МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Э. БАУМАНА

На правах рукописи
УДК 004.4'2

Сидорин Алексей Васильевич

НАЗВАНИЕ ДИССЕРТАЦИОННОЙ РАБОТЫ

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

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
к.т.н., доцент
Романова Т. Н.

Москва — 2015

Оглавление

Введение

Для больших и сложных программно-технических комплексов полное покрытие всех путей выполнения программы становится невозможным, поскольку эта задача соотносится с проблемой останова. Ресурсы, выделенные на тестирование сложных программных комплексов, всегда ограничены, что приводит к необходимости рационального их использования. Проблема поиска подходящего компромисса между повышением надежности разрабатываемых программных средств и эффективным использованием ресурсов становится все актуальнее. Для обеспечения надёжности программных средств активно ведётся разработка новых эффективных методов и средств автоматического тестирования, позволяющих за реальное время предупредить и выявить как можно большее количество дефектов в программе. В настоящее время всё большее распространение получают инструменты, предназначенные для поиска дефектов в программном коде.

Обычно различают статический, динамический и смешанный анализ. Под статическим анализом понимают анализ программы, не требующий её непосредственного выполнения. Часть инструментов, таких, как Clang Static Analyzer [1], PVS-Studio [2], Cppcheck [3], Lint [4], исследует непосредственно код программы или структуры данные, строящиеся на его основе, — абстрактное синтаксическое дерево или граф потока управления. Другая часть инструментов статического анализа использует для анализа более низкоуровневое представление программы — скомпилированный объектный или промежуточный код (Coverity Prevent [5], Svace [6], FindBugs [7]). В отличие от статического анализа, для динамического анализа программы требуется её выполнение — на специальных входных данных, в виртуальной машине (Valgrind [8]), с использованием инструментации (AddressSanitizer [9], ThreadSanitizer [10], UndefinedBehaviorSanitizer), с использованием дополнительных библиотек или их подменой. Наконец, смешанный анализ представляет собой комбинацию статического и динамического анализа и используется в таких инструментах как Mayhem [11], KLEE [12], а также других автоматических генераторов контрпримеров.

Перечисленные виды анализа имеют свои достоинства и недостатки, в частности, различные виды анализа наиболее эффективны для поиска различных видов ошибок.

– Динамический анализ наиболее хорошо зарекомендовал себя для поиска ошибок, связанных с многопоточностью и управлением памятью, однако крайне затратен в случае больших проектов. Значительным недостатком динамического анализа является необходимость явного выполнения программы, что влечёт за собой необходимость подготовки входных данных (или их автогенерации), и быстрый рост длительности такого анализа с увеличением объёма проекта. Это также затрудняет интеграцию инструментов, использующих динамический анализ, в процесс разработки, что снижает шансы быстрого обнаружения ошибки.
– Статический анализ позволяет эффективно производить поиск различных видов дефектов: опечаток, некорректного использования типов, проблем безопасности, неопределённого или недокументированного поведения и многих других видов. Инструменты для выполнения статического анализа могут быть легко интегрированы в процесс разработки. При этом они могут быть использованы как индивидуальные вспомогательные инструменты разработки (например, для подсветки кода, содержащего потенциальную ошибку), так и в качестве инструментов, использующихся группой разработчиков (например, для развёртывания и интеграции в систему непрерывной сборки). Сравнительно небольшое время, затрачиваемое на анализ, вкупе с интеграцией в рабочий процесс позволяет быстро находить дефекты в разрабатываемых программах. Недостатком статических анализаторов является возможность выдачи ими некорректных сообщений об ошибках — ложных срабатываний (ошибок первого рода) и возможность пропуска имеющихся дефектов (ошибки второго рода), вероятность которых стараются снизить при разработке анализаторов. Вред от ошибок второго рода очевиден, но и ошибки первого рода играют не меньшую роль при оценке качества анализатора, поскольку их большое количество отвлекает разработчика на длительное время для просмотра ложных срабатываний, поэтому при большом количестве ложных срабатываний инструмент может стать практически непригодным для использования. Од-

нако при небольшом количестве ложных срабатываний польза от применения анализатора в виде снижения времени, затрачиваемого на обнаружение ошибки, быстро перевешивает недостаток в виде времени, затрачиваемого на просмотр ложных срабатываний.

Первоначально распространение у разработчиков получили инструменты, использующие методы на основе анализа синтаксического дерева программы и её графа потока управления. Преимуществами данных методов анализа программного кода являются:

- высокая скорость работы,
- незначительное потребление памяти,
- возможность его реализации в компиляторе для выполнения дополнительных проверок и предупреждения программиста о потенциально некорректном поведении компилируемого кода. Это становится возможным благодаря малому потреблению системных ресурсов, позволяющему лишь незначительно снижать производительность компилятора,
- возможность интеграции в среды разработки для осуществления анализа «на лету», непосредственно в процессе набора кода программистом, или в качестве дополнительного инструмента для быстрого обнаружения дефекта.

Аналогичные методы применяются в компиляторах для предупреждения программиста о потенциально некорректном поведении программы, поскольку и синтаксическое дерево, и граф потока управления являются основными структурами данных, с которыми работает компилятор. Однако проверка, включаемая в состав компилятора, должна исключать возможность ложных срабатываний, т. е. являться консервативной. Инструменты же статического анализа могут включать также и неконсервативные проверки, с возможностью выдачи ложных срабатываний.

Данные методы могут обнаруживать лишь очень узкие классы дефектов в программном коде: простые ошибки, затрагивающие лишь несколько операторов, расположенных в пределах одной функции. Это может быть простейший поиск использования неинициализированных переменных, ошибок при преобразовании типов, потенциально лишние операции, а также другие дефекты, для поиска которых не требуется анализировать циклы и условные переходы. При наличии циклов и переходов в анализируемой функции эффективность ви-

дов анализа, нечувствительных к путям выполнения, резко падает, поскольку данные методы позволяют корректно определить достижимость одних операторов из других операторов при выполнении программы лишь в тривиальных случаях.

Значительно более ресурсоёмким, но и более подробным является анализ на основе обхода путей выполнения программы. Основы этих методов были заложены ещё в 70-х годах. Метод символьного выполнения был предложен Джеймсом Кингом в 1976 году [13]. В основе метода лежит идея разбиения входных данных на классы эквивалентности в зависимости от встречаемых по пути выполнения условий. Метод абстрактной интерпретации, предложенный в 1977 году супругами Кузо [14], предполагает использовать абстрагирование данных и их анализ на основе алгебры решёток. Однако данные походы стали получать распространение только в последнее время. Это связано с увеличившейся мощностью компьютеров: время анализа растёт пропорционально количеству путей выполнения, что означает экспоненциальный рост времени анализа с увеличением размера программы. (Вообще говоря, абсолютно полный и точный анализ программы невозможен в связи с проблемой останова, независимо от применяемого подхода.) В отличие от базового анализа графа потока управления, анализ путей выполнения способен учитывать условия выполнения тех или иных ветвей программы, следствием чего являются преимущества данного вида анализа — его более высокая точность и способность покрыть намного больший класс дефектов. Такие методы, как абстрактная интерпретация и символьное выполнение, нашли применения в известных инструментах для поиска дефектов, например, Coverity SAVE, Clang Static Analyzer и многих других.

Одними из наиболее актуальных целевых языков для статического анализа традиционно являются языки C и C++. Причин для этого несколько. Во-первых, это связано с большим количеством видов потенциальных ошибок, которые может допустить программист, ведущий разработку с использованием этих языков. Наиболее специфичными среди таких ошибок являются ошибки, связанные с неправильной работой с указателями — переполнение буфера, обращение к неинициализированной памяти или к памяти по некорректному адресу. Во-вторых, стандарты языков трактуют достаточно большое количество ситуаций как не имеющих определённого поведения (например, порядок вычисления

аргументов функций может быть произвольным), что, с одной стороны, позволяет компилятору проводить более глубокие оптимизации и получить наибольшую скорость выполнения результирующего кода, но, с другой стороны, требует от программиста повышенного внимания в процессе написания кода программы для учёта этих особенностей. В-третьих, эти языки являются одними из самых распространённых и известных, с их использованием было разработано и продолжает создаваться большое количество как системного, так и прикладного программного обеспечения. Кроме того, язык C является практически единственным выбором при разработке низкоуровневых компонентов, например, компонентов операционных систем и драйверов, что также предъявляет повышенные требования к качеству программного кода.

Целью данной работы является разработка метода межпроцедурного межмодульного анализа крупных программных комплексов, разработанных с использованием языков C и C++, способного осуществлять анализ проектов масштаба ОС Android и ОС Tizen за приемлемое время и обеспечивающего достаточное покрытие путей выполнения программы.

Для достижения поставленной цели необходимо было решить следующие задачи:

1. Разработать метод межпроцедурного анализа программ с высокой масштабируемостью, пригодный для анализа крупных программных проектов, а также расширяемый на различные классы проверок
2. Разработать метод межмодульного анализа программ, разработанных с использованием языков C и C++
3. Разработать метод отображения результатов анализа при использовании разработанного метода межпроцедурного анализа
4. Реализовать разработанные методы с использованием инфраструктуры статического анализатора Clang Static Analyzer
5. Осуществить проверку разработанных методов на реальных программных проектах
6. Провести анализ разработанного метода на предмет масштабируемости и качества анализа с учётом результатов, полученных при проверке реальных программных проектов.

