Спектр таких подходов чрезвычайно широк. На самом верхнем уровне можно выделить два подхода: динамический анализ, предполагающий изучение конкретных примеров исполнения программы, и статический анализ, представляющий собой анализ программы без её запуска. К методам динамического анализа относятся динамическая и статическая инструментация, динамическое символьное исполнение, фаззинг (см. раздел 1.1.1). Методы статического анализа (раздел 1.1.2) включают формальную верификацию, лексический и синтаксический анализ, чувствительные к потоку и путям подходы.

В языке C доступ к массиву определяется как разыменование указателя на один из элементов этого массива. Значение указателя вычисляется как смещение относительно адреса одного из элементов (либо адреса, следующего непосредственно за концом массива) на некоторое количество элементов. При этом по стандарту языка [12, §6.5.6/8], если вычисленный указатель не содержит адрес одного из элементов массива (выходит за границы), то разыменование такого указателя приведёт к неопределённому поведению[1]. Таким образом, для определения подобной ошибки перед инструментом анализа стоит задача определить для каждой точки разыменования размер массива и величину смещения указателя относительно начала этого массива.

1.1.1 Динамический анализ

В первую очередь рассмотрим динамический анализ кода. К этой категории можно отнести динамическую (Valgrind, DynamoRIO, Pin) и статическую (AddressSanitizer) инструментацию, динамическое символьное исполнение, фаззинг [13]. Коротко говоря, подходы к анализу различаются способами, которыми генерируются входные данные для наблюдаемых конкретных исполнений, и собственно техникой наблюдения. Подробное рассмотрение методов динамического

[1] Вычисленный указатель может указывать на элемент смежного массива — тогда такое разыменование будет корректным.

анализа не является предметом данной работы; тем не менее, отметим, что преимуществами данных методов является отсутствие ложных срабатываний, предоставление пользователю входных данных, на которых происходит ошибка. Кроме того, для проведения динамического 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.

1.1.2 Статический анализ

Исследование всех путей в программе возможно с помощью статического анализа, который предполагает анализ кода программы без её запуска. К преимуществам этого подхода можно также отнести возможность анализа неполной программы. Например, можно производить анализ функций, вызовы которых в проекте отсутствуют (например, если их планируется использовать в будущем, либо исследуемый проект является библиотекой).

К сожалению, следствием теоремы Райса [14] является алгоритмическая неразрешимость задачи поиска критических дефектов 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.

Формальная верификация

Методы формальной верификации программ, в частности, интересующие нас методы проверки моделей изначально применялись для анализа ПО, к которому предъявляются высочайшие требования к надежности (например, ПО для медицинского оборудования), однако в последнее время сфера их применения

интенсивно расширяется (некоторые примеры можно найти в [15]). Такие инструменты могут подтвердить отсутствие ошибок в анализируемой программе, однако могут вводить ограничения на множество используемых конструкций языка [16]; кроме того, используемые тяжеловесные подходы плохо масштабируются на программы большого размера. Эти инструменты не пропускают ошибок, однако могут допускать ложные срабатывания. Среди данных подходов с точки зрения поиска ошибок стоит отметить в первую очередь ограничиваемую проверку моделей (Bounded Model Checking, BMC) [17] и уточнение абстракций по контрпримерам (CounterExample-Guided Abstraction Refinement, CEGAR) [18].

Подход ограничиваемой проверки моделей заключается в итеративной проверке условия корректности программы. Для текущего значения k выполняется разворачивание циклов на глубину не более k итераций. Условие наличия среди рассматриваемых на данной итерации путей ошибочного записывается в виде формулы алгебры логики и проверяется с помощью SAT/SMT-решателя. Если ошибочного пути не найдено и не выполнены условия разворачивания (unwinding assertions), значит, не существует более «глубоких» непроверенных выполнимых путей, и, таким образом, доказано проверяемое свойство корректности программы. В противном случае, если порог максимального числа шагов не достигнут, то происходит следующая итерация с большим k. Такие инструменты, например, CBMC [19], способны с битовой точностью моделировать поведение программы и поддерживать сложные конструкции языков программирования. В то же время данный подход не позволяет находить ошибки на путях, требующих раскрытие циклов на глубину, превышающую заданный порог.

Метод CEGAR, как правило, формулируется для задачи определения достижимости ошибочной точки, к которой, в частности, можно свести задачу поиска переполнения буфера. Идея метода заключается в построении изначально грубой предикатной абстракции, но заведомо сохраняющей достижимость точки, с последующим её уточнением. На очередном шаге алгоритма в текущей абстракции ошибочная точка может уже быть недостижима, что гарантирует корректность программы. В противном случае имеется согласующийся с абстракцией путь, который проверяется на выполнимость. Если он выполним, то в программе найдена ошибка. Если нет — тогда на основе полученных противоречий проводится уточнение абстракции для следующего этапа. Инструменты, реализующие этот подход для поиска переполнения буфера, — SLAM [20], Blast [21], Magic [22], MONA [23].

В случае, когда приемлемы и ложноположительные и ложноотрицательные результаты, но необходимо проанализировать весь код большого проекта (до десятков миллионов строк) с минимальным участием пользователя, требуется искать эвристики, ограничивающие сложность методов статического анализа.

Одним из примеров инструмента, использующего методы верификации для полностью автоматического анализа крупных программ произвольного назначения, можно считать Infer Static Analyzer. Этот инструмент разрабатывается компанией Facebook, имеет открытый исходный код и интенсивно развивается в настоящее время. По утверждениям разработчиков, он активно используется в индустрии, например, в таких крупных IT-компаниях, как Amazon, Spotify, Uber, Mozilla Corporation [24; 25]. Главной особенностью этого инструмента является метод моделирования памяти, основанный на сепарационной логике (separation logic) и биабдукции (bi-abduction) [26]. *Сепарационная логика* с помощью особой операции $*$ — сепарационной конъюнкции — позволяет записывать суждения о свойствах выделенной на куче памяти, при этом известно, что куча может быть разделена на две непересекающиеся части, для которых отдельно верны левый и правый конъюнкт формулы. С помощью этих суждений, записанных в тройках Хоара, составляется спецификация участка программы. *Биабдукция* является расширением логического вывода, которое даёт возможность по формуле-спецификации вывести предусловие, необходимое для выполнимости этой формулы — как правило, такое предусловие описывает состояние кучи, необходимое для корректности описанного спецификацией участка программы. Тем самым можно записывать спецификации функций в виде компактных переиспользуемых резюме, организовать масштабируемый межпроцедурный анализ, проверяющий, выполнимы ли эти спецификации в контексте вызова, и доказать корректность операций с памятью программы либо выдать ошибку.

Сепарационная логика, и в особенности способ записывать в её терминах с помощью биабдукции локальные суждения о свойствах памяти, является прорывным результатом в деле масштабируемой верификации, который по достоинству оценен исследовательским сообществом[2].

[2] Один из авторов Infer Питер О'Хёрн был награжден премией Гёделя в 2016 г. за разработку многопоточной сепарационной логики, членством в Royal Academy of Engineering и наградой 2016 Computer Aided Verification вместе с тремя другими членами команды Infer. За статью [26] в январе 2019 года команда разработчиков Infer получила награду POPL 2019 Most Influential Paper Award [27].

Для поиска переполнения буфера в Infer реализован детектор InferBO [28], который использует символьные интервалы для отслеживания границ буфера и инфраструктуру анализатора для записи резюме функций. Детектор не использует сепарационную логику для моделирования операций с массивами, т.к. исследования таких её расширений появились сравнительно недавно [29] и, насколько известно, пока не реализованы. Сравнение предложенного в работе подхода с детектором InferBO с точки зрения качества анализа приведено в разделе 7.2. Здесь отметим, что автоматическая сборка и анализ программ на C/C++ в Infer по состоянию на ноябрь 2018 г. поддерживается только для программ, которые собираются компиляторами GCC и Clang, при этом анализатор «подменяет» исходные компиляторы собственными, изменяя переменную среды `PATH`. Поэтому сборка сложных программных систем, например, ОС Android, под контролем Infer оказывается невозможной.

Лексические и синтаксические анализаторы

К простым анализаторам, предназначенным для поиска ошибок «на лету» непосредственно в процессе написания кода, относится инструмент ITS4 [30]. Он разбирает код на языках C или C++ в последовательность лексем, в которой производится поиск ошибок путём сопоставления с шаблонами из базы ошибочных последовательностей. Такой подход позволяет анализировать код, находящийся в процессе разработки (даже некорректный с точки зрения компилятора), и, кроме того, не фиксировать для проведения анализа определённую конфигурацию сборки проекта. Разумеется, такой легковесный подход не может обеспечить существенную полноту анализа. Другими примерами инструментов, использующих лексический анализ, являются анализаторы Flawfinder и RATS.

Одним из первых статических анализаторов, предназначенных для поиска переполнения буфера, стал созданный в 2000 г. инструмент BOON [31]. В процессе обхода дерева разбора входной программы на языке C анализатор генерирует систему целочисленных уравнений. Каждой целочисленной переменной ставится в соответствие символьная переменная-интервал, целочисленные выражения моделируются с помощью соответствующих интервальных операций. Каждой строке соответствуют две переменные, описывающие размер выделенной памяти и длину строки. Преобразования строк моделируются изменением этих двух переменных. Для операторов исходной программы генерируются интервальные неравенства. Присваивания с разыменованием указателя в левой части,

указатели на функции, массивы указателей, объединения, указатели на функции игнорируются. После решения полученной системы неравенств для каждой строки анализируется условие безопасности: если не доказано, что длина строки меньше размера буфера, то выдается предупреждение. Анализ является межпроцедурным, нечувствительным к потоку и контексту. Авторы экспериментальных исследований данного инструмента отмечают большое количество пропускаемых ошибок, связанное с тем, что анализатор проверяет только манипуляции со строками [32; 33].

Чувствительный к потоку анализ

Анализ является чувствительным к потоку, если он учитывает порядок инструкций и вычисляет уникальную информацию для каждой точки программы. Как правило, это даёт серьезное преимущество перед потоконечувствительными подходами, позволяя различать больше ситуаций и, как следствие, получать более точные результаты.

Анализатор Splint появился в 2001 году как улучшенная версия анализатора LCLint [34], созданного Эвансом в 1994 г. в рамках магистерской диссертации [35]. Подход данного инструмента заключается в поиске ошибок, опираясь на созданные аналитиками аннотации для функций проекта и стандартной библиотеки. Он выполняет легковесный чувствительный к потоку внутрипроцедурный анализ, а также использует простые эвристики для определения границ циклов. Splint на основании аннотаций и отслеживания создания и обращения к буферам строит систему условий и выдаёт предупреждение каждый раз, когда систему решить не удалось, что приводит к высокому числу ложных срабатываний. Инструмент может работать и в отсутствие пользовательских аннотаций, однако в таких условиях фактически отсутствует межпроцедурный анализ и дополнительно растёт количество ложных срабатываний. Неоспоримыми достоинствами этого инструмента являются его высокая производительность и хорошая масштабируемость.

За годы существования этого инструмента он неоднократно был испытан исследователями в сравнении с другими инструментами. В этих работах отмечается его быстродействие, но в то же время слишком неточный анализ циклов, высокое количество ложных срабатываний [32]. Также некоторые авторы отмечают, что столкнулись с трудностями при попытках проанализировать некоторые

приложения, связанными с ошибками препроцессирования и синтаксического разбора [33; 36].

В 2003 г. был создан инструмент CSSV, целью которого является обнаружение всех ошибок переполнения буфера в программе на языке CoreC (подмножество C) с небольшим количеством ложных срабатываний [37]. Анализ осуществляется отдельно для каждой функции, межпроцедурный анализ организован с помощью аннотаций, предоставленных пользователем. На первом этапе анализа функции все вызовы других функций заменяются их аннотациями. Затем выполняется потоково-нечувствительный анализ псевдонимов. Далее с учетом полученной информации программа сводится к целочисленной программе, в которой, кроме исходных целочисленных переменных, также присутствуют значения, описывающие интересные анализу свойства, например, размер выделенного участка памяти, смещение, длину строки. Также для некоторых исходных конструкций, например, для разыменования, добавляются проверки корректности. Наконец, в ходе анализа полученного представления обнаруживаются ошибочные ситуации и выдаются предупреждения.

Подход, предложенный в работе [38], как и CSSV, гарантирует обнаружение всех ошибок переполнения при работе со строками. Он основан на абстрактной интерпретации с использованием полиэдров в качестве абстракции для моделирования возможных значений размеров массивов и длин строк. В работе [39] предлагается в качестве более точной абстракции использовать *тропические полиэдры* — аналог выпуклых полиэдров в тропической алгебре. Такой подход позволяет вычислять более точные инварианты над длинами строк в результате строковых преобразований. Данные методы позволяют гарантировать обнаружение всех ошибок рассматриваемого типа, однако плохо масштабируются на программы большого размера.

Инструмент Airac [40] — современник CSSV и использует абстрактную интерпретацию с интервальным доменом, дополнительно реализуя межпроцедурный анализ с помощью встраивания функций, аннотируя вычисленные интервалы условиями переходов, а в последующих версиях также удаляя ложные срабатывания, проверяя совместность условий вдоль построенного от места ошибки до начала процедуры обратного слайса с помощью SMT-решателя [41]. Среди других сравнительно недавних инструментов такого типа можно также отметить анализатор Sparrow [42; 43], в котором сделана попытка сохранить одновременно масштабируемость и нахождение всех ошибок за счёт *разреженного* анализа:

состояние абстрактной интерпретации продвигается сразу вдоль зависимостей по данных, консервативно вычисленных отдельным «препроцессирующим» анализом. Тем не менее, современные программные системы на порядок и более превосходят по размеру самую большую программу в 1.3 млн строк кода, про анализ которой упоминается на сайте Sparrow.

Также отметим развивающийся анализатор IKOS [44], разработанный в NASA для нужд авионики, в частности, проверки кода по стандарту DO-178, что требует находить все ошибки. IKOS строит внутреннее представление для анализа с помощью компилятора LLVM и далее выполняет абстрактную интерпретацию, в которой используемый домен может быть выбран конкретным анализом. Также поддерживается межпроцедурный анализ развёрткой вызовов. Авторы позиционируют IKOS как аналог анализатора Astree [16], однако поддерживающий все конструкции языка C. Тем не менее, представляется, что для анализа программ вне авионики IKOS пока нуждается в доработке, так как попытка применить версию от 12 декабря 2018 года (коммит 009dd22) к анализу программы ffmpeg-2.4.14 привела к аварийному завершению анализатора.

Инструмент Carraybounds [45] находит переполнения буфера в случаях, когда на индекс или размер буфера могли повлиять пользовательские данные, однако не поддерживает сложные выражения языка C и библиотечные вызовы, а также ограничен в масштабируемости.

Чувствительный к путям анализ

Для многих инструментов критерием выдачи предупреждения является обнаружение в ходе анализа «ошибочного» состояния, свидетельствующего о наличии ошибки. Например, в работе [46] предлагается следующий критерий: необходимо выдать предупреждение, если будет найдено некоторое ребро графа потока управления такое, что все проходящие через него пути содержат ошибку. В примере на 1.1 этому свойству удовлетворяет пунктирное ребро — на любом проходящем через него пути произойдет доступ к буферу размера 7 по индексу 7. К сожалению, существуют ошибочные ситуации, которые не удовлетворяют этому критерию. В качестве иллюстрации этого тезиса рассмотрим пример, изображенный на 1.2.

Если `cond1` и `cond2` могут одновременно принимать значение «истина», то такая программа содержит ошибку доступа к буферу. В данном графе не существует единственного ребра, прохождение через которое гарантирует

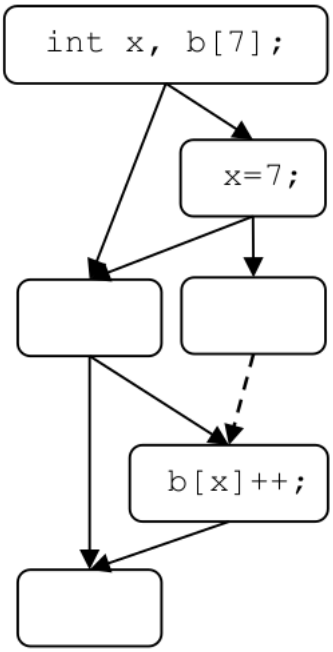

Рисунок 1.1 — Ошибка, определяемая критическим ребром

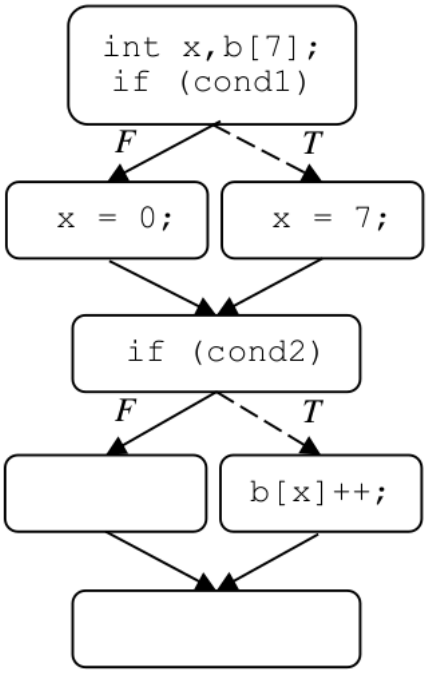

Рисунок 1.2 — Ошибка с критическим путём, но без критического ребра

возникновение ошибки (для каждого ребра существует безошибочный путь через него). Тем не менее, можно предъявить такую последовательность рёбер (на рисунке выделены пунктиром), что любой путь, включающий эту последовательность, будет содержать ошибку. Ещё раз подчеркнём, что для выдачи предупреждения в такой ситуации необходимо прежде проанализировать совместность условий `cond1` и `cond2`. Чтобы обнаруживать такие ошибки, при этом

не превышая допустимое количество ложных срабатываний, необходимо реализовать чувствительный к путям анализ.

Существует ряд инструментов, реализующих анализ, чувствительный к путям. Среди них первым отметим анализатор Marple [47]. Все пути от точки входа в программу до инструкции доступа к буферу классифицируются по 5 категориям: невыполнимые (Infeasible), безопасные (Safe), уязвимые (Vulnerable), содержащие переполнение данными, не контролируемыми извне (Overflow-Input-Independent), неопределенные (Don't-Know). Анализ производится над межпроцедурным графом потока управления программы. На первом этапе определяются невыполнимые пути. Далее для всех инструкций обращения к буферу генерируются запросы на проверку, которые затем последовательно обрабатываются.

Запросы представляют собой совокупность условий на длину строки и размер массива и некоторого набора флагов. Обработка запроса производится с помощью обратного анализа. Для начала инструкция, для которой имеется запрос, обновляет его в соответствии со своей семантикой. Затем, если запрос удаётся решить (проверить условия), то обработка завершается. В противном случае обновлённый запрос продвигается по МГПУ на одно ребро «назад», исключая невыполнимые пути и повторную обработку одного и того же запроса.

В случае обнаружения циклов возможны три исхода. Если установлено, что цикл не влияет на запрос, в таком случае тело цикла «просматривается» анализом один раз. Если количество итераций цикла представимо в виде линейной комбинации некоторых целых переменных, то, зная запрос на выходе из цикла и изменение запроса на одной итерации, анализ может вычислить запрос для инструкции входа в цикл. В противном случае вводится неизвестная переменная, обозначающая количество итераций, с помощью которой формулируется запрос на входе в цикл. В конце для всех разрешённых запросов строятся множества классифицированных путей. Авторы утверждают, что реализованный инструмент масштабируется на сотни тысяч строк кода, при этом допуская в среднем меньше 1 % ложных срабатываний (для MechCommander2 найдено 28 ошибок на 570.9 KLOC).

Авторы S-Looper [48] предлагают методику для анализа циклов обработки строк, суммирующего результаты нескольких путей внутри цикла. Для этого они вводят понятие *счетчиков пути* — символьных значений количества итераций, пройденных по каждому из путей внутри цикла. Эти счётчики далее могут быть использованы для выражения конечных значений переменных, изменяемых

на каждом из путей. Для разрешения условий внутри цикла на содержимое обрабатываемой строки предполагается выводить шаблоны анализируемой строки, удовлетворяющие условиям, и строить на основе этих шаблонов подходящие строковые константы. Результатом анализа является набор условий над целочисленными и строковыми переменными, выполнение которых обеспечивает выполнение цикла, реализующее заданные значения счетчиков путей.

S-Looper без потери точности анализирует циклы, одновременно удовлетворяющие следующим условиям: 1) используемые в цикле переменные имеют целочисленные, символьные или строковые типы; 2) переменные, используемые в условиях внутри цикла являются индуктивными; 3) на результат анализа влияют значения счётчиков путей, но не влияет порядок выбора путей на итерациях цикла; 4) отсутствуют вложенные циклы. Данный алгоритм был реализован в рамках инструмента Marple [47], в результате чего среди всех циклов, которые Marple не смог проанализировать, S-Looper проанализировал 75,7 % циклов.

Parfait [49] — это детектор ошибок, реализованный на базе компиляторной инфраструктуры LLVM. Для каждой SSA-переменной V в произвольной точке программы P вычисляется символьный интервал $S_{V,P} = [S_{V,P_{min}}, S_{V,P_{max}}]$. Границами этого интервала являются атомарные символьные выражения, которые представляют собой аффинные функции над другими атомарными символьными выражениями. Совокупность таких выражений образует частично-упорядоченное множество с дополнительно определёнными верхним и нижним элементами. Детектор выдаёт предупреждение, если в точке доступа к буферу A для переменной индекса I и размера буфера B выполнено $S_{B,A_{max}} \prec S_{I,A_{max}}$.

Вычисление символьного интервала переменной производится лениво, т.е. только в том случае, если его необходимо проверить либо необходимо построить интервал переменной, зависящей от данной. Вычисление символьного интервала происходит в два этапа: сначала обрабатываются зависимости по данным, а затем полученный интервал уточняется с учётом зависимостей по управлению. Зависимость по данным возникает, если переменная получена в результате арифметического выражения, φ-функции, прочитана из константного буфера. Интервал $S_{V,P}$ считается зависимым по управлению от предиката ветвления $pred = (Op_1\ Op_2)$, если 1) существует аффинное отношение между V и $Op_1(Op_2)$ и 2) точка P доминируется инструкцией ветвления, но достижима ровно из одной её ветки. Учёт зависимостей по управлению, по заявлению авторов, обеспечивает чувствительность к путям данного алгоритма. Однако учёт только таких типов

зависимостей не представляется достаточным для обеспечения полноценной чувствительности к путям.

Отдельного упоминания заслуживает инструмент ARCHER [50], реализованный на языке OCaml создателями известного инструмента статического анализа Saturn [51]. Данный инструмент не требует от пользователя никаких дополнительных данных о программе, способен анализировать большие программы (анализ 2158 файлов и 1,6 млн. строк кода ОС Linux версии 2.5.53 занял 4 часа). При этом сохраняется высокий процент истинных срабатываний (65 % из 139 срабатываний на Linux 2.5.53). В основе его подхода к анализу лежит символьное выполнение с ограничением на количество рассмотренных путей и время анализа одной функции (авторы утверждают, что при анализе Linux в среднем в функции было покрыто 96 % путей). Поиск межпроцедурных срабатываний организован с помощью метода резюме. К минусам, присущим инструменту ARCHER, можно отнести отсутствие полной поддержки библиотечных функций работы со строками в языке C.

Перспективными в контексте анализа строк для переполнения буфера представляются решатели с поддержкой строк [52]. В настоящее время они получили широкое применение для анализа значений строк в веб-приложениях, где они позволяют обнаруживать и автоматически анализировать процедуры проверки входных данных («санитайзеры») для устранения потенциальных уязвимостей. Способность данных инструментов анализировать не только размер, но и содержимое строк может быть также полезна и при анализе переполнения буфера, однако число реально встретившихся нам на практике примеров, для которых требуется выносить суждения о содержимом строк, относительно невелико.

Проприетарные инструменты

Ряд современных промышленных анализаторов, таких как Coverity Prevent [53], Klocwork [54], Polyspace [55], IAR Static Analyzer [56] и некоторые другие, включают в себя детекторы переполнения буфера, однако используемые ими алгоритмы закрыты, что затрудняет суждения о качестве этих детекторов. Prevent является наиболее известным анализатором, выполняющим контекстно-чувствительный межпроцедурный анализ, а также подавляющий ложные срабатывания на нереализуемых путях. В статье [57] приводятся результаты сравнения Prevent, Polyspace, CBMC, UNO и ARCHER с точки зрения масштабируемости и качества поиска ошибок переполнения (как количества

ложных срабатываний, так и пропущенных ошибок) на подготовленном искусственном наборе тестов. Prevent показал 20% ложных срабатываний, однако пропустил 52% ошибок. Polyspace не пропустил ни одной ошибки, что неудивительно, учитывая, что этот инструмент ставит целью доказать отсутствие ошибок, но показал 25% ложных срабатываний и не смог проанализировать программы более 15 тыс. строк кода. Также стоит отметить, что о качестве проведённого в статье тестирования сложно судить, не имея самого набора тестов.

1.1.3 Межпроцедурный анализ и контекстная чувствительность

Существенная часть ошибок переполнения буфера возникает из-за нарушения контракта функции. Такие ситуации возникают, если в библиотечную или пользовательскую функцию передаются значения, не соответствующие предусловию функции, реализация некоторой функции не учитывает какой-то из предполагаемых контрактом случаев, либо возвращаемое значение нарушает постусловие или некорректно обрабатывается в вызывающей функции. Чтобы их обнаруживать, статический анализ должен быть межпроцедурным, т.е. при анализе одной функции учитывать информацию о других функциях.

Для легковесных анализаторов, работающих на уровне последовательности лексем, связь между формальными и фактическими параметрами может быть установлена по именам, для случаев, когда они названы одинаково. Более тяжеловесные инструменты могут производить анализ всей программы целиком как единого графа потока управления. В этом случае анализ функций может производиться с разной степенью *контекстной* чувствительности[3]. Полностью нечувствительный анализ проверяет все контексты вызова совместно, при этом результаты анализа одного контекста могут повлиять на анализ других контекстов, что снижает точность анализа. Однако такой подход хорошо масштабируется и потому достаточно популярен. Если же в каждой точке вызова код вызываемой функции анализируется независимо, т.е. де факто для каждого вызова используется новый экземпляр кода функции, анализ является контекстно-чувствительным. Такой подход обеспечивает высокую точность, потому что анализ функции в одном контексте вызова не влияет на результат анализа в другом контексте. Но

[3] Иногда используются термины «мономорфный анализ» и «полиморфный анализ», например, когда природа выполняемого анализа связана с типами переменных (анализ алиасов).

для масштабируемости подхода требуется ограничивать степень контекстной чувствительности, выбирая группу контекстов, для которых функция будет проанализирована единожды. Разумным компромиссом может быть сочетание этих двух подходов. Например, анализатор [58] контекстно чувствительно обрабатывает лишь вызовы библиотечных функций работы со строками, а остальные функции анализирует один раз для всех контекстов.

В качестве альтернативного подхода можно при анализе инструкции вызова функции учитывать некоторое полученное извне приближение контракта функции. Многие анализаторы полностью полагаются только на аннотации пользователя при организации межпроцедурного поиска, предоставляя ему возможность самостоятельно написать аннотации для всех функций, тогда при анализе все вызовы функций заменяются на применение аннотаций. 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, который посещает каждую функцию один раз.

1.2 Требования к детектору переполнения буфера

1.2.1 Тестовые наборы для оценки инструментов поиска переполнения буфера

За последние несколько десятилетий был опубликован ряд работ, посвящённых тестированию и оценке результатов различных инструментов поиска переполнения буфера. Кроме того, существует ряд тестовых наборов, включающих как корректные, так и ошибочные синтетические примеры кода и позволяющих оценить спектр обнаруживаемых ошибок и долю ложных срабатываний исследуемых анализаторов.

Juliet Test Suite Одним из самых известных и крупных тестовых наборов является Juliet Test Suite, разработанный в NSA’s Center for Assured Software

(CAS) [59]. Для языков C/C++ в этом наборе представлено 64 099 тестов, размеченных по классификации CWE [60]. К ошибке переполнения буфера среди всех групп набора относятся группы:

- CWE 121 — «Stack-based Buffer Overflow» (переполнение буфера на стеке при записи, 4 968 тестов),
- CWE 122 — «Heap-based Buffer Overflow» (переполнение буфера на куче при записи, 5 922 теста),
- CWE 124 — «Buffer Underwrite» (запись за левой границей буфера, 2 048 тестов),
- CWE 126 — «Buffer Over-read» (чтение за правой границей буфера, 1 452 теста),
- CWE 127 — «Buffer Under-read» (чтение за левой границей буфера, 2048 тестов).

Для каждого теста из этого набора также указан номер т.н. «варианта потока» (flow variant), который соответствует определённому виду потока управления и потока данных для данного теста. Среди вариантов потока управления рассматриваются различные случаи условий перехода, в том числе: 1) проверка глобальной переменной (`STATIC_CONST_FIVE==5`), 2) условие вычисляется в глобальной функции (`globalReturnsTrueOrFalse()`), 3) различные операторы языка (`switch`, `while`, ...). Варианты потока данных описывают различные случаи межпроцедурной и внутрипроцедурной передачи данных, такие как: 1) через аргументы функции (в т.ч. по указателю, ссылке, как элемент массива или коллекции и пр.), 2) через возвращаемое значение функции, 3) через глобальную переменную. Некоторые из вариантов специфичны для языка C++ и не применимы к тестам на C.

В группах, относящихся к переполнению, тесты распределены примерно равномерно между всеми вариантами. Отмечается большое количество тестов, включающее строки с широкими символами. Также значительное количество тестов проверяют использование библиотечных функций, таких как `memcpy`-подобные функции, функции обработки строк, использующие форматную строку и т.п.

Toyota ITC Benchmark Также часто используется тестовый набор, разработанный Toyota InfoTechnology Center для тестирования инструментов статического анализа [61]. Он содержит 1 276 простых тестовых примеров (638 ошибочных и

638 корректных), разделённых на 9 типов и 51 подтип. Каждый тест представлен парой из ошибочного и корректного случаев. В рамках данной работы интерес представляют следующие типы и подтипы:

- static memory (статическая память):
  - static buffer overrun (выход за правую границу статического буфера, 54 теста),
  - static buffer underrun (выход за левую границу статического буфера, 13 тестов),
- dynamic memory (динамическая память):
  - dynamic buffer overflow (выход за правую границу динамического буфера, 32 теста),
  - dynamic buffer underrun (выход за левую границу динамического буфера, 39 тестов).

Тесты набора покрывают следующие особенности кода в различных комбинациях:

- массивы на стеке, в статической памяти и на куче;
- типы элементов массива (`char`, `int`, `float`, структуры и т.п.);
- способы вычисления индекса (константа, линейные и нелинейные выражения, переданные в качестве аргумента и возвращаемые из функции, итеративные переменные цикла и элементы другого массива);
- варианты получения адреса массива (локальная/глобальная переменная, аргумент функции, арифметика указателей, в том числе в цикле и с учётом псевдонимов);
- размер массива (динамические массивы с константным размером, приведение типа указателя);
- способ обращения к массиву (по индексу, разыменование указателя, через библиотечную функцию, в том числе функцию обработки строк).

1.2.2 Исследование уязвимостей, связанных с переполнением буфера

Изучение инструмента поиска ошибок с помощью тестовых наборов позволяет получить хорошее представление о возможностях и качестве инструмента, однако ни один из представленных тестовых наборов не даёт репрезентативной картины совокупности дефектов в реальном коде программ. Одна из важнейших

задач для статического анализа — обнаружение на ранней стадии разработки потенциальных ошибок, которые впоследствии могут привести к появлению уязвимостей в итоговом продукте. 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. В целом применялась методика из работы [36] для получения выборки ошибок для исследования.

Стоит отметить, что обнаружение эксплуатируемых уязвимостей не является единственной задачей для статического анализатора. Некоторые ошибки не могут быть проэксплуатированы или их эксплуатация сопряжена со значительными затруднениями, и тем не менее, их наличие в исходном коде программы нежелательно. Кроме того, присутствует робкая надежда, что в настоящее время разработчики в большей степени используют различные анализаторы кода (статические и/или динамические) в процессе создания программного продукта. Следствием этого является тот факт, что даже возникающие в ходе разработки простые ошибки быстро исправляются и не попадают в базу данных уязвимостей. С другой стороны, исследование ошибок из базы уязвимостей представляется удачным способом обнаружить слабые стороны современных статических анализаторов и, как следствие, выделить важные направления развития.

Для исследования было выбрано 132 случайных записи из категории «overflow» из базы уязвимостей CVE [62]. Для 33 из них удалось найти исходный код соответствующей версии проекта для изучения, для ещё 14 некоторые данные были извлечены из описания уязвимости. Каждая ошибка была исследована на предмет причин появления уязвимостей и классифицирована по ряду признаков. Набор признаков был основан на таксономии, приведённой в статье [63], с некоторыми изменениями.

В первую очередь следует отметить, что ряд закономерностей в полученной выборке может быть напрямую объяснен особенностями именно эксплуатируемых ошибок:

1. более чем в половине случаев (77 %) ошибка происходит при записи в буфер, и только в 23 % случаев при чтении;
2. все ошибки связаны с выходом только за правую границу буфера;
3. почти все уязвимости происходят из вычисления индекса из данных, полученных из ненадёжных источников (непроверенные данные из сети, чтение из файла, аргументы командной строки и т.п.).

Кроме этого, прослеживается тенденция, согласно которой простые типичные ошибки (например, использование небезопасных функций, таких как `strcpy`) встречаются в более старом коде (до 2010 г.) и почти не встречаются в более поздних версиях проектов. Возможно, эта тенденция по крайней мере отчасти объясняется растущей популярностью инструментов для проверки кода.

В нашей выборке около половины случаев (51 %) составляет переполнение буфера на стеке, почти столько же (47 %) — переполнение буфера на куче, и лишь несколько случаев переполнения статического буфера.

Среди всех способов обращения к буферу 46 % составляет обращение по индексу (`buf[i]`), 12 % через разыменование указателя, 42 % путём вызова библиотечной функции, в том числе 23 % составляют функции обработки строк. Для обнаружения ошибок последней группы с помощью статического анализа необходимо моделирование строк языка C. Когда доступ к буферу осуществлялся внутри библиотечной функции, для дальнейшего исследования в качестве индекса там, где это разумно, использовался аргумент, соответствующий размеру копируемой памяти либо ограничению на длину копируемой строки.

57 % всех ошибок связаны с переполнением буфера константного размера (все буферы на стеке и в статической памяти, некоторые буферы на куче). Ещё в 18 % случаев размер буфера является линейной комбинацией переменных программы. Таким образом, почти в половине случаев требуется анализ взаимосвязей между значениями целочисленных переменных для определения размера буфера.

Также было исследовано соотношение внутрипроцедурных и межпроцедурных ошибок. Выделение буфера и доступ к нему располагаются в одной функции (либо буфер глобальный) только в 22 % случаев. Вычисления индекса полностью располагаются в той же функции, что и доступ по этому индексу, в 37 % случаев. Оба свойства верны только для 12 % ошибок. Это свидетельствует о том, что межпроцедурный анализ критически важен для обнаружения ошибок рассматриваемого типа.

Последнее исследованное свойство — наличие критической точки в рассматриваемых ошибках. Если такая точка отсутствует, то для обнаружения ошибки требуется проведение чувствительного к путям анализа. В рассматриваемой выборке только в 30 % случаев можно предъявить критическую точку. Таким образом, чувствительный к путям анализ является *sine qua non* для обнаружения таких ошибок.

Из вышесказанного следует, что для понимания направлений развития промышленного статического анализатора важно рассматривать оба приведённых источника примеров ошибочных программ. Наборы тестов очерчивают круг ситуаций, которые необходимо поддержать в анализаторе, при этом их легко понять, классифицировать и проверить. С другой стороны, они не отражают распределение таких ситуаций в реальном коде. Выборка уязвимостей из промышленных проектов также представляет интерес для исследования, но оказывается смещённой в сторону эксплуатируемых ошибок и к тому же не включает ошибки, исправленные на стадии разработки (возможно, как раз с использованием статического анализатора). 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.

Глава 2. Определение ошибки доступа к буферу

Для решения задачи поиска ошибок в исходном коде программ следует в первую очередь понять, что именно необходимо искать, то есть некоторым образом определить, что считается ошибкой. Существует несколько принципиальных подходов к ответу на этот вопрос с точки зрения статического анализа.

Можно считать ошибочным только такой код, исполнение которого для некоторых входных данных программы приводит к неопределённому поведению (undefined behaviour, в нашем случае — к некорректному доступу к массиву). Однако такой подход имеет ряд недостатков. Во-первых, данное определение не применимо для анализа несамостоятельных частей программ (например, библиотек). Корректность отдельной функции программы по данному определению устанавливается только для ограниченного набора представленных в программе контекстов её вызова, использование такой функции в новом контексте может привести к ошибочной ситуации. Во-вторых, некоторые аномалии в коде могут свидетельствовать о наличии логической ошибки, но не приводят к аварийному завершению программы. Например, обращение к памяти, расположенной за пределами индексируемого буфера, но являющейся частью того же объекта, что и буфер, разрешено стандартом и может быть как намеренным, так и ошибочным поведением (см. раздел 7.1.2). В качестве другого примера можно привести наличие в программе избыточного, всегда принимающего истинное или ложное значение условия, которое привносит в программу мёртвый код.

Другим подходом является поиск неконсистентностей, несогласующихся частей в коде программы [64]. Подход, используемый в данной работе, основывается на методе определения ошибки, описанном в статье [65]. Далее он будет рассмотрен подробно.

2.1 Внутрипроцедурные ошибки

При реализации конкретной функции разработчик, как правило, стремится к тому, чтобы её поведение было корректным во всех потенциальных контекстах вызова (а не только в реально существующих в проекте на данный момент).

Это требование в первую очередь важно для библиотечных функций, функций, которые ещё будут использоваться при разработке нового кода. Поэтому при проведении анализа каждая функция считается точкой входа в программу. Далее будем считать, что при анализе рассматривается некоторая функция вместе со всеми вызываемыми ею, в том числе транзитивно.