Основные положения, выносимые на защиту:

1. Метод межпроцедурного анализа программ на основе резюме для метода символьного выполнения для программ, разработанных с использованием языков C и C++
2. Метод межмодульного анализа программ, разработанных с использованием языков C и C++, архитектура анализатора, эвристики, связанные с объединением синтаксических деревьев различных модулей трансляции
3. Метод построения отчёта о дефекте при использовании метода резюме для метода символьного выполнения

Научная новизна:

1. Разработан и подробно описан метод межпроцедурного анализа программ на основе резюме для метода символьного выполнения для программ, разработанных с использованием языков C и C++
2. Разработан и подробно описан метод межмодульного анализа программ, разработанных с использованием языков C и C++ для статического анализатора, использующего в качестве входных данных непосредственно исходный код программы
3. Разработан метод построения отчёта о дефекте при использовании метода резюме для метода символьного выполнения
4. Вышеперечисленные методы реализованы с использованием инфраструктуры статического анализатора Clang Static Analyzer и апробированы на реальных проектах (ОС Android)

Научная и практическая значимость. Предложена новая модификация метода межпроцедурного анализа на основе резюме для метода символьного выполнения, а также метод межмодульного анализа и архитектура анализатора, использующего в качестве входных данных непосредственно исходный код программы. Разработанные методы применимы для проектов масштаба операционных систем и их наборов пользовательских приложений. Программное обеспечение, реализующее разработанные методы, внедрено в Samsung Electronics и используется для анализа исходного кода ПО различного назначения, в частности, мобильных приложений и операционных систем, телевизионное ПО, ПО медицинских систем и т. д.

Степень достоверности полученных результатов обеспечивается ... Результаты находятся в соответствии с результатами, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на: перечисление основных конференций, симпозиумов и т. п.

Личный вклад. Автор принимал активное участие ...

Публикации. Основные результаты по теме диссертации изложены в XX печатных изданиях [15–18], X из которых изданы в журналах, рекомендованных ВАК [15; 16], XX — в тезисах докладов [17; 18].

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и двух приложений. Полный объём диссертации составляет 96 страниц с 7 рисунками и 10 таблицами. Список литературы содержит 60 наименований.

1 Методы статического межпроцедурного анализа программ

1.1 Используемая терминология

1.2 Метод анализа программ с помощью символьного выполнения

В данной работе исследуется метод символьного выполнения [13], применяемый для анализа путей выполнения программ. Этот метод подразумевает абстрактное движение по путям программы, имитирующее её выполнение в зависимости от входных данных, сопровождающееся изменением состояния программы в различных точках. Суть метода символьного выполнения заключается в разбиении множества входных данных на классы эквивалентности, что позволяет оперировать при анализе не отдельными входными значениями (число которых может быть очень большим и экспоненциально растёт в зависимости от количества входных аргументов) и их перебором, а целыми классами эквивалентности, число которых может оказаться и не конечным, но не превышает общее количество комбинаций отдельных входных значений. Однако, как правило, количество классов эквивалентности комбинаций входных данных оказывается значительно ниже числа всех возможных комбинаций входных данных, что резко увеличивает возможности анализатора по обработке путей выполнения.

Основной алгоритм символического выполнения [13] заключается в следующем.

1. При старте анализа функции входные значения её аргументов и внешних по отношении к ней переменных неизвестны, т. е. они потенциально могут принимать любые значения. Начальное состояние является корневым узлом специального графа — дерева выполнения программы.
2. Каждой неизвестной величине назначается абстрактное значение, называемое символьным значением.
3. Обработка операторов языка изменяет значения переменных программы, т. е. их символьные значения. Считается, что над символьными зна-

чениями уже определён необходимый набор операций для вычисления новых символьных значений на основе уже имеющихся. Выполнение каждого оператора добавляет узел к дереву выполнения программы с входящим ребром от предыдущего оператора.

4. При обработке условных операторов в дерево выполнения программы добавляется не один, а два узла: в первом узле условие выполняется, во втором — нет. Совокупность условий, при которых достижим данный узел графа выполнения программы, определяет класс эквивалентности входных данных программы или функции. Таким образом, каждый лист дерева выполнения программы соответствует классу эквивалентности входных данных, которая приводит программу в конечное состояние, обозначаемое данным листом.
5. Обычно каждое из условий, определяющих класс эквивалентности входных данных, можно представить в виде уравнения или неравенства. В результате наложения на путь выполнения программы нескольких условий в соответствие каждому классу эквивалентности входных данных ставится система уравнений или неравенств. Решениями этих систем уравнений являются множества реальных значений входных величин, при которых будут выполнены ветви выполнения программы. Если система является несовместной, т. е. не имеет ни одного решения, то путь выполнения, соответствующий этой системе, недостижим.
6. Анализатор производит обход всех путей получившегося дерева выполнения с целью поиска ситуаций, которые могли бы трактоваться как некорректное поведение. В случае обнаружения такой ситуации анализатор сообщает о дефекте и при этом указывает набор условий, при выполнении которых программа проявит некорректное поведение.

Данный алгоритм можно рассмотреть на следующем примере. Пусть имеется следующая программа чтения из файла на языке C:

```
1  int test(int a, int b) {
2    FILE *f = fopen("file.txt");
3    int result;
4    fscanf(f, "%d", &result);
5    fclose(f);
6    if (a == 0 && b > 2) {
```

```
 7      fclose(f);
 8      return 0;
 9    }
10    return result;
11  }
```

В результате выполнения этой программы будут достижимы три конечных состояния с уравнениями $a \neq 0, b \in [INT\_MIN, INT\_MAX]$; $a = 0, b \leqslant 2$; $a = 0, b > 2$. Соответствующий граф представлен на рис. 1.1.

В результате получены три пути с соответствующими ограничениями. Теперь анализатор просматривает каждый из этих путей и обнаруживает дефект: на одном из путей файл `f` закрывается дважды, что приводит к неопределённому поведению. Этому пути соответствует система $a = 0, b > 2$. Таким образом дефекту сопоставляется множество условий, при котором он проявляется при выполнении программы.

Основным преимуществом метода символьного выполнения является простота и очевидность концепции, на которой он основан: метод использует идею «симуляции» выполнения программы, так, как это делает программист. Метод символьного выполнения получил распространение не только в инструментах статического, но и смешанного анализа: так, хорошо зарекомендовали себя инструменты, использующие подход concolic testing [19] (символьно-конкретное — concrete+symbolic). Concolic testing — это метод поиска дефектов, осуществляющий генерацию тестовых данные, при использовании которых программа проявляет некорректное поведение, на основе символьного выполнения.

Наряду с преимуществами, метод имеет ряд недостатков. Так, существует проблема экспоненциального роста количества проходимых путей (path explosion), приводящая к проблемам с масштабируемостью метода. Есть также проблемы при моделировании циклов, поскольку зачастую количество итераций цикла точно неизвестно — оно также является символьной величиной. Тем не менее, метод символьного выполнения активно применяется, в том числе, целым набором широко используемых инструментов анализа программ. Таким образом, разработка подходов для улучшения данного метода является актуальной и практически важной задачей.

Основные проблемы масштабируемости метода связаны с двумя факторами.