Сценарий использования разрабатываемого анализатора предполагает анализ проектов при отсутствии части исходного кода (например, реализаций библиотечных функций). Кроме этого, анализатор должен быть полностью автоматическим, а значит, не предполагается получение от разработчиков дополнительной информации о функциях, такой как предусловия, постусловия и пр. В данном случае при анализе функции подразумеваемое программистом предусловие неизвестно (с учетом того, что в рассмотрение включены в том числе отсутствующие в программе потенциальные контексты вызова). Вытекающая из этого сложность анализа заключается в том, что на значения, получаемые из неизвестных функций или передаваемые в функцию в качестве аргументов, могут быть наложены неизвестные анализу ограничения, например, программисты могут подразумевать существование некоторой взаимосвязи значений аргументов функции. Полное игнорирование возможности наличия таких взаимосвязей приведет к выдаче чрезмерного количества ложных срабатываний, так как анализатор будет сообщать об ошибках, возникающих в случае их нарушения, что нежелательно.

Существуют подходы, позволяющие вычислить контракт функции, если для анализа ограничить множество допустимых контекстов вызова функции только реально представленными в коде во всех точках вызова [66]. В некоторых случаях (например, для статических функций языка C) предположение об отсутствии точек вызова вне анализируемого кода вполне корректно в рамках одной анализируемой версии программы. Однако, это не снимает вопроса о корректности функции в других потенциальных контекстах вызова в общем случае. Выбор множества потенциальных рассматриваемых потенциальных контекстов напрямую влияет на выбор определения ошибки и стратегии анализа, поэтому рассмотрим его подробнее.

Для начала введём необходимый формализм. Для некоторой функции 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. Для произвольного пути p на ГПУ множество соответствующих ему конкретных выполнений будем далее обозначать как $\mathfrak{C}(p)$. Совокупность ограничений на множество значений неизвестных переменных будем называть *контрактом*. Таким образом, при анализе необходимо иметь в виду всё множество возможных контрактов и выдавать предупреждения только в тех случаях, когда ошибка происходит при каждом контракте из этого множества.

Какие именно контракты считать возможными, решается при построении конкретной стратегии анализа. Выше уже было отмечено, что выбор пустого множества в качестве множества возможных контрактов приводит к чрезмерному количеству ложных срабатываний. С другой стороны, если считать, что возможны абсолютно любые контракты, то придется пропустить слишком много реальных ошибок. Как правило, для ошибочной функции можно подобрать искусственное предусловие, исключающее возникновение ошибки, при этом оно будет куда более строгим, чем реальное, задуманное программистом. То есть существование такого искусственного, полностью «безопасного» предусловия (как правило, лишающего функцию смысла) не является препятствием для выдачи предупреждения об ошибке.

В качестве компромисса предлагается ввести априорное предположение о свойствах контрактов функций, которое определит множество возможных 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 неизвестных переменных. Проиллюстрируем это различие на примере листинга 2.1. В функции `foo` может произойти переполнение буфера `buf`, если будет выполнено `idx` $\geqslant$ `S`. Исходя из вышесказанного, данную ситуацию не будем считать ошибочной, так как считаем, что такие значения `idx` запрещены контрактом функции. Заметим, что наличие ошибки напрямую зависит от множества возможных значений `idx`. Теперь рассмотрим функцию `bar`, здесь переполнение

Листинг 2.1 — Примеры функций с неизвестными контрактами

```
1  #define S 10
2
3  int buf[S];
4
5  void foo (int idx) {
6    buf[idx]++;
7  }
8
9  int bar (int a, int b) {
10   if (a >= S-1) {
11     // ...
12   }
13
14   if (b)
15     a++;
16
17   return buf[a];
18 }
```

произойдет, если будет выполнено:

$$(\mathtt{a} \geqslant \mathtt{S} - 1) \wedge \mathtt{b} \neq 0 \tag{2.1}$$

Мы будем считать такую ситуацию дефектом, так как ошибка следует из свойств ГПУ, а именно, из наличия возможно выполнимого пути, на котором гарантированно произойдёт переполнение. При этом также контракт функции может запрещать условие (2.1) — тогда ошибки нет, то есть наличие дефекта зависит не напрямую от множества значений неизвестных переменных, а косвенно, так как контракт влияет на выполнимость некоторых путей на ГПУ. Чтобы строго различить две рассмотренные в функциях `foo` и `bar` ситуации, будем считать, что контракты функций не могут влиять на выполнимость путей, то есть контракта, запрещающего условие (2.1), не существует, тогда очевидно, что функция `bar` содержит ошибку. Заметим, что рассмотренный для функции `foo` контракт удовлетворяет этому предположению, значит, предупреждение об ошибке не будет выдано.

Сформулируем описанное предположение о контрактах строго. 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 (т.е. $\mathfrak{C}(p) \neq \varnothing$). Пусть P^f — подмножество путей графа потока управления программы, 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, 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 $(0 > \texttt{i}) \vee (\texttt{i} \geqslant \texttt{S})$,
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 `S` по индексу `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.*

Доказательство. Докажем необходимость и достаточность.

$\Rightarrow$ Покажем, что если функция удовлетворяет определению 1 и $P^c = P^f$, то для неё существует путь в графе G_k^*, подходящий под условия теоремы.

По определению 1 для такой программы существует путь в графе G_k^*, удовлетворяющий условиям 1 и 2. Так как этот путь принадлежит графу G_k^*, то он

проходит не более k^n раз по каждому обратному ребру цикла вложенности n. Из условия 2 следует, что он выполним, а из $P^c = P^f$ следует, что он выполним и в рамках любого контракта. Таким образом, этот путь и является искомым.

$\Leftarrow$ Покажем, что если для любого контракта функции выполнено $P^c = P^f$ и найдется путь, подходящий под условия теоремы, то данная функция удовлетворяет определению 1.

Т.к. выбранный путь проходит не более k^n раз по каждому обратному ребру цикла вложенности n, то для него существует путь p на графе G_k^*. Т.к. он является выполнимым, то для него найдется конкретное выполнение $c \in \mathfrak{C}(p)$, т.е. p удовлетворяет пункту 2 определения 1. Предположим, что он не удовлетворяет пункту 1, т.е. для p существует другое конкретное выполнение $\hat{c} : \hat{c} \in \mathfrak{C}(p), \hat{c} \neq c$, на котором в точке доступа выполнено условие $0 \leqslant \mathtt{i} < \mathtt{S}$. Значит, выполнение этого условия в точке доступа зависит от значения неизвестных переменных. Следовательно, существует контракт, который запрещает те и только те значения неизвестных переменных, при которых $(0 > \mathtt{i}) \vee (\mathtt{i} \geqslant \mathtt{S})$ и конкретное выполнение принадлежит $\mathfrak{C}(p)$. Такой контракт удовлетворяет условию $P^f = P^c$, т.к. он может повлиять только на выполнимость пути $p \in P^f$, а p остаётся выполнимым ($p \in P^c$), раз существует $\hat{c}$. Существование такого контракта противоречит свойству *(ii)* выбранного пути, поэтому такого конкретного выполнения $\hat{c}$ не может существовать, а значит, верен и пункт 1 определения 1. □

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

Рассмотрение графа G_k^* вместо графа G^* приводит к тому, что игнорируются ошибки, происходящие на итерации цикла с номером, большим k. В разработанном алгоритме анализа используется эвристика, позволяющая частично обойти эти ограничения. Она заключается в изменении семантики арифметических операций, позволяющем моделировать некоторую обобщенную итерацию цикла.

2.2 Межпроцедурные ошибки

В отдельный класс ошибок, обнаружение которых требует привлечения специализированных подходов, можно выделить межпроцедурные ошибки. Под этим термином будем понимать ситуацию, 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.

Листинг 2.2 — Примеры межпроцедурной ошибки

```
1  #define SIZE 10
2
3  extern int check_idx (int, int);
4
5  int find_idx (int val) {
6    int x;
7    for (x = 0; x < SIZE; x++)
8      if (check_idx (x, val))
9        break;
10   return x;
11 }
12
13 void store (int *b, int i, int val) {
14   b[i] = val;
15 }
16
17 void foo (int val) {
18   int buffer[SIZE];
19   int idx = find_idx (val);
20   store (buffer, idx, val);
21 }
```

В качестве иллюстрации межпроцедурной ошибки рассмотрим пример дефекта, аналогичный обнаруженному в реальном проекте (см. листинг 2.2). В данном случае функция `find_idx` вычисляет некоторое значение, которое в том числе может быть равно `SIZE`. Функция `foo` передаёт результат вызова `find_idx` и адрес своего локального буфера размера `SIZE` в функцию `store`, в

которой к переданному буферу происходит обращение по переданному индексу. Сама по себе функция `store` не содержит ошибки, но её контракт подразумевает, что переданный индекс меньше размера переданного буфера. Как мы видим, в точке вызова в функции `foo` этот контракт может нарушаться, следовательно, в этом месте необходимо выдать предупреждение.

Данный пример иллюстрирует, что выделение буфера, вычисление индекса и инструкция доступа к буферу могут находиться в разных функциях, при этом в случае ациклического графа вызовов можно говорить о наличии ошибки в функции, являющейся их ближайшим общим предком. Для обнаружения таких дефектов необходимо вычислять контракты функций, гарантирующие отсутствие ошибок доступа к буферу, анализировать результат и побочные эффекты функции.

Рассмотрим некоторый путь P в графе G_k^*, удовлетворяющий определению 1, то есть содержащий ошибку доступа к буферу. Он представляет собой некоторую конечную последовательность рёбер $\{e_i\}$, включающую ребро, ведущее в инструкцию p_{access} доступа к буферу, в которой происходит ошибка. Выберем из пути произвольную подпоследовательность e_{i_s} и на графе вызовов программы пометим все функции, содержащие рёбра из последовательности e_{i_s}. Далее рекурсивно отметим все функции, вызывающие отмеченные и содержащие рёбра из $\{e_i\}$, вплоть до единого общего предка — функции f. В результате получится некоторый помеченный подграф графа вызовов. Будем считать, что значения, изменяемые и возвращаемые непомеченными вызываемыми функциями, могут быть любыми, но обязаны удовлетворять предположению о контрактах; контекст вызова функции f — корня помеченного подграфа — может быть любым, удовлетворяющим предположению о контрактах.

Определение 2. Если при данных условиях любой путь в функции f, проходящий через рёбра e_{i_s}, либо невыполним, либо ошибочен в точке p_{access} по определению 1, то такой набор рёбер будем называть *критическим*.

Определение 3. Если из некоторого критического набора нельзя исключить ни одного ребра с сохранением данного свойства, то такой набор будем называть *минимальным критическим*.

Заметим, что для некоторых ошибочных путей можно построить более одного минимального критического набора.

Определение 4. Если для некоторого пути, удовлетворяющего определению ошибки, не существует минимального критического набора, целиком состоящего из рёбер единственной функции, то такую ситуацию мы будем называть *межпроцедурной ошибкой*.

Заметим, что функция `foo` (см. листинг 2.2) содержит пример межпроцедурной ошибки, так как в любой критический набор ошибочного пути обязательно входит ребро, ведущее к инструкции доступа к буферу в функции `store`, но одного этого ребра недостаточно для определения ошибочного пути, так как для этой функции можно подобрать безопасный контекст вызова. Таким образом, в критический набор обязательно входят точки из других функций, то есть ошибка межпроцедурная. В данном случае минимальный критический набор состоит из ребра в функции `store` и ребра, соответствующего `false`-ветке условного оператора функции `find_idx` на k-ой итерации цикла.

Глава 3. Поиск ошибок в рамках одной функции

В рамках данной главы остановимся на поиске переполнения массивов, имеющих известный в момент компиляции размер. Анализ переполнения для буферов произвольного размера рассматривается в главе 6.

3.1 Модельный язык

Изложение алгоритма поиска ошибок переполнения буфера будет проводиться для программ на модельном языке. Для его построения было выбрано репрезентативное с точки зрения поставленной задачи подмножество языка C, для краткости не включающее некоторые возможности этого языка. Так, в модельном языке присутствуют локальные целочисленные переменные, указатели, локальные массивы, функции. Анализ значений глобальных переменных является ортогональной задачей, а рассматриваемые алгоритмы могут использовать результат такого анализа, но это выходит за рамки настоящего исследования, поэтому глобальные переменные и массивы в модельном языке не представлены. Аналогично, легко расширить рассматриваемый алгоритм анализа для поддержки структурных типов, но специфичные для задачи переполнения буфера построения принципиально не изменятся, поэтому для теоретического изложения алгоритма наличие структур не является существенным. Особенности реализации алгоритма для реальных программ на языке C рассматривается в разделе 7.1.

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.

Пусть также $C \doteq \bigcup_{b \in B} C_b$, $V \doteq \bigcup_{b \in B} V_b$, $M \doteq \bigcup_{b \in B} M_b$, $A \doteq \bigcup_{b \in B} A_b$, $P \doteq \bigcup_{b \in B} P_b$. Выделим значение $r \in B$, обозначающее разрядность адресов переменных и индексов массивов. Подмножество ячеек памяти, хранящих значения указателей, обозначим $M^p \subseteq M_r$.

$$b, \texttt{e} \in B,\ \texttt{e} \neq b, \quad \texttt{x}, \texttt{y} \in V_b, \quad \texttt{z} \in V_e,$$
$$\texttt{p}, \texttt{s} \in P_b, \quad \texttt{v} \in V \cup P,$$
$$\texttt{h}, \texttt{g} \in V_b \cup C_b, \quad \texttt{i} \in V_r \cup C_r,$$
$$\texttt{f} \in F, \quad \texttt{c} \in C_r.$$

⟨*Program*⟩ ::= ⟨*Function*⟩ | ⟨*Program*⟩ ⟨*Function*⟩

⟨*Function*⟩ ::= `f(` ⟨*FParams*⟩ `) {` ⟨*Statements*⟩ `}`

⟨*FParams*⟩ ::= ε | ⟨*FParam*⟩ { `,` ⟨*FParam*⟩ }

⟨*FParam*⟩ ::= `var x` | `ptr p` | `arr m`

⟨*Statements*⟩ ::= ⟨*Statement*⟩ | ⟨*Statement*⟩ ⟨*Statements*⟩

⟨*Statement*⟩ ::= `x = h` | `p = s`
| `p = alloca(b)` | `p = alloca(b, c)`
| `store(p, h)` | `store(p, s)`
| `h = load(p)` | `p = load(s)`
| `x = p [i]` | `p [i] = h`
| `x = h` ⟨*Arithm*⟩ `g` | `z = trunc(x, e)`
| `z = zext(x, e)` | `z = sext(x, e)`
| `if` ⟨*Cond*⟩ `{` ⟨*Statements*⟩ `}` `else {` ⟨*Statements*⟩ `}`
| `while` ⟨*Cond*⟩ `do {` ⟨*Statements*⟩ `}`
| `v = call f(` ⟨*AParams*⟩ `)` | `return`
| ε

⟨*Arithm*⟩ ::= `+` | `-` | `*` | $/_s$ | $/_u$

⟨*Cond*⟩ ::= `(h` ⟨*Relation*⟩ `g)`

⟨*Relation*⟩ ::= $<_s$ | $<_u$ | $>_s$ | $>_u$
| $\geqslant_s$ | $\geqslant_u$ | $\leqslant_s$ | $\leqslant_u$
| `=` | $\neq$

⟨*AParams*⟩ ::= ε | ⟨*AParam*⟩ { `,` ⟨*AParam*⟩ }

⟨*AParam*⟩ ::= `h` | `p` | `m`

Рисунок 3.1 — Расширенная БНФ для модельного языка

Программа представляет собой набор функций, каждая из которых имеет уникальное имя из множества F и набор параметров из множества $V \cup P \cup A$[1]. Возвращаемое значение функции $f \in F$ после её выполнения сохраняется в специальной переменной $f_{ret} \in V$ (подобно возвращаемому значению функции языка Паскаль). Синтаксис модельного языка приведён на рис. 3.1.

Тело состоит из последовательности операторов, изменяющих состояние программы.

Конкретное состояние программы описывается тремя отображениями:

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

Инструкции `x = h` и `p = s` реализуют присваивание значений в переменную и указатель соответственно. В результате

$$\mathbb{X}(\mathtt{x}) \leftarrow \mathbb{X}(\mathtt{h}), \quad \mathbb{P}(\mathtt{p}) \leftarrow \mathbb{P}(\mathtt{s})^{2}.$$

Инструкция `p = alloca(b)` выделяет новую ячейку памяти m размера `b` (не пересекающуюся ни с какими уже выделенными участками памяти) и сохраняет её адрес в указателе `p`:

$$\mathbb{P}(\mathtt{p}) \leftarrow m.$$

Инструкция `p = alloca(b, c)` выделяет новый локальный массив[3] m из `c` элементов размера `b` (не пересекающийся ни с какими уже выделенными участками памяти) и сохраняет его адрес в указателе `p`:

$$\mathbb{P}(\mathtt{p}) \leftarrow m, \quad \mathbb{B}(\mathtt{m}) \leftarrow \mathtt{c}.$$

В этой главе рассматриваются только массивы константного размера, в главе 6 модельный язык будет расширен для описания программ с массивами произвольного размера.

[1] В отличие от языка C, в модельном языке массивы не могут выступать в роли указателей (array decaying), поэтому для них введён отдельный тип формального параметра.

[2] Здесь и далее с помощью нотации $\mathbb{M}(x) \leftarrow y$ будем обозначать обновление отображения $\mathbb{M}$, в результате которого значение $\mathbb{M}$ не меняется для величин, отличных от x, и $\mathbb{M}(x)$ принимается равным y.

[3] Так как вопросы моделирования содержимого массивов в данной работе не рассматриваются (исключение сделано для строк, см. главу 5.1), то случай статических массивов и массивов на куче принципиально не отличается и поэтому опущен для краткости.

Инструкции записи и чтения значения по указателю могут быть использованы только для указателей, хранящих адреса ячеек памяти из M (но не массивов). Инструкция `store(p, h)` сохраняет значение `h` в ячейку памяти, на которую указывает `p`:

$$\mathbb{X}(\mathbb{P}(\mathtt{p})) \leftarrow \mathbb{X}(\mathtt{h}).$$

Инструкция `store(p, s)` сохраняет адрес, хранящийся в переменной `s`, в ячейку памяти, на которую указывает `p`:

$$\mathbb{P}(\mathbb{P}(\mathtt{p})) \leftarrow \mathbb{P}(\mathtt{s}).$$

При этом ячейка $\mathbb{P}(\mathtt{p})$, на которую указывает `p`, должна принадлежать множеству M^p. При выполнении этой инструкции тип указателя `s` становится *эффективным типом* ячейки $\mathbb{P}(\mathtt{p})$.

Инструкция `x = load(p)` загружает значение, находящееся по адресу `p`, в переменную `x`:

$$\mathbb{X}(\mathtt{x}) \leftarrow \mathbb{X}(\mathbb{P}(\mathtt{p})).$$

Инструкция `p = load(s)` загружает адрес, сохранённый в ячейке по адресу `s`, в переменную `p`:

$$\mathbb{P}(\mathtt{p}) \leftarrow \mathbb{P}(\mathbb{P}(\mathtt{s})).$$

При этом, как и для случая сохранения, ячейка $\mathbb{P}(\mathtt{s})$, на которую указывает `s`, должна принадлежать множеству M^p, и эффективный тип этой ячейки должен совпадать с типом указателя `p`.

Инструкция `x = p[i]` загружает в переменную `x` значение `i`-ого элемента массива, на который указывает `p`, если $\mathbb{B}(\mathbb{P}(\mathtt{p})) >_s \mathbb{X}(\mathtt{i}) \geqslant_s 0$, и завершается с ошибкой в противном случае:

$$\mathbb{X}(\mathtt{x}) \leftarrow \mathbb{X}(\langle \mathbb{P}(\mathtt{p}), \mathbb{X}(\mathtt{i}) \rangle).$$

При этом указатель `p` должен указывать на массив $\mathbb{P}(\mathtt{p})$ из множества A_b.

Инструкция `p[i] = h` сохраняет значение `h` в `i`-й элемент массива `m` (с теми же ограничениями на значение индекса и указателя `p`):

$$\mathbb{X}(\langle \mathbb{P}(\mathtt{p}), \mathbb{X}(\mathtt{i}) \rangle) \leftarrow \mathbb{X}(\mathtt{h}).$$

В инструкциях доступа значение битового вектора `i` рассматривается как знаковое число в дополнительном коде.

Инструкция `x = h` $\circ$ `g`, где $\circ \in \{+, -, *, /_s, /_u\}$, вычисляет арифметическое выражение и сохраняет его значение в переменную `x`:

$$\mathbb{X}(\texttt{x}) \leftarrow \mathbb{X}(\texttt{h}) \circ \mathbb{X}(\texttt{g}).$$

Инструкции деления различаются для случая знакового и беззнакового представления. Для всех операций, кроме знакового деления, итоговый битовый вектор есть представление в прямом коде результата соответствующей операций в кольце вычетов по модулю 2^b для чисел, чьё представление в прямом коде равно битовым векторам $\mathbb{X}(\texttt{h})$ и $\mathbb{X}(\texttt{g})$ соответственно. Эти же операции можно интерпретировать как арифметику над знаковыми числами в дополнительном коде. Знаковое деление интерпретирует свои операнды как числа в дополнительном коде. Единственный случай переполнения — $2^{b-1}/_s(-1)$, в этом случае значение 2^{b-1} принимается в качестве результата[4].

Инструкции `z = zext(x, e)` и `z = sext(x, e)` обозначают беззнаковое и знаковое приведение к типу большего размера (`e` $> b$). В логике битовых векторов может быть записано как *zeroExtend*$(\texttt{x}, \texttt{e} - b)$ — расширение нулём и *signExtend*$(\texttt{x}, \texttt{e} - b)$ — расширение знаковым битом для дополнительного кода. Инструкция `z = trunc(x, e)` выполняет приведение к типу меньшего размера (`e` $< b$) путём отбрасывания старших битов. В логике битовых векторов может быть записано как *extract*$(\texttt{x}, 0, \texttt{e})$.

Инструкции передачи управления представлены инструкцией условного перехода `if (`*`cond`*`) { `*`true-branch`*` } else { `*`else-branch`*` }` и цикла `while (`*`cond`*`) do { `*`loop-body`*` }` с традиционной семантикой.

В качестве условий переходов выступают предикаты равенства и неравенства битовых векторов (анализируют побитовое совпадение), беззнаковые целочисленные сравнения ($<_u, >_u, \geqslant_u, \leqslant_u$ — интерпретируют свои операнды как числа в прямом коде) и знаковые целочисленные сравнения ($<_s, >_s, \geqslant_s, \leqslant_s$ — интерпретируют свои операнды как числа в дополнительном коде).

Множество возможных условий переходов `<Cond>` обозначим $SimpleCond$. Заметим, что это множество замкнуто относительно операции отрицания. Например, $\neg(\texttt{h} \geqslant_u \texttt{g}) = (\texttt{h} <_u \texttt{g}) \in SimpleCond$.

[4] Стандарт С не специфицирует используемое представление целых чисел. В модельном языке дополнительный код был выбран ввиду его популярности, а также ради консистентности с теорией битовых векторов.

Также в языке имеется инструкция вызова функции `call f (`*params*`)`. Количество параметров функции может быть любым, но количество и типы параметров в инструкции вызова функции должны совпадать с прототипом вызываемой функции. Параметры могут иметь следующие типы: `var x` — битовый вектор определённого размера, `ptr p` и `arr m` — указатель на ячейку и массив ячеек определённого размера соответственно.

Возврат из функции осуществляется с помощью оператора `return`. Если он является последним оператором в тексте функции, то его можно опустить.

В ходе анализа текст программы на модельном языке будет преобразовываться, в результате чего в него могут быть вставлены новые инструкции (см. раздел 3.2).

3.2 Общий алгоритм анализа функции

Как уже было сказано, требованиями к анализатору определяется необходимость анализа каждой функции в программе как независимой точки входа. При этом предполагается, что среди параметров-указателей функции нет алиасов. В данном разделе будет рассмотрен общий алгоритм внутрипроцедурного анализа (см. Алгоритм 3.1).

Для проведения анализа текст функции на модельном языке преобразуется в граф потока управления. Вершинами в этом графе являются инструкции исходной программы. При этом при генерации ветвления с условием $c \in SimpleCond$ 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*.

Алгоритм 3.1 — Внутрипроцедурный алгоритм поиска переполнения буфера

Входные данные: F — анализируемая функция
Параметр: k — число анализируемых итераций
Выходные данные: *warning-List* — набор предупреждений

```
G ← Build-CFG(F) ;
G_k ← Unroll-Loops(G, k) ;
instr-List ← Top-Sort-Vertices(G_k) ;
warning-List ← ∅ ;
foreach instr ∈ instr-List do
    if instr не имеет предков then
        A_in ← A_entry ;
    end
    else if instr имеет более одного предка then
        n ← количество предков instr ;
        A_in1 ← Out-State(pred(instr)[1]) ;
        ...
        A_inN ← Out-State(pred(instr)[n]) ;
        A_in ← Join-States(A_in1, ... ,A_inN) ;
    end
    else
        A_in ← Out-State(pred(instr)[1])
    end
    if instr — это инструкция доступа к буферу then
        warning ← Check-Safety(A_in, instr) ;
        if warning ≠ ∅ then
            warning-List.Add(warning) ;
        end
    end
    A_out ← Perform-Single-Step(A_in, instr) ;
end
```

Полученный ориентированный граф G_k не содержит циклов, поэтому его вершины (соответствующие инструкциям из *Instr*) могут быть упорядочены с помощью топологической сортировки [67]. Теперь при использовании данного порядка при обходе инструкций программы в процессе анализа гарантируется, что при обработке любой инструкции все её предки уже были проанализированы. С другой стороны, использование развёртки вместо самого ГПУ при анализе приводит к тому, что анализируется поведение программы только на тех конкретных выполнениях, которые проходят не более k^n раз по каждому обратному ребру цикла вложенности n. Будем считать *рассматриваемыми анализом* такие конкретные выполнения и соответствующие им конкретные состояния.

3.2.1 Абстрактное состояние анализа

Анализ осуществляется путём продвижения абстрактного состояния программы $\mathcal{A}$ 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 исполнением. 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:
  1. $c_b \in C_b \;\Rightarrow\; c_b \in SE_b$,
  2. $s_b \in S_b \;\Rightarrow\; s_b \in SE_b$,
  3. $s_1, s_2 \in SE_b,\ \circ \in \{+, -, *, /_s, /_u\} \;\Rightarrow\; s_1 \circ s_2 \in SE_b$,
  4. $s_b \in SE_b,\ e \in B, e < b \;\Rightarrow\; trunc(s_b, e) \in SE_e$ (приведение к типу меньшего размера),

5. $s_b \in SE_b,\ e \in B, e > b \ \Rightarrow\ \mathit{zext}(s_b, e),\ \mathit{sext}(s_b, e) \in SE_e$ (беззнаковое и знаковое расширение типа соответственно),
6. $s_1, s_2 \in SE_b \ \Rightarrow\ s_1 \,|\, s_2,\ s_1 \,\&\, s_2 \in SE_b$ (побитовое «или» и побитовое «и»),
7. $\tau \in \mathcal{C},\ s_1, s_2 \in SE_b \ \Rightarrow\ \mathit{ite}(\tau, s_1, s_2) \in SE_b$ — условное значение, которое равно s_1, если выполнено условие τ, или s_2 в противном случае (множество условий $\mathcal{C}$ определено ниже).

– $SE \doteq \bigcup_{b \in B} SE_b$.
– 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) [68]. Тождественную истину и ложь будем обозначать $\top$ и $\bot$ соответственно.

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

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

Значения, хранящиеся в переменных и ячейках памяти в абстрактном состоянии, представлены символьными выражениями, как при символьном исполнении. Это соответствие определяется отображением σ. Отображение для инструкции $q \in \mathit{Instr}$ будем обозначать σ_q.

Формула $\pi \in \mathcal{C}$, определённая для некоторой точки в функции, является необходимым условием достижимости данной точки. Это условие аналогично предикату пути в символьном исполнении, однако в данном случае π определяется не для пути, а для точки, что соответствует символьному исполнению с объединением состояний [51; 69]. Для краткости условие достижимости точки $q \in \mathit{Instr}$ будем обозначать как $\pi(q)$.

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» [70]. Неизвестная ячейка μ будет использоваться в качестве значения по умолчанию, т.е. если для некоторого указателя явно не задано значение отображения $\mathcal{P}$, то оно равно $\langle \mu, \top \rangle$. Будем считать, что $p \in P_b \Rightarrow \forall\, i : a_i \neq \mu$, т.е. переменные-указатели не могут указывать на неизвестные ячейки. При этом новые абстрактные ячейки и массивы для значений указателей из памяти будут создаваться лениво при разыменовании этих указателей (при чтении/записи, обращении к массиву), а до тех пор в качестве значения $\mathcal{P}$ будет использоваться μ.

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 $m \in \mathcal{B}$.

Отображение $\mathcal{V}$ будет подробно описано в последующих разделах. На текущем этапе для определения начального абстрактного состояния упомянем лишь, что специальное значение $\varepsilon \in Summary$ означает отсутствие информации о значении символьного выражения, а также что $C \subset Summary$ и что константы отображением $\mathcal{V}$ всегда переводятся в себя. По аналогии с отображением $\mathcal{P}$ в данном случае значением по умолчанию будем считать значение ε.

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 $\psi : SE_b \to C_b$ (по аналогии с данным выше определением SE_b 3.2.1) и формул в теории битовых векторов $\psi : \mathcal{C} \to \{\top, \bot\}$.

Определение 5. 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. $\forall a \in \mathit{AM}^p : \exists \langle d, \gamma \rangle \in \mathcal{P}(a) : \psi(\gamma) = \top \wedge (\psi(d) = \mathbb{P}(\psi(a)) \vee d = \mu)$ (эквивалентность значений адресов в памяти),
5. $\forall m \in \mathit{AArr} : m \in \mathcal{B} \Rightarrow \mathcal{B}(m) = \mathbb{B}(\psi(m))$ (эквивалентность размеров массивов).

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

Определение 6. 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 7. 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.

Начальное состояние

Обозначим множество входных параметров функции, являющихся обычными переменными, указателями на ячейки памяти и массивы, как *VArgs*, *PArgs* и *ArrArgs* соответственно. Состояние программы на входном ребре в функцию инициализируется начальным состоянием $\mathcal{A}_{\mathit{entry}}$:

$$
\begin{aligned}
\mathcal{A}_{\mathit{entry}} = \langle\, \top,\ \sigma_{\mathit{entry}}, \mathcal{P}_{\mathit{entry}},\ \mathcal{B}_{\mathit{entry}},\ \mathcal{V}_{\mathit{entry}} \,\rangle,& \\
\forall b \in B : \forall x_b \in \mathit{VArgs} :\quad & v = \mathit{initVar}(v, b), \\
& \sigma_{\mathit{entry}}(x_b) = v, \\
\forall p_b \in \mathit{PArgs} :\quad & m = \mathit{initCell}(p_b, b), \\
& \mathcal{P}_{\mathit{entry}}(p_b) = \{\langle\, m, \top \rangle\}, \\
& \sigma_{\mathit{entry}}(m) = \mathit{initVar}(m, b), \\
\forall m_b \in \mathit{ArrArgs} :\quad & a = \mathit{initArr}(m_b, b), \\
& \mathcal{P}_{\mathit{entry}}(m_b) = \{\langle\, a, \top \rangle\}, \\
\forall c \in C_b :\quad & \mathcal{V}_{\mathit{entry}}(c) = c,
\end{aligned}
$$

где функция $initVar(a, b) : VArgs \cup AM \to S$ возвращает всегда одну и ту символьную переменную размера b для одинаковых a и разные переменные для разных a, функция $initCell(a, b) : PArgs \cup AM^p \to AM$ возвращает всегда одну и ту же ячейку размера b для одинаковых a и разные новые ячейки для разных a, функция $initArr(a, b) : ArrArgs \to AArr$ аналогичным образом инициализирует массивы (корректность такого подхода следует из предположения об отсутствии алиасов среди аргументов функции). Таким образом, входное состояние функции параметризуется значениями входных переменных и состоянием памяти. Т.к. размеры массивов, переданных по указателю как параметр, неизвестны, то отображение $\mathcal{B}$ для них не определяется.

Заметим, что начальное состояние является точным и корректным.

Продвижение состояния

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 3.3.7). Далее, если инструкция осуществляет доступ к буферу на запись или чтение, то проверяется отсутствие ошибки (см. раздел 3.2.2). Затем с учётом семантики текущей инструкции происходит вычисление выходного состояния с помощью передаточных функций (см. раздел 3.3), которое будет считаться состоянием на всех выходных рёбрах из данной инструкции.

3.2.2 Построение достаточных условий ошибки

Рассмотрим общий подход к проверке инструкции доступа к буферу на наличие потенциальной ошибки. Для рассматриваемой инструкции доступа необходимо установить, существует ли путь на графе развёртки, проходящий через данную инструкцию и удовлетворяющий определению 1.

Рассмотрим произвольную инструкцию доступа к буферу *ac*, расположенную в функции f. Обозначим $\vec{t}$ произвольный набор конкретных значений неизвестных переменных этой функции. Множество путей графа G_k от входа в функцию до точки *ac* обозначим P_{ac}, а условие достижимости *ac* по некоторому пути $p \in P_{ac}$, записанное как формула над $\vec{t}$, обозначим $ReachCond^p_{ac}(\vec{t})$. Условие ошибки в инструкции *ac* для конкретных значений обозначим $Error_{ac}(\vec{t})$. Тогда

условие ошибки по определению 1 может быть записано как:

$$\exists p \in P_{ac}: \quad \exists \vec{t}: \mathit{ReachCond}^p_{ac}(\vec{t}) \ \wedge \ \forall \vec{t}: (\mathit{ReachCond}^p_{ac}(\vec{t}) \Rightarrow \mathit{Error}_{ac}(\vec{t})). \quad (3.1)$$

Проверка этого условия напрямую (с помощью SMT-решателя) сопряжена с двумя трудностями: во-первых, условие $\mathit{ReachCond}^p_{ac}(\vec{t})$ в общем случае уникально для каждого пути, что затрудняет построение общей формулы; во-вторых, современные SMT-решатели не слишком эффективно справляются с проверкой совместности формул с кванторами всеобщности (по сравнению с проверкой свободных от кванторов формул). Таким образом, предлагается построить как можно более слабое достаточное условие для (3.1), имеющее более простую и свободную от кванторов формулировку.

Оказывается, что для многих видов ошибок (в т.ч. для многих ошибок переполнения буфера, как будет показано в этой главе) естественным образом может быть построено для произвольной точки q условие $\mathit{FaultyPoint}_q(\vec{t})$ (форма которого не зависит от пути до q), обладающее следующим свойством:

$$\begin{aligned} \forall p \in P_q: \quad & (\exists \vec{t}: \mathit{FaultyPoint}_q(\vec{t}) \ \wedge \ \mathit{ReachCond}^p_q(\vec{t})) \ \Rightarrow \\ & \Rightarrow \ (\forall \vec{t}: \mathit{ReachCond}^p_q(\vec{t}) \Rightarrow \mathit{Error}_q(\vec{t})). \end{aligned} \quad (3.2)$$

Если для некоторого пути $p \in P_q$ выполнена посылка импликации из формулы (3.2), то, во-первых, выполнено условие $\exists \vec{t} \ : \ \mathit{ReachCond}^p_{ac}(\vec{t})$ как часть конъюнкции, а во-вторых, для этого пути выполнено $\forall \vec{t} \ : \ \mathit{ReachCond}^p_q(\vec{t}) \ \Rightarrow \ \mathit{Error}_q(\vec{t})$ (следует из (3.2)). Значит, для точки q выполнено условие (3.1), т.к. для пути p выполнено условие под квантором существования. Т.к. наличие ошибки хотя бы на одном из путей до q означает наличие ошибки в этой точке, то итоговое достаточное условие ошибки в точке q можно записать как

$$\exists \vec{t}: \mathit{FaultyPoint}_q(\vec{t}) \ \wedge \ \mathit{ReachCond}_q(\vec{t}), \quad (3.3)$$

где $\mathit{ReachCond}_q(\vec{t}) \doteq \bigvee_{p \in P_q} \mathit{ReachCond}^p_q(\vec{t})$. Данное условие не зависит от пути и не включает кванторов всеобщности, а, значит, легко может быть проверено SMT-решателем. Дальнейшее изложение в этом разделе будет посвящено построению аналога условия $\mathit{FaultyPoint}_q(\vec{t})$ для абстрактного состояния.

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 $b \in B$. Переменные, являющиеся элементами этого множества, будут описаны позднее,

там, где они будут требоваться. Множество символьных выражений над такими переменными обозначим SE_b^{Aux}, пусть также $SE^{Aux} \doteq \bigcup_{b \in B} SE_b^{Aux}$. Отметим лишь, что так как размеры всех буферов константны и не зависят от входных значений, то они принадлежат SE_r^{Aux}. Значения этих выражений также не зависят от конкретных значений $\vec{t}$.

Искомое условие наличия ошибки переполнения $FaultyPoint_q(\vec{t})$ 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 $p = \{q_i\},\ q_i \in Instr$ 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 8. 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 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 в q_{ac} *является формула:*

$$\pi(q_{ac}) \wedge \bigvee_{\substack{\langle m, \gamma \rangle \in \mathcal{P}(\mathtt{p}),\\ m \in \mathcal{B}}} \gamma \wedge (NotLess(q_{ac}, \sigma(\mathtt{i}), \mathcal{B}(m)) \vee NotGreater(q_{ac}, \sigma(\mathtt{i}), -1)) \tag{3.4}$$

Доказательство. Исходя из определения ошибки 1, необходимо показать, что если формула (3.4) выполнима, то найдётся конкретное состояние $\langle \mathbb{X}, \mathbb{P}, \mathbb{B} \rangle$ в точке q_{ac} такое, что

(i) выполнено $(\mathbb{X}(\mathtt{i}) \geqslant_s \mathbb{B}(\mathbb{P}(\mathtt{p})))$ или $(\mathbb{X}(\mathtt{i}) <_s 0)$,

(ii) то же условие в этой точке выполнено для любого конкретного состояния, соответствующего тому же пути.

Пусть (3.4) выполнима. Значит, существует функция конкретизации ψ, которая переводит эту формулу в истину. Тогда для этой функции заведомо $\psi(\pi(q_{ac})) = \top$. Из условия о точности абстрактного состояния следует, что найдётся эквивалентное функции ψ конкретное состояние. Обозначим за l любой путь, по которому достижимо это конкретное состояние.

Из однозначности выбора значения для $\mathcal{P}(\mathtt{p})$ следует, что существует не более одной пары $\langle m, \gamma \rangle \in \mathcal{P}(\mathtt{p})$, для которой $\psi(\gamma) = \top$. Т.к. (3.4) выполнима, то найдётся ровно одна такая пара, причём для неё $m \in \mathcal{B}$. Обозначим $s = \mathcal{B}(m)$.

Не ограничивая общности, проведём доказательство для условия переполнения правой границы — пусть $\psi(\mathit{NotLess}(q_{ac}, \sigma(\mathtt{i}), s)) = \top$. Выбор пути l гарантирует, что существует как минимум одно соответствующее ему конкретное состояние. Осталось показать, что для любого соответствующего l конкретного состояния выполнено $\mathbb{X}(\mathtt{i}) \geqslant_s \mathbb{B}(\mathbb{P}(\mathtt{p}))$.

Рассмотрим соответствующую l последовательность абстрактных состояний $\{\mathcal{A}_i\}$ от входа в функцию до точки q_{ac}. Т.к. по условию теоремы каждое из них корректно, то для любого соответствующего l конкретного состояния $\langle \mathbb{X}, \mathbb{P}, \mathbb{B} \rangle$ в точке q_{ac} найдётся такая последовательность функций конкретизаций $\{\widetilde{\psi}_i\}$, что $\forall i : \widetilde{\psi}_i(\pi_i) = \top$ и $\widetilde{\psi}_{q_{ac}}$ эквивалентна этому конкретному состоянию. Из определения *NotLess* следует, что раз $\psi(\mathit{NotLess}(q_{ac}, \sigma(\mathtt{i}), s)) = \top$, то для любой такой последовательности верно $\widetilde{\psi}_{q_{ac}}(v) \geqslant_s \widetilde{\psi}_{q_{ac}}(x)$. Из эквивалентности следует, что $\mathbb{X}(\mathtt{i}) \geqslant_s \mathbb{B}(\mathbb{P}(\mathtt{p}))$. □

Таким образом, задача поиска ошибок переполнения описанного типа сводится к вычислению как можно более слабых условий $\mathit{NotLess}(q, v, x)$ и $\mathit{NotGreater}(q, v, x)$. Для её решения для символьного выражения v в точке q определяется значение $\mathcal{V}(v) \in \mathit{Summary}$, суммирующее информацию о значениях v по всем путям, заканчивающимся в q. Искомые условия $\mathit{NotLess}(q, v, x)$ и $\mathit{NotGreater}(q, v, x)$ 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 $\mathcal{V}(ac, \sigma(\texttt{i}))$ составляется формула (3.4) и проверяется на выполнимость.
3. В случае, если формула выполнима, т.е. подобраны значения символьных переменных S, приводящие к переполнению, из $\mathcal{V}(ac, \sigma(\texttt{i}))$ путём подстановки конкретных значений переменных извлекается конкретное выполнение, приводящее к ошибке, и выдается предупреждение, указывающее на соответствующий этому выполнению путь.

Рассмотрим подробнее, что представляет собой отображение $\mathcal{V}$ и как вычислять условия $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ с его помощью.

$$\mathcal{V} : SE \to Summary$$

Каждое значение $s_v \in Summary,\ v \in SE_b$ представляет собой набор элементов, первым из которых всегда является символьное выражение v, информацию о котором данное значение s_v суммирует по разным путям графа развёртки. Таким образом, если $\mathcal{V}(v) = s_v$, то первым элементом набора s_v является символьное выражение v. Далее будем, как и здесь, обозначать данный факт с помощью нижнего индекса.

Определение 9. Для вычисления с помощью s_v условий $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ определены вспомогательные функции:

$$\mathfrak{U}, \mathfrak{L} : Summary \times SE \to \mathcal{C}.$$

Для любых $x \in SE_b^{Aux}$, $q \in Instr$, если выполнено

(i) $\mathcal{V}(q, v) = s$,

(ii) найдётся такая последовательность функций конкретизации $\{\psi_i\}$ для $\{\mathcal{A}_i\}$, что $\forall i : \psi_i(\pi_i) = \top$,

(iii) абстрактное состояние $\mathcal{A}_n$ точно и $\psi_n(\mathfrak{U}(s, x)) = \top \ \left(\psi_n(\mathfrak{L}(s, x)) = \top\right)$,

то для любой последовательности функций конкретизации $\{\widetilde{\psi}_i\} : \forall i : \widetilde{\psi}_i(\pi_i) = \top$ заведомо выполнено $\widetilde{\psi}_n(v) \geqslant_s \widetilde{\psi}_n(x) \ \left(\widetilde{\psi}_n(v) \leqslant_s \widetilde{\psi}_n(x)\right)$.

С помощью этих функций будем вычислять условия $\mathit{NotLess}(q, v, x)$ и $\mathit{NotGreater}(q, v, x)$:

$$\begin{aligned} \mathit{NotLess}(q, v, x) &\doteq \mathfrak{U}(\mathcal{V}(q, v), x), \\ \mathit{NotGreater}(q, v, x) &\doteq \mathfrak{L}(\mathcal{V}(q, v), x). \end{aligned} \tag{3.5}$$

Таким образом, задача свелась к построению отображения $\mathcal{V}$ и вычислению условий $\mathfrak{U}, \mathfrak{L}$ (опять, чем слабее будут эти условия, тем лучше). Введение вспомогательных функций удобно, так как данные функции будут впоследствии определены рекурсивно.

3.2.3 Структура множества *Summary* и построение условий $\mathfrak{U}, \mathfrak{L}$

Значения множества *Summary* принадлежат одному из шести типов:

$$\mathit{Summary} \doteq \{\varepsilon\} \cup C \cup \mathit{Assume} \cup \mathit{Arithm} \cup \mathit{Cast} \cup \mathit{Join}.$$

Рассмотрим каждый из этих типов с соответствующими условиями $\mathfrak{U}, \mathfrak{L}$ и покажем, что они удовлетворяют определению 9.

Пустое значение

Отсутствие какой-либо полезной информации о значении символьного выражения будет обозначаться специальным элементом $\varepsilon \in \mathit{Summary}$. В этом случае доказать наличие ошибки в функции на основании этого значения невозможно, поэтому

$$\mathfrak{U}(\varepsilon, x) \doteq \mathfrak{L}(\varepsilon, x) \doteq \bot.$$

Значение $\bot$ всегда удовлетворяет определению 9, т.к. условие (iii) никогда не выполняется.

Константы

Значения констант и переменных из множества SE^{Aux} не зависят от входных параметров по определению (т.е. на каждом пути имеют всегда одно и то же значение для любых значений переменных из S), поэтому если $c \in C_b$, $x \in SE_b^{Aux}$ и для некоторой функции конкретизации $c \geqslant_s x$ ($c \leqslant_s x$), то это условие верно и для любой другой функции конкретизации. Отсюда следует

***Лемма* 1.** *Для значений Summary, являющихся константами, в качестве условий* $\mathfrak{U}, \mathfrak{L}$ *могут быть выбраны следующие формулы:*

$$\mathfrak{U}(c, x) \doteq c \geqslant_s x, \qquad \mathfrak{L}(c, x) \doteq c \leqslant_s x.$$

Целочисленные сравнения

Элементы данного типа отражают факт истинности отношения $\diamond \in \{>_s, <_s, \geqslant_s, \leqslant_s, >_u, <_u, \geqslant_u, \leqslant_u, =, \neq\}$ для пары символьных выражений $v = \sigma(\mathtt{g})$ и *comparand* $= \sigma(\mathtt{h})$, вытекающий из перехода по условию $\mathtt{g} \diamond \mathtt{h}$. Значение типа *Assume* является результатом отображения $\mathcal{V}$ для символьного выражения v в точке q в том случае, когда $\mathcal{V}(q, \mathit{comparand}) = s_{\mathit{comparand}}$, и у некоторого условного оператора ветка с условием $\mathtt{g} \diamond \mathtt{h}$ доминирует над точкой q. Очевидно, что условие перехода по данной ветке гарантированно выполнено в текущей точке.

$$\begin{aligned} \mathit{Assume} \doteq \{\langle v, s_{\mathit{comparand}}, \diamond\rangle \quad | \quad & v \in \mathit{SE}, \\ & s_{\mathit{comparand}} \in \mathit{Summary}, \quad \diamond \in \{>_s, <_s, \geqslant_s, \leqslant_s, >_u, <_u, \geqslant_u, \leqslant_u, =, \neq\}\} \end{aligned}$$

Пусть t^u означает битовый вектор длины u, значения каждого бита которого равны $t \in \{0, 1\}$; $e \cdot j$ означает конкатенацию битовых векторов e и j.

Таблица 1 — Условия $\mathfrak{U}$ и $\mathfrak{L}$ для целочисленных знаковых сравнений

Условие	$\mathfrak{U}(s_v, x)$	$\mathfrak{L}(s_v, x)$
$v >_s \mathit{comp}$	$\mathfrak{U}(s_{\mathit{comp}}, x-1) \wedge (x >_s 1^1 \cdot 0^{b-1})$	$\bot$
$v \geqslant_s \mathit{comp}$	$\mathfrak{U}(s_{\mathit{comp}}, x)$	$\bot$
$v <_s \mathit{comp}$	$\bot$	$\mathfrak{L}(s_{\mathit{comp}}, x+1) \wedge (x <_s 0^1 \cdot 1^{b-1})$
$v \leqslant_s \mathit{comp}$	$\bot$	$\mathfrak{L}(s_{\mathit{comp}}, x)$
$v = \mathit{comp}$	$\mathfrak{U}(s_{\mathit{comp}}, x)$	$\mathfrak{L}(s_{\mathit{comp}}, x)$

***Лемма* 2.** *Искомые условия для всех рассматриваемых целочисленных отношений приведены в таблице 1.*

Доказательство. Вывод искомых достаточных условий рассмотрим на примере отношения $\geqslant_s$. Пусть в точке q выполнено $\mathcal{V}(q, v) = s_v$, где

$$s_v = \langle v, s_{\mathit{comp}}, \geqslant_s\rangle \in \mathit{Assume} \quad | \quad s_{\mathit{comp}} \in \mathit{Summary}.$$

Т.к. ветка с условием $\mathtt{g} \geqslant_s \mathtt{h}$ доминирует над точкой q, то для любого конкретного состояния в этой точке выполнено $\mathbb{X}(g) \geqslant_s \mathbb{X}(h)$. Из условия о точности абстрактного состояния в q следует, что для любой функции конкретизации ψ заведомо $\psi(v \geqslant_s \mathit{comp}) = \top$. Таким образом, если выполнено $\psi(\mathit{comp} \geqslant_s x) = \top$, также верно $\psi(v \geqslant_s x)$, поэтому для точки q можно записать:

$$\mathfrak{U}(s_v, x) = \mathfrak{U}(s_{\mathit{comp}}, x).$$

Дополнительной информации о верхней границе значения v данное сравнение не даёт, поэтому $\mathfrak{L}(s_v, x) \doteq \bot$. Условия для других отношений рассматриваются аналогично. Для случаев строгого неравенства также добавлены условия на отсутствие целочисленного переполнения, которое нарушило бы транзитивность отношений $>_s$ и $<_s$. Для битового вектора размера b, если отрицательные числа представлены в дополнительном коде, то $\mathtt{INT\_MIN} = 1^1 \cdot 0^{b-1}$, $\mathtt{INT\_MAX} = 0^1 \cdot 1^{b-1}$. □

Беззнаковые сравнения могут также быть использованы для построения значений *Assume*, если добавить в формулы $\mathfrak{U}$ и $\mathfrak{L}$ условия на интервалы операндов сравнений, но в данной работе для краткости изложения будем считать $\mathfrak{U}(s_v, x) \doteq \mathfrak{L}(s_v, x) \doteq \bot$ при $\diamond \in >_u, <_u, \geqslant_u, \leqslant_u$.

Арифметические операции

Значения данного типа представляют результат арифметической операции $\mathtt{h} \circ \mathtt{g}$, для каждого из операндов которой в данной точке $q_\circ$ определено значение отображения: $v_h = \sigma(\mathtt{h})$, $\mathcal{V}(q_\circ, v_h) = s_h$, $v_g = \sigma(\mathtt{g})$, $\mathcal{V}(q_\circ, v_g) = s_g$.

$$\begin{aligned} \mathit{Arithm} \doteq \{\langle v_h \circ v_g, s_h, s_g, \circ\rangle \quad | \\ v_h, v_g \in \mathit{SE}_b, \quad s_h, s_g \in \mathit{Summary}, \quad \circ \in \{+, -, *, /_s, /_u\}\} \end{aligned}$$

В данном случае очевидного способа оценить значение $v_h \circ v_g$ с помощью значений v_h и v_g нет. Предлагается построить нетривиальные (отличные от $\bot$) достаточные условия для случая, когда для значений v_h и v_g на некотором рассматриваемом пути на графе развёртки можно подобрать некоторые оценки, справедливые для всех соответствующих конкретных выполнений. В таком случае эти оценки можно доказать привычным способом, показав выполнимость условий $\mathfrak{U}$ и $\mathfrak{L}$. Далее с помощью этих оценок получить оценку на значение результата арифметической операции (константную для пути на графе развёртки),

которую в свою очередь использовать для доказательства условий $\mathfrak{U}$ и $\mathfrak{L}$ для результата арифметической операции.

Таблица 2 — Условия $\mathfrak{U}$ и $\mathfrak{L}$ для сложения и вычитания

$\mathfrak{U}(\langle v_h + v_g, s_h, s_g, + \rangle, x)$	$\exists \hat{h}\, \exists \hat{g} \quad \mathfrak{U}(s_h, \hat{h}) \wedge \mathfrak{U}(s_g, \hat{g}) \wedge$ $\wedge (\hat{h} + \hat{g} \geqslant_s x) \wedge \mathit{NoIntOverflow}_+(h, g, \hat{h}, \hat{g})$
$\mathfrak{L}(\langle v_h + v_g, s_h, s_g, + \rangle, x)$	$\exists \hat{h} \exists \hat{g} \quad \mathfrak{L}(s_h, \hat{h}) \wedge \mathfrak{L}(s_g, \hat{g}) \wedge$ $\wedge (\hat{h} + \hat{g} \leqslant_s x) \wedge \mathit{NoIntOverflow}_+(h, g, \hat{h}, \hat{g})$
$\mathfrak{U}(\langle v_h - v_g, s_h, s_g, - \rangle, x)$	$\exists \hat{h} \exists \hat{g} \quad \mathfrak{U}(s_h, \hat{h}) \wedge \mathfrak{L}(s_g, \hat{g}) \wedge$ $\wedge (\hat{h} - \hat{g} \geqslant_s x) \wedge \mathit{NoIntOverflow}\_(h, g, \hat{h}, \hat{g})$
$\mathfrak{L}(\langle v_h - v_g, s_h, s_g, - \rangle, x)$	$\exists \hat{h} \exists \hat{g} \quad \mathfrak{L}(s_h, \hat{h}) \wedge \mathfrak{U}(s_g, \hat{g}) \wedge$ $\wedge (\hat{h} - \hat{g} \leqslant_s x) \wedge \mathit{NoIntOverflow}\_(h, g, \hat{h}, \hat{g})$

***Лемма* 3.** *Для элементов Arithm, соответствующих сумме и разности двух переменных, в качестве искомых условий могут быть выбраны достаточные условия, приведённые в таблице 2.*

Доказательство. Проведём доказательство для условия $\mathfrak{U}(s_v, x)$ на примере вычитания. Необходимо показать, что если для некоторого пути и соответствующей ему последовательности абстрактных состояний $\{\mathcal{A}_i\}$ выполнено:

(i) $\mathcal{V}(q, v_h - v_g) = s_v = \langle v_h - v_g, s_h, s_g, - \rangle \in \mathit{Arithm}$,

(ii) найдётся такая последовательность функции конкретизации $\{\psi_i\}$ для $\{\mathcal{A}_i\}$, что $\forall i : \psi_i(\pi_i) = \top$,

(iii) абстрактное состояние $\mathcal{A}_{q_\circ}$ точно и $\psi_{q_\circ}(\mathfrak{U}(s_v, x)) = \top$,

то для любой последовательности функций конкретизации $\{\widetilde{\psi}_i\} : \forall i : \widetilde{\psi}_i(\pi_i) = \top$ заведомо выполнено $\widetilde{\psi}_{q_\circ}(v_h - v_g) \geqslant_s \widetilde{\psi}_{q_\circ}(x)$.

Заметим, что для любых l,$\hat{l}$,k,$\hat{k} \in \mathbb{Z}$ верно

$$(l \geqslant \hat{l}) \wedge (k \leqslant \hat{k}) \wedge (\hat{l} - \hat{k} \geqslant x) \Rightarrow l - k \geqslant x. \tag{3.6}$$

Данное условие верно для целых чисел, но неверно для логики битовых векторов из-за возможности машинного переполнения при вычитании. В рамках

данной работы будем считать, что конкретные выполнения, содержащие знаковое переполнение, не рассматриваются, т.е. доказывать наличие ошибки для них не является необходимостью, если на всех остальных конкретных выполнениях для того же абстрактного пути (на которых нет переполнения) ошибка есть.[5]

Таким образом, к условию (3.6) добавляется также условие отсутствия целочисленного переполнения при вычислении $l - k$ и $\hat{l} - \hat{k}$. Введём две вспомогательные переменные $\hat{h}, \hat{g} \in SE_b^{Aux}$ в качестве аналогов $\hat{l}, \hat{k}$ в битовых векторах. Если рассматривать значения битовых векторов как знаковые числа, то условие отсутствия переполнения для случая разности в предположении $(h \geqslant_s \hat{h}) \wedge (g \leqslant_s \hat{g})$ можно записать как

$$\mathit{NoIntOverflow\_}(h, g, \hat{h}, \hat{g}) = \bigwedge \begin{array}{l} (\hat{g} \leqslant_s 0) \vee (\hat{h} \geqslant_s \texttt{INT\_MIN} + \hat{g}) \\ (g \geqslant_s 0) \vee (\texttt{INT\_MAX} + g \geqslant_s h). \end{array}$$

Аналогично можно представить условие для сложения $\mathit{NoIntOverflow}_+$.

Используем этот факт для доказательства леммы. Пусть для некоторой функции конкретизации ψ следующая формула обращается в истину:

$$\mathfrak{U}(s_v, x) \doteq \mathit{NoIntOverflow\_}(h, g, \hat{h}, \hat{g}) \wedge \exists\hat{h}\exists\hat{g}\, \mathfrak{U}(s_h, \hat{h}) \wedge \mathfrak{L}(s_g, \hat{g}) \wedge (\hat{h} - \hat{g} \geqslant_s x)).$$

Рассмотрим произвольную последовательность функций конкретизации $\{\widetilde{\psi}_i\} : \forall i : \widetilde{\psi}_i(\pi_i) = \top$. Покажем, что для неё выполнено $\widetilde{\psi}_{q_\circ}(v_h - v_g) \geqslant_s \widetilde{\psi}_{q_\circ}(x)$.

Из условий $\psi(\mathfrak{U}(s_h, \hat{h})) = \top$ и $\psi(\mathfrak{L}(s_g, \hat{g})) = \top$ следует, что $\widetilde{\psi}_{q_\circ}(h) \geqslant_s \widetilde{\psi}_{q_\circ}(\hat{h})$ и $\widetilde{\psi}_{q_\circ}(g) \leqslant_s \widetilde{\psi}_{q_\circ}(\hat{g})$ соответственно (по определению условий $\mathfrak{U}$ и $\mathfrak{L}$). Условие $\psi_{q_\circ}(\hat{h}) - \psi_{q_\circ}(\hat{g}) \geqslant_s \psi_{q_\circ}(x)$ не зависит от значений переменных из множества

[5] Такое ограничение оправдывается тем фактом, что переполнение при арифметической операции над знаковыми переменными приводит к неопределённому поведению согласно стандарту языка C [12, §6.5/5]. Таким образом, разумно считать, что конкретные выполнения, содержащие целочисленное знаковое переполнение, запрещены контрактом функции, поэтому достаточно построить условие только для случаев без переполнения.

В то же время ошибки, связанные с целочисленным переполнением, зачастую в свою очередь являются причиной переполнения буфера. Однако можно заметить, что если есть ошибочный путь на графе развёртки в соответствии с определением 1 и переполнение буфера возникает на конкретных выполнениях с целочисленным переполнением, значит, для этого пути целочисленное переполнение происходит на всех не запрещённых контрактом конкретных выполнениях. Следовательно, такая ситуация удовлетворяет аналогичному данному нами для переполнения буфера определению ошибки для целочисленного переполнения, и она может быть обнаружена предложенным в данной работе подходом. Для этого, например, можно моделировать значения переменных битовыми векторами ширины большей, чем размер типа переменных, а вместо размера буфера использовать константы `INT_MAX`, `INT_MIN` и аналогичные.

S, поэтому если оно выполнено для одной функции конкретизации, то оно выполнено и для всех прочих. Условие $\psi(\mathit{NoIntOverflow\_}(h, g, \hat{h}, \hat{g}))$ гарантирует, что $\hat{h} - \hat{g}$ не приводит к переполнению, и найдется хотя бы одно конкретное выполнение для этого пути без переполнения в точке $q_\circ$. Отсюда следует, что $\widetilde{\psi}_{q_\circ}(v_h - v_g) \geqslant_s \widetilde{\psi}_{q_\circ}(x)$.

Остальные формулы из таблицы 2 рассматриваются аналогично. □

Введение дополнительных переменных $\hat{h}$ и $\hat{g}$ здесь необходимо, т.к. заранее (до анализа совместности условий переходов) определить нижние и верхние границы значений переменных h и g невозможно, поэтому значения $\hat{h}$ и $\hat{g}$ определит решатель.

Для случая умножения и деления можно применить такой же подход. Для этого необходимо разобрать все возможные знаки операндов арифметической операции и для каждого из случаев построить формулу, аналогичную (3.6).

Операции приведения типа

Элементы данных типов ставятся в соответствие результатам инструкций преобразования типов `cast`: *Trunc* для случая приведения типа большего размера к типу меньшего размера, *ZExt* и *SExt* для беззнакового и знакового расширения соответственно.

$$\mathit{Cast} \doteq \mathit{Trunc} \cup \mathit{ZExt} \cup \mathit{SExt}$$

В данном подразделе примем следующие обозначения: $x \in \mathit{SE}_e^{\mathit{Aux}}$, $t = \mathit{trunc}(x, b)$, $\bar{x} = x \,\&\, 1^{e-b} \cdot 0^b$.

Беззнаковое расширение.

$$\mathit{ZExt} \doteq \{\langle v, s_a, e\rangle \mid e \in B,\ v \in \mathit{SE}_e,\ s_a \in \mathit{Summary}\}$$

Пусть `z` — результат беззнакового расширения `z = zext(a, e);` до вектора ширины $e \in B$, $e > b$, для аргумента `a` которого в данной точке p_{zext} определено значение отображения $a = \sigma(\mathtt{a}) \in \mathit{SE}_b$, $\mathcal{V}(p_{\mathit{zext}}, a) = s_a$. Тогда $v = \sigma(\mathtt{z}) \in \mathit{SE}_e$, $\mathcal{V}(p_{\mathit{zext}}, v) = s_v = \langle v, s_a\rangle$.

***Лемма* 4.** *Для значений Summary, являющихся результатом беззнакового расширения, в качестве условий* $\mathfrak{U}, \mathfrak{L}$ *могут быть выбраны следующие формулы:*

$$\mathfrak{U}(s_v, x) \doteq x \leqslant_s 0 \ \vee\ (\bar{x} = 0 \wedge ((\mathfrak{U}(s_a, t) \wedge t >_s 0) \vee (\mathfrak{L}(s_a, t) \wedge t \leqslant_s 0)),$$

$$\mathfrak{L}(s_v, x) \doteq x \geqslant_s 0 \ \wedge\ (\bar{x} \neq 0 \vee ((\mathfrak{U}(s_a, 0) \wedge \mathfrak{L}(s_a, t)) \vee (\mathfrak{L}(s_a, 0) \wedge \mathfrak{U}(s_a, t)))).$$

Доказательство. Из семантики инструкции очевидно, что $\mathtt{z} \geqslant_s 0$. Поэтому $x \leqslant_s 0 \Rightarrow \mathtt{z} \geqslant_s x$. Для $x \geqslant_s 0$ следует рассмотреть два случая. Если $\bar{x} \neq 0$, то заведомо $x >_s \mathtt{z}$. В противном случае $\mathtt{z} \geqslant_s x \;\Leftrightarrow\; \mathtt{a} \geqslant_u t \;\Leftrightarrow\; (\mathtt{a} \geqslant_s t \wedge t >_s 0) \vee (\mathtt{a} \leqslant_s t \wedge t \leqslant_s 0)$. Аналогичные рассуждения можно провести для условия $\mathtt{z} \leqslant_s x$. Отсюда следует утверждение леммы. □

Знаковое расширение.

$$\mathit{SExt} \doteq \{\langle v, s_a, e\rangle \mid e \in B,\ v \in \mathit{SE}_e,\ s_a \in \mathit{Summary}\}$$

Пусть z — результат знакового расширения `z = sext(a, e);` до вектора ширины $e \in B,\ e > b$, для аргумента a которого в данной точке p_{sext} определено значение отображения $a = \sigma(\mathtt{a}) \in \mathit{SE}_b,\ \mathcal{V}(p_{sext}, a) = s_a$. Тогда $v = \sigma(\mathtt{z}) \in \mathit{SE}_e,\ \mathcal{V}(p_{sext}, v) = s_v = \langle v, s_a\rangle$. Искомые условия выводятся аналогично предыдущей инструкции.

***Лемма* 5.** *Для значений Summary, являющихся результатом знакового расширения, в качестве условий $\mathfrak{U}, \mathfrak{L}$ могут быть выбраны следующие формулы:*

$$\begin{aligned} \mathfrak{U}(s_v, x) &\doteq \bar{x} = 0 \ \wedge\ \mathfrak{U}(s_a, t), \\ \mathfrak{L}(s_v, x) &\doteq \bar{x} = 0 \ \wedge\ \mathfrak{L}(s_a, t). \end{aligned}$$

Приведение к типу меньшего размера.

$$\mathit{Trunc} \doteq \{\langle v, s_a, e\rangle \mid e \in B,\ v \in \mathit{SE}_e,\ s_a \in \mathit{Summary}\}$$

Пусть z — результат инструкции `z = trunc(a, e);` приведения a к типу меньшего размера, равного $e \in B,\ e < b$, и в этой точке p_{trunc} определено значение отображения $a = \sigma(\mathtt{a}) \in \mathit{SE}_b,\ \mathcal{V}(p_{trunc}, a) = s_a$. Обозначим $v = \sigma(\mathtt{z}) \in \mathit{SE}_e,\ \mathcal{V}(p_{trunc}, v) = s_v = \langle v, s_a, e\rangle$. Тогда верна

***Лемма* 6.** *Для значения $s_v \in \mathit{Trunc}$ в качестве условий $\mathfrak{U}, \mathfrak{L}$ могут быть выбраны следующие формулы:*

$$\begin{aligned} \mathfrak{U}(s_v, x) &\doteq \exists\, m:\ \mathit{trunc}(m, e) = x \ \wedge\ \mathfrak{U}(s_a,\, m) \ \wedge\ \mathfrak{L}(s_a,\, m \mid (0^{b-e} \cdot 1^e)), \\ \mathfrak{L}(s_v, x) &\doteq \exists\, m:\ \mathit{trunc}(m, e) = x \ \wedge\ \mathfrak{U}(s_a,\, m \,\&\, (1^{b-e} \cdot 0^e)) \ \wedge\ \mathfrak{L}(s_a,\, m). \end{aligned} \tag{3.7}$$

Доказательство. Для вывода достаточных условий обозначим за $m \in \mathit{SE}_b^{Aux}$ битовый вектор длины b, равной длине вектора a, зачение которого совпадает с a во

всех разрядах, кроме младших e разрядов, а значения в этих последних разрядах совпадают с x:

$$m = a \,\&\, 1^{b-e} \cdot 0^{e} \mid 0^{b-e} \cdot x.$$

Обозначим также $m_0 = m \,\&\, (1^{b-e} \cdot 0^{e})$ — вектор, равный вектору m, за исключением младших e разрядов, которые равны 0, и аналогичный вектор $m_1 = m \mid 0^{b-e} \cdot 1^{e}$, последние e разрядов которого равны 1.

Нетрудно заметить, что $v \geqslant_s x \Leftrightarrow a \in [m, m_1]$. Аналогично, $v \leqslant_s x \Leftrightarrow a \in [m_0, m]$. Таким образом, искомые формулы будут иметь вид (3.7). □

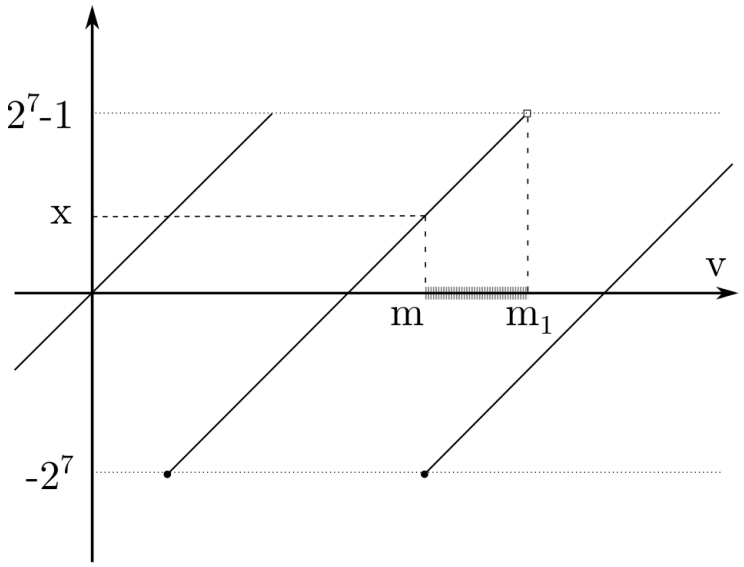

Рисунок 3.2 — Вычисление достаточного условия для нижней границы для результата инструкции приведения типа

Данное рассуждение для случая $e = 8$ проиллюстрировано на рисунке 3.2, где показана зависимость результата инструкции приведения значения v к однобайтному целому типу. Искомый интервал $[m, m \mid (0^{b-e} \cdot 1^{e})]$ для условия $\mathfrak{U}(s_v, x)$ на рисунке выделен отрезком на оси абсцисс. Формула для $\mathfrak{L}$ доказывается аналогично.

Слияние значений

Элементы данного типа соответствуют значениям, полученным в результате слияния нескольких символьных выражений, хотя бы для одного из которых определено значение $\mathcal{V}$.

$$\begin{aligned} \mathit{Join} \doteq \{ \langle j, \{ \langle s_{v_i}, \gamma_i \rangle \} \rangle \quad \mid \\ j \in \mathit{SE}_b,\ s_{v_i} \in \mathit{Summary},\ n \in \mathbb{N},\ \forall i \in [1{,}n] : \gamma_i \in \mathcal{C} \} \end{aligned}$$

Лемма 7. *Пусть p_{join} является точкой, в которой непустые значения $s_{v_1}, \ldots s_{v_n}$ объединяются в значение s_j. Условия, соответствующие этим значениям, обозначим как $\gamma_1, \ldots \gamma_n \in \mathcal{C}$ соответственно. Тогда*

$$\mathcal{V}(p_{join}, j) = \langle j, \{\langle s_{v_i}, \gamma_i \rangle\}\rangle$$

и корректны следующие условия:

$$\begin{aligned} \mathfrak{U}(s_j, x) &\doteq \bigvee\nolimits_{i=1}^{n} \mathfrak{U}(s_{v_i}, x) \wedge \gamma_i, \\ \mathfrak{L}(s_j, x) &\doteq \bigvee\nolimits_{i=1}^{n} \mathfrak{L}(s_{v_i}, x) \wedge \gamma_i. \end{aligned} \tag{3.8}$$

Доказательство. Если хотя бы для одного из объединяемых значений существует такая функция конкретизации ψ, что $\psi(\mathfrak{U}(s_{v_i}, x)) = \top$ и $\psi(\gamma_i) = \top$, то для той же функции ψ, кроме того, выполнено $\psi(\mathfrak{U}(s_j, x))$, так как $\psi(j) = \psi(v_j)$ (что гарантируется условием γ_i) и верно $\psi(\mathfrak{U}(s_{v_i}, x))$. Таким образом, искомое условие будет иметь вид (3.8). Формула для $\mathfrak{L}$ доказывается аналогично. □

Также значения данного типа будут использованы в случае, когда для одного символьного выражения становится известно несколько полезных фактов о его значении (например, несколько вложенных сравнений с разными величинами на одном пути).

Для создания значений данного типа введём вспомогательную функцию

$$\mathit{joinVals} : \mathit{SE} \times 2^{\mathit{Summary} \times \mathcal{C}} \to \mathit{Summary},$$

которая для символьного выражения $j \in \mathit{SE}$ и последовательности пар $\{\langle s_i, \gamma_i \rangle \mid s_i \in \mathit{Summary}, \gamma_i \in \mathcal{C}\}$ возвращает ε, если все значения s_i пустые, либо значение $\langle j, \{\langle s_{i_k}, \gamma_{i_k} \rangle\}\rangle$ типа *Join*, где $\{i_k\}$ — подпоследовательность индексов непустых значений s_i:

$$\mathit{joinVals}(j, \{\langle s_i, \gamma_i \rangle\}) \doteq \begin{cases} \varepsilon, & \forall i : s_i = \varepsilon, \\ \langle j, \{\langle s_{i_k}, \gamma_{i_k} \rangle\}\rangle \in \mathit{Join}, & \forall i_k : s_{i_k} \neq \varepsilon, \{i_k\} \neq \varnothing. \end{cases}$$

В следующем разделе будет показано, как в ходе анализа функции создаются значения из множества *Summary* (в т.ч. с помощью функции *joinVals*) и как они ставятся в соответствие символьным выражениям для построения отображения $\mathcal{V}$.

3.3 Передаточные функции

Обозначим $q \in$ *Instr* текущую обрабатываемую инструкцию, пусть также отображения $\mathcal{A}_{in}, \mathcal{A}_{out}$ определяют для q входное и выходное состояния соответственно. Для произвольного отображения M записью $M\{x \mapsto a\}$ будем обозначать отображение, совпадающее с M на всём множестве определения, кроме значения x, которому в новом отображении будет соответствовать значение a.

В этом разделе рассмотрим передаточные функции для каждой из инструкций и убедимся, что все инструкции сохраняют корректность абстрактного состояния (в предположении об отсутствии алиасов) и за исключением инструкции чтения значения из буфера сохраняют его точность. Будем считать, что на входе в каждую инструкцию состояние было точно и корректно. Под конкретными состояниями в данном разделе будем понимать рассматриваемые анализом конкретные состояния.

3.3.1 Инструкция условного перехода

Инструкция `assume(sc)`, `sc` $\in SimpleCond$ не изменяет состояние памяти или значения переменных, но влияет на условие достижимости текущей точки: условие `sc` формулируется в символьных выражениях с учётом текущего состояния и добавляется к предикату на выходящем ребре. Кроме того, сравнение может быть использовано для создания нового значения типа *Assume* для одного из операндов сравнения, если для второго из операндов значение $\mathcal{V}$ не пусто. Если отображение $\mathcal{V}$ для этого операнда уже содержит непустое значение, то необходимо сохранить оба значения, так как они оба верны в данной точке. Для этого можно создать значение типа *Join*, снабдив старое и новое значения тождественно истинными предикатами, что и будет означать, что они оба являются одновременно корректными в данной точке.

Запишем это формально в виде передаточной функции. Обозначим множество рассматриваемых отношений $R = \{>_s, <_s, \geqslant_s, \leqslant_s, >_u, <_u, \geqslant_u, \leqslant_u, =, \neq\}$. Будем обозначать для каждого из этих отношений $\diamond \in R$ обратное отношение как $\diamond^{-1}$, (например, $>_u^{-1} = <_u$). Введём вспомогательную функцию *createAssume*:

$$\mathit{createAssume} : \mathit{SE} \times \mathit{Summary} \times R \to \mathit{Summary},$$

$$\mathit{createAssume}(v, s, \diamond) = \begin{cases} \varepsilon, & s = \varepsilon, \\ \langle v, s, \diamond \rangle \in \mathit{Assume}, & s \neq \varepsilon. \end{cases}$$

Тогда передаточная функция для инструкции `assume(sc)` будет иметь вид:

$$\text{ASSUME}\frac{\begin{array}{c} \mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{assume}(\texttt{h} \diamond \texttt{g}) \\ \sigma \vdash \texttt{h} \to v_h \quad \sigma \vdash \texttt{g} \to v_g \quad \mathcal{V} \vdash v_h \to s_h \quad \mathcal{V} \vdash v_g \to s_g \\ \widetilde{s_h} = \mathit{joinVals}(\langle s_h, \top \rangle, \langle \mathit{createAssume}(v_h, s_g, \diamond), \top \rangle) \\ \widetilde{s_g} = \mathit{joinVals}(\langle s_g, \top \rangle, \langle \mathit{createAssume}(v_g, s_h, \diamond^{-1}), \top \rangle) \end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi \wedge (v_h \diamond v_g),\ \sigma,\ \mathcal{P},\ \mathcal{B}, \mathcal{V}\{v_h \mapsto \widetilde{s_h}, v_g \mapsto \widetilde{s_g}\} \rangle}$$

Рассмотрим произвольное конкретное состояние на выходе из инструкции. Ему соответствует некоторое конкретное состояние на входе, а для него по условию существует эквивалентная ему функция конкретизации ψ такая, что $\psi(\pi) = \top$. Из эквивалентности также следует, что $\psi(\sigma(\texttt{h})) \diamond \psi(\sigma(\texttt{g}))$, поэтому та же функция конкретизации для абстрактного состояния $\mathcal{A}_{out}$ будет эквивалентна этому конкретному состоянию на выходе и $\psi(\pi \wedge (v_h \diamond v_g)) = \top$, а, значит, состояние $\mathcal{A}_{out}$ корректно.