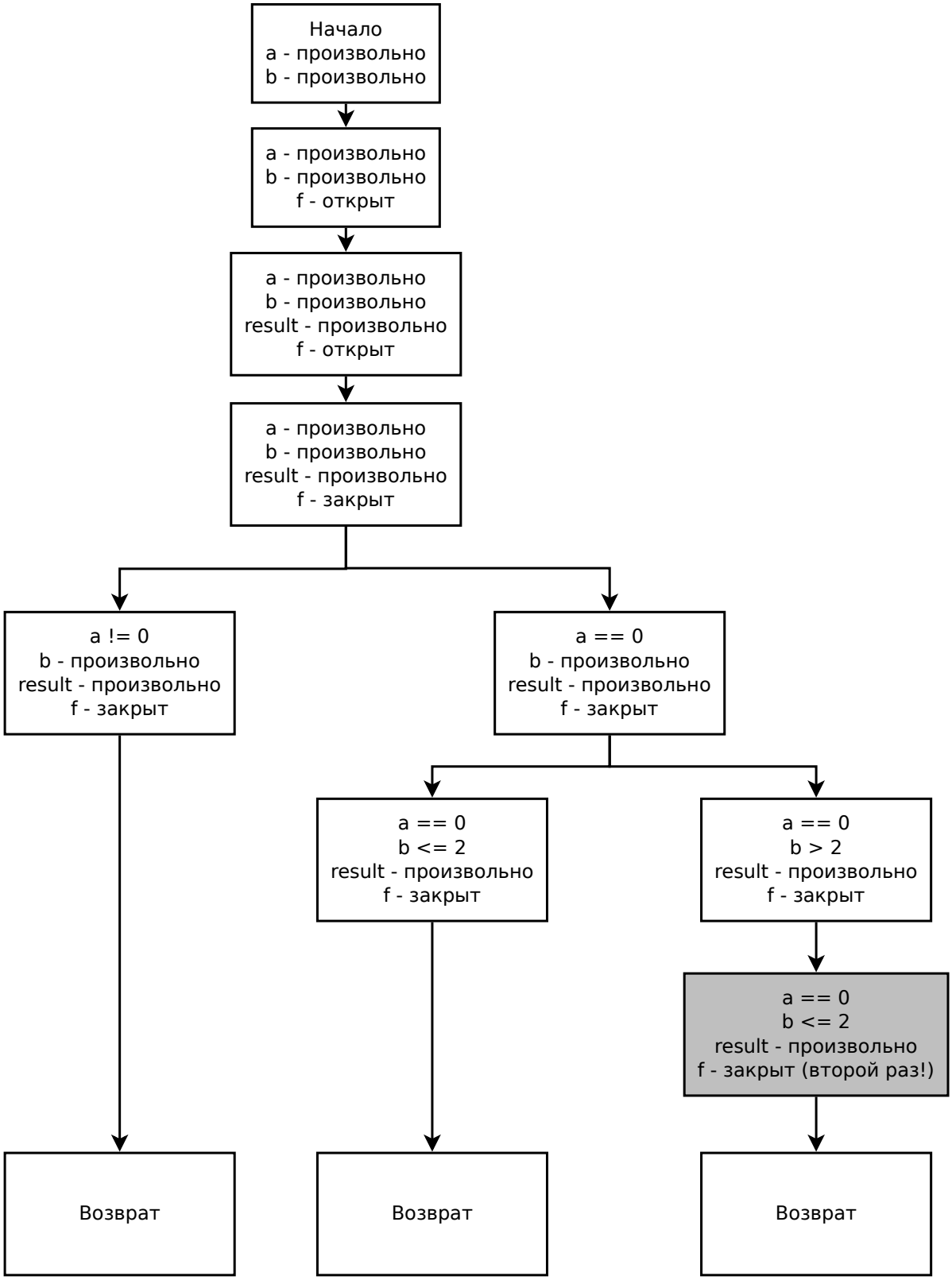

Рисунок 1.1 — Граф выполнения программы чтения из файла

1. При моделировании циклов время анализа линейно зависит от количества итераций, проходимых программой при выполнении цикла. Даже если число итераций известно, но велико, анализ программы выполняется длительное время, резко возрастающее при наличии в программе вложенных циклов. При использовании смешанного анализа обычно эту проблему анализом выполнения программы для установления реального количества итераций. При статическом анализе наиболее распространённым решением является ограничение количества итераций циклов каким-либо максимальным константным значением. Этот подход позволяет ограничить время анализа, однако не решает проблему роста времени анализа при наличии вложенных циклов. Кроме того, это ограничение приводит к потере точности моделирования, что, в свою очередь, приводит к ложным срабатываниям или отсутствию срабатываний в тех случаях, когда они ожидаются.
2. Время анализа быстро растёт при использовании межпроцедурного анализа.

Отличие межпроцедурного анализа (МПА) от внутрипроцедурного (ВПА) заключается в том, что анализатор позволяет использовать доступные определения пользовательских функций для моделирования эффектов их вызовов. МПА используется для решения двух основных проблем, связанных с использованием внутрипроцедурного анализа. Во-первых, межпроцедурный анализ позволяет определить эффект, оказываемый на состояние программы в результате вызова из анализируемой функции другой функции. В отсутствие межпроцедурного анализа вызов функции можно моделировать либо самостоятельно (с помощью спецификаций эффектов), либо приближённо. Первый подход, как правило, используется для функций, эффекты которых специфицированы. Таковы, например, функции различных публичных интерфейсов взаимодействия. Наиболее распространено такое моделирование для Posix API, встречаются также реализации, моделирующие вызовы Windows API. Кроме того, хорошими кандидатами на специфицирование являются функции, принадлежащие стандартной библиотеке языка, поскольку она, как правило, стандартизирована, реже — функции других распространённых библиотек (например, STL и др.).

Во-вторых, в случае, если полная спецификация функции недоступна, при внутрипроцедурном анализе может использоваться приближённое моделирование. В этом случае считается, что вызов функции может произвести любые действия с данными, которые доступны внутри функции (для языков, имеющих операции арифметики с указателями, например, C/C++, в общем случае можно считать, что программа может модифицировать любые данные), и вернуть произвольное значение. Для уточнения эффектов может использоваться анализ атрибутов доступной декларации функции, например, информация о модификаторах типов аргументов функции, атрибуты аргументов и самой функции. Так, например, функция, объявленная с GNU-атрибутом `__attribute__((pure))`, не имеет прав на изменение глобальной памяти и аргументов, а функция с атрибутом `__attribute__((noreturn))` никогда не вернёт управление в вызывающую функцию. Могут также использоваться различные эвристики. Вместе с тем, приближённое моделирование может решить проблему анализа лишь частично. Из-за невозможности оценки влияния вызова на состояние программы анализатор может сделать некорректные выводы о текущем состоянии выполнения программы, что может привести как к ложным срабатываниям анализатора (ошибка первого рода), так и к отсутствию срабатывания в условиях, когда анализатор должен выдавать диагностическое предупреждение (ошибка второго рода).

1.3 Ссылки

Сошлёмся на библиографию. Одна ссылка: [15, с. 54] [16, с. 36]. Две ссылки: [15; 16]. Много ссылок: [17; 18; 20, с. 54] [17; 18; 20–34]. И ещё немного ссылок: [35–47]. [48–56]

Сошлёмся на приложения: Приложение А, Приложение Б.2.

Сошлёмся на формулу: формула (1.1).

Сошлёмся на изображение: рисунок 2.4.

1.4 Формулы

Благодаря пакету *icomma*, LaTeX одинаково хорошо воспринимает в качестве десятичного разделителя и запятую (3,1415), и точку (3.1415).

1.4.1 Ненумерованные одиночные формулы

Вот так может выглядеть формула, которую необходимо вставить в строку по тексту: $x \approx \sin x$ при $x \to 0$.

А вот так выглядит ненумерованая отдельностоящая формула с подстрочными и надстрочными индексами:

$$(x_1 + x_2)^2 = x_1^2 + 2x_1x_2 + x_2^2$$

При использовании дробей формулы могут получаться очень высокие:

$$\frac{1}{\sqrt(2) + \dfrac{1}{\sqrt{2} + \dfrac{1}{\sqrt{2} + \cdots}}}$$

В формулах можно использовать греческие буквы:

$$\alpha\beta\gamma\delta\epsilon\varepsilon\zeta\eta\theta\vartheta\iota\kappa\lambda m u\nu\xi\pi\varpi\rho\varrho\sigma\varsigma\tau\upsilon\phi\varphi\chi\psi\omega\Gamma\Delta\Theta\Lambda\Xi\Pi\Sigma\Upsilon\Phi\Psi\Omega$$

1.4.2 Ненумерованные многострочные формулы

Вот так можно написать две формулы, не нумеруя их, чтобы знаки равно были строго друг под другом:

$$f_W = \min\left(1, \max\left(0, \frac{W_{soil}/W_{max}}{W_{crit}}\right)\right),$$
$$f_T = \min\left(1, \max\left(0, \frac{T_s/T_{melt}}{T_{crit}}\right)\right),$$

Выровнять систему ещё и по переменной x можно, используя окружение `aligned`at из пакета `amsmath`. Вот так:

$$|x| = \begin{cases} x, & \text{если } x \geqslant 0 \\ -x, & \text{если } x < 0 \end{cases}$$

Здесь первый амперсанд означает выравнивание по левому краю, второй — по x, а третий — по слову «если». Команда `\quad` делает большой горизонтальный пробел.

Ещё вариант:

$$|x| = \begin{cases} x, \text{если } x \geqslant 0 \\ -x, \text{если } x < 0 \end{cases}$$

Можно использовать разные математические алфавиты:

$$\mathcal{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$
$$\mathfrak{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$
$$\mathbb{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$

Посмотрим на систему уравнений на примере аттрактора Лоренца:

$$\begin{cases} \dot{x} = & \sigma(y - x) \\ \dot{y} = & x(r - z) - y \\ \dot{z} = & xy - bz \end{cases}$$

А для вёрстки матриц удобно использовать многоточия:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}$$

1.4.3 Нумерованные формулы

А вот так пишется нумерованая формула:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \tag{1.1}$$

Нумерованых формул может быть несколько:

$$\lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k^2} = \frac{\pi^2}{6} \tag{1.2}$$

Впоследствии на формулы (1.1) и (1.2) можно ссылаться.