Аналогично, для произвольной функции конкретизации ψ для $\mathcal{A}_{out}$, если $\psi(\pi \wedge (v_h \diamond v_g)) = \top$, то для состояния $\mathcal{A}_{in}$ для той же функции заведомо $\psi(\pi) = \top$. Из точности $\mathcal{A}_{in}$ следует, что для ψ найдется эквивалентное конкретное состояние. Из эквивалентности и условия $\psi(v_h \diamond v_g) = \top$ следует, что для этого конкретного состояния выполнено условие перехода, и найдётся конкретное состояние на выходе из инструкции, которое будет эквивалентно ψ для $\mathcal{A}_{out}$. Значит, состояние $\mathcal{A}_{out}$ точно.

3.3.2 Инструкции выделения памяти

Инструкция `p = alloca(`b`)`, `p` $\in P_b$ выделяет новую абстрактную ячейку памяти. Так как было условлено, что все выделяемые ячейки в памяти не пересекаются, то в данной ситуации требуется создать новую ячейку. Пусть функция $\mathit{freshCell}(b)$ всегда возвращает новую ячейку памяти размера b. Тогда:

$$\text{ALLOCAM}\frac{\begin{array}{c} \mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{p = alloca(}b\texttt{)} \\ m = \mathit{freshCell}(b), v = \mathit{freshVar}(b) \end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{m \mapsto v\}, \mathcal{P}\{\texttt{p} \mapsto \{\langle m, \top \rangle\}\}, \mathcal{B}, \mathcal{V} \rangle}$$

Инструкция $\texttt{p = alloca(}b\texttt{, c)}$, $\texttt{p} \in P_b$, $\texttt{c} \in C_r$ выделяет новый массив из c элементов размера b. Необходимо создать новый абстрактный массив и создать для него отображение $\mathcal{B}$. Пусть функция $\mathit{freshArray}(b)$ всегда возвращает новый абстрактный массив.

$$\text{ALLOCAA}\frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{p = alloca(}b\texttt{, c)} \qquad m = freshArray(b)}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma, \mathcal{P}\{\texttt{p} \mapsto \{\langle m, \top \rangle\}\}, \mathcal{B}\{m \mapsto c\}, \mathcal{V} \rangle}$$

Корректность и точность покажем для инструкции выделения массива (для инструкции выделения новой ячейки можно провести аналогичные рассуждения).

Для произвольного конкретного состояния на входе для $\mathcal{A}_{in}$ найдется эквивалентная функция конкретизации ψ, которую для соответствующего конкретного состояния на выходе можно доопределить $\psi(m) = \mathbb{P}(\texttt{p})$, и тогда заведомо будет выполнено $\mathcal{B}(m) = c = \mathbb{B}(\psi(m))$. Т.к. вновь выделенная память не пересекается с выделенной ранее, то ψ будет также эквивалентна конкретному состоянию на выходе, из чего следует корректность $\mathcal{A}_{out}$.

Аналогично, рассмотрим произвольную функцию конкретизации ψ для $\mathcal{A}_{out}$, для которой $\psi(\pi) = \top$. Из точности $\mathcal{A}_{in}$ следует, что для ψ найдется эквивалентное конкретное состояние на входе в инструкцию. Тогда найдётся такое конкретное состояние на выходе, совпадающее с ним для всех переменных, и дополнительно $\mathbb{P}(\texttt{p}) = \psi(m)$ (т.к. $freshArray(b)$ выделяет новый массив), причём заведомо $\mathbb{B}(\mathbb{P}(\texttt{p})) = c = \mathcal{B}(m)$. Это конкретное состояние будет эквивалентно ψ для $\mathcal{A}_{out}$, откуда следует, что $\mathcal{A}_{out}$ точно.

3.3.3 Инструкции присваивания и записи значения указателя

Рассмотрим передаточную функцию для инструкции копирования указателей $\texttt{p = s}, \texttt{p}, \texttt{s} \in P_b$. После выполнения этой инструкции указатели p и s указывают на одну и ту же ячейку памяти. Напомним, что для переменных-указателей (в отличие от ячеек памяти) значения $\mathcal{P}$ всегда полностью инициализированы, т.е. $\langle d, \gamma \rangle \in \mathcal{P}(\texttt{s}) \Rightarrow d \neq \mu$.

$$\text{PASSIGN}\frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{p = s} \qquad \mathcal{P} \vdash \texttt{s} \to pt_s}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma, \mathcal{P}\{\texttt{p} \mapsto pt_s\}, \mathcal{B}, \mathcal{V} \rangle}$$

Для произвольного конкретного состояния на входе соответствующее состояние на выходе отличается только значением $\mathbb{P}(\mathtt{p}) = \mathbb{P}(\mathtt{s})$. Состоянию на входе эквивалентна некоторая функция конкретизации ψ для $\mathcal{A}_{in}$, т.е. для неё $\psi(\pi) = \top$ и найдётся пара $\langle d \neq \mu, \gamma\rangle \in pt_s \ : \ \psi(\gamma) \wedge (\psi(d) = \mathbb{P}(\mathtt{s}))$. Та же функция для конкретного состояния на выходе будет эквивалентна $\mathcal{A}_{out}$, т.к. $\langle d, \gamma\rangle \in \mathcal{P}(\mathtt{p}),\ \psi(\gamma) \wedge (\psi(d) = \mathbb{P}(\mathtt{p}))$. Таким образом, состояние $\mathcal{A}_{out}$ корректно.

Аналогично можно показать, что для произвольной функции конкретизации $\psi : \psi(\pi) = \top$ для $\mathcal{A}_{in}$ найдется конкретное состояние на входе, и соответствующее ему конкретное состояние на выходе будет эквивалентно той же функции ψ для $\mathcal{A}_{out}$, откуда следует точность состояния $\mathcal{A}_{out}$.

Следующей рассмотрим инструкцию `store(p, s)` сохранения значения указателя $\mathtt{s} \in P_b$ в ячейку памяти по адресу $\mathtt{p} \in P_r$. Пусть значение $\mathcal{P}(\mathtt{p}) = \{\langle a_i, \gamma_i\rangle | a_i \neq \mu\}$, а значение $\mathcal{P}(\mathtt{s}) = \{\langle m_k, \zeta_k\rangle | m_k \neq \mu\}$. Тогда значение $\mathcal{P}(a_i)$ для каждой из ячеек обновится на $\mathcal{P}(\mathtt{s})$ при условии γ_i и останется прежним при $\neg\gamma_i$. Тогда, если $\mathcal{P}(a_i) = \{\langle n_i^j, \lambda_i^j\rangle\}$, то новые значения $\mathcal{P}(a_i)$ представимы в виде $\{\langle n_i^j,\ \lambda_i^j \wedge \neg\gamma_i\rangle\} \cup \{\langle m_k,\ \zeta_k \wedge \gamma_i\rangle\}$[6]. Заметим, что из попарной несовместности условий ζ_k и для любого i попарной несовместности λ_i^j следует попарная несовместность условий из множества $\{\lambda_i^j \wedge \neg\gamma_i\} \cup \{\zeta_k \wedge \gamma_i\}$, что гарантирует однозначность выбора значений для $\mathcal{P}(a_i)$. Таким образом, передаточную функцию можно записать следующим образом:

$$\text{STOREP}\ \frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}\rangle \qquad\qquad q \vdash \mathtt{store(p,\ s)} \\ \mathcal{P} \vdash \mathtt{p} \to \{\langle a_i, \gamma_i\rangle\} \quad \mathcal{P} \vdash \mathtt{s} \to \{\langle m_k, \zeta_k\rangle\} \quad \mathcal{P} \vdash a_i \to \{\langle n_i^j, \lambda_i^j\rangle\}\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma, \mathcal{P}\{\forall i : a_i \mapsto \{\langle n_i^j,\ \lambda_i^j \wedge \neg\gamma_i\rangle\} \cup \{\langle m_k,\ \zeta_k \wedge \gamma_i\rangle\}\}, \mathcal{B}, \mathcal{V}\rangle}$$

Для произвольного конкретного состояния на входе существует некоторая функция конкретизации ψ, эквивалентная ему для $\mathcal{A}_{in}$. Значит, $\psi(\pi) = \top$ и найдутся такие $a \neq \mu, m \neq \mu, n \in AM, \gamma, \lambda, \zeta \in \mathcal{C}$, что

- $\psi(\gamma) = \top,\ \langle a, \gamma\rangle \in \mathcal{P}(\mathtt{p}),\ \psi(a) = \mathbb{P}(\mathtt{p})$;
- $\psi(\zeta) = \top,\ \langle m, \zeta\rangle \in \mathcal{P}(\mathtt{s}),\ \psi(m) = \mathbb{P}(\mathtt{s})$;
- $\psi(\lambda) = \top,\ \langle n, \lambda\rangle, \in \mathcal{P}(a),\ \psi(n) = \mathbb{P}(\mathbb{P}(\mathtt{p})) \vee n = \mu$.

[6] Здесь и далее под такой записью будем понимать следующее:

$$\bigcup_i \bigcup_j \{\langle n_i^j, \lambda_i^j \wedge \neg\gamma_i\rangle\} \quad \cup \quad \bigcup_i \bigcup_k \{\langle m_k, \zeta_k \wedge \gamma_i\rangle\}.$$

В соответствующем конкретном состоянии на выходе $\mathbb{P}(\mathbb{P}(\mathtt{p})) = \mathbb{P}(\mathtt{s})$. Заметим, что для всех ячеек памяти, кроме $\mathbb{P}(\mathtt{p})$, значение отображений не изменилось. Из однозначности выбора $\mathcal{P}(\mathtt{p})$ и $\mathcal{P}(\mathtt{s})$ следует, что на выходе $\forall i : \gamma_i \neq \gamma \Rightarrow \psi(\gamma_i) = \bot \Rightarrow \mathcal{P}(a_i) = \{\langle n_i^j, \lambda_i^j \rangle\}$. Значит, значение $\mathcal{P}$ для абстрактных ячеек, которым соответствуют указанные ячейки памяти, тоже не изменилось. Необходимо показать эквивалентность значения адреса в абстрактной ячейке a, т.е. что $\psi(m) = \mathbb{P}(\psi(a))$. Заметим, что $\psi(m) = \mathbb{P}(\mathtt{s}) = \mathbb{P}(\mathbb{P}(\mathtt{p})) = \mathbb{P}(\psi(a))$. Отсюда следует корректность состояния $\mathcal{A}_{out}$.

Покажем теперь точность абстрактного состояния $\mathcal{A}_{out}$. Для произвольной функции конкретизации $\psi : \psi(\pi) = \top$ найдётся соответствующее эквивалентное конкретное состояние для $\mathcal{A}_{in}$. Значит, как уже было показано, соответствующее ему конкретное состояние на выходе из инструкции будет эквивалентно ψ для $\mathcal{A}_{out}$, значит, $\mathcal{A}_{out}$ точно.

Инструкция `p = load(s)` загружает в переменную-указатель $\mathtt{p} \in P_b$ значение указателя по адресу из $\mathtt{s} \in P_r$. Некоторые из абстрактных ячеек, на которые могут указывать ячейки по адресу `s`, могут быть неизвестны (равны μ). Поэтому в первую очередь необходимо выполнить ленивую инициализацию, т.е. для всех неизвестных ячеек, на которые могут указывать ячейки по адресу `s`, создать абстрактные ячейки. Для этого введём вспомогательную функцию $\mathit{initCells} : 2^{AM \times \mathcal{C}} \to 2^{AM \times \mathcal{C}} \times 2^{AM}$, которая для некоторого значения отображения $\mathcal{P}$ возвращает полностью инициализированное значение, а также множество новых созданных ячеек:

$$\mathit{initCells}(\{a_i, \gamma_i\}) = \langle \{\langle \widetilde{a}_i, \gamma_i \rangle\}, F \rangle, \forall i : \widetilde{a}_i = \begin{cases} a_i, & \text{если} \quad a_i \neq \mu; \\ f_i = \mathit{initCell}(a_i, b) \in F & \text{иначе}, \end{cases}$$

Применим её для значения $\mathcal{P}(\mathtt{s})$:

$$\begin{aligned} \mathcal{P}(\mathtt{s}) &= \{\langle a_i, \gamma_i \rangle\}, \\ \langle pt_{a_i}, F_i \rangle &= \mathit{initCells}(\mathcal{P}(a_i)), \\ pt_{a_i} &= \{\langle m_i^j, \lambda_i^j \rangle\}. \end{aligned}$$

Множество ячеек, на которые может указывать `p` после выполнения данной инструкции, представляет собой объединение всех множеств $\{m_i^j\}$, на которые могут указывать значения из ячеек по адресу из `s`. При этом `p` будет указывать на ячейку m_i^j, если `s` указывал на a_i (что соответствует условию γ_i) и одновременно

значение в a_i указывало на m_i^j (соответствует условию λ_i^j). Таким образом, итоговое значение для $\mathcal{P}(\mathtt{p})$ равно $\widetilde{pt}_p = \{\langle m_i^j, \gamma_i \wedge \lambda_i^j \rangle\}$. Попарная несовместность γ_i и для любого i попарная несовместность λ_i^j гарантирует попарную несовместность условий из множества $\{\gamma_i \wedge \lambda_i^j\}$, чем обеспечивается однозначность выбора значений в $\widetilde{pt}_p$. Если при вычислении $\widetilde{pt}_p$ для некоторых i, j будет установлено, что $\gamma_i \wedge \lambda_i^j = \bot$, то соответствующая пара со значением m_i^j может быть исключена из итогового множества.

$$\text{LOADP}\frac{\begin{array}{cc}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle & q \vdash \mathtt{p = load(s)} \\ \mathcal{P} \vdash \mathtt{s} \to \{\langle a_i, \gamma_i \rangle\} & \mathcal{P}' = \mathcal{P}\{\mathtt{p} \mapsto \widetilde{pt}_p, \forall i : a_i \mapsto pt_{a_i}\}\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\forall f_i^j \in F_i : f_i^j \mapsto \mathit{initVar}(f_i^j, b)\}, \mathcal{P}', \mathcal{B}, \mathcal{V} \rangle}$$

Для произвольного конкретного состояния на входе существует некоторая функция конкретизации ψ, эквивалентная ему для $\mathcal{A}_{in}$. В соответствующем конкретном состоянии на выходе $\mathbb{P}(\mathtt{p}) = \mathbb{P}(\mathbb{P}(\mathtt{s}))$. Значение отображения для ячеек и переменных, отличных от p, не изменилось. В абстрактном состоянии $\mathcal{A}_{out}$, кроме значения $\mathcal{P}(\mathtt{p})$, также в результате ленивой инициализации изменились значения $\mathcal{P}(a_i)$, точность и корректность для значений адресов в этих ячейках, очевидно, сохраняется. Для $\mathcal{A}_{in}$ верно $\psi(\pi) = \top$ и найдутся такие $i, j : a_i \neq \mu, m_i^j \in AM, \gamma_i, \lambda_i^j \in \mathcal{C}$, что

- $\psi(\gamma_i) = \top$, $\langle a_i, \gamma \rangle \in \mathcal{P}(\mathtt{s})$, $\psi(a_i) = \mathbb{P}(\mathtt{s})$;
- $\psi(\lambda_i^j) = \top$, $\langle m_i^j, \lambda \rangle \in \mathcal{P}(a_i)$, $\psi(m_i^j) = \mathbb{P}(\psi(a_i))$.

Чтобы показать, что конкретное состояние на выходе эквивалентно ψ для $\mathcal{A}_{out}$, необходимо убедиться, что $\psi(m_i^j) = \mathbb{P}(\mathtt{p})$. Заметим, что для $\mathcal{A}_{out}$ верно $\psi(m_i^j) = \mathbb{P}(\psi(a_i)) = \mathbb{P}(\mathbb{P}(\mathtt{s})) = \mathbb{P}(\mathtt{p})$. Отсюда следует эквивалентность этого конкретного состояния ψ для $\mathcal{A}_{out}$, что обеспечивает корректность $\mathcal{A}_{out}$.

Пусть для произвольной функции конкретизации $\psi : \psi(\pi) = \top$ найдётся конкретное состояние на входе, эквивалентное ей для $\mathcal{A}_{in}$. Аналогично можно показать, что соответствующее ему конкретное состояние на выходе эквивалентно ψ для $\mathcal{A}_{out}$. Таким образом, доказана точность абстрактного состояния $\mathcal{A}_{out}$.

3.3.4 Инструкции присваивания, чтения и записи значения переменной

При обработке инструкции присваивания переменных `x = h` происходит копирование символьного выражения переменной или константы `h` для переменной `x`:

$$\text{VASSIGN}\frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{x = h} \qquad \sigma \vdash \texttt{h} \to v_h}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v_h\}, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle}$$

Очевидно, что если некоторая функция конкретизации ψ была эквивалентна некоторому конкретному состоянию на входе для $\mathcal{A}_{in}$, то она же будет эквивалентна соответствующему конкретному состоянию на выходе для $\mathcal{A}_{out}$, из чего следует корректность и точность абстрактного состояния $\mathcal{A}_{out}$.

Инструкция `store(p, h)`, $\texttt{p} \in P_b$, $\texttt{h} \in V_b \cup C_b$ сохраняет значение переменной или константы `h` в ячейку памяти по адресу `p`. С точки зрения изменения абстрактного состояния необходимо рассмотреть два случая: когда `p` однозначно определяет ячейку и необходимо т.н. *сильное обновление*, и когда `p` может указывать на разные ячейки в зависимости от пути до данной точки — тогда выполняется *слабое обновление*.

В первом случае (сильное обновление) символьное выражение, соответствующее `h`, ставится в соответствие единственной абстрактной ячейке, на которую всегда указывает `p`:

$$\text{STOREV-1}\frac{\begin{array}{cc}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle & q \vdash \texttt{store(p, h)} \\ \mathcal{P} \vdash \texttt{p} \to \langle a, \top \rangle & \sigma \vdash \texttt{h} \to v_h\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{a \mapsto v_h\}, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle}$$

Доказательство корректности и точности абстрактного состояния $\mathcal{A}_{out}$ аналогично предыдущему.

В случае слабого обновления, если `p` указывает на ячейку a_i при условии γ_i, то значение a_i изменится на значение `h` в результате выполнения данной инструкции, если выполнено γ_i, и останется прежним в противном случае. Для соблюдения этого правила ячейкам a_i сопоставим условные символьные выражения.

Также необходимо создать значение $\mathcal{V}$ для каждого из новых условных символьных выражений, представляющее собой объединение нового и старого значений с условием γ_i. Таким образом, передаточная функция будет иметь вид:

$$\text{STOREV-2}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{store(p, h)} \\ \mathcal{P} \vdash \texttt{p} \to \{\langle a_i, \gamma_i \rangle\} \qquad \sigma \vdash \texttt{h} \to v_h \qquad \mathcal{V} \vdash \texttt{h} \to s_h \\ \forall a_i.\ \sigma \vdash a_i \to v_i,\ \widetilde{v_i} = \mathit{ite}(\gamma_i, v_h, v_i) \\ \mathcal{V}' = \mathcal{V}\{\forall i :\ \widetilde{v_i} \mapsto \mathit{joinVals}(\widetilde{v_i},\ \langle s_h, \gamma_i \rangle,\ \langle \mathcal{V}(v_i), \neg\gamma_i \rangle)\}\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\forall a_i : a_i \mapsto \widetilde{v_i}\}, \mathcal{P}, \mathcal{B}, \mathcal{V}' \rangle}$$

Точность и корректность абстрактного состояния $\mathcal{A}_{out}$ следует из того, что если некоторое конкретное состояние на входе эквивалентно некоторой функции

конкретизации ψ для $\mathcal{A}_{in}$, то для соответствующего конкретного состояния на выходе выполнено:

$$\forall i:\ \psi(\gamma_i) = \top \Rightarrow \quad \psi(\sigma(a_i)) = \psi(\widetilde{v_i}) = \psi(v_h) = \mathbb{X}(\mathtt{h}) = \mathbb{X}(\mathbb{P}(\mathtt{p})) = \mathbb{X}(\psi(a_i)),$$
$$\forall i:\ \psi(\gamma_i) = \bot \Rightarrow \quad \psi(\sigma(a_i)) = \psi(\widetilde{v_i}) = \psi(v_i) = \mathbb{X}(\psi(a_i)).$$

Похожим образом два случая уместно рассмотреть для инструкции `x = load(p)`, $\mathtt{x} \in V_b$, `p` $\in P_b$, которая загружает значение из ячейки памяти по адресу `p` в переменную `x`. В первом случае `p` однозначно определяет ячейку, тогда символьное выражение из этой ячейки можно просто скопировать в `x`:

$$\text{LOADV-1}\ \frac{\begin{array}{cc}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}\rangle & q \vdash \mathtt{x = load(p)} \\ \mathcal{P} \vdash \mathtt{p} \to \langle a, \top\rangle\} & \sigma \vdash a \to v\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\mathtt{x} \mapsto v\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\rangle}$$

Доказательство корректности и точности абстрактного состояния $\mathcal{A}_{out}$ аналогично доказательству для инструкции присваивания битовых векторов.

Если `p` указывает на ячейку a_i при условии γ_i, то загруженное значение будет равно значению из a_i при γ_i для всех i. Для значения `x` в данном случае будет создано условное символьное выражение, и для него будет создано значение в $\mathcal{V}$. При выборе условного символьного выражения v используется тот факт, что $\forall i \neq j:\ \gamma_i \wedge \gamma_j = \bot$.

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

Для обоснования точности и корректности абстрактного состояния $\mathcal{A}_{out}$ достаточно заметить, что если некоторое конкретное состояние на входе эквивалентно некоторой функции конкретизации ψ для $\mathcal{A}_{in}$, то для соответствующего конкретного состояния на выходе найдется единственное $i : \psi(\gamma_i) = \top$, откуда следует:

$$\psi(\sigma(\mathtt{x})) = \psi(v) = \psi(v_i) = \psi(\sigma(a_i)) = \mathbb{X}(\psi(a_i)) = \mathbb{X}(\mathbb{P}(\mathtt{p})) = \mathbb{X}(\mathtt{x}).$$

3.3.5 Инструкции арифметики битовых векторов

Передаточная функция для инструкции вычисления арифметического выражения $\texttt{x = h} \circ \texttt{g}$, $\texttt{x} \in V_b$, $\texttt{h,g} \in V_b \cup C_b$ изменяет значение символьного выражения и отображения $\mathcal{V}$ для переменной $\texttt{x}$.

Введём вспомогательную функцию

$$\mathit{createArithm} : \mathit{SE} \times \mathit{Summary} \times \mathit{Summary} \times \{+, -, *, /_s, /_u\} \to \mathit{Summary},$$

$$\mathit{createArithm}(v, a, b, \circ) = \begin{cases} \varepsilon, & a = \varepsilon \text{ или } b = \varepsilon, \\ \langle v, a, b, \circ \rangle \in \mathit{Arithm} & \text{иначе}. \end{cases}$$

С её помощью запишем передаточную функцию:

$$\text{ARITHM}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \qquad q \vdash \texttt{x = h} \circ \texttt{g} \\ \circ \in \{+, -, *, /_s, /_u\} \qquad \sigma \vdash \texttt{h} \to v_h \qquad \sigma \vdash \texttt{g} \to v_g \\ \mathcal{V} \vdash v_h \to s_h \quad \mathcal{V} \vdash v_g \to s_g \quad s = \mathit{createArithm}(v_h \circ v_g, s_h, s_g, \circ)\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v_h \circ v_g\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\{v_h \circ v_g \mapsto s\}\rangle}$$

Если некоторое конкретное состояние на входе эквивалентно некоторой функции конкретизации ψ для $\mathcal{A}_{in}$, то для соответствующего конкретного состояния на выходе выполнено:

$$\psi(\sigma(x)) = \psi(v_h \circ v_g) = \psi(v_h) \circ \psi(v_g) = \mathbb{X}(\texttt{h}) \circ \mathbb{X}(\texttt{g}) = \mathbb{X}(\texttt{x}).$$

Отсюда следует корректность и точность абстрактного состояния $\mathcal{A}_{out}$.

Теперь рассмотрим инструкции приведения типа к некоторому типу размера $\texttt{e} \in B$, отличного от размера исходного типа. Их передаточные функции по сути аналогичны функции ARITHM. Функции *createZExt*, *createSExt* и *createTrunc* явно выписывать для краткости не будем, они работают аналогично *createArithm*: возвращают ε, если переданное значение для аргумента пусто, либо значения *ZExt*, *SExt* или *Trunc* соответственно в противном случае.

$$\text{ZEXT}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \quad q \vdash \texttt{x = zext(h, e)} \quad \texttt{e} > b \\ \sigma \vdash \texttt{h} \to v_h \qquad \mathcal{V} \vdash v_h \to s_h \qquad v_x = \mathit{zext}(v_h, \texttt{e})\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v_x\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\{v_x \mapsto \mathit{createZExt}(v_x, s_h, \texttt{e})\}\rangle}$$

$$\text{SEXT}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \quad q \vdash \texttt{x = sext(h, e)} \quad \texttt{e} > b \\ \sigma \vdash \texttt{h} \to v_h \qquad \mathcal{V} \vdash v_h \to s_h \qquad v_x = \mathit{sext}(v_h, \texttt{e})\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v_x\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\{v_x \mapsto \mathit{createSExt}(v_x, s_h, \texttt{e})\}\rangle}$$

$$\text{TRUNC}\frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \quad q \vdash \texttt{x = trunc(h, e)} \quad \texttt{e} < b \quad \sigma \vdash \texttt{h} \to v \quad \mathcal{V} \vdash v_h \to s_h \quad v_x = trunc(v, \texttt{e})}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v_x\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\{v_x \mapsto \mathit{createTrunc}(v_x, s_h, \texttt{e})\}\rangle}$$

Обоснования корректности и точности абстрактного состояния $\mathcal{A}_{out}$ для инструкций приведения типа полностью аналогичны доказательству для инструкции вычисления арифметического выражения.

3.3.6 Инструкции обращения к массивам

Инструкция `x = m[h]` записывает в переменную $\texttt{x} \in V_b$ значение из массива $\texttt{m} \in AArr_b$ по индексу $\texttt{h} \in V_r \cup C_r$. Как уже было отмечено, вопросы моделирования значения массивов в данной работе не рассматриваются, поэтому переменной `x` ставится в соответствие новая символьная переменная.

$$\text{ACCESS}\frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle \quad q \vdash \texttt{x = m[h]} \quad v = freshVar(b)}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma\{\texttt{x} \mapsto v\}, \mathcal{P}, \mathcal{B}, \mathcal{V}\rangle}$$

Очевидно, что при обработке данной инструкции сохраняется корректность абстрактного состояния. Однако точность абстрактного состояния $\mathcal{A}_{out}$ в общем случае не может быть гарантирована, т.к. если в эту ячейку массива ранее в ходе выполнения функции было записано некоторое значение, то в данной точке значение переменной `x` нельзя считать произвольным, т.е. для некоторых функций конкретизации может не найтись соответствующего конкретного состояния на выходе.

Инструкция `m[h] = g` не меняет абстрактное состояние анализа, поэтому передаточная функция для неё тождественна и для краткости изложения не приводится. Данная инструкция, очевидно, также не нарушает корректности и точности входного абстрактного состояния.

3.3.7 Слияние состояний

Как было описано в разделе 3.2, алгоритм 3.1 предполагает слияние входных состояний в случае, если в некоторую инструкцию входит больше одного ребра графа развёртки. Пусть для некоторой инструкции $q \in Instr$ необходимо

объединить состояния

$$\mathcal{A}_1 = \langle\, \pi_1, \sigma_1, \mathcal{P}_1, \mathcal{B}_1\,, \mathcal{V}_1\,\rangle, \quad \ldots \quad , \mathcal{A}_n = \langle\, \pi_n, \sigma_n, \mathcal{P}_n, \mathcal{B}_n\,, \mathcal{V}_n\,\rangle.$$

Объединённое состояние, которое будет использоваться как входное состояние для рассматриваемой инструкции, обозначим $\mathcal{A}_j = \langle\, \pi_j, \sigma_j, \mathcal{P}_j, \mathcal{B}_j\,, \mathcal{V}_j\,\rangle$.

Будем писать $x \in M$, если значение отображения M определено для x.

Сначала рассмотрим, как объединяются условие достижимости π и значения σ, $\mathcal{V}$ для переменных программы (см. алгоритм 3.2).

Условие достижимости для точки, в которую входит более одного ребра на графе G_k, представляет собой дизъюнкцию условий достижимости π_i с рёбер, т.к. достижимость любого из них означает достижимость этой точки:

$$\pi_j = \pi_1 \vee \pi_2 \vee \ldots \vee \pi_n.$$

Для обоснования алгоритма объединения σ потребуется следующая лемма:

***Лемма* 8.** *Для точки объединения состояний $\mathcal{A}_1, \ldots, \mathcal{A}_n$ верно:*

$$\forall 1 \leqslant i < j \leqslant n: \quad \pi_i \wedge \pi_j = \bot.$$

Доказательство. Для начала покажем, что если произвольная точка q доминируется некоторой инструкцией a : `assume h◇g`, то условие достижимости в точке q может быть записано как $\pi(q) = (\sigma_a(h) \diamond \sigma_a(g)) \wedge \gamma$, где σ_a — отображение σ в точке a. Т.к. a доминирует q, то существует как минимум один путь от a до q. Покажем, что для всех точек на любом таком пути условие достижимости может быть представлено в требуемом виде.

Доказательство можно провести по индукции по длине самого длинного пути от a. База индукции — это точка a: т.к. передаточная функция инструкции a вычисляет конъюнкцию π с условием $\sigma_a(h) \diamond \sigma_a(g)$, то для состояния после инструкции a утверждение леммы заведомо выполняется. Передаточные функции всех инструкций, за исключением инструкций $assume$, не меняют предикат π, а инструкции $assume$ добавляют к π по одному конъюнкту, и, следовательно, все передаточные функции сохраняют искомое свойство предиката π.

Осталось показать, что это свойство сохраняется и при объединении состояний. Если на пути от a до q в некоторой вершине происходит объединение состояний с нескольких входных рёбер, то инструкции, откуда выходят эти рёбра,

Алгоритм 3.2 — Алгоритм объединения значений переменных

Входные данные: $\mathcal{A}_1, \ldots \mathcal{A}_n$ — состояния на входных рёбрах

Выходные данные: $\pi_j, \sigma_j, \mathcal{V}_j$ — объединённые отображения $\sigma, \mathcal{V}$ и условие достижимости.

foreach $b \in B$ **do**
  foreach $x \in V_b$ **do**
    if $x \in \sigma_1 \wedge \ldots \wedge x \in \sigma_n$ **then**
      $v_j \leftarrow \sigma_1(x)$;
      foreach $i \in [2..n]$ **do**
        $v_j \leftarrow \mathit{ite}(\pi_i, \sigma_i(x), v_j)$;
      end
      $\sigma_j(x) \leftarrow v_j$;
      $\mathcal{V}_j(v_j) \leftarrow \mathit{joinVals}(v_j, \{\langle \mathcal{V}_i(\sigma_i(x)), \pi_i \rangle\})$;
    end
  end
  foreach $a \in AM_b$ **do**
    if $a \in \sigma_1 \vee \ldots \vee a \in \sigma_n$ **then**
      $v_j \leftarrow \varepsilon$;
      $\mathit{branchSet} \leftarrow \varnothing$;
      foreach $i \in [1..n]$ **do**
        if $a \in \sigma_i$ **then**
          $\mathit{branchSet} \leftarrow \mathit{branchSet} \cup \langle \sigma_i(a), \pi_i \rangle$;
          if $v_j = \varepsilon$ **then** $v_j \leftarrow \sigma_i(a)$;
          else $v_j \leftarrow \mathit{ite}(\pi_i, \sigma_i(a), v_j)$;
        end
      end
      $\sigma_j(a) \leftarrow v_j$;
      $\mathcal{V}_j(v_j) \leftarrow \mathit{joinVals}(v_j, \mathit{branchSet})$;
    end
  end
end

также доминируются a, и длина пути от a до данной вершины превышает длины любых путей от a до этих инструкций. Таким образом, для каждого из объединяемых состояний верно предположение индукции, т.е. их условия достижимости представимы в виде $\pi_i = (\sigma_a(h) \diamond \sigma_a(g)) \wedge \gamma_i$. Значит, предикат объединённого состояния можно записать как $\pi_j = (\sigma_a(h) \diamond \sigma_a(g)) \wedge \bigvee_i \gamma_i$, т.е. он тоже удовлетворяет искомому условию.

Теперь докажем утверждение леммы для произвольной пары объединяемых состояний. Рассмотрим инструкцию, являющуюся ближайшим общим доминатором для двух инструкций, из которых выходят объединяемые рёбра. Заметим, что у этой инструкции ровно два выходных ребра, и они ведут в инструкции `assume` с одинаковыми входными состояниями и противоположными условиями. Каждая из этих инструкции доминирует ровно одно из объединяемых рёбер (и эти рёбра различны). Получается, что для этих состояний предикаты достижимости можно записать как $\pi_i = (\sigma_a(h) \diamond \sigma_a(g)) \wedge \gamma_i$ и $\pi_j = \neg(\sigma_a(h) \diamond \sigma_a(g)) \wedge \gamma_j$, из чего следует, что они несовместны. □

Теперь перейдем к объединению значений σ. Если символьное выражение для $x \in V$ определено не на всех входящих рёбрах, то переменная x далее использоваться не будет (переменная использовалась для локальных вычислений в рамках только одной ветки), поэтому значение для неё можно не определять.

В противном случае объединённое состояние строится по аналогии с объединением значений при загрузке из памяти (правило [LOADV-2]). Для x создаётся условное символьное выражение, равное $\sigma_n(x)$, если выполнено π_n, иначе $\sigma_{n-1}(x)$, если выполнено π_{n-1}, и так далее. Корректность выбора такого выражения следует из леммы 8.

Значения отображений для ячеек памяти создаются в объединённом состоянии, если эти значения присутствуют хотя бы на одном входящем ребре. Новые значения строятся аналогично с учётом того, что не на всех входящих рёбрах может быть определено отображение σ (тогда используется вспомогательное значение ε такое, что ему заведомо не равно никакое символьное выражение). Вычисляется множество *branchSet* определённых символьных выражений с условиями, которое затем используется при создании значения $\mathcal{V}$ для новой символьной переменной.

Алгоритм 3.3 — Алгоритм объединения значений указателей

Входные данные: $\mathcal{A}_1, \ldots \mathcal{A}_n$ — состояния на входных рёбрах
Выходные данные: $\mathcal{A}_j$ — объединённое состояние
foreach $b \in B$ **do**
  foreach $p \in P_b$ **do**
    if $p \in \mathcal{P}_1 \wedge \ldots \wedge p \in \mathcal{P}_n$ **then**
      $\mathcal{P}_j(p) \leftarrow$ *mergePT*($\langle \mathcal{P}_1(p), \pi_1 \rangle, \ldots, \langle \mathcal{P}_n(p), \pi_n \rangle$) ;
    end
  end
  foreach $a \in AM^p$ **do**
    if $a \in \mathcal{P}_1 \vee \ldots \vee p \in \mathcal{P}_n$ **then**
      branchSet $\leftarrow \varnothing$;
      foreach $i \in [1..n]$ **do**
        if $a \in \mathcal{P}_i$ **then** *branchSet* $\leftarrow$ *branchSet* $\cup \langle \mathcal{P}_i(a), \pi_i \rangle$;
      end
      $\mathcal{P}_j(a) \leftarrow$ *mergePT*(*branchSet*) ;
    end
  end
end

Теперь рассмотрим алгоритм объединение отображений $\mathcal{P}$ для переменных-указателей и ячеек памяти из AM^p (см. алгоритм 3.3). Для этого будет использоваться вспомогательная функция

$$\textit{mergePT} : 2^{2^{AM \times \mathcal{C}} \times \mathcal{C}} \to 2^{AM \times \mathcal{C}}$$
$$\textit{mergePT}(\langle \{\langle a_{i_1}, \gamma_{i_1} \rangle\}, \pi_1 \rangle, \ldots, \langle \{\langle a_{i_n}, \gamma_{i_n} \rangle\}, \pi_n \rangle) =$$
$$\textit{mergeEquals}(\langle a_{i_1}, \gamma_{i_1} \wedge \pi_1 \rangle, \ldots, \langle a_{i_n}, \gamma_{i_n} \wedge \pi_n \rangle),$$

где функция *mergeEquals* принимает на вход множество пар ячеек памяти с условиями и объединяет пары с одинаковой ячейкой $\langle a, \gamma_1 \rangle, \langle a, \gamma_2 \rangle$ в одну пару $\langle a, \gamma_1 \vee \gamma_2 \rangle$.

По аналогии со значениями переменных, значения указателей $p \in P$ будут использованы далее в функции только в случае, если для них были определены значения $\mathcal{P}$, поэтому достаточно в объединённом состоянии определить

значения для тех указателей, для которых они определены на всех рёбрах. Значение на выходном ребре вычисляется с помощью функции *mergePT*, которая принимает значения на входных рёбрах с соответствующими условиями достижимости, добавляет условия с рёбер в условия для ячеек памяти, объединяет полученные множества и сливает пары с одинаковыми ячейками памяти, объединяя их условия. Лемма 8 и попарная несовместность условий $\{\gamma_{i_k}\}$ для каждого $k \in [1,n]$ гарантирует попарную несовместность условий $\bigcup_{1 \leqslant k \leqslant n} \bigcup_{i_k} \gamma_{i_k} \wedge \pi_k$. Значит, любые пары конъюнкции из этих условий будут также несовместны, что обеспечивает однозначность выбора значений в $\mathcal{P}_j$. Если условие для какой-то из ячеек оказалось несовместным, её можно исключить из итогового множества (как при обработке загрузки значения указателя).

Значения $\mathcal{P}$ для ячеек памяти из AM^p обрабатываются аналогично, но с учётом того, что не на всех входящих рёбрах может быть определено отображение $\mathcal{P}$.

Объединённое отображение $\mathcal{B}_j$ вычисляется как объединение отображений:

$$\mathcal{B}_j = \mathcal{B}_1 \cup \mathcal{B}_2 \cup \ldots \cup \mathcal{B}_n.$$

Покажем корректность и точность объединённого абстрактного состояния $\mathcal{A}_j$. Пусть известно, что все объединяемые абстрактные состояния корректны и точны. Рассмотрим произвольное конкретное состояние на входе в следующую инструкцию. Корректностью объединяемых состояний гарантируется, что для i-ого ребра, по которому было достигнуто это конкретное состояние, существует функция конкретизации $\psi : \psi(\pi_i) = \top$, эквивалентная состоянию $\mathcal{A}_i$. Отсюда следует, что $\psi(\pi_j) = \top$. Покажем, что рассматриваемое конкретное состояние эквивалентно ψ для $\mathcal{A}_j$, что будет означать корректность этого абстрактного состояния.

$$\begin{aligned}
\forall\, \mathtt{x} \in V \cap \sigma_i: &\quad \psi(\sigma_j(\mathtt{x})) = \psi(\sigma_i(\mathtt{x})) = \mathbb{X}(\mathtt{x}),\\
\forall\, a \in AM \cap \sigma_i: &\quad \psi(\sigma_j(a)) = \psi(\sigma_i(a)) = \mathbb{X}(\psi(a)),\\
\forall\, \mathtt{p} \in P \cap \mathcal{P}_i: &\quad \exists \langle d, \gamma\rangle \in \mathcal{P}_i(\mathtt{p}):\ (\psi(\gamma) = \top) \wedge (\psi(d) = \mathbb{P}(\mathtt{p})) \Rightarrow\\
&\Rightarrow \langle d, \gamma \wedge \pi_i\rangle \in \mathcal{P}_j(\mathtt{p}):\ (\psi(\gamma \wedge \pi_i) = \top) \wedge (\psi(d) = \mathbb{P}(\mathtt{p})),\\
\forall\, a \in AM^p \cap \mathcal{P}_i: &\quad \exists \langle d, \gamma\rangle \in \mathcal{P}_i(a):\ (\psi(\gamma) = \top) \wedge (\psi(d) = \mathbb{P}(\psi(a)) \vee d = \mu) \Rightarrow\\
&\Rightarrow \langle d, \gamma \wedge \pi_i\rangle \in \mathcal{P}_j(a):\ (\psi(\gamma \wedge \pi_i) = \top) \wedge (\psi(d) = \mathbb{P}(\psi(a)) \vee d = \mu),\\
\forall\, m \in AArr \cap \mathcal{B}_i: &\quad \mathcal{B}_j(m) = \mathcal{B}_i(m) = \mathbb{B}(\psi(m)).
\end{aligned}$$

Для произвольной функции конкретизации $\psi(\pi_j) = \top$ найдётся единственное $i : \psi(\pi_i) = \top$. Из точности состояния $\mathcal{A}_i$ следует, что найдётся такое конкретное состояние, эквивалентное ψ для $\mathcal{A}_i$. Проведя рассуждения, аналогичные предыдущим, можно показать, что это конкретное состояние будет эквивалентно ψ для $\mathcal{A}_j$. Таким образом, абстрактное состояние $\mathcal{A}_j$ точно.

3.4 Анализ циклов

Наличие в графе развёртки ГПУ лишь тех путей, которые проходят не более k^n раз по каждому ребру вложенности n, приводит к тому, что некоторые инструкции ГПУ могут оказаться недостижимыми. Например, для цикла с фиксированным числом итераций, превосходящим k, любые инструкции после цикла, доминируемые его заголовком, окажутся недостижимы, т.к. условия выхода из цикла на каждой из первых k итераций окажутся несовместными (равными $\bot$ для любой функции конкретизации). Следовательно, найти ошибку в ставшем таким образом недостижимым коде будет невозможно (даже если она никак не связана с предшествующим циклом).

Для разрешения этой проблемы применяется изменение семантики инструкций на последней итерации. При обработке инструкций k-ой итерации цикла все используемые значения отображения σ (присваиваемые в переменную, загружаемые в ячейку памяти и из неё, аргументы арифметических выражений, условных операторов и инструкций приведения типа, индексы в инструкциях доступа к массиву) заменяются на новые символьные переменные.

Покажем, что такое преобразование не нарушает корректности построенных состояний. Пусть при обработке некоторой инструкции последней итерации вместо символьного выражения $e \in SE_b$ была использована новая переменная $s = \mathit{freshVar}(b)$. Если для некоторого рассматриваемого анализом конкретного состояния нашлась эквивалентная ему функция конкретизации ψ для $\mathcal{A}_{out}$, то её можно доопределить следующим образом: $\psi(s) = \psi(e)$, и тогда полученная функция также будет эквивалентна этому конкретному состоянию.

К сожалению, данный подход может нарушить точность абстрактных состояний.

3.5 Корректность анализа

В данном разделе буду доказаны точность и корректность абстрактных состояний при внутрипроцедурном анализе.

***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.

Доказательство. Доказательство проведём по индукции по длине n самого длинного пути от входа в функцию до данной точки.

База Начальное состояние $\mathcal{A}_{entry}$ корректно описывает конкретное состояние на входе в функцию, т.е. предположение индукции верно для $n = 1$.

Переход Рассмотрим некоторое ребро, для которого длина самого длинного пути до входа равна n. Из предположения индукции следует, что абстрактное состояние на любом ребре, входящем в инструкцию, из которой выходит рассматриваемое ребро, корректно. В разделе 3.3 было показано, что передаточные функции для всех инструкций сохраняют корректность абстрактного состояния на входе и корректность сохраняется при объединении состояний. В разделе 3.3 было показано, что изменение семантики инструкций на последней итерации цикла также не нарушает свойство корректности. Таким образом, абстрактное состояние на данном ребре тоже корректно. □

***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.

Доказательство. Доказательство этого утверждения может быть проведено по аналогии с доказательством корректности. Начальное абстрактное состояние точно, передаточные функции всех инструкций, кроме инструкции чтения из буфера, и алгоритм объединения состояний сохраняют точность. Отсутствие последней итерации развёртки цикла на любом пути до данной точки гарантирует, что изменение семантики на k-ой итерации также не повлияет на точность абстрактного состояния. Таким образом, также по индукции по длине n самого длинного пути от входа в функцию до данной точки доказывается утверждение теоремы. □

Неточное моделирование условия достижимости точки может приводить к появлению ложных срабатываний, т.к. использование недостаточно строгого условия (не учитывающего некоторых зависимостей между переменными) в некоторых случаях не позволит обнаружить несовместность условия ошибки для этой точки, вследствие чего для неё будет выдано ложное предупреждение. Надо отметить, что условия теоремы 4 очень строги и не выполняются для значительной доли реальных функций. Однако из-за свойств определения ошибки (исследование путей, а не конкретных значений) нарушение условий теоремы хотя и приводит к получению неточных абстрактных состояний, но это нечасто становится причиной выдачи ложных срабатываний. Для появления ложного срабатывания необходимо, чтобы некоторый путь из-за неточности абстракции оказался выполнимым и ошибочным с точки зрения анализа, но был невыполнимым в реальности.

Условие отсутствия на пути инструкции чтения из массива вызвано тем, что моделирование содержимого массивов не предусмотрено рассматриваемым алгоритмом, т.к. рассмотрение методов решения этой задачи выходит за рамки данной работы. Однако это ограничение может быть ослаблено, если абстрактное состояние будет расширено с целью анализа значений в ячейках массива, и по крайней мере для некоторых ячеек удастся построить точную абстракцию их значений.

Таким образом, если на любом ребре от входа в функцию до инструкции обращения к буферу выполнены условия теорем 3 и 4, то выполнены условия теоремы 2 и достаточным условием наличия ошибки в данной инструкции будет выполнимость формулы (3.4).

3.6 Пример обнаружения ошибки

Рассмотрим предложенный подход на примере поиска ошибки в функции `bar` из раздела 2.1 (приведена на листинге 2.1). На листинге 3.1 код этой функции записан на модельном языке.

Листинг 3.1 — Функция, содержащая ошибку

```
1  a, b, ret ∈ V_32,  buf ∈ M_32
2  bar(var a, var b) {  // σ(a) = a_0, σ(b) = b_0
3    buf = alloca(32, 10)
4    if (a ⩾_s 9) {  // V(a_0) = s_1 = ⟨a_0, 9, ⩾_s⟩ ∈ Assume
5      // …
6    } else {}
7    // V(a_0) = s_2 = ⟨a_0, {⟨s_1, a_0 ⩾_s 9⟩}⟩ ∈ Join
8    if (b ≠ 0) {
9      a = a + 1   // σ(a) = a_0 + 1, V(a_0 + 1) = s_3 = ⟨a_0 + 1, s_2, 1, +⟩ ∈ Arithm
10   } else {}
11   // σ(a) = a_1, V(a_1) = s_4 = ⟨a_1, {⟨s_2, b_0 = 0⟩, ⟨s_3, b_0 ≠ 0⟩}⟩ ∈ Join
12   bar_ret = buf[a]
13 }
```

Значения отображения σ и отображения $\mathcal{V}$ для значений $\sigma(\texttt{a})$, вычисленные в результате обработки соответствующих инструкций, также приведены на листинге 3.1 (значения $\mathcal{V}$ для символьных выражений $\sigma(\texttt{b})$ опущены для краткости, т.к. переменная `b` не используется в качестве индекса буфера).

На строке 12 происходит обращение к буферу размера 10 по индексу `a`, таким образом, ошибка произойдёт в случае $\texttt{a} \geqslant_s 10$. Для проверки данной инструкции необходимо проверить на выполнимость условие

$$\mathit{NotLess}(p_{12}, a_1, 10) = \mathfrak{U}(\mathcal{V}(p_{12}, a_1), 10) = \mathfrak{U}(s_4, 10).$$

Здесь и далее запись p_l будет означать точку в программе на входе в инструкцию на строке l.

Значение a_1 получилось из слияния значений a_0 и a_0+1, что отражено в значении $s_4 = \mathcal{V}(p_{12}, a_1) \in \mathit{Join}$, поэтому будут проверены оба значения с учётом условий:

$$\mathfrak{U}(s_4, 10) = \bigvee \begin{array}{l} a_1 = a_0 \wedge \mathfrak{U}(s_2, 10) \wedge b = 0 \\ a_1 = a_0 + 1 \wedge \mathfrak{U}(s_3, 10) \wedge b \neq 0 \end{array}$$

Рассмотрим вторую (нижнюю) ветвь дизъюнкции, соответствующую ветке истинности второго условного оператора. Вывод условия для этого случая

$$\dfrac{\dfrac{\dfrac{\dfrac{9 \geqslant_s \hat{a_0}}{\mathfrak{U}(s_1, \hat{a_0})} \quad a_0 \geqslant_s 9}{\mathfrak{U}(s_2, \hat{a_0})} \quad \dfrac{\dfrac{1 \geqslant_s \hat{c_1}}{\mathfrak{U}(1, \hat{c_1})} \quad \hat{a_0} + \hat{c_1} \geqslant_s 10}{\mathfrak{U}(s_3, 10)} \quad a_1 = a_0 + 1 \quad b_0 \neq 0}{\mathfrak{U}(s_4, 10)}$$

Рисунок 3.3 — Пример вывода условия для одного из путей

изображен на рис. 3.3. Значение в этой ветви является суммой двух значений: $s_3 = \langle a_0 + 1, s_2, 1, + \rangle \in \textit{Arithm}$.

В общем случае для произвольной суммы нижние границы каждого из слагаемых неизвестны (т.к., как уже пояснялось ранее, в графе их значений могут быть узлы *Join*, а совместность условий путей еще не анализировалась), поэтому для проверки этого условия вводятся вспомогательные переменные $\hat{a_0}$ и $\hat{c_1}$. Тогда искомое условие будет иметь вид:

$$\mathfrak{U}(s_3, 10) = \exists \hat{a_0} \exists \hat{c_1}\, \mathfrak{U}(s_2, \hat{a_0}) \wedge \mathfrak{U}(1, \hat{c_1}) \wedge (\hat{a_0} + \hat{c_1} \geqslant_s 10).$$

Для значения a_0 информация о нём есть только на одном из путей первого условного оператора, поэтому:

$$\mathfrak{U}(s_2, \hat{a_0}) = \mathfrak{U}(s_1, \hat{a_0}) \wedge a_0 \geqslant_s 9.$$

Про значение a_0, пришедшее с ветки истинности первого условного оператора, точно известно, что для него условие этого оператора `a` $\geqslant_s 9$ выполнено. Очевидно, что если $9 \geqslant_s \hat{a_0}$, то заведомо $a_0 \geqslant_s \hat{a_0}$. Отсюда получаем:

$$\mathfrak{U}(s_1, \hat{a_0}) = 9 \geqslant_s \hat{a_0}.$$

Для `else`-ветки второго условного оператора значение $\mathfrak{U}(s_2, 10)$ раскрывается аналогично значению $\mathfrak{U}(s_2, \hat{a_0})$:

$$\mathfrak{U}(s_2, 10) = \mathfrak{U}(s_1, 10) \wedge a_0 \geqslant_s 9 = \mathbf{9} \geqslant_{\mathbf{s}} \mathbf{10} \wedge a_0 \geqslant_s 9 = \bot.$$

Уже на этом этапе видно, что эта ветвь дизъюнкции несовместна, и, следовательно, её можно исключить.

Развернув оставшиеся тривиальные значения, получим итоговое условие (кванторы существования удалены в результате сколемизации):

$$\begin{aligned} \mathfrak{U}(s_4, 10) = &(9 \geqslant_s \hat{a_0}) \wedge (a_0 \geqslant_s 9) \wedge (1 \geqslant_s \hat{c_1}) \wedge \\ &\wedge (\hat{a_0} + \hat{c_1} \geqslant_s 10) \wedge (a_1 = a_0 + 1) \wedge (b \neq 0) \end{aligned} \tag{3.9}$$

Получившаяся формула (3.9) является выполнимой, и в таблице 3 представлены соответствующие значения переменных.

Таблица 3 — Модель для условия ошибки (3.9)

Переменная	$\hat{a_0}$	$\hat{c_1}$	a_0	$a_0 + 1$	a_1	b_0
Значение	9	1	9	10	10	1

Чтобы получить ошибочный путь, необходимо подставить полученные значения в условия в узлах типа *Join*. В результате подстановки получим, что любой путь, для которого выполнено $(a_0 \geqslant_s 9) \wedge (b \neq 0)$, будет ошибочным. В данном случае этому условию удовлетворяет единственный путь `(3)-(4)-(5)-(7)-(8)-(9)-(11)-(12)`, и он действительно содержит ошибку доступа к буферу.

Глава 4. Поиск межпроцедурных ошибок

4.1 Метод резюме

Межпроцедурный анализ реализуется с помощью резюме. Данный подход заключается в том, что 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). При этом значениям отображений в вызываемой функции сопоставляются соответствующие значения в контексте вызывающей функции. Значения последних вычисляются на основе соответствующих значений в резюме (например, просто копируются из резюме). К преимуществам данного подхода можно отнести однократный анализ каждой функции и возможность построить контекстно-чувствительный анализ.

В нашем случае будем считать, что в резюме функции записывается объединённое абстрактное состояние из состояний на выходе из инструкций `return`. Вопросы оптимизации и сокращения размера резюме в данной работе не рассматриваются, но для отображений π, σ, $\mathcal{P}$ подробно освещены в работах [46; 70; 71]. Сокращение размера отображения $\mathcal{V}$ будет рассмотрено в разделе 7.1 при описании реализации.

Рассмотрим в общих чертах, как при вызове функции происходит применение её резюме, т.е. с помощью сохранённого абстрактного состояния вызванной функции модифицируется текущее состояние в точке вызова. Применение резюме происходит в несколько этапов:

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». Граф достраивается каждый раз при вызове

функций *initCell* и *initArr*. Кроме этого, при вызове функции *initVar* для параметров и абстрактных ячеек памяти, хранящих битовые векторы, запоминаются созданные символьные переменные.

2. При применении резюме с помощью этого графа строится функция соответствия его вершин абстрактным ячейкам памяти и массивам в контексте вызова:

$$\mathfrak{M} : P^{callee} \cup AM^{callee} \cup AArr^{callee} \to 2^{(P^{caller} \cup AM^{caller} \cup AArr^{caller}) \times \mathcal{C}^{caller}}.$$

   Это отображение строится рекурсивно, начиная с сопоставления формальных аргументов фактическим и далее продвигаясь по дугам графа памяти. Наборы абстрактных ячеек в контексте вызова, которым может соответствовать ячейка вызывающей функции, снабженных условиями, вычисляются по аналогии с передаточной функцией инструкции `p = load(s)`.
3. Также на основе информации из п. 1 символьным переменным, полученным в результате вызова функции *initVar*, 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 $\mathfrak{E} : SE^{callee} \to SE^{caller}$. Естественным образом по $\mathfrak{E}$ определяется соответствие условий, которое мы обозначим $\mathfrak{C} : \mathcal{C}^{callee} \to \mathcal{C}^{caller}$.
5. Абстрактное состояние, помещённое в резюме, используется в качестве передаточной функция 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 транслированное в контекст вызова символьное выражение, соответствующее в резюме переменной f_{ret}.
6. При сопоставлении символьных выражений мигрируют также значения $\mathcal{V}$ для этих выражений. Построение отображения $\mathfrak{V} : \mathcal{V} \to \mathcal{V}$, реализующее миграцию значений $\mathcal{V}$, будет рассмотрено в разделе 4.2.

Подробное описание алгоритма миграции можно также найти в работах [46; 70; 71][1]. В данной главе остановимся на расширении отображения $\mathcal{V}$ для межпроцедурного случая.

4.2 Ошибки с межпроцедурным вычислением индекса

Рассмотрим произвольную ошибку доступа к буферу, возникающую, к примеру, 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. Для начала рассмотрим первый случай, пример такой ситуации приведён в листинге 4.1.

Функция `access_1` содержит межпроцедурную ошибку доступа к буферу, так как можно привести ошибочный путь (9)-(10)-*(1)-(2)-(3)-(5)-(6)*-(11)-(12). Точку доступа к буферу на строке 12 обозначим p_{12}. Обнаружить такую ошибку можно, используя резюме. Значения отображения σ для интересующих нас переменных выписаны на листинге. Из формулы (3.4), а также из того, что в точке p_{12} выполнено $\pi \equiv \top$, следует, что достаточным условием ошибки будет являться формула $\mathit{NotLess}(p_{12}, i_1, 11)$.

Индексом, по которому происходит доступ на строке 12, является возвращаемое значение функции `inc`. Таким образом,

$$\mathit{NotLess}(p_{12}, i_1, 11) = \mathit{NotLess}(p_6, r, 11).$$

[1] Отметим, что вопросы моделирования памяти и организации анализа указателей не являются целью данной работы. В частности, предлагаемый подход также может применяться совместно с другими методами анализа указателей, помимо перечисленных, например, с межпроцедурным анализом из [72].

Листинг 4.1 — Ошибка с межпроцедурным вычислением индекса

```
1  inc(var x) {          // σ(x) = x_0
2    if (x ⩾_s 10) {
3      x = 10            // σ(x) = 10
4    } else {}
5                        // σ(x) = x_1
6    inc_ret = x + 1     // σ(inc_ret) = r
7  }
8
9  access_1(var a) { // σ(a) = a_0
10   buf11 = alloca(32, 11)
11   i = inc(a)        // σ(i) = i_1
12   access_1_ret = buf11[i]
13 }
14
15 access_2(){
16   buf5 = alloca(32, 5)
17   i = inc(4)        // σ(i) = i_2
18   access_2_ret = buf5[i]
19 }
```

В ходе анализа функции `inc` было установлено, что $\mathcal{V}(p_6, r) = s_3$. Это значение можно представить в виде графа, изображённого на рис. 4.1.

Исходя из этого, можно построить достаточное условие $\mathit{NotLess}(p_6, r, 11)$:

$$\mathit{NotLess}(p_6, r, 11) = \mathfrak{U}(s_3, 11).$$

Таким образом, для вычисления достаточного условия ошибки в вызывающей функции необходимо поместить в резюме значение атрибута $\mathcal{V}(p_6, r)$. При

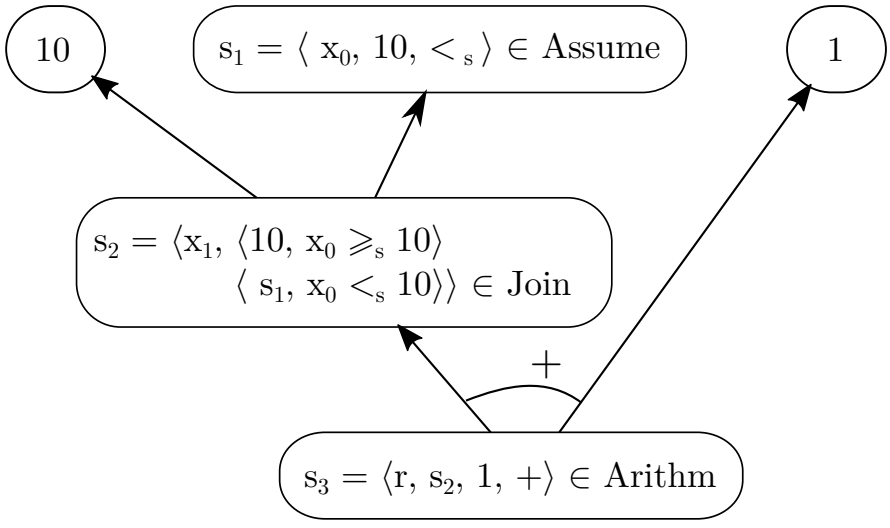

Рисунок 4.1 — Граф значения $s_3 = \mathcal{V}(p_6, r)$

применении резюме следует сопоставить формальные и фактические аргументы, результат вызова и возвращаемое значение и т.п., например, символьной переменной x_0 будет сопоставлено символьное выражение, соответствующее `a` в контексте вызывающей функции (см. раздел 4.1). По этому правилу будет вычислено значение $\mathcal{V}(p_{12}, i_1)$ путём последовательной миграции узлов дерева $s_3 = \mathcal{V}(p_6, r)$ от листьев к корню. Таким образом, достаточное условие ошибки можно представить как

$$\mathit{NotLess}(p_{12}, i_1, 11) = \mathfrak{U}(\mathcal{V}(p_6, r), 11) =$$
$$= \exists\, \hat{x_1}\, \exists\, \hat{c_1} \,:\, (x_1 = 10 \,\wedge x_0 \geqslant_s 10 \,\wedge\, 10 \geqslant_s \hat{x_1}) \wedge (1 \geqslant_s \hat{c_1}) \wedge (\hat{x_1} + \hat{c_1} \geqslant_s 11).$$

Данная формула выполнима при следующих значениях: $\hat{c_1} = 1$, $x_0 = 20$, $x_1 = 10$, $\hat{x_1} = 10$, $r = 11$, $a_0 = 20$, следовательно, необходимо выдать предупреждение об ошибке.

К сожалению, данный подход сам по себе не позволит обнаружить ошибку, происходящую в функции `access_2` из листинга 4.1. Так как анализ функций производится снизу вверх, то при анализе функции `inc` не было ничего известно о возможных значениях параметра `x`. Поэтому информация о возвращаемом значении, вычисляемом из параметра по пути (1)-(2)-(4)-(5)-(6), отсутствует в значении $\mathcal{V}(p_6, r)$.

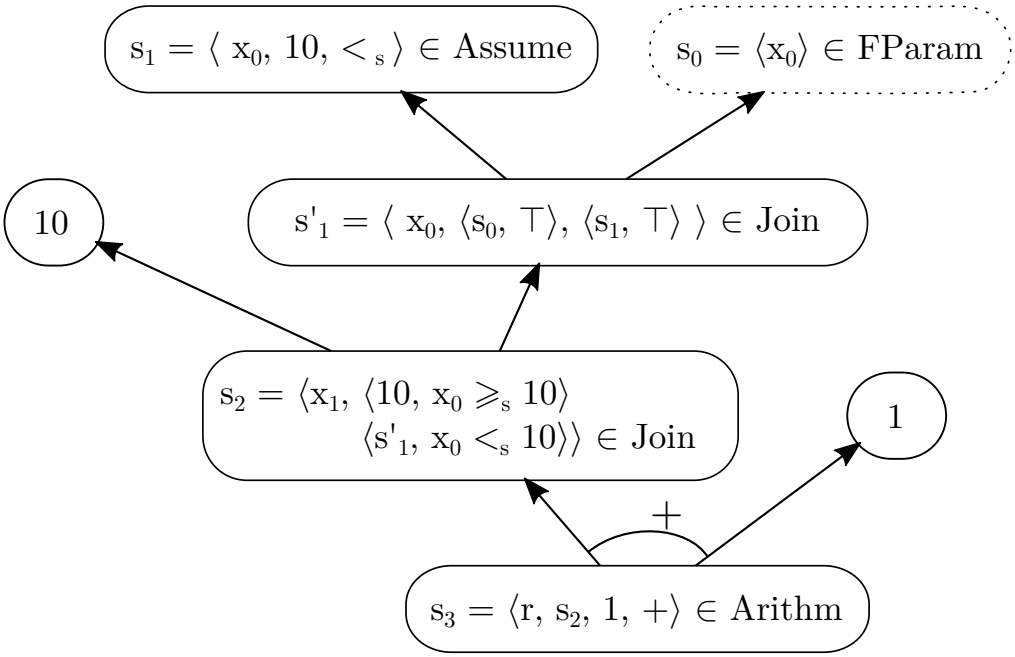

Рисунок 4.2 — Значение $\mathcal{V}(p_6, r)$

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. В результате значение $\mathcal{V}(p_6, r)$ будет иметь вид, изображённый на рис. 4.2.

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 $s_v = \langle \mathfrak{E}(v), s^{actual} \mid v \in \mathit{SE}, s^{actual} \in \mathit{Summary} \rangle \in \mathit{AParam}$. Так как в точке вызова $\mathcal{V}(p_{17}, 4) = 4$ и $\mathfrak{E}(x_0) = 4$, то вместо $s_0 = \langle x_0 \rangle \in \mathit{FParam}$ будет подставлено значение $s_4 = \langle 4, 4 \rangle \in \mathit{AParam}$. В результате значение $\mathcal{V}(p_{18}, i_2) = s_8$ будет иметь вид, изображённый на рис. 4.3.

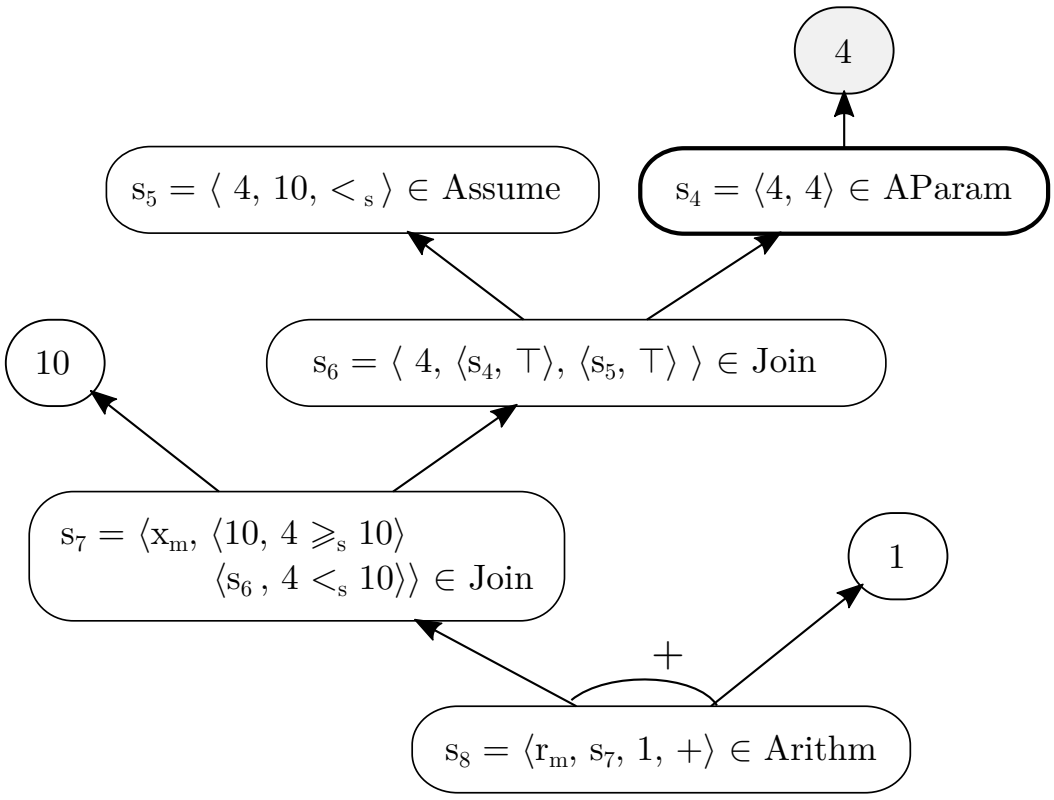

Рисунок 4.3 — Значение $\mathcal{V}(p_{18}, i_2)$

Во время анализа процедуры никакой априорной информации о значениях её параметров нет, поэтому

$$s_v = \langle v \rangle \in \mathit{FParam} \quad \Rightarrow \quad \mathfrak{U}(s_v, x) \doteq \bot \; \mathfrak{L}(s_v, x) \doteq \bot.$$

Для значений, обозначающих фактические параметры, выполнено

$$s_v = \langle v, s^{actual} \rangle \in \mathit{AParam} \Rightarrow \begin{array}{l} \mathfrak{U}(s_v, x) \doteq (v = w) \ \wedge \ \mathfrak{U}(s^{actual}, x), \\ \mathfrak{L}(s_v, x) \doteq (v = w) \ \wedge \ \mathfrak{L}(s^{actual}, x), \end{array}$$

где $w \in \mathit{SE}$ — первый элемент s^{actual}.

Миграция значений *Summary* выполняется с помощью отображения $\mathfrak{V}$, определённого по формуле (4.1). Оно сопоставляет нетривиальное значение выражению s_v, только если символьному выражению v, определённому в контексте вызванной функции, найдено соответствующее выражение $\mathfrak{E}(v)$ в контексте вызывающей функции. Далее при анализе вызывающей функции в качестве значения $\mathcal{V}(\mathfrak{E}(v))$ используется $\mathfrak{V}(s_v)$.

$$\mathfrak{V}(s) \doteq \begin{cases} \varepsilon, & s = \varepsilon \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_{cmp}), \diamond \rangle \in \mathit{Assume}, & s = \langle v, s_{comp}, \diamond \rangle \in \mathit{Assume} \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_a), \mathfrak{V}(s_b), \circ \rangle \in \mathit{Arithm}, & s = \langle v, s_a, s_b, \circ \rangle \in \mathit{Arithm} \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_a) \rangle \in \mathit{ZExt}, & s = \langle v, s_a \rangle \in \mathit{ZExt} \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_a) \rangle \in \mathit{SExt}, & s = \langle v, s_a \rangle \in \mathit{SExt} \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_a), w \rangle \in \mathit{Trunc}, & s = \langle v, s_a, w \rangle \in \mathit{Trunc} \\ \langle \mathfrak{E}(j), \ \{\langle \mathfrak{V}(s_{v_i}), \mathfrak{C}(\gamma_i) \}\rangle \in \mathit{Join}, & s = \langle j, \ \{\langle s_{v_i}, \gamma_i \rangle\}\rangle \in \mathit{Join} \\ \varepsilon, & s = \langle v \rangle \in \mathit{FParam}, \mathcal{V}(\mathfrak{E}(v)) = \varepsilon \\ \langle \mathfrak{E}(v), s_a \rangle \in \mathit{AParam}, & s = \langle v \rangle \in \mathit{FParam}, \mathcal{V}(\mathfrak{E}(v)) = s_a \neq \varepsilon \\ \langle \mathfrak{E}(v), \mathfrak{V}(s_a) \rangle \in \mathit{AParam}, & s = \langle v, s_a \rangle \in \mathit{AParam} \end{cases} \tag{4.1}$$

Константы не нуждаются в миграции, остальные значения *Summary* мигрируют рекурсивно с помощью отображения $\mathfrak{E}$ (см. (4.1)). Единственная нетривиальная процедура выполняется при миграции значения *FParam*, этот механизм уже был описан 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.*

Доказательство. Доказательство проведём по индукции по высоте ациклического графа вызовов.

База Так как при межпроцедурном анализе граф вызовов был приведён к ациклическому виду, то можно в качестве базы индукции выбрать функцию, которая

не вызывает других функций. Раз при её анализе не происходило применение резюме функции, то для неё напрямую применима теорема 2.

Переход Предположим, что для всех функций с высотой графа вызовов не больше n выполнено утверждение теоремы. Докажем его для некоторой функции f с высотой графа вызовов $n+1$. Произведём встраивание всех функций $\{g_i\}$ в тело f ($f \notin \{g_i\}$, т.к. граф вызовов ациклический). Полученная функция $\widetilde{f}$ будет иметь высоту графа вызовов n, поэтому для неё выполнено предположение индукции. Покажем, что каждому значению $\mathcal{V}$ в функции f соответствует аналогичное значение в $\widetilde{f}$. Таким образом, если для всех значений $\widetilde{f}$ верно утверждение теоремы, то и для f оно тоже верно.

В каждой функции g_i для всех параметров и значений памяти, заданных в вызывающей функции, были созданы значения из *FParam*. При миграции некоторые из них будут заменены на значения *AParam*, если для соответствующих символьных выражений в f было определено непустое значение $s_{actual} \in \mathcal{V}$. Таким образом, каждому из этих значений $\langle v, s_{actual} \rangle \in$ *AParam* из функции f можно сопоставить значение s_{actual} в функции $\widetilde{f}$.

Далее заметим, что если в двух различных точках g_i некоторой переменной было сопоставлено одно и то же символьное выражение, то в функции $\widetilde{f}$ соответствующей переменной в тех же точках будет также соответствовать одинаковое символьное выражение[2]. Таким образом, если в g_i для аргументов некоторой инструкции `assume`, `arithm`, `zext`, `sext`, `trunc`, `call` либо для веток в точке слияния были определены непустые значения $\mathcal{V}$ и, следовательно, было создано значение $s \in \mathcal{V}$ для результата инструкции, то для соответствующей инструкции $\widetilde{f}$ значения $\mathcal{V}$ для операндов также были непусты, а, значит, было создано аналогичное значение $t \in \mathcal{V}$.

После миграции значений s в f полученное значение $\mathfrak{V}(s)$ будет содержаться в t (здесь имеется в виду, что либо $\mathfrak{V}(s) = t$, либо $t \in$ *Join* и одна из веток t содержит значение $\mathfrak{V}(s)$). Таким образом, если значения $\mathcal{V}$ в $\widetilde{f}$ корректны, то и значения в f также корректны, и выполнено утверждение теоремы. □

[2]Обратное, однако, не всегда верно, т.к. если два фактических аргумента функции g_i одинаковы, то в $\widetilde{f}$ они будут иметь одинаковые символьные выражения, но в g_i соответствующим формальным аргументам будут сопоставлены разные символьные переменные.

4.3 Переполнение буфера, переданного как параметр функции

В предыдущих разделах были рассмотрены только ситуации, когда инструкция выделения буфера находится в той же функции, что и инструкция доступа. 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. Для этого было введено два типа факта доступа к буферу внутри функции, сохраняемых в резюме для последующего анализа в вызывающей функции:

$$\begin{aligned} \mathit{KnownBufferAccess} \doteq \{\langle s_i, \tau, \mathit{bufferSizes}\rangle \mid s_i \in \mathit{Summary}, \\ \tau \in \mathcal{C},\ \mathit{bufferSizes} = \{\langle n, \lambda\rangle \mid n \in C, \lambda \in \mathcal{C}\}\ \}, \\ \mathit{UnknownBufferAccess} \doteq \{\langle s_i, \tau, \mathit{bufferVars}\rangle \mid s_i \in \mathit{Summary}, \\ \tau \in \mathcal{C},\ \mathit{bufferVars} = \{\langle m, \lambda\rangle \mid m \in \mathit{AArr}, \lambda \in \mathcal{C}\}\ \}. \end{aligned}$$

В данном разделе для краткости изложения будем рассматривать только ошибку выхода за правую границу буфера, т.к. для обнаружения выхода за левую границу не обязательно знать размер этого буфера и, следовательно, нет необходимости анализировать инструкции выделения памяти, которые могут быть расположены в другой функции.

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 с помощью рассматриваемого подхода невозможно. Предположим, что значение атрибута $\mathcal{V}$ определено для индекса 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* — слияние значений с условиями, 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.

Обозначим $\mathcal{K}$ — множество абстрактных массивов, для которых известен размер $\mathcal{B}(m) \in C$ в данной точке. Аналогично, $\mathcal{U} \doteq \{m \mid m \in AArr, m \notin \mathcal{B}\}$ — множество абстрактных массивов, для которых неизвестен размер в этом абстрактном состоянии. Т.к. для всех массивов, к которым можно обращаться в данной точке, уже должна быть выделена память с помощью инструкции `alloca`, то $\mathcal{U}$ — это массивы, чей адрес передан в данную функцию в качестве одного из параметров *ArrArgs*.