Сделать так, чтобы номер формулы стоял напротив средней строки, можно, используя окружение `multlined` (пакет `mathtools`) вместо `multline` внутри окружения `equation`. Вот так:

$$\begin{aligned} &1 + 2 + 3 + 4 + 5 + 6 + 7 + \cdots + \\ &+ 50 + 51 + 52 + 53 + 54 + 55 + 56 + 57 + \cdots + \\ &+ 96 + 97 + 98 + 99 + 100 = 5050 \end{aligned} \tag{1.3}$$

2 Межпроцедурный анализ на основе резюме для метода символьного выполнения

2.1 Математическая модель. Разработанный алгоритм метода резюме для символьного выполнения

Рассмотрим основные источники затрат при встраивании и при использовании резюме. При встраивании функций вызываемая функция анализируется каждый раз при её вызове. При этом в случае контекстно-чувствительного анализа функция анализируется не полностью: анализируются лишь те пути выполнения, которые являются достижимыми при контексте на момент вызова. Суммарное время, затраченное на анализ функции при допустимой степени вложенности, равной 1, можно вычислить по формуле:

$$T_{\text{встраивания}} = \sum_{i=0}^{n} t_i \tag{2.1}$$

где i — номер вызова, t_i — время, затраченное на анализ i-го вызова (с учётом контекста). С учётом того, что сама вызываемая функция также анализируется отдельно, формула приобретает вид:

$$T_{\text{встраивания полное}} = T_{\text{анализа}} + \sum_{i=0}^{n} t_i \tag{2.2}$$

При использовании подхода, основанного на резюме, временные затраты вычисляются следующим образом. Вызываемая функция анализируется один раз, но полностью (независимо от вида анализа). Однократный характер также носят затраты, связанные с составлением резюме функции. Применение резюме выполняется в каждой точке вызова функции. Таким образом, общие затраты рассчитываются по формуле:

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + \sum_{i=0}^{n} t_{i\text{применения}} \tag{2.3}$$

что, с учётом примерного равенства времён применения резюме (поскольку резюме имеет не меняющийся размер), приближённо равно

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + nt_{i\text{применения}} \tag{2.4}$$

Таким образом, выигрыш от применения резюме будет получен при выполнении следующего соотношения:

$$T_{\text{сбора}} + nt_{i\text{применения}} < \sum_{i=0}^{n} t_i \tag{2.5}$$

откуда следует, что для получения ускорения необходимо выполнение соотношения

$$t_{\text{ср. применения}} < t_{\text{ср.}} \tag{2.6}$$

Пусть $s_1, \ldots, s_n$ — некоторая последовательность операторов программы. Каждый оператор имеет набор эффектов, который он оказывает на состояние выполнения программы. Таким образом, каждый оператор программы представляет собой передаточную функцию: $p_i = s_i(p_{i-1})$, где p_{i-1} — состояние программы непосредственно перед выполнением оператора, p_i — состояние программы непосредственно после выполнением оператора s_i (и перед выполнением оператора s_{i+1}). Тогда в результате выполнения всего блока операторов программа из начального состояния p_0 перейдёт в состояние p_n: $p_n = s_n(s_{n-1}(\ldots s_i(\ldots(s_1(p_0))\ldots))$. Тогда суммарный эффект последовательности операторов можно представить в виде композиции их передаточных функций:

$$s = s_1 \circ s_2 \circ \ldots \circ s_n \tag{2.7}$$

Данная формула справедлива для последовательности операторов без переходов, то есть для базовых блоков программы. Кроме того, формула справедлива для последовательности операторов, содержащей безусловный переход, поскольку такая последовательность также представляет собой путь выполнения без ветвлений. Однако при наличии условных переходов в блоке последовательности операторов, оказывающих эффект на выполнение программы, могут различаться. Это означает, что суммарный эффект выполнения блока зависит от пути выполнения внутри блока, а следовательно, и от значения выражения в условии.

Пусть c_j — условие, принадлежащее анализируемому блоку, $0 \leqslant j \leqslant m$, в ветках `if` и `else` которого находятся непрерывные последовательности операторов $s_0, \ldots, s_k$ и $s_{k+1}, \ldots, s_n$ соответственно, возможно, пустые. Тогда будет справедливо следующее соотношение:

$$s_c = \begin{cases} s_1 \circ \ldots \circ s_k & \text{при } c_j \equiv true \\ s_{k+1} \circ \ldots \circ s_n & \text{при } c_j \equiv false \end{cases} \tag{2.8}$$

С использованием данных правил можно строить композиции эффектов произвольных последовательностей операторов.

Поскольку тело функции также является последовательностью операторов языка, эффект от вызова функции можно рассчитать по тем же правилам. На основе зависимости полученного эффекта от условий на пути выполнения внутри функции и строится резюме. При этом в резюме сохраняются не все эффекты, производимые операторами, содержащимися в теле функции, а лишь те из них, которые сохраняются после выхода из неё и могут повлиять на дальнейшее выполнение программы после выхода из функции. Таким образом, сократить время анализа при использовании резюме в сравнении со встраиванием можно получить за счёт отсутствия необходимости затрачивать время на анализ эффектов, действия которых локальны или не учитываются при дальнейшем анализе. Так, связывание символьного значения с выражением имеет только локальный эффект, поскольку все выражения становятся неактивными при выходе из контекста анализа функции. Аналогично, локальный эффект имеют записи в локально видимую память и т. д. Кроме того, модель анализатора заведомо допускает упрощения, поскольку анализатор не может досконально смоделировать поведение программы. Это означает, что ряд эффектов операторов не будет учтён, т. е. на моделирование некоторых эффектов операторов будет затрачено время, однако результат этого моделирования не будет отражён в изменении состояния. Учёт этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, однако, что некоторые эффекты самого применения резюме не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Но, поскольку набор эффектов, получаемых в результате применения резюме, включается строго или совпадает

с набором эффектов, моделируемых при анализе методом встраивания, время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания.

2.2 Алгоритм метода резюме для символьного выполнения

В результате проведённого анализа в данной работе построен алгоритм метода межпроцедурного анализа с помощью резюме для метода символьного выполнения.

1. Провести анализ вызываемой функции, получив в результате её граф выполнения.
2. Для каждого конечного узла графа выполнения функции осуществить сбор эффектов, оказываемых на выполнение программы при выполнении данной ветви выполнения. Полученным результатом является набор ветвей резюме.
3. В каждой точке вызова проанализированной функции создать новые узлы графа выполнения (узлы применения резюме) со следующими характеристиками:
   - Дуги графа выполнения ведут из узла, соответствующего вызову функции (узел вызова) в каждый из узлов применения резюме.
   - Каждая точка применения резюме соответствует листу графа выполнения вызываемой функции и, соответственно, своей ветви резюме.
   - Состояние программы в каждой точке применения резюме есть композиция состояния программы в узле вызова и функции, описываемой соответствующей ветвью резюме.

Таким образом, множество узлов графа выполнения вызываемой функции отображается в множество узлов применения резюме. Поскольку множество всех узлов графа выполнения, как правило, многократно превосходит по количеству элементов множество листов графа, данный метод имеет значительно большую потенциальную масштабируемость.

2.3 Модель анализатора

В терминологии Clang Static Analyzer, разработанной на основе [57], выполнение программы представляет собой множество последовательных переходов между состояниями из одного состояния в другие. Каждому состоянию соответствует точка выполнения (`ProgramPoint`), для которого это состояние актуально. (Здесь и далее в скобках приводятся названия соответствующих классов из фреймворка Clang Static Analyzer). Переходы между состояниями обусловлены либо эффектами интерпретации отдельных выражений, определёнными стандартом языка, либо событиям, связанных с выполнением проверок проверяющими модулями (`Checker`). Из одной точки выполнения может идти не один переход в другое, а более — это происходит в случаях, когда условие перехода невозможно однозначно разрешить в пользу выбора какой-либо одной ветви выполнения, например, при обработке условных операторов (это и есть разделение на классы эквивалентности). Кроме того, проверки также могут разделять состояние программы, сохраняя в разных структурах состояния различающиеся данные состояния. Результирующее множество узлов в виде состояний и переходов из одного состояния программы в другое образует граф выполнения программы (`ExplodedGraph`).

За базовое моделирование процесса выполнения, без каких-либо проверок корректности исходного кода, отвечает ядро анализатора. Ядро анализатора представляет собой во-первых, набор методов, связанных с построением графа состояний выполнения программы (класс `CoreEngine`), а, во-вторых, набор методов, отвечающие за моделирование эффектов, специфичных для языка программирования, т. е. моделирование эффектов выражений и правил их выполнения (класс `ExprEngine`). Кроме того, в процедуре построения графа выполнения могут принимать участие проверяющие модули, анализируя события, наступающие в процессе выполнения. Эти проверки могут останавливать построение графа на заданном пути в случае обнаружения критического дефекта, разделять состояние программы и вносить в него дополнительную информацию для работы проверяющего модуля.

Структура состояния является представлением состояния программы в точке выполнения. Структура состояния включает следующие данные:

1. Содержимое памяти программы (модель памяти — *RegionStore*) [58], представляемое как отображение между регионами памяти и символьными значениями, связанными с этими регионами. Для создания записи в модели памяти необходимо произвести непосредственное связывание региона памяти и его значения, например, при обработке оператора присваивания. Операциями, изменяющими содержимое модели памяти, являются непосредственное связывание региона со значением (происходящее, например, при присваивании переменной значения), пометка регионов памяти как не содержащих первоначальное значение (инвалидация) и удаление имеющихся привязок, происходящее при потере регионом памяти активности, а также иногда используемое вместо инвалидации. В случае, если необходимо получить символьное значение для региона памяти, не имеющего записи в модели памяти (например, для аргументов функции), происходит неявное связывание с помощью символьного значения специального вида, при этом записи в модели памяти не создаётся.
2. Окружение (*Environment*) ставит в соответствие активным выражениям их символьные значения, как левосторонние, так и правосторонние: левосторонними значениями выражений являются абстрактные области памяти кода программы, где располагаются выражения, а правосторонними — вычисленные символьные значения выражений. Значения выражений, ставших неактивными, удаляются из окружения.
3. Нетипизированное хранилище (*GDM*, Generic Data Map) является контейнером для хранения данных проверок, а также используется для хранения некоторых специфических структур данных ядра анализатора, связанных с состоянием программы. Наиболее важными такими данными является карта соответствия символов и их диапазонов возможных значений, с помощью которой производится анализ достижимости. GDM позволяет хранить произвольные структуры данных, однако наиболее часто используются простейшие типы — указатели и целочисленные типы, а также неизменяемые словари, наборы и списки (`ImmutableMap`, `ImmutableSet` и `ImmutableList` соответственно), имеющие специальную поддержку, упрощающую их использование. За помещение данных в GDM и удаление их оттуда отвечают непосредственно

используют эти данные модули — проверки и модули ядра анализатора (в частности, ответственный за арифметические и логические вычисления решатель — `ConstraintManager`).

Символьное значение, в терминологии Clang Static Analyzer, является абстракцией переменного или константного значения какого-либо типа данных. Абстрактным значением можно представить, например, значение выражения, содержимое области памяти, саму область памяти и указатель на неё. В модели Clang Static Analyzer символьное значение может иметь несколько основные видов.

Первым видом символьных значений являются целочисленная константа. Clang Static Analyzer на данный момент может корректно работать лишь со значениями, представляемыми с помощью целочисленных типов. Константа имеет знаковость, разрядность и значение. Целочисленными константами также представляются символы строк (`char`-типы). Константы могут быть как правосторонними значениями (`nonloc::ConcreteInt`), так и левосторонними (`loc::ConcreteInt`). Система арифметики позволяет производить вычисления над константами с использованием бинарных и унарных операторов с получением новой константы в качестве результата.

Вторым видом символьного значения является символьное значение (`SymbolVal`). Символьное значение является представлением для *символьного выражения*, или *символа* — абстрактного неконстантного выражения. Символьным выражением могут быть:

- атомарный символ — атомарное правостороннее значение, которое нельзя однозначно определить как константу. Различные виды атомарных символов используются анализатором для различных целей. Существуют следующие виды атомарных символов:
  - `SymbolRegionValue` — символ, являющийся значением некоторого региона памяти по умолчанию, если его исходное значение неизвестно.
  - `SymbolExtent` — символ, представляющий размер соответствующего региона памяти в байтах, если этот регион имеет неконстантный размер.
  - `SymbolConjured` — символ, являющийся результатом выражения в случае, если правило вычисления данного выражения

неивестно. В частности, `SymbolConjured` может быть получен в качестве возвращаемого результата функции, определение которой недоступно анализатору.

  - `SymbolMetadata` — символ, описывающий некоторый регион памяти. Такие символы обычно используются проверяющими модулями для хранения информации об отслеживаемом регионе
  - `SymbolDerived` — символ, представляющий значение некоторого региона памяти, чей родительский регион имеет символьное значение.

- бинарное отношение символьного выражения и константы, выражаемое с помощью бинарных операторов (в CSA им соответствуют классы `SymIntExpr` и `IntSymExpr`);
- бинарное отношение двух символьных выражений, представленную в виде бинарного выражения с помощью бинарных операторов (`SymSymExpr`).

Символьное выражение, таким образом, в общем случае представляет собой вычисляемое дерево, листьями которого являются атомарные символы и константы.

Ещё одним видом символьных значений являются значение вида региона памяти (`MemRegionVal`) — значение, являющееся *регионом памяти*, т. е. абстрактным представления некоторого последовательного набора байт в памяти программы. Регион памяти является левосторонним выражением, будучи контейнером для некоторого значения, расположенного в памяти. Таким образом, регион памяти может быть как ключом для хранения в Store, так и его значением — например, значением указателя. В виде регионов памяти представляются левосторонние значения выражений (например, переменных, объектов). Каждый регион памяти, кроме *областей памяти*, имеет родительский регион. Таким образом, регионы памяти образуют иерархию памяти, корневыми регионами которой являются области памяти, а прочие регионы памяти являются их подрегионами и подрегионами друг друга. Память делится на области согласно расположению объектов в оперативной памяти (таблица 2.1)

Сами подрегионы разделяются на классы в зависимости от назначения (актуальные для C и C++ классы регионов перечислены в таблице 2.2. Вместе они образуют иерархию наследования **??**. Классы регионов памяти играют

Таблица 2.1

Виды областей памяти

Название региона	Хранимые данные
NonStaticGlobalSpaceRegion	Глобальные переменные внешней области видимости
StaticGlobalSpaceRegion	Глобальные статические переменные
HeapSpaceRegion	Переменные, располагающиеся в «куче»
StackArgumentsSpaceRegion	Переменные, являющиеся аргументами функции и располагающиеся на стеке
StackLocalsSpaceRegion	Переменные, локальные для функции
UnknownSpaceRegion	Вспомогательная область памяти для случаев, когда область хранения неизвестна

важную роль при построении и применении резюме функций, поскольку несут в себе важную служебную информацию, необходимую для определения новых регионов в заданном контексте.

Для экономии ресурсов при обработке операций присваивания сложных объектов используются символьные значения отложенной обработки (`LazyCompoundVal`). Их задача заключается в отображении крупных блоков хранилища на другой регион памяти до момента, пока в новый регион памяти не будет произведена запись. Таким образом производится увеличение скорости работы анализатора за счёт уменьшения объёмов копируемой и хранимой в структуре состояния информации.

Вспомогательными видами символьных значений является неизвестное значение (`UnknownVal`) и неопределённое значение (`UndefinedVal`). Неизвестное значение является может являться результатом выражения, которое анализатор не может корректно смоделировать — например, битовые операции над неконстантными значениями; кроме того, неизвестное выражение может получиться, если операндом выражения является неизвестное выражение. Нередко вместо неизвестного значения создаётся новый атомарный символ без наложенных на него ограничений, что позволяет восстановить контекстную чувствительность для символьных операций. Неопределённое значение является результатом операций, результат которых не определён стандартом — в частности, разыменование нулевого указателя, а также при выполнении операций, операндом которого является другое неопределённое значение. Использование

Таблица 2.2

Виды регионов памяти

Название региона	Хранимые данные
AllocaRegion	Память на стеке, выделенная функцией `alloca()`
SymbolicRegion	Регион памяти, располагающийся по адресу, заданному указателю. Не имеет определённого размера и типа, они задаются его подрегионами
BlockDataRegion	Данные блоковых конструкций языка C и лямбда-выражений C++
BlockTextRegion	Код блоковых конструкций языка C и лямбда-выражений C++
FunctionTextRegion	Регион кода функции
CompoundLiteralRegion	Регион составного литерала
CXXBaseObjectRegion	Регион базового подобъекта класса. Задаётся определением базового класса
CXXTempObjectRegion	Регион временного объекта в C++
CXXThisRegion	Регион, на который указывает указатель `this` (C++). Используется при анализе методов класса вне контекста
FieldRegion	Регион поля структуры или класса. Задаётся определением поля
VarRegion	Регион переменной
ElementRegion	Регион элемента массива
StringRegion	Регион строкового литерала

неопределённых значений также является одним из видов потенциальных проверок, и введение для этого отдельного типа символьного значения упрощает работу с анализатором.

Символьное значение можно получить следующими способами.

Левостороннее символьное значение можно получить при обращении к выражению, результат которого является левосторонним выражением — переменная, элемент массива, поле объекта и т. д. При этом результирующее значение будет связано с объявлением той области памяти, к которой происходит обращение.

Символьное значение можно также получить в результате выполнения загрузки символьного значения из региона памяти (при преобразовании левостороннего выражения в правостороннее выражения, lvalue-to-rvalue cast). При этом, если регион уже имел непосредственную привязку, то будет получено символьное выражение, связанное с этой привязкой. В противном (и более частом) случае произойдёт создание атомарного символа, являющимся значением этого региона по умолчанию, без каких-либо наложенных на него ограничений, или возврат уже созданного символа.

Наконец, символьное значение можно получить, осуществив бинарную операцию над другими символьными значениями.

2.4 Эффекты, учитываемые в резюме функции

Каждый оператор при своём выполнении производит эффект, заключающийся в изменении состояния программы. В случае анализа речь идёт о моделировании эффектов операторов, то есть о моделировании действия, которое моделируемый оператор оказывает на модель состояния программы.