Если в s_i есть непомеченные вершины, то необходимо проверить переполнение в данной точке в ходе анализа текущей функции с помощью формулы (3.4). 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 $\mathcal{K} \neq \varnothing$, то в резюме записывается факт доступа к буферу известного размера

$$\langle\ s_i,\ \pi(p_{access}),\ \{\langle n, \lambda\rangle \mid \exists\, m : \langle m, \lambda\rangle \in \mathcal{P}(p), \mathcal{B}(m) = n\}\ \rangle \in \textit{KnownBufferAccess}.$$

Если $\mathcal{U} \neq \varnothing$, то для некоторых массивов, к которым может происходить обращение в данной точке, размер известен только в вызывающей функции (адрес буфера передан в качестве параметра), и в резюме записывается факт доступа к буферу неизвестного размера

$$\langle\ s_i,\ \pi(p_{access}),\ \{\langle m, \lambda\rangle \mid \exists\, m : \langle m, \lambda\rangle \in \mathcal{P}(p), m \notin \mathcal{B}\}\ \rangle \in \textit{UnknownBufferAccess}.$$

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 $\langle s_i, \tau, \textit{bufferSizes}\rangle \in \textit{KnownBufferAccess}$. Тогда, если в s_{actual} есть непомеченные вершины, то необходимо проверить наличие ошибки в данной точке, установив выполнимость формулы

$$\pi(p_{call}) \wedge \mathfrak{C}(\tau) \wedge \bigvee_{\langle n,\lambda\rangle\ \in\ \textit{bufferSizes}} \mathfrak{U}(s_{actual}, n) \wedge \mathfrak{C}(\lambda). \tag{4.2}$$

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

$$\langle\ s_{actual},\ \pi(p_{call}) \wedge \mathfrak{C}(\tau),\ \{\langle n, \mathfrak{C}(\lambda)\rangle \mid \langle n, \lambda\rangle \in \textit{bufferSizes}\}\ \rangle \in \textit{KnownBufferAccess}.$$

Если в резюме записан факт доступа к буферу неизвестного размера $\langle s_i, \tau, \mathit{bufferVars}\rangle \in \mathit{UnknownBufferAccess}$ и в s_{actual} есть непомеченные вершины, то проверяется выполнимость следующей формулы:

$$\pi(p_{\mathit{call}}) \wedge \mathfrak{C}(\tau) \wedge \bigvee_{\langle m,\lambda\rangle \,\in\, \mathit{bufferVars}} \; \bigvee_{\langle a,\zeta\rangle \,\in \mathfrak{M}(m),\; a\in\mathcal{B}} \zeta \wedge \mathfrak{U}(s_{\mathit{actual}}, \mathcal{B}(a)) \wedge \mathfrak{C}(\lambda). \quad (4.3)$$

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.

$$\mathit{newBS} = \{\langle \mathcal{B}(a),\ \zeta \wedge \mathfrak{C}(\lambda)\rangle \mid a \in \mathcal{B},\ \exists\, m : \langle m, \lambda\rangle \in \mathit{bufferVars},\ \langle a, \zeta\rangle \in \mathfrak{M}(m)\}$$
$$\langle\ s_{\mathit{actual}},\ \pi(p_{\mathit{call}}) \wedge \mathfrak{C}(\tau),\ \mathit{newBS}\ \rangle \in \mathit{KnownBufferAccess},$$
$$\mathit{newBV} = \{\langle a,\ \zeta \wedge \mathfrak{C}(\lambda)\rangle \mid a \notin \mathcal{B},\ \exists\, m : \langle m, \lambda\rangle \in \mathit{bufferVars},\ \langle a, \zeta\rangle \in \mathfrak{M}(m)\}$$
$$\langle\ s_{\mathit{actual}},\ \pi(p_{\mathit{call}}) \wedge \mathfrak{C}(\tau),\ \mathit{newBV}\ \rangle \in \mathit{UnknownBufferAccess}.$$

Сформулируем и докажем теорему о корректности построенных условий.

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

1. *в резюме функции g содержится факт доступа к буферу неизвестного размера $ac = \langle s_i, \tau, \mathit{bufferVars}\rangle \in \mathit{UnknownBufferAccess}$,*
2. *выполнима формула (4.3),*
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.

Доказательство. Т.к. формула (4.3) выполнима, то существует функция конкретизации $\psi : \psi(\pi(p_{\mathit{callG}}) \wedge \mathfrak{C}^g(\tau)) = \top$, и для некоторых $\langle m, \lambda\rangle \in \mathit{bufferVars}$ найдутся такие $\langle a, \zeta\rangle \in \mathfrak{M}^g(m),\ a \in \mathcal{B}$, что выполнено $\psi(\mathfrak{U}(\mathfrak{V}(s_i), \mathcal{B}(a))) = \top$, $\psi(\mathfrak{C}^g(\lambda)) = \top, \psi(\zeta) = \top$.[3].

Покажем, что существует конкретное выполнение, проходящее через инструкцию вызова p_{callG} и точку доступа p_{access}, эквивалентное ψ во всех точках

[3]Здесь и далее обозначение $\mathfrak{C}^g$ означает функцию миграции $\mathfrak{C}$ для резюме функции g (и аналогично для других отображений).

от входа в f до p_{callG}, и на соответствующем пути буфер, факт доступа к которому содержится в *ac*, равен a. Тогда из теоремы 5 будет следовать, что условие $\psi(\mathfrak{U}(\mathfrak{V}(s_i)), \mathcal{B}(a)) = \top$ гарантирует, что на данном пути в условиях настоящей теоремы всегда выполнено $\mathtt{i} \geqslant_s \mathcal{B}(a)$. Таким образом, в точке вызова p_{callG} будет доказано наличие ошибки переполнения буфера.

Докажем это утверждение по индукции по n — количеству миграций факта ac.

База В качестве базы индукции рассмотрим случай $n = 1$, когда факт доступа *ac* создан в результате анализа инструкции p_{access} : `p[i] = h` в функции g.

Условие $\pi(p_{callG})$ является условием достижимости точки p_{callG} от входа в функцию f, условие $\mathfrak{C}^g(\tau)$ является условием достижимости точки p_{access} от точки вызова p_{callG}. Так как абстрактные состояния p_{callG} и p_{access} точны, то для f найдётся такое эквивалентное ψ конкретное выполнение, что функция g будет вызвана в точке p_{callG}, и выполнение в ней пройдёт через точку p_{access}. По построению факта *ac* выполнено $\exists\, m : \langle m, \lambda \rangle \in \mathcal{P}(p)$ и $\psi(\mathfrak{C}^g(\lambda)) = \top$. Отсюда и из корректности абстрактных состояний следует, что доступ в функции g на этом пути осуществляется к массиву, переданному в g в качестве параметра m. Так как выполнено $\psi(\zeta) = \top$, то в точке p_{callG} в качестве параметра m передаётся массив a. Таким образом, на этом пути доступ к буферу, зафиксированный в *ac*, осуществляется к буферу a.

Переход Пусть факт доступа *ac* получен в результате n-ой миграции из резюме функции h в точке вызова p_{callH} факта доступа к буферу неизвестного размера $ac_h = \langle\, s_i^h,\ \tau_h,\ \{\langle d, \gamma \rangle\}\, \rangle \in \mathit{UnknownBufferAccess}$. Тогда $\tau = \pi(p_{callH}) \wedge \mathfrak{C}^h(\tau_h)$ и $\lambda = \xi \wedge \mathfrak{C}^h(\gamma)$, $\langle m, \xi \rangle \in \mathfrak{M}^h(d)$. Для функции g можно по ψ определить функцию конкретизации $\psi_h : \forall\, v \in SE : \psi_h(v) = \psi(\mathfrak{E}(v))$. Тогда $\psi_h(\pi(p_{callH}) \wedge \mathfrak{C}^h(\tau_h)) = \top$, $\psi_h(\mathfrak{C}^h(\tau_h)) = \top$. Значит, для точки p_{callH} выполнены условия текущей теоремы, при этом факт ac_h мигрировался меньше n раз, следовательно, для него верно предположение индукции и найдётся такое эквивалентное ψ_h на всех точках от входа в g до p_{callH} конкретное выполнение, что на соответствующем пути буфер, к которому производился доступ, равен m. Из условия $\psi(p_{callG}) = \top$ следует, что это конкретное выполнение пройдет через точку p_{callG}, причём из $\psi(\zeta) = \top$ следует, что в точке p_{callG} в качестве параметра m передаётся массив a. Значит, на этом пути будет осуществлен доступ к буферу a. □

Проведя аналогичные по структуре рассуждения, может быть доказана и

***Теорема* 7.** *Пусть функция f содержит инструкцию $p_{callG} \in Instr$ вызова функции g. Если*

1. *в резюме функции g содержится факт доступа к буферу известного размера $ac \in UnknownBufferAccess$,*
2. *выполнима формула* (4.2)*,*
3. *в любой точке на любом пути от входа в функцию f до 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.

Если дополнительно предположить, что функция, содержащая доступ к буферу, не содержала ошибки, то заведомо множество критических рёбер включает точку доступа и хотя бы одну точку в одной из функций выше по графу вызовов, и тогда данная ошибка удовлетворяет определению 4, т.е. является межпроцедурной.

Теперь, когда полностью определены все этапы алгоритма анализа, становится очевидной следующая теорема:

***Теорема* 8.** *Рассмотренный алгоритм анализа программы для поиска переполнения буфера завершается для любой программы.*

Доказательство. Так как метод резюме подразумевает анализ каждой функции один раз, то достаточно показать сходимость внутрипроцедурного алгоритма, включая алгоритмы создания и применения резюме.

Внутрипроцедурный анализ производится над развёрткой G_k графа потока управления G, поэтому каждая инструкция обрабатывается n^k раз, где n — вложенность цикла, в котором находится инструкция. Из этого следует конечное число разыменований каждого указателя и, следовательно, конечное число абстрактных ячеек памяти, что в свою очередь означает конечное число созданных символьных выражений. Значит, множество созданных значений *Summary* тоже конечно. Таким образом, процесс копирования абстрактного состояния в резюме всегда завершается.

Сходимость алгоритма применения резюме для компонент $\pi, \sigma, \mathcal{P}$ также следует из конечности множеств абстрактных ячеек и символьных выражений.

Чтобы убедиться, что заданный формулами 4.1 алгоритм трансляции значений *Summary* также завершается, достаточно заметить, что любое из значений *Summary* всегда представляет собой ориентированный ациклический граф, и, следовательно, не существует бесконечного пути в таком графе, по которому мог бы бесконечно двигаться рекурсивный обход. □

4.3.1 Переполнение при использовании библиотечных функций

Описанный выше подход может быть применён для анализа корректности использования библиотечных функций. Традиционным подходом к анализу вызовов библиотечных функций является использование написанных вручную спецификаций (от пользователя этого не требуется, спецификации библиотечных функций являются частью анализатора). Спецификации представляют собой краткое описание интересующего анализ особенностей поведения функции в терминах, удобных для самого анализа. При вызове специфицированной функции её спецификация применяется так же, как резюме для пользовательской функции.

В нашем случае нас будет интересовать факт доступа внутри библиотечной функции к массиву, переданному в качестве параметра, с помощью индекса, также переданного как параметр. Например, таким свойством обладают функции `memcpy` и `strncpy`. Для них (и других функций с аналогичным поведением) в спецификации будет создан факт доступа к буферу. Например, для функции

```
void *memset(void *s, int c, size_t n);
```

будет создан факт доступа к буферу неизвестного размера

$$\langle \mathcal{V}(\sigma(\mathtt{n}) - 1),\ \top,\ \{\langle \mathtt{s}, \top \rangle\} \rangle \in \mathit{UnknownBufferAccess}.$$

Заметим, что в качестве индекса используется значение $\mathtt{n} - 1$, так как это наибольший индекс, по которому произойдет доступ к массиву `s` в функции `memset`.

Таким образом, задача анализа переполнения при использования библиотечных функций сводится к обычной задаче межпроцедурного анализа.

Глава 5. Расширения базового алгоритма

В данной главе будут рассмотрены расширения базового алгоритма для поиска переполнения буфера при обработке строк (5.1), данных из недоверенного источника (5.2) и в циклах (5.3).

5.1 Переполнение при работе со строками языка C

Важным случаем ошибки переполнения буфера является переполнение при работе со строками. Строка представляет собой массив символов, концом строки считается позиция ближайшего к началу нулевого элемента. Особенностью таких ошибок переполнения является тот факт, что работа со строками в языке C преимущественно осуществляется с помощью специальных 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 буфера при работе со строками расширим модельный язык. Для краткости изложения при формальном описании алгоритмов будем считать, что один символ всегда кодируется одним байтом (вопросы анализа строк с широкими символами будут рассмотрены в разделе 5.1.4). Пусть имеется множество строковых литералов, обозначим его как *ConstString*. Строками будем считать элементы множества $String = A_8 \cup ConstString$ — массивы из однобайтовых элементов и строковые литералы. Последние можно использовать только в качестве аргументов функции:

$$\texttt{x} \in V_b, \quad \texttt{p} \in P_b, \quad \texttt{h} \in V_b \cup C_b, \quad \texttt{m} \in A_b,$$
$$\texttt{s} \in A_8, \quad \texttt{t} \in \mathit{String}.$$

⟨*FParam*⟩ ::= `var x|ptr p|arr m|`**`string s`**

⟨*AParam*⟩ ::= `h|p|m|`**`t`**

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.

5.1.1 Расширение абстрактного состояния для работы со строками

Содержимое каждой строки в абстрактном состоянии будет представлено одним символьным выражением, означающем длину данной строки (будем считать, что значениями этих выражений могут быть только положительные знаковые значения C_r). 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

$$\mathcal{S} : \mathit{String} \to \mathit{SE}_r.$$

Будем считать, что для строковых литералов длина известна и является константой:

$$\mathcal{S}|_{\mathit{ConstString}} : \mathit{ConstString} \to C_r.$$

Рассмотрим отображение $\mathcal{S}_{\mathit{entry}}$, являющееся инициализацией отображения $\mathcal{S}$ в начальном состоянии. Обозначим множество формальных входных параметров функции, являющихся строками, как $\mathit{SArgs} \in A_8$. В начальном состоянии длина каждой строки-параметра обозначается новой символьной переменной:

$$\forall s \in \mathit{SArgs} : \quad \mathcal{S}_{\mathit{entry}}(s) = \mathit{initVar}(r).$$

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 5, добавим также условие эквивалентности длин строк:

$$\forall s \in \mathit{String}: \quad \forall i \in C_r : \bigwedge \begin{array}{ll} i = \psi(\mathcal{S}(s)) & \Rightarrow \mathbb{X}(\langle \mathbb{P}(\mathtt{p}), i \rangle) = 0 \\ i <_s \psi(\mathcal{S}(s)) & \Rightarrow \mathbb{X}(\langle \mathbb{P}(\mathtt{p}), i \rangle) \neq 0 \end{array}$$

Модификации передаточных функций анализа будут рассмотрены в разделе 5.1.3.

5.1.2 Расширение отображения $\mathcal{V}$ для поиска ошибок переполнения при работе со строками

Для обнаружения переполнения буфера при вызове библиотечных функций работы со строками доопределим отображение $\mathcal{V}$ для символьных выражений, соответствующих длинам строк. Т.к. было условлено, что длина любого строкового литерала является константой, а на множестве констант отображение $\mathcal{V}$ тождественно, то расширение $\mathcal{V}$ для строковых литералов тривиально.

Как будет показано в следующем разделе 5.1.3, 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 будет использован новый тип элементов $\mathit{Min} \subset \mathit{Summary}$:

$$\mathit{Min} = \{\langle l, s_{str}, s_n \rangle \quad | \quad l \in \mathit{SE}, \quad s_{str}, s_n \in \mathit{Summary}\}.$$

Значение данного типа $\langle l, s_{str}, s_n \rangle \in \mathit{Min}$ будет сопоставлено в точке q, например, длине строки `dst`, полученной в результате вызова функции `strncpy(dst, src, n)`, при условиях $\mathcal{V}(q, \mathcal{S}(\sigma(\mathtt{src}))) = s_{str}$ и $\mathcal{V}(q, \sigma(\mathtt{n})) = s_n$. Очевидна следующая лемма:

Лемма 9. *Для значений $\langle l, s_{str}, s_n \rangle \in \mathit{Min}$ в качестве условий $\mathfrak{U}, \mathfrak{L}$ могут быть выбраны следующие формулы:*

$$\mathfrak{U}(l, x) = \mathfrak{U}(s_{str}, x) \wedge \mathfrak{U}(s_n, x),$$
$$\mathfrak{L}(l, x) = \mathfrak{L}(s_{str}, x) \vee \mathfrak{L}(s_n, x).$$

Введём вспомогательную функцию *createMin*:

$$createMin : SE \times Summary \times Summary \to Summary,$$

$$createMin(l, s_{str}, s_n) = \begin{cases} \varepsilon, & s_{str} = \varepsilon \ \vee \ s_n = \varepsilon, \\ \langle l, s_{str}, s_n \rangle \in Min, & s_{str} \neq \varepsilon \ \wedge \ s_n \neq \varepsilon. \end{cases}$$

5.1.3 Расширение передаточных функций анализа для поддержки строк

Работа со строками в модельном языке осуществляется с помощью вызова функций обработки строк либо с помощью инструкций работы с массивами. Таким образом, именно для этих операторов следует переопределить передаточные функции для поддержки операций со строками. Как и ранее, при обсуждении корректности и точности абстрактного состояния на выходе будем предполагать, что абстрактное состояние на входе в функцию было корректно и точно.

Первой переопределим передаточную функцию инструкции выделения массива для случая, когда размер элемента равен размеру символа.

$$\text{ALLOCAS}\ \frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \qquad q \vdash \texttt{p = alloca(8, c)} \qquad m = freshArray(8)}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma, \mathcal{P}\{\texttt{p} \mapsto \{\langle m, \top \rangle\}\}, \mathcal{B}\{m \mapsto c\}, \mathcal{V}, \mathcal{S}\{\texttt{p} \mapsto freshVar(r)\}\rangle}$$

Данная передаточная функция сохраняет корректность абстрактного состояния, т.к. всегда можно определить $\psi(\mathcal{S}(p)) = l$, где l — длина выделенной строки. Т.к. значение элементов выделенного массива может быть любым, то данная передаточная функция также сохраняет точность абстрактного состояния.

Функция `strcpy(string dst, string src)` осуществляет копирование содержимого строки `src` в строку `dst`, поэтому новой длиной `dst` после вызова становится длина строки `src`.

$$\text{STRCPY}\ \frac{\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \qquad q \vdash \texttt{x = call strcpy(dst, src)} \qquad \mathcal{S} \vdash \texttt{src} \to l_{src}}{\mathcal{A}_{out} \vdash q \to \langle \pi,\ \sigma,\ \mathcal{P},\ \mathcal{V},\ \mathcal{S}\{\texttt{dst} \mapsto l_{src}\}\rangle}$$

Заметим, что данная передаточная функция, очевидно, сохраняет корректность и точность абстрактного состояния.

Функция `strncpy(string dst, string src, var n)` принимает три параметра: адрес `dst`, куда копируются данные, адрес `src`, откуда копируется строка, и целочисленное значение `n`, определяющее наибольшее количество

скопированных байт. Пусть на ребре, входящем в инструкцию вызова данной функции, значения отображения $\mathcal{S}$ для строк `dst` и `src` равны l_{dst} и l_{src} соответственно, символьным выражением для переменной `n` является v_n.

Для определения значения длины `dst` следует рассмотреть три случая. Если значение переменной `n` больше длины `src` ($v_n >_s l_{src}$), то длина `dst` после вызова будет равняться длине `src`. Если значение `n` не больше длины `src` ($v_n \leqslant_s l_{src}$), то среди первых `n` байт строки `src` заведомо нет нулевого, и здесь возможны два случая. Если значение `n` также не превосходит длину `dst` ($v_n \leqslant_s l_{dst}$), то длина `dst` останется прежней, так как положение ближайшего к началу нулевого байта не изменится. В противном же случае про длину `dst` можно сказать лишь, что она заведомо не меньше `n`, поэтому для неё будет создана новая символьная переменная. Условие на значение этой переменной будет добавлено к предикату π.

$$\text{STRNCPY}\ \frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \qquad q \vdash \texttt{x = call strncpy(dst, src, n)} \\ \mathcal{S} \vdash \texttt{src} \to l_{src} \qquad \mathcal{S} \vdash \texttt{dst} \to l_{dst} \qquad \sigma \vdash n \to v_n \\ l_u = freshVar(r) \qquad l_{res} = \mathit{ite}(v_n >_s l_{src},\ l_{src},\ \mathit{ite}(v_n \leqslant_s l_{dst},\ l_{dst},\ l_u)) \\ s_l = \mathit{createMin}(l_{res}, \mathcal{V}(q, l_{src}), \mathcal{V}(q, v_n))\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi \wedge (l_u \geqslant_s v_n),\ \sigma,\ \mathcal{P},\ \mathcal{V}\{l_{res} \mapsto s_l\},\ \mathcal{S}\{\texttt{dst} \mapsto l_{res}\} \rangle}$$

Заметим, что по-прежнему остаётся верна лемма 8 о попарной несовместности предикатов π для различных веток в точке объединения. Корретность абстрактного состояния очевидна по построению. Точность абстрактного состояния для первых двух случаев доказывается тривиально. Для третьего случая точность не гарантируется.

Функция `strcat(string dst, string src)` дописывает значение строки `src` в конец строки `dst`, таким образом, длина последней становится равна сумме исходных длин строк.

$$\text{STRCAT}\ \frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \qquad q \vdash \texttt{x = call strcpy(dst, src)} \\ \mathcal{S} \vdash \texttt{src} \to l_{src} \qquad \mathcal{S} \vdash \texttt{dst} \to l_{dst} \\ s_l = \mathit{createArithm}(l_{src} + l_{dst}, \mathcal{V}(q, l_{dst}), \mathcal{V}(q, l_{src}), +)\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi,\ \sigma,\ \mathcal{P},\ \mathcal{V}\{l_{dst} + l_{src} \mapsto s_l\},\ \mathcal{S}\{\texttt{dst} \mapsto l_{dst} + l_{src}\} \rangle}$$

Корректность и точность абстрактного состояния на выходе доказываются по аналогии с инструкцией сложения.

Функция `strncat(string dst, string src, var n)` аналогична предыдущей функции, но копирует не более чем `n` символов. Аналогично, здесь рассматриваются два варианта: когда длина строки `src` меньше `n` и обратный

случай. Заметим, что функция `strncat` всегда записывает нулевой байт после скопированных символов, т.е. длина получившейся строки всегда известна.

$$\text{STRNCAT}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \quad q \vdash \texttt{x = call strncat(dst, src, n)} \\ \mathcal{S} \vdash \texttt{dst} \to l_{dst} \quad \mathcal{S} \vdash \texttt{src} \to l_{src} \quad \sigma \vdash \texttt{n} \to v_n \quad \mathcal{V} \vdash v_n \to s_n \\ \mathcal{V} \vdash l_{src} \to s_{src} \quad l_{min} = \mathit{ite}(l_{src} <_s v_n,\ l_{src},\ v_n) \\ \mathcal{V} \vdash l_{dst} \to s_{dst} \quad l_{res} = \mathit{ite}(v_n >_s l_{src},\ l_{dst} + l_{src},\ l_{dst} + v_n) \\ s_{min} = \mathit{createMin}(l_{min}, s_{src}, s_n) \quad s_l = \mathit{createArithm}(l_{res}, s_{dst}, s_{min}, +)\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi,\ \sigma,\ \mathcal{P},\ \mathcal{V}\{l_{res} \mapsto s_l,\ l_{min} \mapsto s_{min}\},\ \mathcal{S}\{\texttt{dst} \mapsto l_{res}\} \rangle}$$

Корректность и точность абстрактного состояния очевидна по построению.

Связь между значениями целочисленных переменных и длинами строк возникает при вызове функций, вычисляющих длину строки, и работе с массивами посимвольно. Так, например, при обработке инструкции `x = strlen(string str)`, которая принимает на вход строку и возвращает её длину, символьное выражение для длины строки `str` копируется для переменной `x`.

$$\text{STRLEN}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \quad q \vdash \texttt{x = call strlen(str)} \\ \mathcal{S} \vdash \texttt{str} \to l_{str}\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi,\ \sigma\{\texttt{x} \mapsto l_{str}\},\ \mathcal{P},\ \mathcal{V},\ \mathcal{S} \rangle}$$

При присваивании нового значения в элемент массива `str[i] = x` возможны два случая: если присваиваемое значение равно нулю и индекс меньше текущей длины, то длина строки `str` станет равна `i`, а в противном случае не изменится.

$$\text{BUFASSIGN}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \quad q \vdash \texttt{str[i] = x} \\ \mathcal{S} \vdash \texttt{str} \to l_{str} \quad \sigma \vdash \texttt{x} = v_x \quad \sigma \vdash \texttt{i} = v_i\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi,\ \sigma,\ \mathcal{P},\ \mathcal{V},\ \mathcal{S}\{\texttt{str} \mapsto \mathit{ite}(v_x \neq 0 \vee l_{str} <_s v_i,\ l_{str},\ v_i)\} \rangle}$$

Корректность и точность абстрактного состояния на выходе для этих двух передаточных функций очевидна.

Аналогично при чтении значения массива `x = str[i]` следует рассмотреть три случая: если индекс меньше текущей длины, то значение по этому индексу заведомо ненулевое. Если длина строки `str` равна `i`, то по этому индексу располагается нуль-терминатор. Иначе значение символа может в общем случае быть любым.

$$\text{BUFACCESS}\frac{\begin{array}{c}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V}, \mathcal{S} \rangle \quad q \vdash \texttt{x = str[i]} \\ \mathcal{S} \vdash \texttt{str} \to l_{str} \quad \sigma \vdash \texttt{i} = v_i \quad v_x = \mathit{ite}(l_{str} = v_i,\ 0,\ \mathit{freshVar}(8))\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi \wedge (l_{str} <_s v_i \ \Rightarrow\ v_x \neq 0),\ \sigma\{\texttt{x} \mapsto v_x\},\ \mathcal{P},\ \mathcal{V},\ \mathcal{S} \rangle}$$

Корректность абстрактного состояния на выходе очевидна. Как уже отмечалось ранее, вопросы моделирования содержимого массивов выходят за рамки данной работы, и поэтому при чтении значения из массива создаётся новая символьная переменная, что в общем случае может нарушить точность абстрактного состояния. В данном случае, если для всех функций конкретизации, для которых $\psi(\pi) = \top$, выполнено $\psi(l_{str} = v_i) = \top$, то абстрактное состояние на выходе будет точным. В противном случае точность гарантировать нельзя, но дополнительно помещённая в абстрактное состояние информация потенциально может уменьшить количество таких функций конкретизаций $\psi : \psi(\pi) = \top$, для которых не найдётся эквивалентного конкретного состояния. На практике это позволяет избавиться от некоторых ложных срабатываний.

Поддержка нового отображения $\mathcal{S}$ требуется и при объединении состояний в точках слияния: при объединении значений переменных из множества *String* объединяются не только их символьные выражения, но и длины их строк объединяются аналогичным способом с учётом условий на объединяемых ветках по аналогии с отображением σ (см. раздел 3.3.7).

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 (см. раздел 4.1). Значения типа *Min* транслируются естественным образом: значение в контексте вызова составляется из соответствующих оттранслированных значений аргументов.

5.1.4 Поддержка анализа строк с широкими символами

Полностью аналогично реализуется поддержка широких строк — нультерминированных массивов из широких символов (в языке C это массивы элементов типа `wchar_t`). Широкие символы имеют фиксированный размер `sizeof(wchar_t)` $= w \geqslant 8$, как правило, 16 или 32 бита.

Для поддержки широких строк необходимо ввести понятия широкого строкового литерала *WCSLit* (по аналогии с обычным строковым литералом) и широкой строки $\mathit{WCSLit} \cup A_w$. Далее потребуется расширить отображение $\mathcal{S}$ для

широких строк по аналогии с обычными строками и вместо функций `strlen`, `strcpy` и т.п. рассмотреть аналоги для широких строк `wcslen`, `wcscpy` и т.п.

5.2 Обнаружение переполнения данными, полученными из недоверенного источника

Одним из наиболее серьезных источников уязвимостей, связанных с переполнением буфера, является обращение к буферу по индексу, вычисленному на основе данных, полученных из недоверенного источника. Если злоумышленник контролирует значения входных данных, которые без должной проверки на корректность используются для вычисления индекса, то в некоторых случаях ему удастся подобрать такие значения входных данных, что произойдёт переполнение. Последствиями этого переполнения в свою очередь могут стать отказ в обслуживании, перехват управления [9], утечка чувствительных данных [10] и пр.

Описанный в предыдущем разделе базовый алгоритм может быть расширен для организации поиска подобных ошибок. Заметим, что так как достаточно предъявить один-единственный набор входных данных, приводящий к переполнению, чтобы гарантированно утверждать о наличии ошибки в программе, то требование независимости ошибки от конкретных входных данных, сформулированное в разделе 2.1, в данном случае можно снять.

Рассмотрим предлагаемый подход формально. Пусть некоторые функции программы помечены как возвращающие данные из недоверенного источника. Тогда в результате анализа инструкции вызова такой функции

```
t = call get_tainted_data()
```

для переменной $\mathtt{t} \in V_b$ будет создана новая символьная переменная $\sigma(\mathtt{t}) = v = freshVar(b)$ и будет создано новое значение в отображении $\mathcal{V}$

$$\mathcal{V}(v) = s_v = \langle v \rangle \in \textit{Tainted} \subset \textit{Summary}.$$

Значения из множества *Tainted* сохраняют только символьную переменную, значения которой считаются пришедшими из недоверенного источника. Так как достаточно подобрать лишь одно любое значение такой переменной, приводящее

к переполнению, чтобы утверждать о наличии ошибки, то условия $\mathfrak{U}$, $\mathfrak{L}$ выглядят следующим образом:

$$\mathfrak{U}(s_v, x) = v \geqslant_s x, \qquad \mathfrak{L}(s_v, x) = v \leqslant_s x.$$

Остальной алгоритм анализа остаётся неизменным. Таким образом, переполнение буфера данными, полученными из недоверенного источника, может быть выполнено с помощью описываемого подхода.

5.3 Поиск переполнения буфера в циклах

Как было отмечено в разделе 1.2, для обнаружения дефектов рассматриваемого типа анализ циклов особенно важен, так как 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. Однако при текущем подходе анализ производится над развёрткой графа, т.е. первые $k-1$ итераций представлены в графе развёртке индивидуальным набором инструкций и анализируются точно, а k-ая итерация представляет собой обобщенную итерацию, объединяющую свойства всех итераций после $(k-1)$-ой. Как следствие, в общем случае вычисляемая анализом информация о переменных, изменяющихся внутри цикла, для неразвёрнутых итераций (больших, чем $(k-1)$-ая) является довольно грубой абстракцией, и её, как правило, недостаточно для анализа значений с последней итерации. Для улучшения ситуации может быть использована информация об индуктивных переменных, полученная с помощью консервативного анализа. Рассмотрим расширение базового алгоритма для поддержки простых циклов.

Пусть для некоторой переменной $i \in V_b$ известно, что её значение изменяется в цикле по закону арифметической прогрессии $i = a + l \cdot d$, $a, d \in C_b$, $l \in \mathbb{N} \cup \{0\}$ (a — начальное значение, d — изменение за одну итерацию, l — номер итерации). Пусть также одним из условий выхода из этого цикла является условие $\neg(i \diamond n)$. Значение $\sigma(i)$ для последней, обобщённой итерации обозначим $v_i \in SE_b$. Потерю точности анализа этого значения, произошедшую в результате объединения итераций, будем компенсировать дополнительной информацией об индуктивных переменных.

Для этого вводится новый тип элементов множества $\mathit{InductiveVar} \subset \mathit{Summary}$:

$$\begin{aligned}\mathit{InductiveVar} = \{\langle v_i, s_n, a, d, \diamond\rangle \quad | \\ v_i \in \mathit{SE}_b, \quad s_n \in \mathit{Summary}, \quad a,\ d \in C_b, \\ \diamond \in \{>_s, <_s, \geqslant_s, \leqslant_s, >_u, <_u, \geqslant_u, \leqslant_u\}\}\end{aligned}$$

В этом случае будем считать, что переменная i внутри цикла может принимать любое значение, удовлетворяющее условию $i \diamond n$ и формуле $a + l \cdot d$. Отсюда можно вывести искомые достаточные условия. Рассмотрим их вычисление на примере условия $i \leqslant_s n$ при $a \geqslant_s 0,\ d >_s 0$.

Пусть $\mathcal{V}(n) = s_n,\ l' \in \mathit{SE}_b^{\mathit{Aux}}$. Тогда

$$\mathfrak{U}(s_i, x) = \exists l' : (a + l' \cdot d \geqslant_s x) \wedge ((1^b{-}a)/d \geqslant_s l') \wedge \mathfrak{U}(s_n, a + l' \cdot d).$$

Таким образом, искомое условие $\mathfrak{U}(s_i, x)$ будет выполнено, если будет выполнено $\mathfrak{U}(s_n, a + l' \cdot d)$, т.е. на некотором пути до входа в цикл n будет достаточно большим, чтобы нашлась такая итерация l' рассматриваемого цикла, что итеративная переменная i на этой итерации будет не меньше, чем x. Условие $(1^b{-}a)/d \geqslant_s l'$ необходимо для того, чтобы в качестве l' рассматривать только значения, соответствующие монотонно возрастающим значениям i (без переполнения). Тем самым гарантируется, что при вычислении условия $i \leqslant_s n$ также не произойдет переполнения и выхода из цикла по этому условию, что привело бы к недостижимости итерации l'. Формулы для других случаев a, d и остальных условий $\diamond$ вычисляются аналогично.

Глава 6. Анализ переполнения буфера произвольного размера

Данная глава будет посвящена поиску переполнения буфера в массивах произвольного размера. Заметим, что ситуация переполнения левой границы буфера от размера буфера никак не зависит, поэтому такие ошибки могут быть найдены методами из предыдущих глав. Здесь будем рассматривать только переполнение правой границы.

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).

$$\texttt{h} \in V_r \cup C_r, \quad \texttt{p} \in P_b.$$

$\langle \mathit{Statements} \rangle ::= \langle \mathit{Statement} \rangle \mid \langle \mathit{Statement} \rangle \, \langle \mathit{Statements} \rangle \mid$ **`m = alloca(b, h)`**

Инструкция `p = alloca(b, h)` выделяет новый локальный массив $m \in A_b$ из `h` элементов размера `b` (не пересекающийся ни с какими уже выделенными участками памяти) и сохраняет его адрес в указателе `p`:

$$\mathbb{P}(\texttt{p}) \leftarrow m, \quad \mathbb{B}(\texttt{m}) \leftarrow \mathbb{X}(\texttt{h}).$$

Расширим также отображение $\mathcal{B}$ — теперь размер массива будет моделироваться произвольным символьным выражением ширины r:

$$\mathcal{B} : A \to SE_r.$$

Передаточные функции для уже имеющихся инструкций не изменятся, но для добавленной инструкции потребуется новая передаточная функция:

$$\text{ALLOCAA2}\frac{\begin{array}{cc}\mathcal{A}_{in} \vdash q \to \langle \pi, \sigma, \mathcal{P}, \mathcal{B}, \mathcal{V} \rangle & q \vdash \texttt{p = alloca(b, h)} \\ \sigma \vdash \texttt{h} \to v_h & m = freshArray(b)\end{array}}{\mathcal{A}_{out} \vdash q \to \langle \pi, \sigma, \mathcal{P}\{\texttt{p} \mapsto \{\langle m, \top \rangle\}\}, \mathcal{B}\{m \mapsto v_h\}, \mathcal{V} \rangle}$$

Покажем, что эта инструкция сохраняет точность и корректность абстрактного состояния. Пусть некоторому конкретному состоянию на входе была эквивалентна функция конкретизации ψ для $\mathcal{A}_{in}$. Тогда можно её доопределить значениями конкретного состояния на выходе: $\psi \leftarrow \psi\{m \mapsto \mathbb{P}(\texttt{p})\}$, причём $\mathbb{B}(\mathbb{P}(\texttt{p})) = \mathbb{X}(\texttt{h}) = \psi(\sigma(\mathcal{B}(m)))$. Значит, она станет эквивалентна этому состоянию для $\mathcal{A}_{out}$, что гарантирует сохранение корректности.

Рассмотрим произвольную функцию конкретизации ψ для $\mathcal{A}_{out}$, для которой $\psi(\pi) = \top$. Из точности $\mathcal{A}_{in}$ следует, что для ψ найдётся эквивалентное конкретное состояние на входе в инструкцию. Тогда найдётся такое конкретное состояние на выходе, совпадающее с ним для всех переменных, и дополнительно $\mathbb{P}(\mathtt{p}) = \psi(m)$, причём $\mathbb{B}(\mathbb{P}(\mathtt{p})) = \sigma(\mathcal{B}(m))$. Это конкретное состояние будет эквивалентно ψ для $\mathcal{A}_{out}$, откуда следует, что $\mathcal{A}_{out}$ точно.

Таким образом, вычисленные абстрактные состояния для программы на расширенном языке будут точными и корректными. Однако это не позволяет напрямую использовать условия *NotLess* и *NotGreater*, т.к. по определению третьим параметром (в качестве которого ранее использовался константный размер буфера) может быть только символьное выражение из множества SE^{Aux}, значение которого не зависит от неизвестных переменных.

Далее будет рассмотрено два альтернативных подхода к решению этой задачи: с использованием отображения $\mathcal{V}$ и путём проверки условия (3.1) с кванторами всеобщности.

6.1 Использование отображения $\mathcal{V}$ для размера буфера

Первый подход 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* 9.** *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:

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

Доказательство. Пусть данная формула выполнима, т.е. найдётся такое значение χ, для которого выполнима формула под квантором существования. Проведя рассуждения, аналогичные приведённым в доказательстве теоремы 2, можно показать, что найдётся путь l в графе развёртки, проходящий через q_{ac}, и на одном из соответствующих ему конкретных выполнений в точке q_{ac} реализуется конкретное состояние $\langle \mathbb{X}, \mathbb{P}, \mathbb{B} \rangle$, причём:

(i) $\mathbb{X}(\mathtt{i}) \geqslant_s \chi$,

(ii) $\mathbb{B}(\mathbb{P}(\mathtt{p})) \leqslant_s \chi$,

(iii) условия (i) и (ii) в этой точке выполнены для любого конкретного выполнения, соответствующего l.

Выбор l гарантирует, что он выполним. Кроме этого, для любого конкретного выполнения, соответствующего этому пути, можно записать: $\mathbb{X}(\mathtt{i}) \geqslant_s \chi \geqslant_s \mathbb{B}(\mathbb{P}(\mathtt{p}))$. Значит, в точке q_{ac} имеется ошибка переполнения буфера. □

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)`.

6.2 Перебор значений условий переходов

Второй подход заключается в использовании достаточного условия ошибки (3.1). Как уже было отмечено в разделе 3.2.2, проверка условия в таком виде затруднительна, во-первых, в связи с уникальностью условия $\mathit{ReachCond}_q^p(\vec{t})$ для каждого пути p, и во-вторых, в связи с наличием квантора всеобщности.

Первое затруднение может быть преодолено путём некоторого ослабления (возможно, с потерей точности) формулы (3.1) для существенного упрощения её вида. 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 единственный (возможно, невыполнимый) путь на графе развёртки.

С учетом этого замечания предлагается рассматривать в формуле (3.1) не пути на графе, а конкретные наборы значений $\{w_i \mid w_i \in \{\top, \bot\}\}$ последовательности $\{W_i\}$:

$$\exists\{w_i\}: \quad \exists\vec{t}: \mathit{ReachCond}_q^{\{w_i\}}(\vec{t}) \wedge \forall\vec{t}: (\mathit{ReachCond}_q^{\{w_i\}}(\vec{t}) \Rightarrow \mathit{Error}_q(\vec{t})),$$

где $\mathit{ReachCond}_q^{\{w_i\}} \doteq \mathit{ReachCond}_q \wedge \bigwedge_i W_i = w_i$.