С учётом описанной модели анализатора, сократить время анализа при использовании резюме в сравнении со встраиванием можно получить за счёт отсутствия необходимости затрачивать время на анализ эффектов, действия которых локальны или не учитываются при дальнейшем анализе. Например, связывание символьного значения с выражением имеет только локальный эффект, поскольку все выражения становятся неактивными при выходе из контек-

ста анализа функции. Аналогично, локальный эффект имеют записи в локально видимую память и т. д. Кроме того, модель анализатора заведомо допускает упрощения, поскольку анализатор не может досконально смоделировать поведение программы. Это означает, что ряд эффектов операторов не будет учтён, т. е. на моделирование некоторых эффектов операторов будет затрачено время, однако результат этого моделирования не будет отражён в изменении состояния. Учёт этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, однако, что некоторые эффекты самого применения резюме не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Тем не менее, время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания. Это объясняется тем, что набор эффектов, получаемых в результате применения резюме, включается строго или совпадает с набором эффектов, моделируемых при анализе методом встраивания.

В данной работе рассмотрен следующий набор эффектов, оказывающих влияние на состояние анализируемой программы в процессе её выполнения:

1. Принятие решений о выборе пути выполнения. Выбор пути выполнения сопровождается наложением ограничений на символьные значения, относительно которых принимается решение о выборе пути. Если эти символьные значения содержат ссылки на внешние по отношении к вызываемой функции регионы памяти, накладываемые ограничения должны быть отражены в резюме. Кроме того, как было показано выше, каждое принятие решения влияет на присутствие и порядок операторов в последовательности выполнения, а следовательно, и на набор эффектов, включаемых в резюме. Наконец, наложение ограничений на входные данные функции в зависимости от выбора пути выполнения позволяет сохранить контекстную чувствительность при анализе, поскольку определённые пути выполнения могут быть достижимы лишь при ограниченном наборе входных значений аргументов функции и значений, находящихся во внешней по отношении к ней памяти.
2. Модификация регионов памяти с областью видимости, отличной от локальной, то есть находящихся в статической или глобальной области

видимости, принадлежащих куче, а также модификация аргументов, переданных по неконстантному указателю или неконстантной ссылке, и областей памяти, относящихся к ним (возможно, с использованием арифметики указателей).

3. Инвалидация регионов памяти, то есть пометка некоторых регионов как изменивших значение на неизвестное. Данное действие обычно выполняется при моделировании оператора, все эффекты которого учесть по каким-либо причинам невозможно — например, при вызове функции с недоступным определением.
4. Возврат вызываемой функцией некоторого значения. Это значение связывается с выражением вызова функции как элемент окружения.
5. Пометки проверяющих модулей:
    (a) пометки символов, регионов памяти и символьных значений;
    (b) события, которые необходимо проверить отложенно, когда контекст вызываемой функции станет достаточно определён для того, чтобы утверждать наличие потенциального дефекта;
    (c) иные действия, связанные с процедурами проверок (в зависимости от логики работы проверяющего модуля).

Поскольку проверяющие модули самостоятельно отвечают за свои данные, логику обработки резюме для проверок имеет смысл включать непосредственно в логику работы этих модулей.

2.5 Порождение новых ветвей выполнения программы и отсечение недостижимых путей

Одним из результатов сбора резюме являются пары «регион памяти — символьное значение». В результате актуализации символьных значений из резюме могут получиться символьные значения, имеющие диапазон, отличный от диапазона этого символьного значения в контексте вызывающей функции. Это является следствием того, что при моделировании условий внутри вызываемой

функции может произойти разделение входных данных функции (аргументов и внешних переменных) на классы эквивалентности.

Рассмотрим пример. Пусть вызывается функция:

```
1  void f(int a) {
2    if (a > 10) {
3      ...
4    } else {
5      ...
6    }
7  }
```

В результате анализа этой функции в её резюме войдут две ветви выполнения. В первой ветви `a` будет иметь диапазон конкретных значений от `INT_MIN` до 10, во второй — от 11 до `INT_MAX`.

Пусть происходит вызов функции при `a` от 5 до 13. Тогда в первой создаваемой ветви выполнения `a` будет иметь диапазон от 5 до 10, а во второй — от 11 до 13.

Далее, пусть в вызывающей функции `a` имеет диапазон конкретных значений от 5 до 9. Тогда в первой ветви выполнения `a` будет иметь диапазон от 5 до 9, а во второй ветви выполнения множество значений будет пустым. Наличие символического значения с пустым множеством допустимых значений означает, что вторая ветвь является недостижимой и может не рассматриваться. Действительно, при вызове функции при заданном `a` выполняется только `else`-ветвь, но не `if`-ветвь.

Введём следующие обозначения:

- n – количество ветвей выполнения, полученных в резюме анализа вызываемой функции,
- i – номер ветви выполнения, где $0 \leqslant i < n$,
- p_i – количество символьных правосторонних входных значений в i-ой ветви выполнения,
- j – номер символьного значения, где $0 \leqslant j < p_i$,
- s_{ij} – символьное значение с номером j в i-ой ветви выполнения,
- $r_{\text{входные } ij}$ – множество значений для s_{ij} в контексте вызывающей функции в точке непосредственно перед вызовом функции,

– $r_{\text{резюме } ij}$ – множество значений для s_{ij} в контексте вызываемой функции,
– $state_{\text{входное}}$ — состояние программы в контексте вызывающей функции в точке непосредственно перед вызовом функции,
– $state_{\text{выходное}}$ — состояние программы после вызова функции (после применения резюме).

Тогда при применении i-й ветви выполнения резюме:

$$\forall i \in [0; n], \forall j \in [0; p_i] : \; r_{\text{выходные } ij} = r_{\text{входные } ij} \cap r_{\text{резюме } ij},$$

то есть результирующее множество является пересечением множеств входных конкретных значений символического значения и множества конкретных значений символического значения из применяемой ветви резюме.

В случае, если результирующее множество конкретных значений является пустым хотя бы для одного символического значения, то данная ветвь выполнения является недостижимой и не принимается в дальнейшее рассмотрение, что может быть выражено формулой 2.5.

$$(\exists i, j : r_{\text{входные } ij} \cap r_{\text{резюме } ij} = \varnothing) \Rightarrow (state_{\text{входное}} \nrightarrow state_{\text{выходное}})$$

2.6 Сбор данных для создания резюме

Пусть некоторое значение относится к региону памяти. Поскольку при передаче аргумента в функцию не по значению его значение может измениться, необходимо различать входное значение региона до его изменения в функции и выходное значение. Для получения информации о разбиении данных на классы эквивалентности можно отслеживать события ветвлений (assume), однако данный подход неудобен на практике, поскольку, во-первых, требует отдельного отслеживания событий изменений региона для разделения входных и выходных значений, и, во-вторых, требует активного участия сборщика резюме в процессе принятия решения о ветвлении. Вместо этого в настоящем решении предложен и использован подход, основанный на проверке активно-

сти символов и регионов. Поскольку входные данные, внешние по отношении к анализируемой функции, всегда являются символическими, отслеживая событие потери актуальности (активности), можно определить диапазон возможных значений символа, связанного с данным регионом, в данной ветви выполнения. При этом первое событие потери активности соответствует диапазону входных значений, а последнее соответствует диапазону выходных значений, причём если они совпадают, это означает, что никаких присваиваний или инвалидаций региона входного символа не было, и входной диапазон является также выходным диапазоном. Данный метод позволяет избежать использования сложных алгоритмических схем для сбора входных и выходных значений аргументов функций, а также входных значений глобальных регионов памяти.

Сбор выходных значений также бывает необходимо проводить по окончании пути выполнения функции. Это необходимо для обработки символьных значений, привязанных к региону памяти непосредственным присваиванием или иным видом связывания. Для этого используется итерация по хранилищу с сохранением диапазонов внешних по отношении к функции регионов памяти в резюме.

Инвалидация региона памяти в терминологии Clang Static Analyzer означает связывание с данным регионом нового символа, без наложенных на него ограничений, т. е. способного принимать произвольные значения. Поскольку символьные значения, связанные с регионами памяти, обрабатываются при завершающем проходе по хранилищу, а значения регионов, актуальные до инвалидации, обрабатываются по событию потери активности, непосредственно предшествующему событию связывания нового значения, инвалидация обрабатывается автоматически, и дополнительных действий для обработки инвалидаций регионов памяти не требуется.

Обработка события возврата функцией значения достаточно тривиальна. Результирующее символьное значение сохраняется в резюме целиком, а ограничения, накладываемые на него и на его части, обрабатываются отдельно.

За хранение данных проверок проверяющие модули отвечают самостоятельно. Основными видами данных проверок является отметка отложенной проверки и данные состояния проверки. Отложенные проверки используются для выдачи предупреждений в тех ситуациях, когда из-за отсутствия данных о контексте вызова невозможно однозначно утверждать наличие дефекта или его

отсутствие. Данные состояния используются для построения нового состояния проверки при применении резюме.

2.7 Актуализация символьных значений

В результате сбора резюме на предыдущем шаге мы получаем некоторое множество регионов памяти, с которыми связаны некоторые символьные значения. Кроме того, регионы памяти сами могут входить в символьные значения как их составная часть. Однако, полученные регионы памяти адресуются в контексте объявлений имён внутри функции. В контексте вызывающей функции эти регионы могут иметь уже другое значение, то есть регионы, используемые внутри функции, являются относительными по отношении к вызывающей функции. Так, например, в контексте вызываемого метода класса регион памяти, связанный с указателем `this`, будет адресоваться безотносительно какого-либо объекта, а в контексте вызывающей функции этот регион будет регионом, в котором находится объект, метод которого вызывается. Аналогично, аргумент функции, фигурирующий в ней как самостоятельная переменная, (и, соответственно, как самостоятельный регион памяти), может быть подрегионом в контексте вызывающей функции — полем структуры, элементом массива. Кроме того, с регионом памяти в контексте вызываемой функции может быть связан не символ, относящийся к региону памяти, а константа, символьное выражение или иное значение, не имеющее в своей основе регион аргумента. Всё это означает, что для корректного применения резюме необходимо производить актуализацию символьных значений, то есть их перевод из контекста имён и значений вызываемой функции в контекст имён и значений вызывающей функции.

Идея актуализации заключается в следующем. Пусть имеется символьное значение. В его состав могут входить символы, регионы памяти и константы. Они, согласно модели анализатора, образуют дерево. Непосредственно актуализации подвергаются только регионы памяти. Таким образом, в символьном значении происходит подмена регионов памяти, содержащихся в нём, на актуализированные. Затем, если это возможности, символы полученного символьного

выражения вычисляются в константы, заменяя исходные поддеревья. Данную процедуру можно выразить следующим алгоритмом:

1. Для всех регионов памяти, содержащихся в символьном значении:
    (a) Создать новый регион, являющийся актуализацией данного региона
    (b) Заменить в символьном значении исходный регион актуализированным
2. Для всех символов, содержащихся в символьном значении, полученном на шаге 1:
    (a) Проверить, не вычисляется ли символ в константу
    (b) Если символ вычисляется в константу, заменить данный символ вычисленной константой.

В соответствии с типами символьных значений, можно производить актуализацию символьных значений в зависимости от их типа.

Константные значения являются неизменяемыми при их актуализации, поскольку они не содержат элементов, зависящих от контекста. Вместе с тем, типы константного значения в контексте вызываемой функции и в контексте вызывающей функции могут быть различающимися, поэтому может понадобиться осуществление дополнительного приведения типов для константы.

Символьные значения, относящиеся к регионам памяти, можно актуализировать, используя следующие правила для различных категорий регионов памяти.

2.7.1 Регионы, относящиеся к пространству аргументов вызываемой функции

Можно выделить два различных случая передачи, в зависимости от того, является ли передаваемый тип ссылочным или типом указателя, или не является.

В случае, если аргумент передаётся по ссылке, левостороннее значение фактического аргумента становится левосторонним значением формального параметра, а правостороннее значение фактического параметра становится право-

сторонним значением формального параметра. Это значит, что значение объекта в результате выполнения функции может отличаться от значения на момент вызова, поскольку в результате передачи по ссылке с фактическим аргументом может быть связано другое значение.

Таким образом, при передаче аргумента по ссылке:

1. Адрес формального параметра является адресом фактического аргумента.
2. Левостороннее значение формального параметра является левосторонним значением фактического аргумента.
3. Правостороннее значение формального параметра является правосторонним значением фактического аргумента.
4. Базовым регионом для построения подрегионов доступа, относящихся к региону памяти формального параметра, является левостороннее значение фактического аргумента.

В случае, если аргумент передаётся по указателю, правостороннее значение фактического параметра становится правосторонним значением формального параметра. Для левосторонних значений данное утверждение неверно: выполнение присваивания формальному аргументу внутри функции не влияет на значение фактического аргумента. Однако в случае, если указываемый тип не является константным, в результате вызова функции может измениться привязка региона памяти, адрес которой задаёт указатель, и его субрегионов.

Таким образом, при передаче аргумента по указателю:

1. Правостороннее значение формального параметра является правосторонним значением фактического аргумента.
2. Регион памяти с адресом, задаваемым указателем формального аргумента является регионом памяти с адресом, задаваемым указателем фактического аргумента (поскольку значения указателей совпадают). Привязки этого региона и его субрегионов могут изменяться в зависимости от модификатора константности их типов.
3. Базовым регионом для построения подрегионов доступа, относящихся к региону памяти с адресом, задаваемым указателем формальным параметром, является регион памяти с адресом, задаваемым указателем фактического аргумента.

В случае, если аргумент передаётся по значению правостороннее значение фактического параметра становится правосторонним значением формального параметра.

В случае передачи в качестве аргумента структурных типов по ссылке или указателю, правила изменения их полей аналогичны. Так, в случае передачи по значению структуры, содержащей ссылку в качестве поля, данное поле можно считать передающимся по ссылке. Фактически, в случае структурных типов можно считать, что передаётся набор аргументов по их типам с тем отличием, что при потере актуальности их окружающего региона памяти эти поля также могут потерять актуальность.

2.7.2 Регионы памяти внешней области видимости

Помимо передаваемых аргументов, вызываемая функция может иметь доступ к ряду других данных — переменных, имеющих области видимости выше, чем функция. К этим регионам относятся глобальные переменные и члены класса и его предков, если вызываемой функцией является метод класса. Их изменения и наложения ограничений на них также необходимо отслеживать.

Регионы глобальных переменных (включая статические) сохраняются неизменными, дополнительные действия по из актуализации предпринимать не нужно. Это так, поскольку регионы глобальных переменные не зависят от контекста вызова и связаны лишь с объявлением соответствующих переменных.

Методам класса, включая конструкторы, деструкторы и операторы-члены класса, могут быть доступны для чтения и записи поля как самого класса, так и его предков в иерархии наследования. При актуализации регионы памяти, относящиеся к статическим полям класса, не изменяются, поскольку они не относятся к конкретному объекту поля и, следовательно, их адресация не зависит от контекста вызова. Нестатические поля в контексте вызываемого метода адресуются относительно условного объекта, связанного с указателем `this`, и, поскольку в контексте вызываемой функции фактическим объектом будет объект, метод которого вызывается, при актуализации эти поля становятся соответствующими полями вызываемого объекта. Если определение нестатического поля

принадлежит классу, определение которого находится ниже по иерархии наследования, то поле отображается в соответствующее поле родительского класса объекта в соответствии с компоновкой полей дочернего класса.

2.7.3 Актуализация составных и служебных символьных значений

Актуализация символьных значений, обозначающих бинарные операции над символами (бинарные символьные значения), выполняется следующим образом. Сначала выполняется актуализация правой и левой частей символьного значения. Если результатом бинарной операции является константа, эта константа становится результирующим символьным значением. В противном случае результатом актуализации является новое символьное значение в виде бинарной операции. Фактически, бинарные символы актуализируются рекурсивно, с возможными упрощениями в виде свёртывания отдельных элементов или всего выражения в константу. Это свёртывание объясняется уменьшением диапазонов входных значений каждого из символьных значений, входящего в состав бинарного символьного значения, при уточнении контекста вызова.

Актуализация метасимволов является отдельной подзадачей. Под метасимволами понимают символические значения, относящиеся к объекту анализа, но не являющиеся его непосредственной характеристикой. Выделение метасимволов связано с тем, что источником метаданных является не ядро анализатора, а сами проверяющие модули, которые связывают необходимую информацию в виде метасимволов с интересующими их регионами памяти и выражениями. Соответственно, регионы памяти и выражения могут иметь более одного связанного с ними метасимвола. Поскольку каждый проверяющий модуль может реализовывать свой подход к использованию метаинформации и обозначениям интересующих его объектов, для актуализации метасимволов используется отдельное событие, на которое должны подписываться проверяющие модули, использующие метаданные. В результате проверяющие модули самостоятельно обновляют представление связанных с ними символьных значений и нарушение принципа инкапсуляции данных не происходит. В качестве примера можно привести проверку, связанную с длиной строки. Длина строки не входит в число

данных, о которых известно самому анализатору — за её моделирование отвечает проверяющий модуль, связывающий символьное значение (метасимвол) с регионом памяти данной подстроки. Таким образом, задачей проверяющего модуля при актуализации метазначения, связанного с регионом, является поиск существующего метазначения для этого региона и его возврат.

Построение и актуализация сложных структур данных в резюме производится с помощью сохранения цепочки родительских регионов. Для каждого региона памяти, имеющего связанное с ним символьное значение (как явно, так и неявно), строится упорядоченный список родительских регионов, начиная от региона верхнего уровня — $M_0 \dots M_n$. При этом регионы $M_1 \dots M_n$ может быть только регионами элемента массива, регионами поля структуры или регионами данных базового класса. Этот список сохраняется в резюме и используется для актуализации значений по следующему алгоритму.