Данный подход позволяет существенно сократить размер формулы за счёт единого предиката $\mathit{ReachCond}_q^{\{w_i\}}$ для всех путей, но по-прежнему требует проверки на разрешимость формулы с кванторами всеобщности, что ограничивает его масштабируемость. Кроме того, в данной формуле никак не учитывается тот факт, что для произвольного пути выполнения некоторые значения $\{w_i\}$ могут остаться невычисленными, но, т.к. условия на их значения входят в предикат $\mathit{ReachCond}_q^{\{w_i\}}$, то это может стать причиной выдачи ложного предупреждения.

Пусть $V \doteq \{V_i = \sigma_{q_i}(\mathtt{h}) \diamond \sigma_{q_i}(\mathtt{g}) \mid q_i: \texttt{assume h} \diamond \texttt{g} \in \mathit{Instr}\} \subset \mathcal{C}$ — множество элементарных условий над символьными переменными, созданных при интерпретации инструкций `assume`. Это множество будет использоваться в качестве аналога W для абстрактного состояния.

В качестве условия Error_q для инструкции `p[i] = x` будет использоваться формула $E \doteq \bigvee_{\langle m,\gamma\rangle\in\mathcal{P}(p),m\in\mathcal{B}} \gamma \wedge (\sigma(i) \geqslant_s \sigma(\mathcal{B}(m)))$. В качестве $\mathit{ReachCond}_q^{\{w_i\}}$ возьмём $\pi_q^{\{v_i\}} \doteq \pi \wedge \bigwedge_{V_i} V_i = v_i$, где $\forall i: v_i \in \{\top, \bot\}$.

Роль независимых переменных $\vec{t}$ будут выполнять символьные переменные из S, набор конкретных значений которых будем обозначать $\vec{s}$.

$$\exists\{v_i\}: \quad \exists\vec{s}: \pi_q^{\{v_i\}} \wedge \forall\vec{s}: \pi_q^{\{v_i\}} \Rightarrow E.$$

На листинге 6.1 приведён пример функции, в которой при использовании данного подхода будет выдано предупреждение на строке 7, хотя эта точка не удовлетворяет определению ошибки. Действительно, в данном случае $V = \{f = 0,\ f \neq 0,\ s <_s n,\ s \geqslant_s n\}$ и $E = n \geqslant_s s$. Конкретные значения условий ветвлений $\{v_i\} = \{(f = 0) = \bot,\ (f \neq 0) = \top,\ (s <_s n) = \top,\ (s \geqslant_s n) = \bot\}$ действительно определяют выполнимый путь и гарантируют ошибку при любых значениях $\vec{s}$. Однако ошибки в этой функции по определению 1 нет, т.к., возможно, контракт функции гарантирует, что $\mathtt{f} \neq 0 \Rightarrow \mathtt{s} \geqslant_s \mathtt{n}$ (такой контракт удовлетворяет предположению о контрактах). Ложное срабатывание произошло

Листинг 6.1 — Пример ложного срабатывания

```
1 foo(var f, var s, var n) { // σ(f) = f,  σ(s) = s,  σ(n) = n
2   a = alloca(s, 4)         // P(a) = ⟨a, ⊤⟩,  B(a) = s
3   if (f = 0) {
4     if (s <_s n)
5       a[0] = 7;
6   } else {
7     a[n] = 42;
8   }
9 }
```

из-за того, что условие $W_3 = \mathtt{s} <_s \mathtt{n}$ не вычисляется на рассматриваемом пути, но из него следует условие E в данной точке.

Глава 7. Реализация и результаты

7.1 Реализация методов поиска переполнения в анализаторе Svace

7.1.1 Анализатор Svace

Svace — статический анализатор исходного кода, разрабатываемый в ИСП РАН [73]. Поддерживается анализ программ на языках C, C++, Java. Ключевыми особенностями данного анализатора является хорошая масштабируемость, высокая доля истинных срабатываний (60-80 %) и большое количество типов обнаруживаемых ошибок.

Анализатор представляет собой совокупность ядра и детекторов, отвечающих за поиск конкретных типов дефектов. Ядро анализатора Svace использует подходы анализа потока данных и символьного исполнения с объединением состояний для внутрипроцедурного анализа и метод резюме для межпроцедурного анализа. На уровне ядра решаются задачи построения ГПУ и графа развёртки, поиска недостижимого кода, анализа алиасов, анализа функций, завершающих выполнение программы. Всем детекторам доступна информация о результатах этих видов анализов. Ядром производится нумерация значений, т.е. вычисляются классы эквивалентности значений переменных, называемые идентификаторами значений. Детекторы ассоциируют с идентификаторами значений вычисленные свойства программы. Ядро проводит символьное выполнение программы с объединением состояний. При этом вычисляются необходимые условия достижимости каждой точки программы в виде формул алгебры логики, где роль переменных играют идентификаторы значений. Детекторы оповещаются о всех событиях, происходящих внутри функции. Реализация детектора заключается в описании обработчиков для этих событий. Рассмотренный подход был реализован в виде нескольких детекторов анализатора Svace.

7.1.2 Особенности реализации поиска переполнения буфера

В общих чертах поиск ошибок осуществляется в соответствии с алгоритмом 3.1. Обход графа G_k производится ядром анализатора, при этом построения графа в явном виде не происходит. Компоненты π, σ, $\mathcal{P}$ абстрактного состояния в каждой точке программы также вычисляются ядром. Условие π упрощается с использованием информации о доминировании вершин ГПУ [71]. Отображения $\mathcal{B}$, $\mathcal{V}$, $\mathcal{S}$ и множество фактов доступа к буферу вычисляются в виде отдельных атрибутов, реализованных в рамках настоящей работы. Эти атрибуты вычисляются, распространяются, сохраняются в резюме и транслируются в контекст вызова детекторами переполнения буфера.

Значения $\mathcal{V}$ хранятся в виде множества графов с общими вершинами, таким образом, общие поддеревья сохраняются только один раз. Это также позволяет организовать кэширование результатов для наиболее затратных по времени операций над значениями *Summary*; к таким операциям относятся миграция значения в конкретный контекст вызова (см. 4.2), рекурсивная «обрезка» веток, не актуальных в контексте вызова, при создании резюме (см. раздел 4.3).

Для улучшения производительности были введены ограничения на размер значения атрибута $\mathcal{V}$ как в рамках внутрипроцедурного анализа, так и (более строгие) для сохранения в резюме. С той же целью используется упрощение помещаемых в резюме формул (условий в узлах типа *Join* и в фактах доступа к буферу *KnownBufferAccess* и *UnknownBufferAccess*).

При проверке инструкций доступа и вызова функций для выдачи предупреждения сначала производится быстрая проверка с помощью интервалов значений. Если отсутствие ошибки доказано не было, то рекурсивно строятся достаточные условия ошибки. Полученные условия формулируются в виде запроса на языке SMTLib2 [74] и передаются решателю Z3 [75] для проверки на выполнимость. Если формула оказалась выполнима, то с помощью 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.

Типы предупреждений

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`);
- `BUFFER_UNDERFLOW` — переполнение левой границы буфера (обращение по отрицательному индексу);
- `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;
- `TAINTED_ARRAY_INDEX.EX` — доступ к буферу по индексу, вычисленному из недоверенных данных (предупреждения детектора, основанного на механизме, описанном в 5.2);
- `BUFFER_OVERFLOW.STRING` — переполнение буфера при работе со строками (см. главу 5.1);
- `DYNAMIC_OVERFLOW.EX` — переполнение буфера неизвестного на момент компиляции размера (детектор, основанный на подходе из раздела 6.1).

Пример срабатывания

На рисунке 7.1 приведён пример реального срабатывания, выданного на проекте LibreOffice 6.0.7.1. Svace сообщает о неконсистентом сравнении: из сравнения на строке 660 следует, что значение `pPara->nDepth` может превышать 4, и в этом случая для использования вместо него создаётся безопасная переменная `nDepth`. Однако впоследствии в качестве аргумента функции `mpStyleSheet->IsHardAttribute` используется старое значение `pPara->nDepth`, где оно выступает в качестве индекса буфера размера 5. В качестве одного из возможных значений `pPara->nDepth`, приводящих к ошибке,

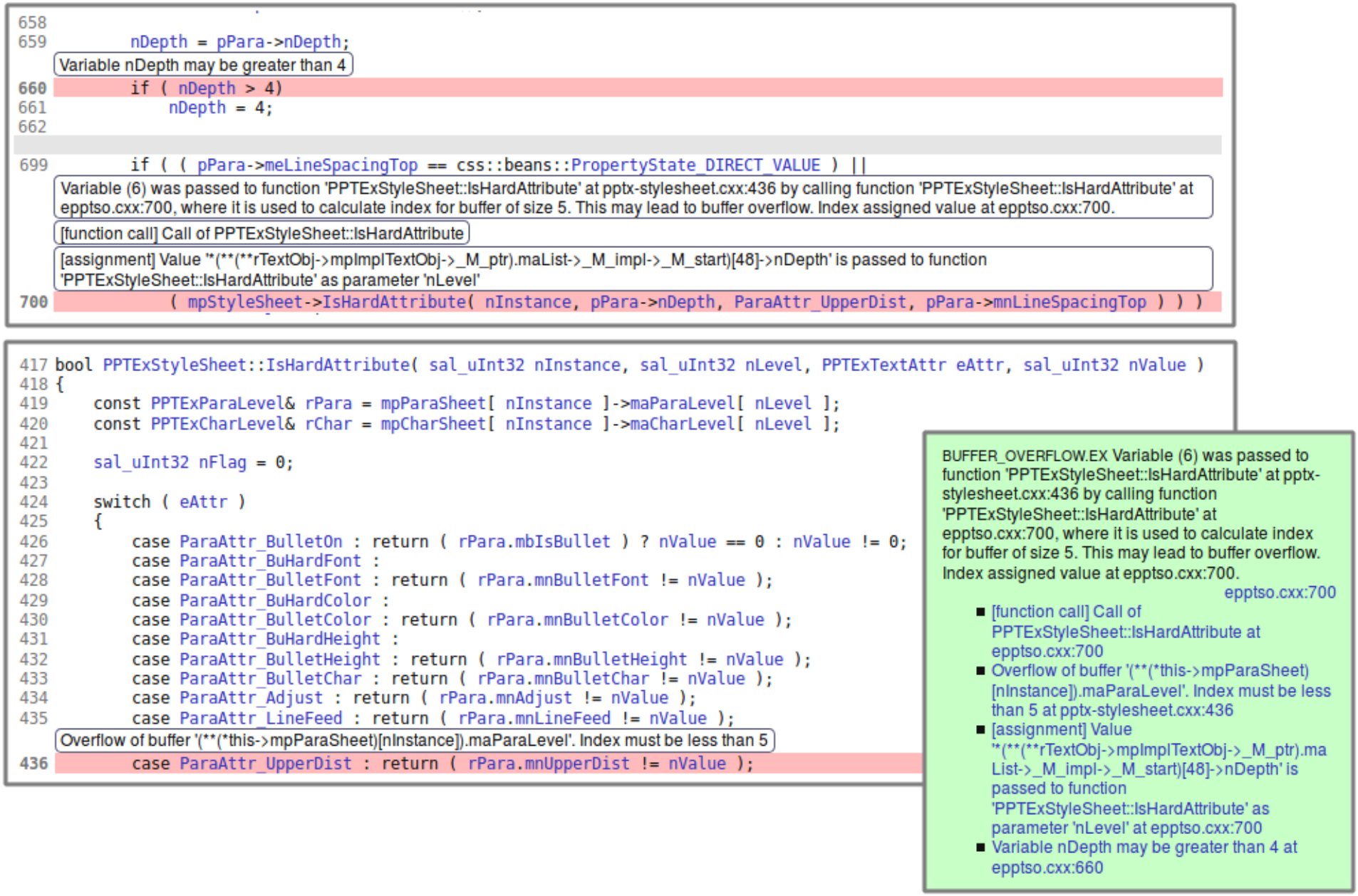

Рисунок 7.1 — Ошибка из проекта LibreOffice 6.0.7.1

в трассе срабатывания приводится значение 6. В результате внесения информации об этом срабатывании в систему отслеживания ошибок LibreOffice (bug 121795 [76]) приведённый код был исправлен разработчиками.

Подавление ложных срабатываний

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.

Листинг 7.1 — Сравнение с параметром

```
1 void foo(int x) {
2   ...
3   if (a > x) { ... }
4   ...
5 }
6
7   foo(1000);
```

Листинг 7.2 — Множественные сравнения

```
1   if (x >= 255) {
2     ...
3   }
4   if (y >= x) {
5     ...
6   }
```

– Зачастую сравнения некоторой переменной с параметром функции во многих контекстах всегда имеют одинаковый результат, поэтому нельзя полагаться на то, что обе ветки сравнения достижимы, и такие сравнения игнорируются. Например, в программе, приведённой на листинге 7.1, возможно, условие $\texttt{a} > \texttt{x}$ невыполнимо при $\texttt{x} = 1000$, поэтому на основании этого сравнения нельзя делать вывод, что на строке 4 значение a может быть больше 1000.
– Ещё одним источником нарушения предположения о контрактах являются множественные невложенные сравнения одной переменной, как, например, в участке кода на листинге 7.2. В данном случае, если оба сравнения вычисляются в истину, то можно заключить, что $\texttt{y} \geqslant 255$ на строке 5. Однако зачастую по контракту функции одна из веток второго сравнения оказывается недостижимой из одной из веток первого сравнения, таким образом, нарушается предположение о контрактах. Для решения этой проблемы значение типа *Assume* не создаётся на основе другого значения типа *Assume*, если его условие не доминирует текущую точку.
– Вследствие того, что анализ производится над развёрткой цикла с изменением семантики инструкций, для циклов, в реальности выполняющихся менее k итераций, на k-ой итерации анализа могут возникать ложные предупреждения. Такие срабатывания легко выявить и отфильтровать перед выдачей.

Листинг 7.3 — Пример неоднозначного определения размера буфера

```
1 struct st {
2   char buf[10];
3   char x;
4   char y;
5 } s;
6
7 char *p = &s.buf[0];
```

Листинг 7.4 — Пример структуры с массивом переменного размера

```
1 struct flex {
2   int len;
3   char data[];
4 };
5   ...
6   struct flex *fl =
7     malloc(sizeof(*fl) + x);
8   fl->len = x;
```

В ситуациях, когда буфер является подобъектом некоторого объемлющего объекта памяти (например, полем структуры или частью многомерного массива), вопрос определения корректности доступа может быть затруднителен. Рассмотрим пример, приведённый на листинге 7.3. Каким следует считать размер массива, адрес которого находится в переменной `p`? Ответ на этот вопрос может зависеть от того, как именно происходит обращение к этому буферу. Рассмотрим два случая:

1. `for (int i = 0; i <= 10; i++) p[i] = 0;`
2. `memset(p, 0, sizeof(struct st));`

Оба этих случая не приведут к неопределённому поведению с точки зрения стандарта языка C. Однако в первом случае такой доступ выглядит как типичная ошибка (т.н. *off-by-one error*) — похоже, что случайно в условии цикла вместо строго неравенства оказалось нестрогое. Хотя поведение данной программы полностью определено, вероятно, значение переменной `s.x` будет неправильным после выполнения такого цикла. Второй случай, хотя и очень похож на первый, но является полностью корректным и часто встречающимся примером кода.

Таким образом, для проверки некоторых инструкций (таких как вызов функций `memcpy`, `memset`, `memmove`, других функций, вызывающих перечисленные) следует в качестве размера массива, доступ к которому нужно проверить, использовать размер корневого (самого крупного) объекта, частью которого является этот массив, за вычетом смещения начала массива от начала этого объекта.

Отдельно в этой связи следует упомянуть структуры с полем-массивом переменного размера (*flexible array member* [12, §6.7.2.1/18], см. листинг 7.4). Последним полем такой структуры является пустой массив,[1], а реальный его размер

[1] В коде, написанном до введения этого пункта в стандарт, можно встретить и массивы из 1-2 элементов, выступающие в той же роли.

определяется при выделении памяти (зачастую он хранится в одном из полей этой же структуры). В текущей реализации доступ к таким массивам игнорируется.

7.2 Оценка качества анализатора с помощью тестовых пакетов и сравнение с Infer

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 [28].

Для тестирования использовался набор синтетических тестов Juliet Test Suite C/C++ 1.3 (см. раздел 1.2.1).

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). На рассмотренных группах тестов были запущены анализаторы Svace и Infer. 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). Далее приведён анализ результатов тестирования.

Ложные срабатывания детекторов ошибки переполнения буфера у инструмента Svace на выбранных группах тестов составили 0,02 % от числа корректных

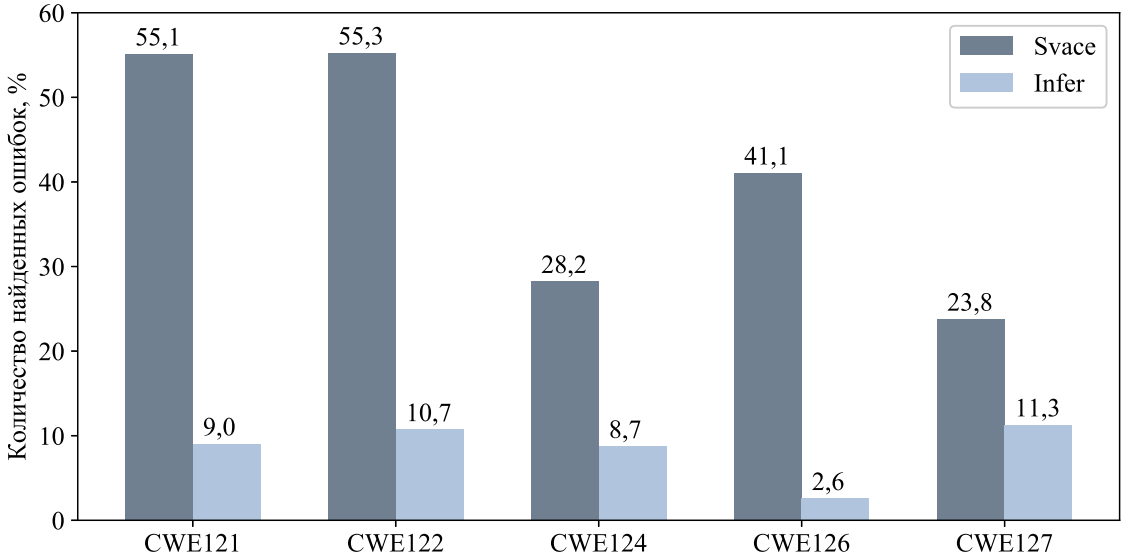

Рисунок 7.2 — Общие результаты на CWE 121, CWE 122, CWE 126

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.

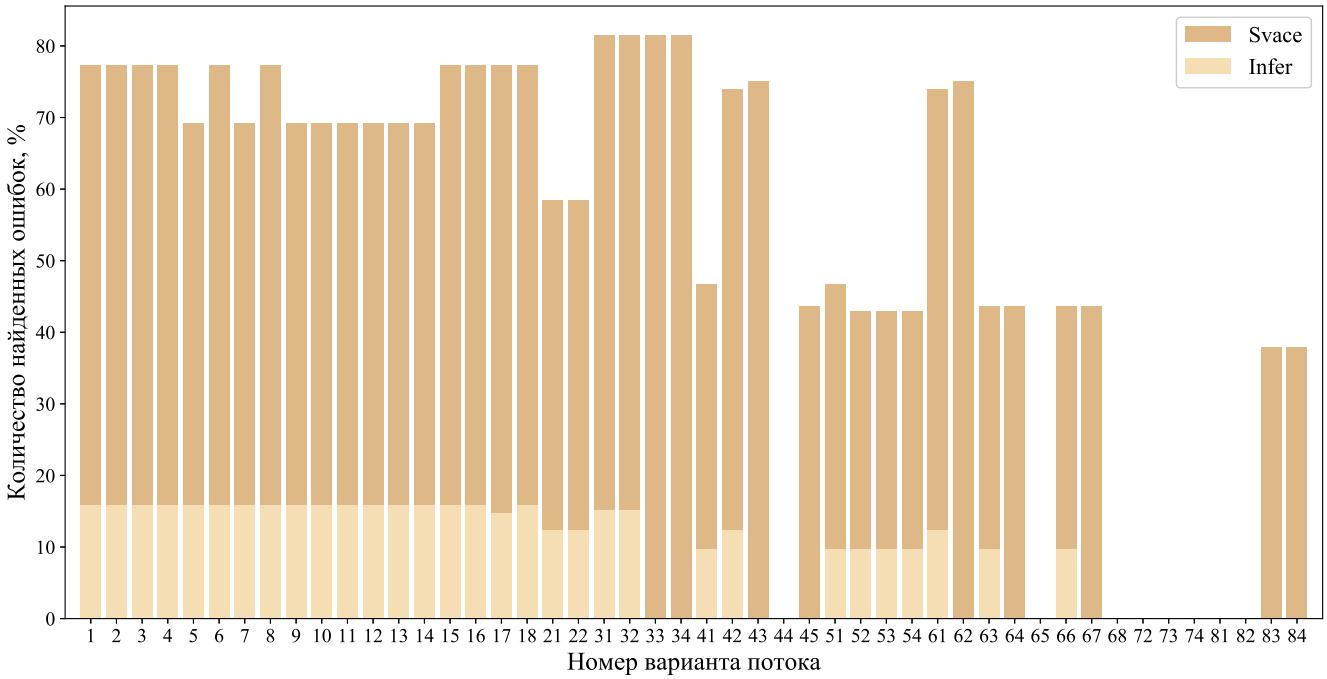

Рисунок 7.3 — Результаты Svace на различных вариантах потока

На рисунке 7.3 приведены результаты Svace на различных вариантах потока для тестов на переполнение правой границы. Можно заметить, что для 16 из 48 вариантов покрытие превышает 75 %, для 12 вариантов находится в диапазоне 50-75 %. Таким результатам соответствуют различные внутрипроцедурные варианты потока с использованием разнообразных управляющих конструкций. В эти же две группы попадают варианты потока, предполагающие передачу данных через возвращаемое значение функции. Для 12 вариантов покрытие составляет 25-50 % — это варианты потока, связанные с передачей данных через аргументы функции. И, наконец, 8 вариантов потока не поддерживаются вовсе — к ним относятся вызов функции по указателю, вызов виртуальной функции, передача данных через коллекции языка C++. Реализация подобной функциональности лежит за пределами детекторов переполнения буфера и не связана с рассматриваемыми алгоритмами. В то же время поддержка в ядре анализатора этих вариантов потока позволит имеющимся детекторам находить ошибки и в этих тестах.

Отметим также, что результаты Svace по каждому из категорий CWE и каждому из вариантов потока, как правило, превосходят результаты, полученные анализатором Infer, никогда им не уступая.

В приложении A приведены подробные результаты работы детекторов на различных типах ошибок.

Также тестирование Svace было проведено с помощью пакета Toyota ITC Benchmark на типах тестов, приведённых в разделе 1.2.1. Ложных срабатываний и на этом пакете выдано не было. Среднее покрытие на двух типах составляет 47,1 %. Для типа static memory покрытие достигло 83,6 % (85,2 % для переполнения правой границы и 76,9 % для переполнения левой границы). Для типа dynamic memory покрытие составило 12,7 % (9,4 % для переполнения правой границы и 15,4 % для переполнения левой границы). При этом включение экспериментального детектора, основанного на подходе анализа буферов произвольного размера из раздела 6.2, не привнесло ложных срабатываний и позволило увеличить покрытие до 68 % для переполнения правой границы динамического буфера, до 48,7 % для левой границы, и как следствие общее покрытие на динамических массивах поднялось до 57,7 % и покрытие в среднем до 70,3 %.

7.3 Результаты тестирования на проектах Android и Tizen

Результаты работы детекторов на исходном коде операционной системы Android 5.0.2 (в части, написанной на C/C++) приведены в таблице 4. Результаты на проекте Tizen 5.0 детекторов переполнения правой границы буфера приведены в таблице 5. Для каждого из типов предупреждений, перечисленных в разделе 7.1.2 и выданных на этом проекте, приведено общее число предупреждений 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. Например, к таким ситуациям относятся запись в два расположенных подряд буфера с помощью `strcpy`, наличие в тексте функции вызова `assert`, проверяющего корректность доступа, но отсутствующего в анализируемой сборке проекта. Для каждой из этих категорий приведено абсолютное и относительное количество срабатываний для каждого типа предупреждения в отдельности и всех в сумме.

Таблица 4 — 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 %**

Таблица 5 — Результаты тестирования на проекте Tizen 5.0

Тип срабатывания	Кол-во	TP		WF		FP	
		n	%	n	%	n	%
BUFFER_OVERFLOW.EX	77	34	44 %	19	25 %	24	31 %
BUFFER_OVERFLOW.LIB.EX	19	11	58 %	1	5 %	7	37 %
OVERFLOW_AFTER_CHECK.EX	88	41	47 %	23	26 %	24	27 %
TAINTED_ARRAY_INDEX.EX	4	2	50 %	0	0 %	2	50 %
DYNAMIC_OVERFLOW.EX	12	3	25 %	0	0 %	9	75 %
Всего	**200**	**91**	**46 %**	**43**	**21 %**	**66**	**33 %**

Как следует из таблиц, количество ложных срабатываний находится в заданных ограничениях (в среднем не превосходит 35 %). Основной причиной ложных срабатываний является неточность построенных условий, которая в свою очередь вызвана неточным межпроцедурным анализом, недостаточно точным моделированием памяти (в т.ч. содержимого массивов) и т.п. Другой причиной ложных срабатываний является нарушение предположения о контрактах.

Таблица 6 — Время работы детекторов на Android 5

Операция	Общее время, с	Кол-во вызовов	Среднее время, нс	Макс. время, с
Атрибуты для целых чисел				
Объединение $\mathcal{V}$	1 995	43 947 986	45	78
Создание резюме для $\mathcal{V}$	1 035	4 286 055	242	384
Применение резюме для $\mathcal{V}$	120	12 647 151	10	27
Создание фактов доступа	312	68 075 409	5	127
Проверка доступа к массиву	125	1 323 522	94	39
Проверка фактов доступа	314	7 269 900	43	115
Атрибуты для строк				
Объединение $\mathcal{V}$	313	4 322 961	72	22
Создание резюме для $\mathcal{V}$	122	111 545	1099	50
Применение резюме для $\mathcal{V}$	15	1 592 787	9	3
Объединение $\mathcal{S}$	404	29 786 914	14	31
Создание резюме для $\mathcal{S}$	209	66 590 846	3	204
Применение резюме для $\mathcal{S}$	41	5 259 411	8	12

В таблице 6 приведено время работы обработчиков событий анализа, реализующих описанную в данной работе функциональность. 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. В таблице приведено процессорное время, потребовавшееся для выполнения перечисленных операций (некоторые вспомогательные обработчики, работавшие менее 10 с, не были учтены).

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. Эта информация полезна не только для детекторов переполнения буфера, но и для любого чувствительного к путям детектора. В контексте детекторов переполнения буфера можно заметить, что наиболее затратными операциями в общем являются слияние $\mathcal{V}$ для целых чисел (т.к. она вызывается очень часто и включает вычисление новых условий) и

создание резюме для $\mathcal{V}$ для целых чисел (т.к. упрощение условий затратно). Наиболее долго в рамках одного запуска выполняется операция создания резюме для $\mathcal{V}$ в особенности для строк по той же причине. Также затратной по времени является проверка одного доступа, т.к. она предполагает вызов решателя (частично это скомпенсировано быстрой проверкой несовместности).

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»).

Заключение

В работе предложены методы межпроцедурного поиска ошибок доступа к буферу на основе статического символьного выполнения с объединением состояний, обладающие контекстной чувствительностью и чувствительностью к путям, которые могут быть использованы во время разработки больших программных систем.

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.

В будущем предложенные методы обнаружения некорректного доступа планируется расширить для поддержки операций над коллекциями языка C++, а также распространить их на другие языки программирования (Java). Подход к анализу длин строк также может быть применён для анализа размера коллекций с целью получения более точных условий для чувствительного к путям анализа. Одним из перспективных способов доработки предложенного метода представляется интеграция решателей с поддержкой строк (Z3str2, Z3str3, CVC4 и т.п.) для получения точных условий для тех случаев, когда текущая абстракция (длина строки) является слишком грубой.

Благодарности

В заключение хотелось бы выразить глубокую признательность моему научному руководителю Андрею Белеванцеву, который консультировал и направлял меня в ходе исследования, оказывал колоссальную поддержку на протяжении всего процесса подготовки диссертации, и с которым очень приятно и интересно работать. Я также признательна директору института Арутюну Ишхановичу Аветисяну и руководителю отдела компиляторных технологий Сергею Суреновичу Гайсаряну, благодаря которым эта работа стала возможной. Большое спасибо всей команде Svace за тёплую и дружескую рабочую атмосферу и в частности Алексею Бородину за терпеливые ответы на возникающие вопросы по реализации. Выражаю свою признательность Михаилу Соловьёву и Валерию Савченко за обсуждение текста диссертации, которое много способствовало к его улучшению. И в первую очередь я благодарна своей семье и главным образом моей маме Евгении Аркадьевне Дудиной, которая установила эту высокую планку и горячо поддерживала меня на протяжении всего пути.

Список литературы

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.
9. *One A.* Smashing the Stack for Fun and Profit // Phrack. — 1996. — Nov. — Vol. 7, no. 49.
10. The Heartbleed Bug [Электронный ресурс]. — URL: http://heartbleed.com/ (дата обр. 17.08.2018).

11. NDV – CWE Over Time [Электронный ресурс]. — URL: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time (дата обр. 01.12.2018).

12. ISO/IEC 9899:2018 Standard. Information technology – Programming languages – C. — International Organization for Standardization, 2018. — URL: https://www.iso.org/standard/74528.html.

13. *Каушан В. В.* Поиск ошибок выхода за границы буфера в бинарном коде программ : автореф. дис. ... канд. техн. наук : 05.13.11. — М., 2007. — 26 с.

14. *Хопкрофт Д.*, *Мотвани Р.*, *Ульман Д.* Введение в теорию автоматов, языков и вычислений. — 2-е изд.:Пер. с англ. — М. : Издательский дом «Вильямс», 2008. — 528 с.

15. Formal Methods: Practice and Experience / J. Woodcock [et al.] // ACM Computing Surveys. — New York, NY, USA, 2009. — Oct. — Vol. 41, no. 4. — P. 1—36.

16. The ASTREÉ Analyzer / P. Cousot [et al.] // Programming Languages and Systems. — Springer Berlin Heidelberg, 2005. — P. 21—30.

17. Symbolic Model Checking without BDDs / A. Biere [et al.] // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 1999. — P. 193—207.

18. Counterexample-Guided Abstraction Refinement / E. Clarke [et al.] // Computer Aided Verification. — Springer Berlin Heidelberg, 2000. — P. 154—169.

19. *Kroening D.*, *Tautschnig M.* CBMC – C Bounded Model Checker // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 2014. — P. 389—391.

20. SLAM verification engine [Электронный ресурс]. — URL: https://www.microsoft.com/en-us/research/project/slam/ (дата обр. 21.12.2018).

21. Software Verification with BLAST / T. A. Henzinger [et al.] // Proceedings of the 10th International Conference on Model Checking Software. — Portland, OR, USA : Springer-Verlag, 2003. — P. 235—239. — (SPIN'03).

22. Modular Verification of Software Components in C / S. Chaki [et al.] // Proceedings of the 25th International Conference on Software Engineering. — IEEE Computer Society, 2003. — P. 385—395. — (ICSE '03).

23. Mona: Monadic second-order logic in practice / J. G. Henriksen [et al.] // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 1995. — P. 89—110.

24. Moving Fast with Software Verification / C. Calcagno [et al.] // NASA Formal Methods. — Cham : Springer International Publishing, 2015. — P. 3—11.

25. Infer static analyzer [Электронный ресурс]. — URL: https://fbinfer.com/ (дата обр. 21.09.2018).

26. Compositional Shape Analysis by Means of Bi-abduction / C. Calcagno [et al.] // Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Savannah, GA, USA : ACM, 2009. — P. 289—300. — (POPL '09).

27. POPL 2019 Most Influential Paper Award for research that led to Facebook Infer [Электронный ресурс]. — URL: https://research.fb.com/popl-2019-most-influential-paper-award-for-research-that-led-to-facebook-infer/ (дата обр. 29.01.2019).

28. Inferbo: Infer-based buffer overrun analyzer [Электронный ресурс]. — URL: https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/ (дата обр. 21.09.2018).

29. *Brotherston J.*, *Gorogiannis N.*, *Kanovich M.* Biabduction (and Related Problems) in Array Separation Logic // Automated Deduction – CADE 26. — Cham : Springer International Publishing, 2017. — P. 472—490.

30. ITS4: A static vulnerability scanner for C and C++ code / J. Viega [et al.] // Proceedings - Annual Computer Security Applications Conference, ACSAC. — 2000. — P. 257—267.

31. A first step towards automated detection of buffer overrun vulnerabilities / D. Wagner [et al.] // Network and Distributed System Security Symposium. — 2000. — P. 3—17.

32. *Kratkiewicz K. J.* Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code Kendra: tech. rep. / Harvard University. — 2005.

33. *Zitser M.*, *Lippmann R.*, *Leek T.* Testing static analysis tools using exploitable buffer overflows from open source code // ACM SIGSOFT Software Engineering Notes. — 2004. — Vol. 29, no. 6. — P. 97.

34. *Larochelle D.*, *Evans D.* Statically Detecting Likely Buffer Overflow Vulnerabilities // Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10. — Washington, D.C. : USENIX Association, 2001. — (SSYM'01).

35. *Evans D.* Using Specifications to Check Source Code: tech. rep. / Massachusetts Institute of Technology. — Cambridge, MA, USA, 1994.

36. An Empirical Study on Detecting and Fixing Buffer Overflow Bugs / T. Ye [et al.] // Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016. — 2016. — P. 91—101.

37. *Dor N.*, *Rodeh M.*, *Sagiv M.* CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. — 2003. — Vol. 2. — P. 155—167.

38. *Simon A.*, *King A.* Analyzing String Buffers in C // Algebraic Methodology and Software Technology. — Springer Berlin Heidelberg, 2002. — P. 365—380.

39. *Allamigeon X.* Static analysis of memory manipulations by abstract interpretation – Algorithmics of tropical polyhedra, and application to abstract interpretation: Theses / Allamigeon Xavier. — Ecole Polytechnique X, 11/2009.

40. *Jung Y.*, *Shin J.* Soundness by Static Analysis and False-alarm Removal by Statistical Analysis: Our Airac Experience // BUGS 2005 Workshop on the Evaluation of Software Defect Detection Tools. — 2005.

41. Filtering false alarms of buffer overflow analysis using SMT solvers / Y. Kim [et al.] // Information and Software Technology. — 2010. — Vol. 52, no. 2. — P. 210—219.

42. Sparrow static analyzer [Электронный ресурс]. — URL: http://ropas.snu.ac.kr/sparrow/ (дата обр. 23.12.2018).

43. Selective Context-sensitivity Guided by Impact Pre-analysis / H. Oh [et al.] // SIGPLAN Not. — 2014. — June. — Vol. 49, no. 6. — P. 475—484.

44. IKOS: A Framework for Static Analysis Based on Abstract Interpretation / G. Brat [et al.] // Software Engineering and Formal Methods / ed. by D. Giannakopoulou, G. Salaün. — Cham : Springer International Publishing, 2014. — P. 271—277.

45. Carraybound: Static Array Bounds Checking in C Programs Based on Taint Analysis / F. Gao [et al.] // Proceedings of the 8th Asia-Pacific Symposium on Internetware. — New York, NY, USA : ACM, 2016. — P. 81—90. — (Internetware ’16).

46. *Бородин А. Е.* Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++ : дис. ... канд. физ.-мат. наук : 05.13.11. — М., 2016. — 137 с.

47. *Le W.*, *Soffa M. L.* Marple // Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT ’08/FSE-16. — 2008. — P. 272—282.

48. S-looper: automatic summarization for multipath string loops / X. Xie [et al.] // Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015. — 2015. — P. 188—198.

49. *Cifuentes C.*, *Scholz B.* Parfait: designing a scalable bug checker // SAW ’08: Proceedings of the 2008 workshop on Static analysis. — New York, NY, USA : ACM, 2008. — P. 4—11.

50. *Xie Y.*, *Chou A.*, *Engler D.* ARCHER : Using Symbolic , Path-sensitive Analysis to Detect Memory Access Errors // Access. — 2003. — Vol. 28, no. 5. — P. 327—336.

51. *Xie Y.*, *Aiken A.* Scalable Error Detection Using Boolean Satisfiability // SIGPLAN Not. — New York, NY, USA, 2005. — Jan. — Vol. 40, no. 1. — P. 351—363.

52. Z3str2: an efficient solver for strings, regular expressions, and length constraints / Y. Zheng [et al.] // Formal Methods in System Design. — 2017. — June. — Vol. 50, no. 2. — P. 249—288.

53. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World / A. Bessey [et al.] // Commun. ACM. — New York, NY, USA, 2010. — Feb. — Vol. 53, no. 2. — P. 66—75.

54. Klocwork static analyzer [Электронный ресурс]. — URL: https://www.roguewave.com/products-services/klocwork/static-code-analysis (дата обр. 21.12.2018).

55. Polyspace bug finder [Электронный ресурс]. — URL: https://www.mathworks.com/products/polyspace-bug-finder.html (дата обр. 21.12.2018).

56. IAR C-STAT static analysis [Электронный ресурс]. — URL: https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/c-stat-static-analysis/ (дата обр. 21.12.2018).

57. *Chimdyalwar B.* Survey of Array out of Bound Access Checkers for C Code // Proceedings of the 5th India Software Engineering Conference. — Kanpur, India : ACM, 2012. — P. 45—48. — (ISEC '12).

58. Buffer overrun detection using linear programming and static analysis / V. Ganapathy [et al.] // Proceedings of the 10th ACM conference on Computer and communication security - CCS '03. — 2003. — P. 345—354.

59. Juliet Test Suite v1.2 for C/C++. User Guide / Center for Assured Software National Security Agency. — 9800 Savage Road Fort George G. Meade, MD 20755 - 6738, 12/2012.

60. Common Weakness Enumeration [Электронный ресурс]. — URL: https://cwe.mitre.org/index.html (дата обр. 07.08.2018).

61. *Shiraishi S.*, *Mohan V.*, *Marimuthu H.* Test Suites for Benchmarks of Static Analysis Tools // Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). — IEEE Computer Society, 2015. — P. 12—15. — (ISSREW '15).

62. CVE security vulnerability database. Security vulnerabilities, exploits, references and more [Электронный ресурс]. — URL: https://www.cvedetails.com/index.php (дата обр. 08.04.2018).

63. *Kratkiewicz K.*, *Lippmann R.* A taxonomy of buffer overflows for evaluating static and dynamic software testing tools // Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics. — 2006. — Vol. 500. — P. 44.

64. Bugs as deviant behavior / D. Engler [et al.] // ACM SIGOPS Operating Systems Review. — 2001. — Vol. 35, no. 5. — P. 57—72.

65. *Кошелев В. К.* Формализация определения ошибок при статическом символьном выполнении // Труды Института системного программирования РАН. — 2016. — Т. 28. — С. 105—118.

66. *Cousot P.*, *Cousot R.*, *Logozzo F.* Precondition Inference from Intermittent Assertions and Application to Contracts on Collections // Verification, Model Checking, and Abstract Interpretation. — Springer Berlin Heidelberg, 2011. — P. 150—168.

67. Introduction to Algorithms, Third Edition / T. H. Cormen [et al.]. — 3rd. — The MIT Press, 2009.

68. *Brummayer R.*, *Biere A.* Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 2009. — P. 174—177.

69. *Babic D.*, *Hu A. J.* Calysto: Scalable and Precise Extended Static Checking // Proceedings of the 30th International Conference on Software Engineering. — New York, NY, USA : ACM, 2008. — P. 211—220. — (ICSE '08).

70. *Белеванцев А. А.* Многоуровневый статический анализ исходного кода для обеспечения качества программ : дис. ... д-ра физ.-мат. наук : 05.13.11. — М., 2017. — 229 с.

71. *Кошелев В. К.* Межпроцедурный статический анализ для поиска ошибок в исходном коде программ на языке C# : дис. ... канд. физ.-мат. наук : 05.13.11. — М., 2017. — 104 с.

72. Precise and compact modular procedure summaries for heap manipulating programs / I. Dillig [et al.] // ACM SIGPLAN Notices. Vol. 46. — ACM. 2011. — P. 567—577.

73. *Borodin A.*, *Belevantcev A.* A static analysis tool Svace as a collection of analyzers with various complexity levels // Proceedings of ISP RAS. — 2015. — Vol. 27. — P. 111—134.

74. The SMT-LIB Standard: "— Version 2.0: tech. rep. / C. Barrett [et al.]. — 2010.

75. *De Moura L.*, *Bjørner N.* Z3: An Efficient SMT Solver // Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — Budapest, Hungary : Springer-Verlag, 2008. — P. 337—340. — (TACAS'08/ETAPS'08).

76. Bug 121795 - Potential buffer overflow in PPTWriter::ImplWriteParagraphs [Электронный ресурс]. — URL: https://bugs.documentfoundation.org/show_bug.cgi?id=121795 (дата обр. 28.02.2019).

Приложение А

Результаты на пакете Juliet Test Suite 1.3

На графиках А.1–А.9 приводятся результаты разработанных детекторов переполнения буфера на наборах тестов из пакета Juliet Test Suite 1.3, соответствующих переполнению правой границы буфера (CWE 121, CWE 122, CWE 126). Все тесты каждого набора разбиты на т.н. «варианты ошибки» (functional variant, flaw type), что позволяет дать качественную оценку возможностям анализатора. Ложных срабатываний выдано не было, поэтому на графиках приводится покрытие каждого варианта ошибки срабатываниями рассматриваемых детекторов.

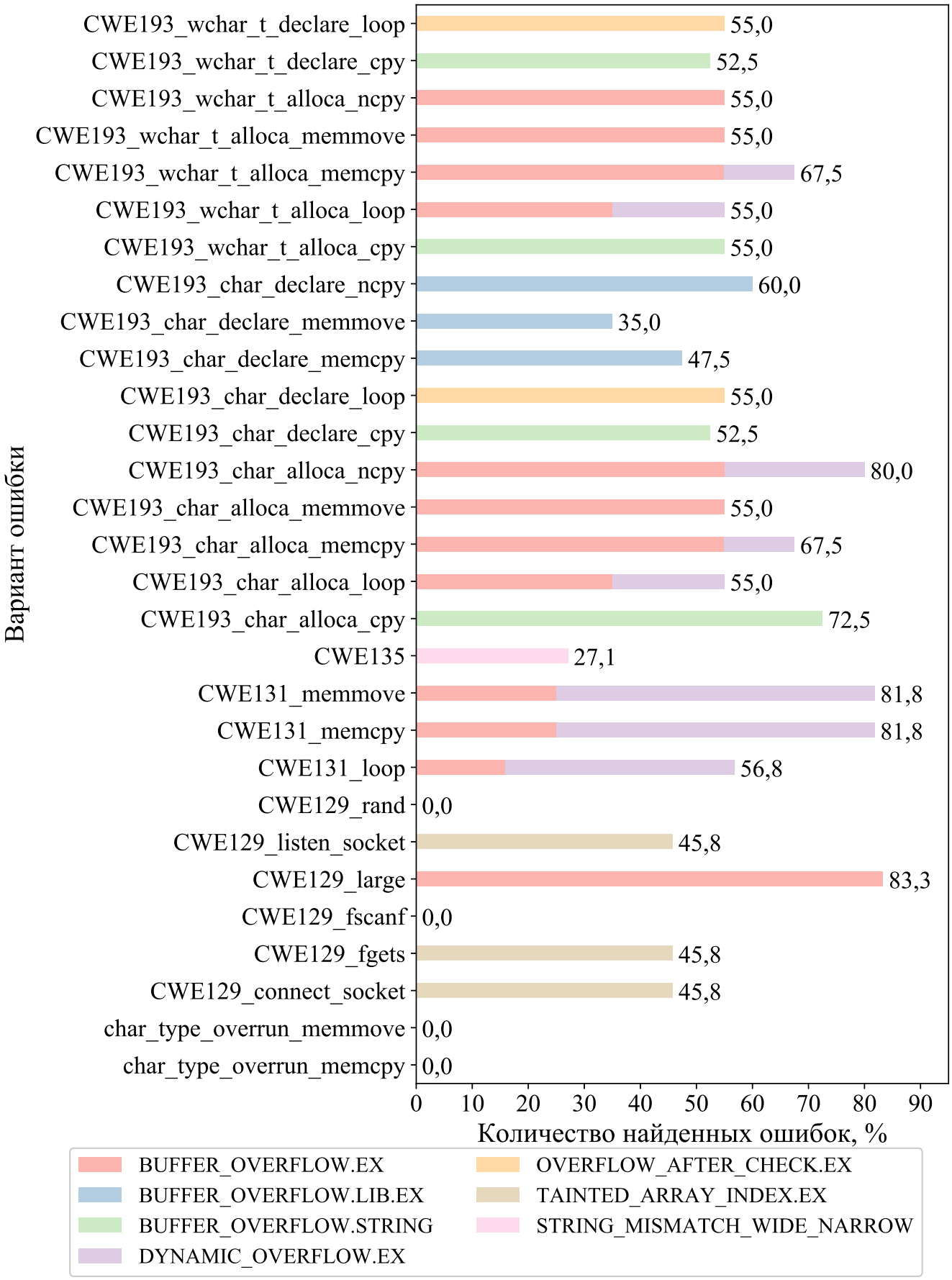

Рисунок А.1 — CWE 121 (1/4)

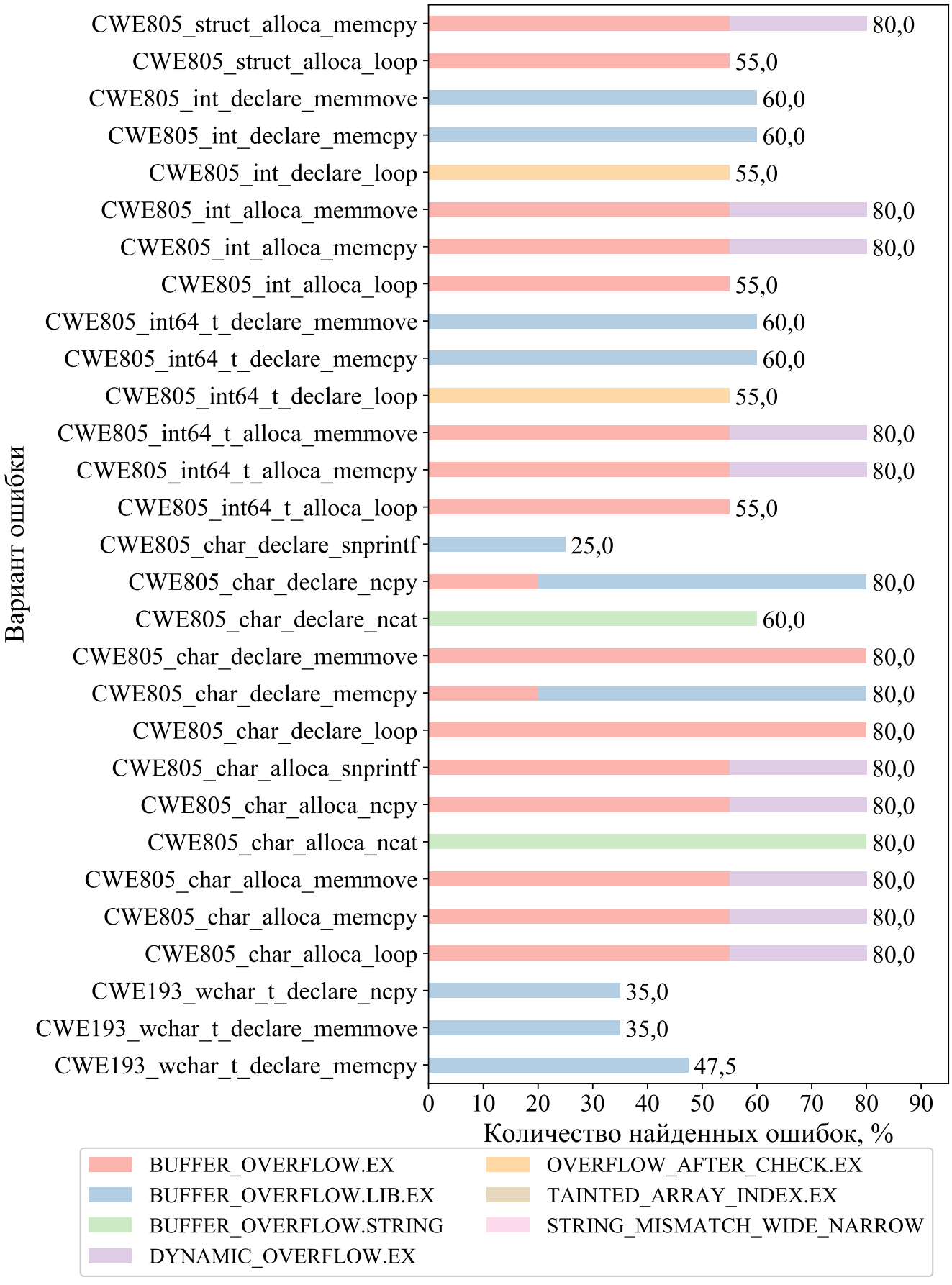

Рисунок А.2 — CWE 121 (2/4)

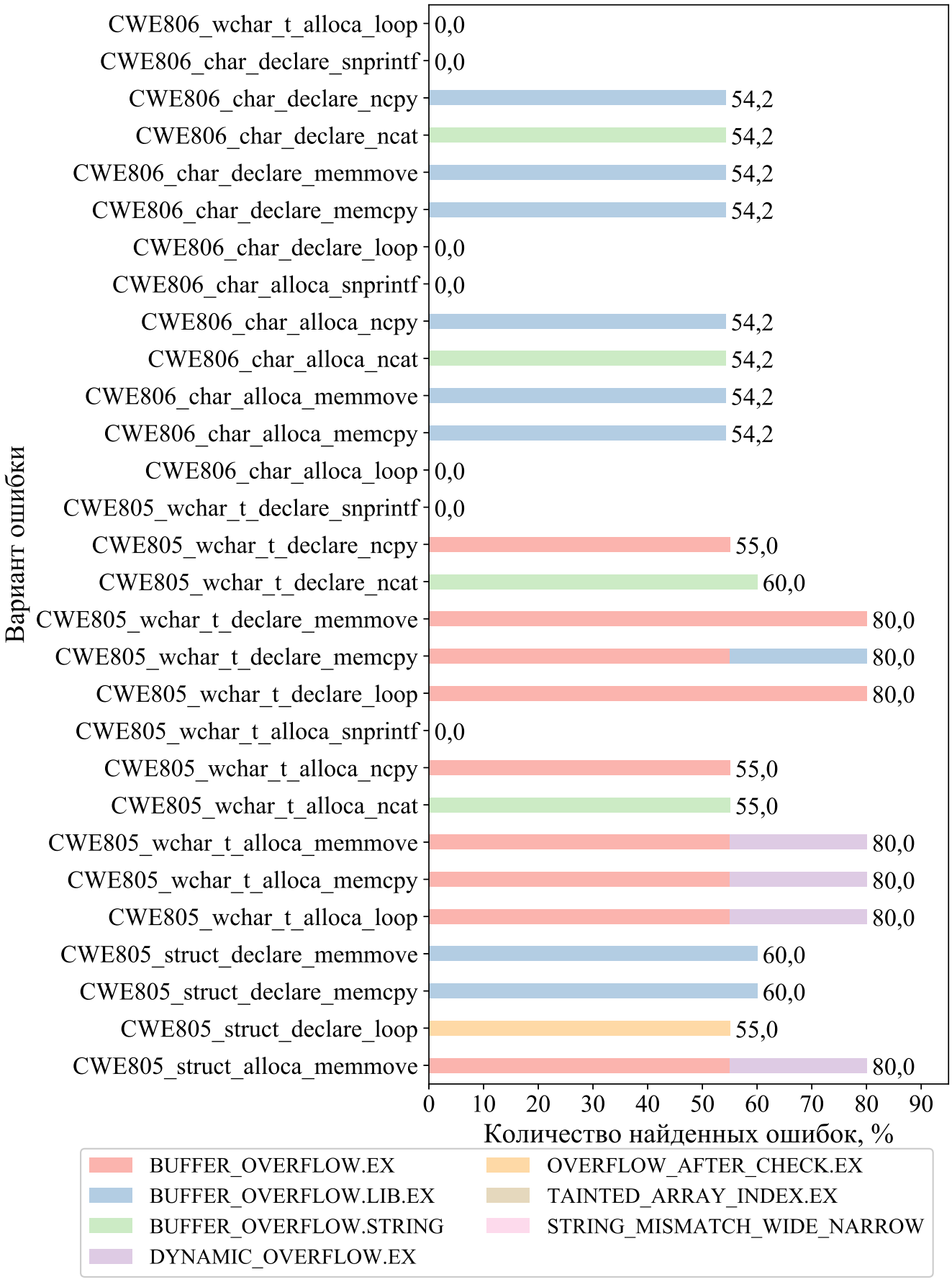

Рисунок А.3 — CWE 121 (3/4)

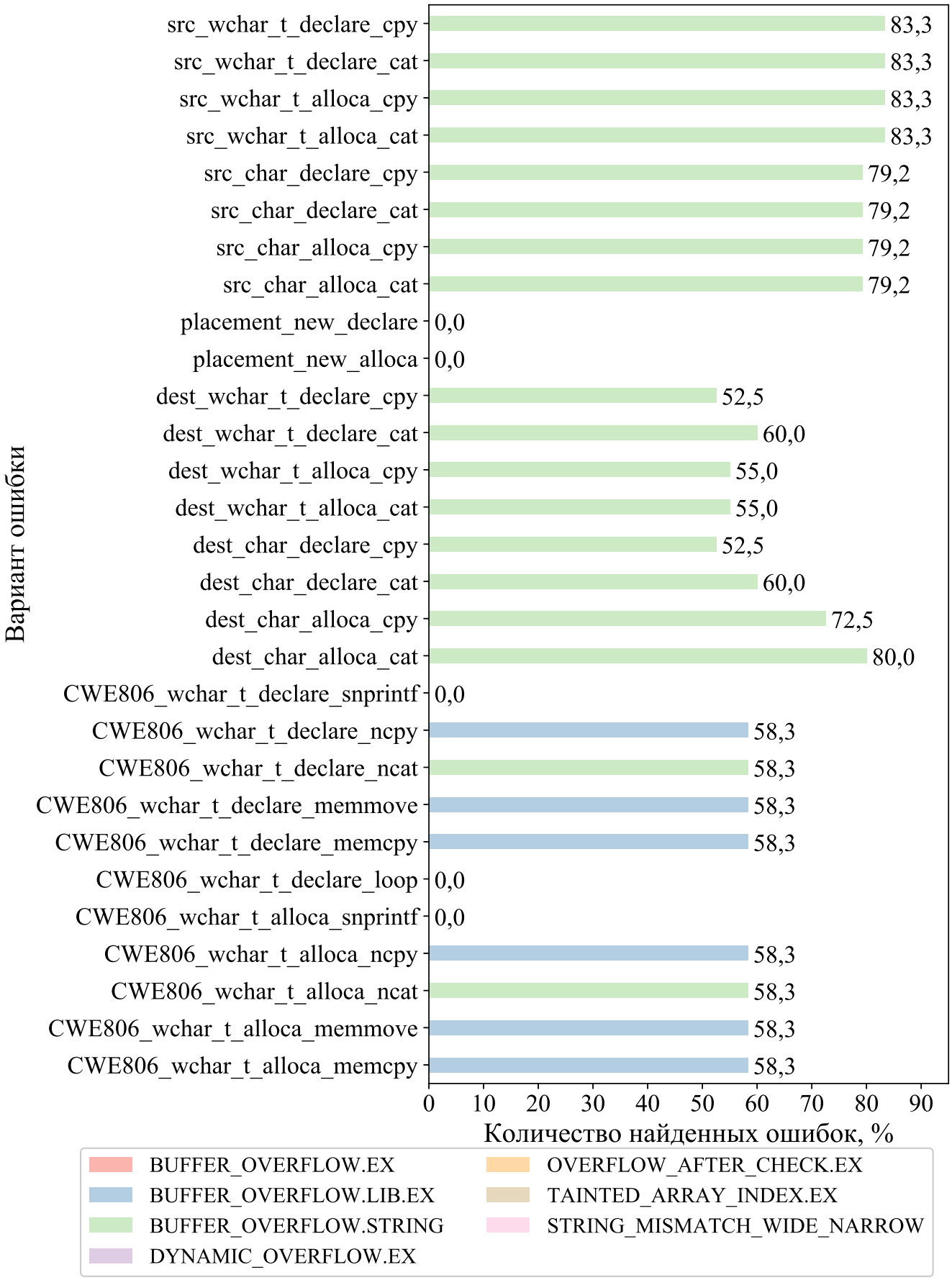

Рисунок А.4 — CWE 121 (4/4)

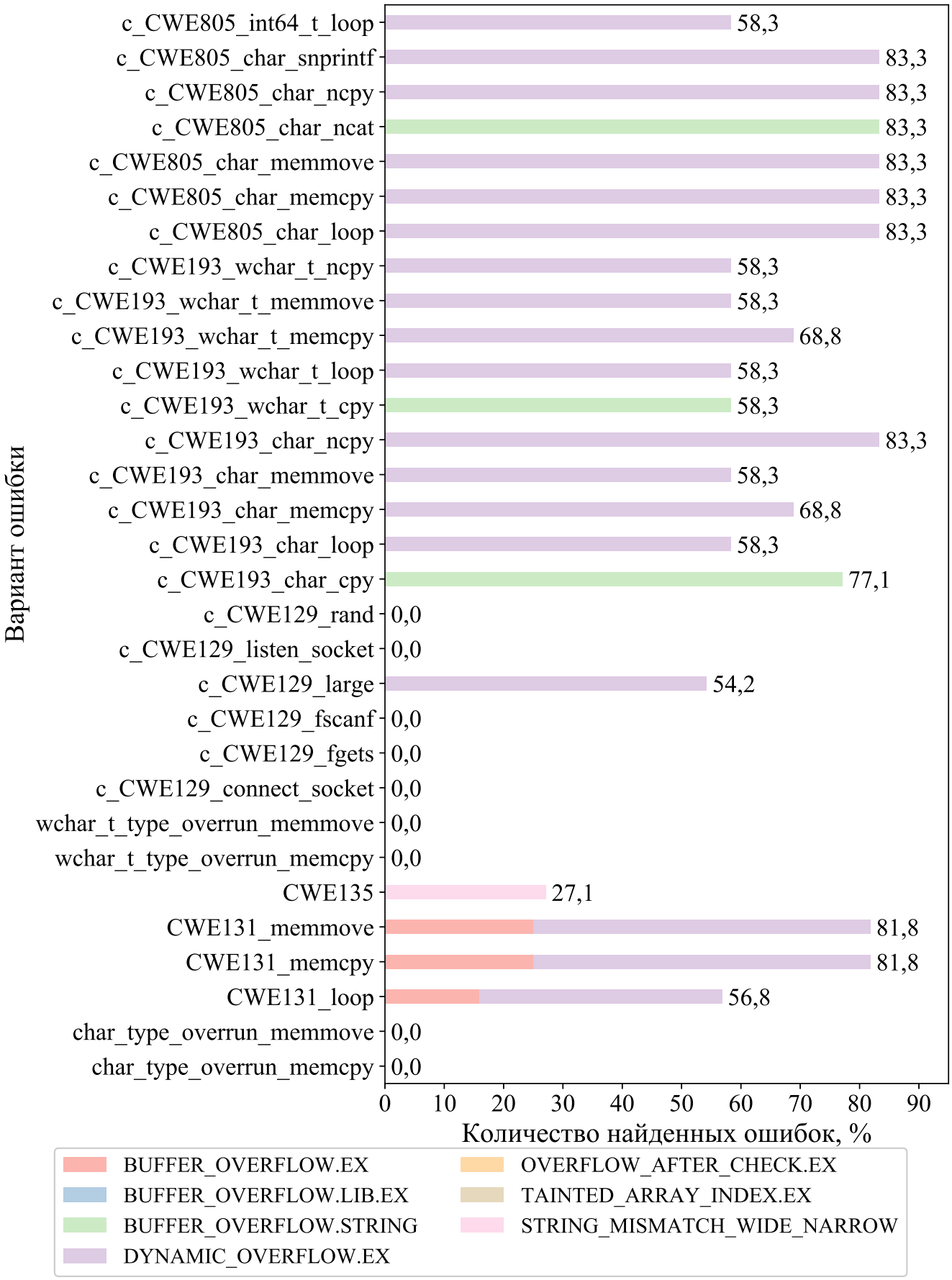

Рисунок А.5 — CWE 122 (1/4)

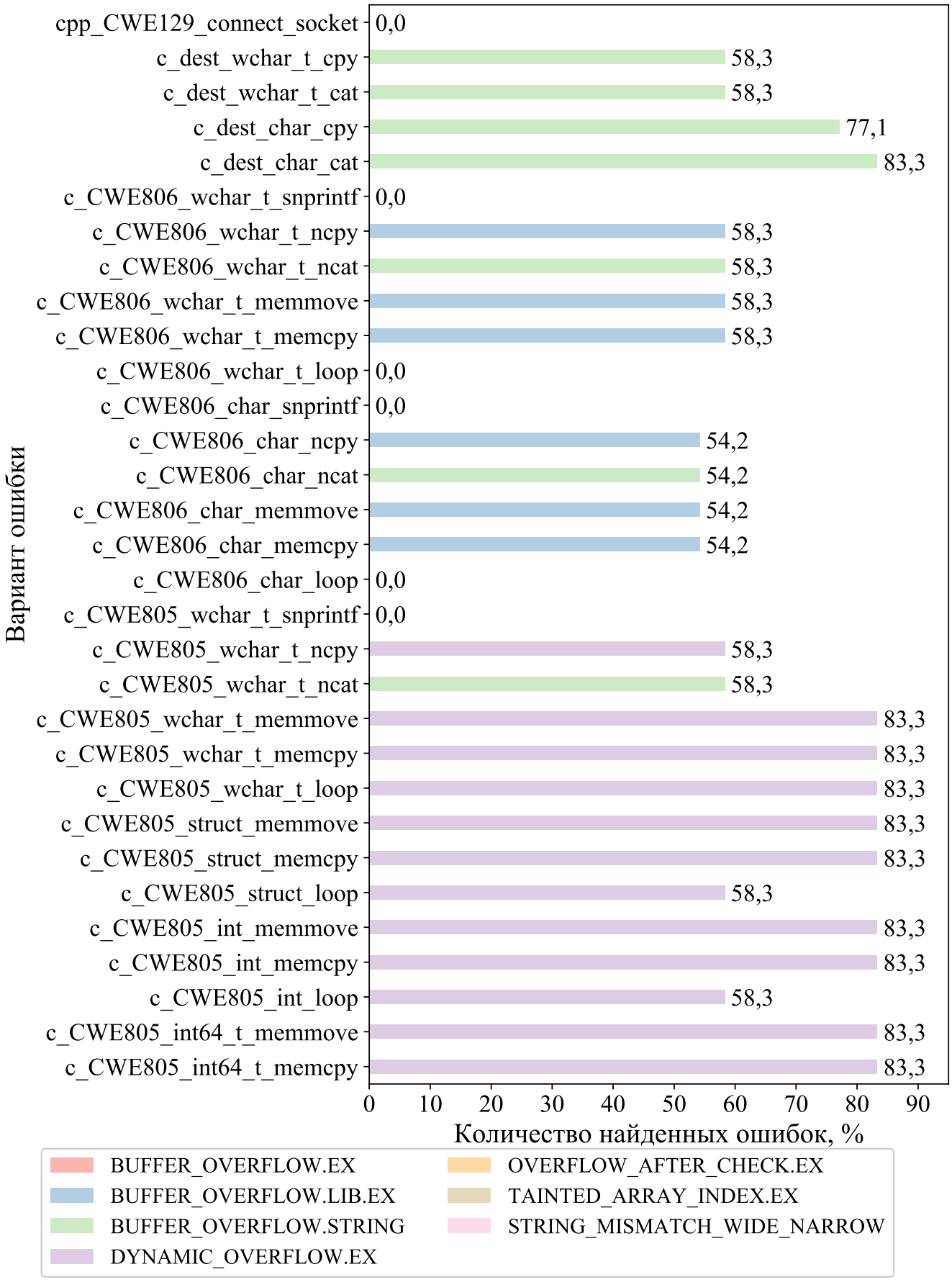

Рисунок А.6 — CWE 122 (2/4)

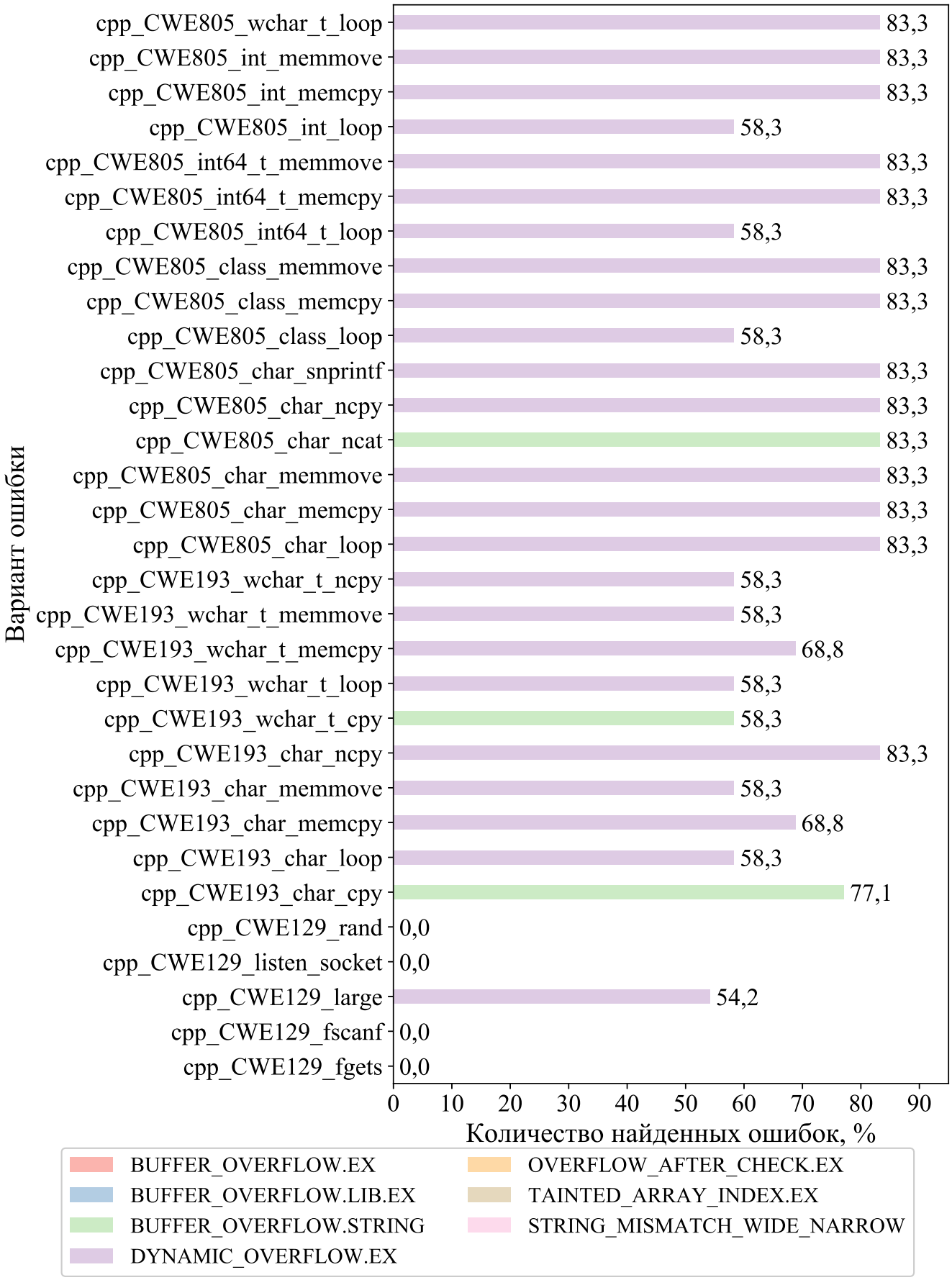

Рисунок А.7 — CWE 122 (3/4)

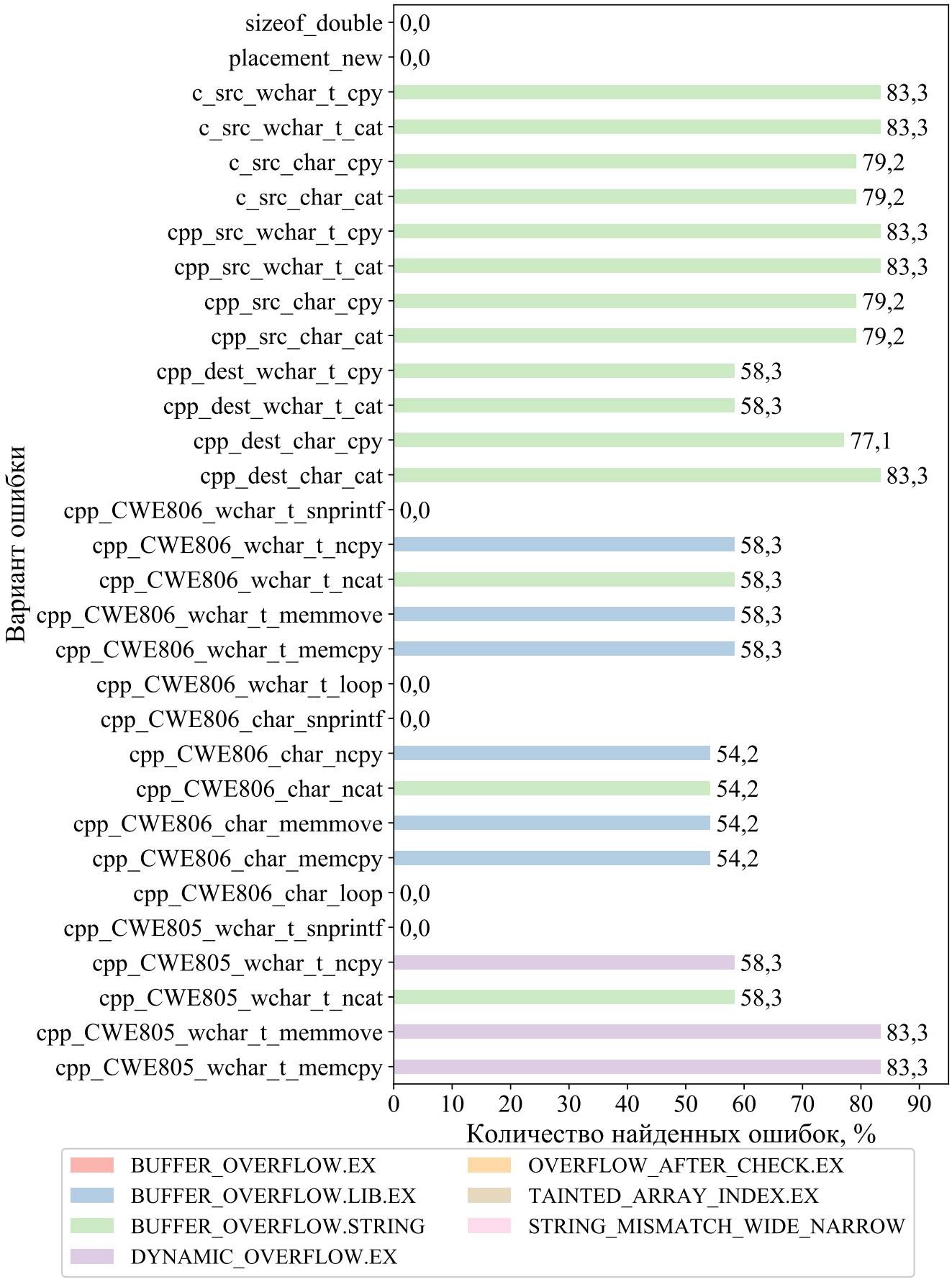

Рисунок А.8 — CWE 122 (4/4)

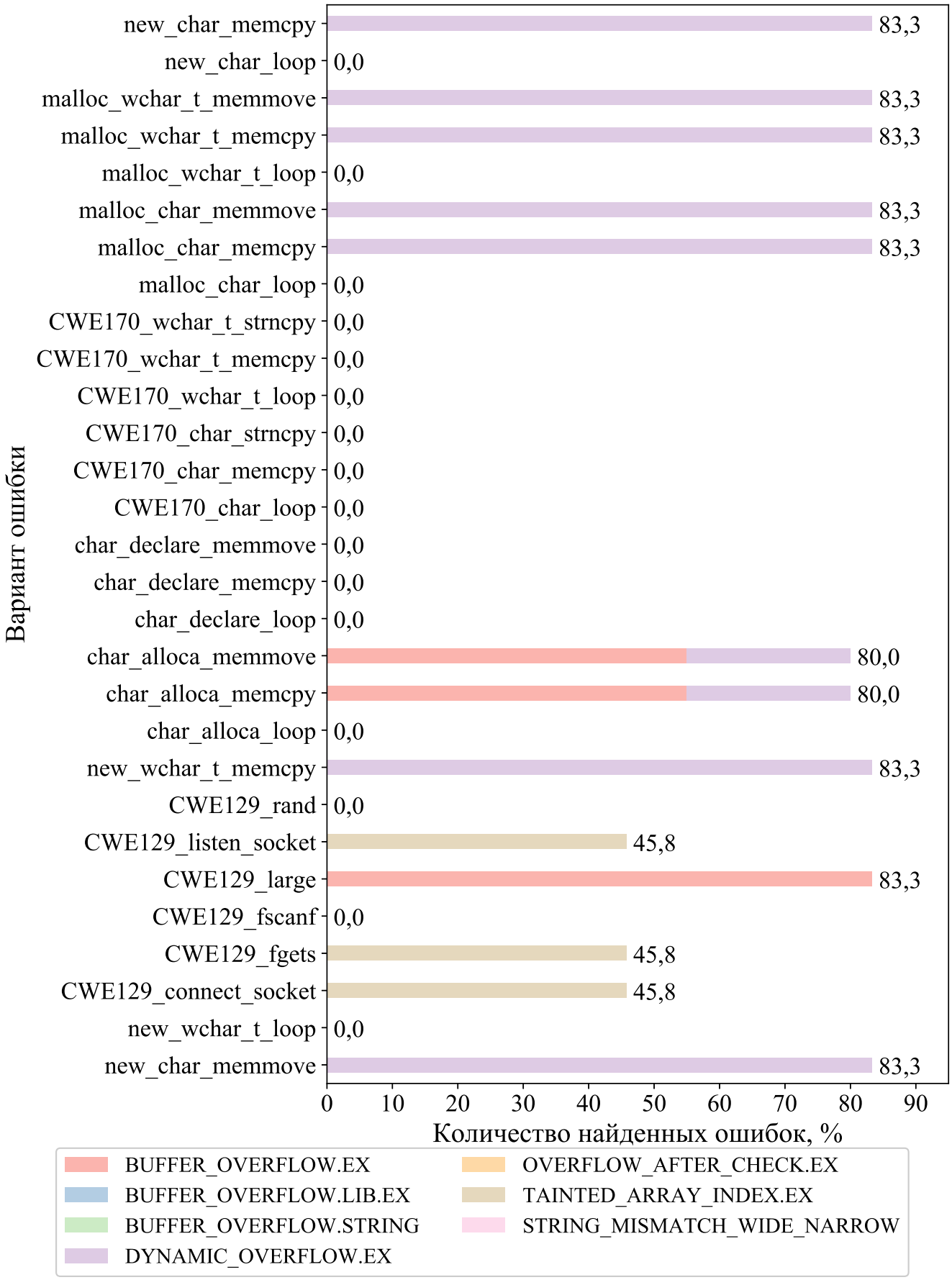

Рисунок А.9 — CWE 126