1. Актуализация региона M_0.
2. Для всех регионов $M_1 \dots M_n$ согласно их положению в списке:
   (a) если M_i — регион элемента массива, то в резюме сохраняется символьное значение индекса, а результатом актуализации является элемент массива от региона, полученного на $(i-1)$-ом шаге алгоритма, с символьным значением индекса, полученным в результате актуализации сохранённого значения индекса.
   (b) если M_i — регион поля структуры, то в резюме сохраняется объявление поля структуры, а результатом актуализации является поле структуры от региона, полученного на $(i-1)$-ом шаге алгоритма, с тем же определением.
   (c) если M_i — регион данных базового класса, то в резюме сохраняется ссылка на определение базового класса, а результатом актуализации является подрегион данных базового класса от региона, полученного на $(i-1)$-ом шаге, с тем же определением.

Цепочка родительских регионов может строиться как явно — при построении резюме для региона памяти, так и неявно — при сохранении и последующем разборе символьного значения, для которого строится резюме.

2.7.4 Актуализация литеральных регионов

Литеральные регионы, т. е. регионы, относящиеся к строковым или составным литералам, актуализируются с использованием выражения языка, по которому они были построены. Поскольку эти регионы хранят исходное выражение как служеюную информацию, именно это выражение записывается в резюме, и по нему строится актуализированное выражение. Фактически, литералы не изменяются вместе с контекстом, и обычно хранятся в статической памяти, являясь видом констант, что упрощает работу с ними при использовании метода резюме.

2.8 Применение резюме проверяющими модулями

Схемы применения резюме, описанные выше, затрагивают отсечение недостижимых ветвей выполнения программы и уточнение множества конкретных значений для символьных значений. Однако для того, чтобы анализатор имел возможность выполнять проверки при вложенном вызове функции, необходима доработка проверяющих модулей. Для получения проверяющими модулями возможностей анализа при использовании резюме мы вводим две дополнительных функции обратного вызова. Первая из них (названная `evalSummaryPopulate`) вызывается для сбора резюме проверяющим модулем, вторая (названная `evalSummaryApply`) вызывается при применении резюме.

Проверяющий модуль, имеющий возможность выполнения действий при событии `SummaryPopulate`, должен сохранить информацию, которая может понадобиться для обновления состояния или для выполнения отложенной проверки. Информация, содержащая в резюме, не освобождается до окончания работы анализатора, поэтому проверяющий модуль может использовать произвольный формат хранения данных, лучшим образом отвечающий задаче проверки. Как показала практика модификации проверяющих модулей, для каждой проверки, проводимой модулем, в GDM обычно помещается две дополнительных записи, которые затем будут использоваться для заполнения резюме — для обновления

состояния и для отложенной проверки. В качестве примера рассмотрим проверку двойного закрытия файлового дескриптора. В резюме помещаются две секции: первая отвечает за обновление состояния дескриптора (является ли он открытым или закрытым), а вторая — за выполнение отложенной проверки: в ней запоминаются события закрытия дескрипторов, исходное состояние которых неизвестно.

При обработке события `SummaryApply` проверяющий модуль должен произвести обновление состояния в соответствии с информацией, хранящейся в выбранной ветви резюме. Так, если при выполнении вызова функции дескриптор был закрыт, он должен быть помечен как закрытый в состоянии вызывающей функции. Если же в контексте вызывающей функции уже известно, что дескриптор закрыт, то отложенная проверка должна выдать предупреждение.

Кроме того, если проверяющий модуль использует метаданные для хранения информации о состоянии контролируемых объектов, он может потребовать реализации функциональности актуализации метасимвола. Для этого вводится функция обратного вызова `evalSummarySVal`. Реализующие эту функциональность модули должны определять, на какое символьное значение метаданных отображается актуализированный метасимвол, и возвращать это символьное значение в качестве результата функции обратного вызова.

2.9 Методы реализации резюме для различных видов проверок

Как говорилось выше, проверяющие модули должны самостоятельно обеспечивать поддержку резюме. В настоящем разделе дано описание методов обеспечения резюме в проверяющих модулях различного назначения.

2.9.1 ConstModifiedChecker — проверка модификации константных данных

Задачей данной проверки является определение записи в область памяти, имеющую константный квалификатор, с использованием приведения типа к неконстантному (ссылка на CWE). Данная проверка преследует две цели. Во-первых, модификация данных таким образом представляет собой плохой стиль програмирования и указывает на необходимость пересмотра интерфейсов программы; в случае константного метода класса (C++) в ряде случаев нужный эффект можно получить, используя ключевое слово `mutable` для определения члена класса, который может быть изменён при вызове семантически константного метода. Во-вторых, попытка записи в память, инициализируемую компилятором как константную, с использованием приведения типов, приводит к неопределённому поведению. (ссылка!)

Исходный ринцип действия проверки следующий. Проверка отслеживает все регионы памяти, встречаемые в процессе анализа программы и имеющие изначально константный тип. Данные регионы записываются в GDM. В случае выполнения явного приведения типов (с использованием одного из выражений `ExplicitCastExpr`) — с использованием приведения типов в стиле C, операторов `const_cast` или `reinterpret_cast` — от константного типа к неконстантному, регионы, для типа которых было выполнено приведение, записываются в отдельный список.

2.10 Построение отчёта о дефекте

Построение отчёта является важным элементом работы статического анализатора. Недостаточно просто найти дефект и указать место его возникновения. В случае анализа путей выполнения программы между различными значимыми для данного дефекта точками программы (например, открытие и закрытие файла) может проходить длинный путь через большое количество операторов. При этом путь выполнения может включать в себя условные операторы,

циклы и вызовы других функций. Однако субъективная сложность отслеживания пути выполнения быстро растёт при увеличении длины пути. Если дефект не локализован в пределах нескольких строк кода или небольшой функции, разработчику становится практически невозможно определить путь выполнения, на котором проявляется дефект. Таким образом, при анализе программы методом анализа её путей выполнения необходимо выполнять построение подробного отчёта, однозначно указывающего условия, при которых проявляется дефект, и соответствующий им путь выполнения.

При использовании межпроцедурного анализа методом встраивания путь, проходимый внутри функции, отображается в графе выполнения как часть общего пути выполнения, поэтому проблем при построении пути при генерации отчёта не возникает. Однако при применении метода резюме возникает проблема потери информации о части пути, проходимом внутри вызываемой функции, поскольку явного построения поддеревьев графа выполнения программы более не происходит. В результате замены встраивания функции на применение её резюме граф выполнения программы изменяется следующим образом. Вместо подграфа вложенного вызова в графе выполнения появляются отдельные точки выполнения, условно соответствующие конечным точкам подграфа вложенного вызова функции. В связи с этим при построении отчёта о найденном дефекте нельзя традиционным образом указать путь, который прошло выполнение программы при вызове функции. Для построения отчёта при межпроцедурном анализе методом резюме в настоящей работе предложен следующий метод.

Задачей проверяющих модулей является отложенная проверка: на основе резюме вызываемой функции и состояния на момент вызова проверяющий модуль должен сделать вывод о необходимости выдачи предупреждения (срабатывании). Имеется два варианта срабатывания проверки. В первом случае выдача предупреждения происходит внутри вызываемой функции. Допустим, что уровень вложенности вызова равен единице. Такое допущение допустимо, поскольку к нему можно свести стек вызовов произвольного уровня вложенности. В этом случае конечной точкой трассы является узел графа выполнения вызываемой функции. В случае, если этот узел известен, от него можно построить трассу до корневого узла графа выполнения. Данная трасса-подграф будет являться путём, проходимым внутри функции при выполнении программы до критической точки.

Второй случай предполагает построение пути при необходимости показать путь внутри вызванной функции целиком, от точки входа до точки выхода. Данная задача сводится к первой при условии, что в качестве конечной точки выбирается лист графа выполнения, соответствующий выбранной ветви выполнения резюме.

Таким образом, для корректного построения пути при моделировании вложенного вызова функции достаточно иметь информацию или об узле графа выполнения вызываемой функции, для которого выдаётся срабатывание, или о листе графа выполнения, который относится выбранный путь выполнения при применении резюме. Для этого достаточно хранить ссылку на этот узел. В случае построения пути изнутри вызываемой функции за хранение ссылки может отвечать проверяющий модуль, генерирующий срабатывание. В случае построения полного пути по вложенному вызову ссылку на лист графа выполнения вызываемой функции можно хранить в узле применения резюме в качестве дополнительной информации.

Поясним описанный метод примером. Рассмотрим функцию следующего вида:

```
1  void close_file(bool flag, FILE *f) {
2    ...
3    if (flag)
4      fclose(f);
5  }
```

Пусть задачей проверяющего модуля является проверка двойного закрытия файлового дескриптора. Резюме функции `close_file()` будет состоять из двух ветвей: в первой ветви ($flag \equiv false$) никакой дополнительной информации не хранится, во второй ($flag \equiv true$) имеется событие закрытия файла. При анализе приведённой функции вне контекста вызывающей функции проверяющий модуль не может доказать, что внутри этой функции дескриптор закрывается во второй раз, поскольку на всех путях выполнения дескриптор закрывается не более одного раза. Отсюда следует необходимость выполнения отложенной проверки при применении резюме этой функции, когда контекст вызова позволит однозначно определить состояние файлового дескриптора при вызове `fclose(f)` внутри `close_file()`.

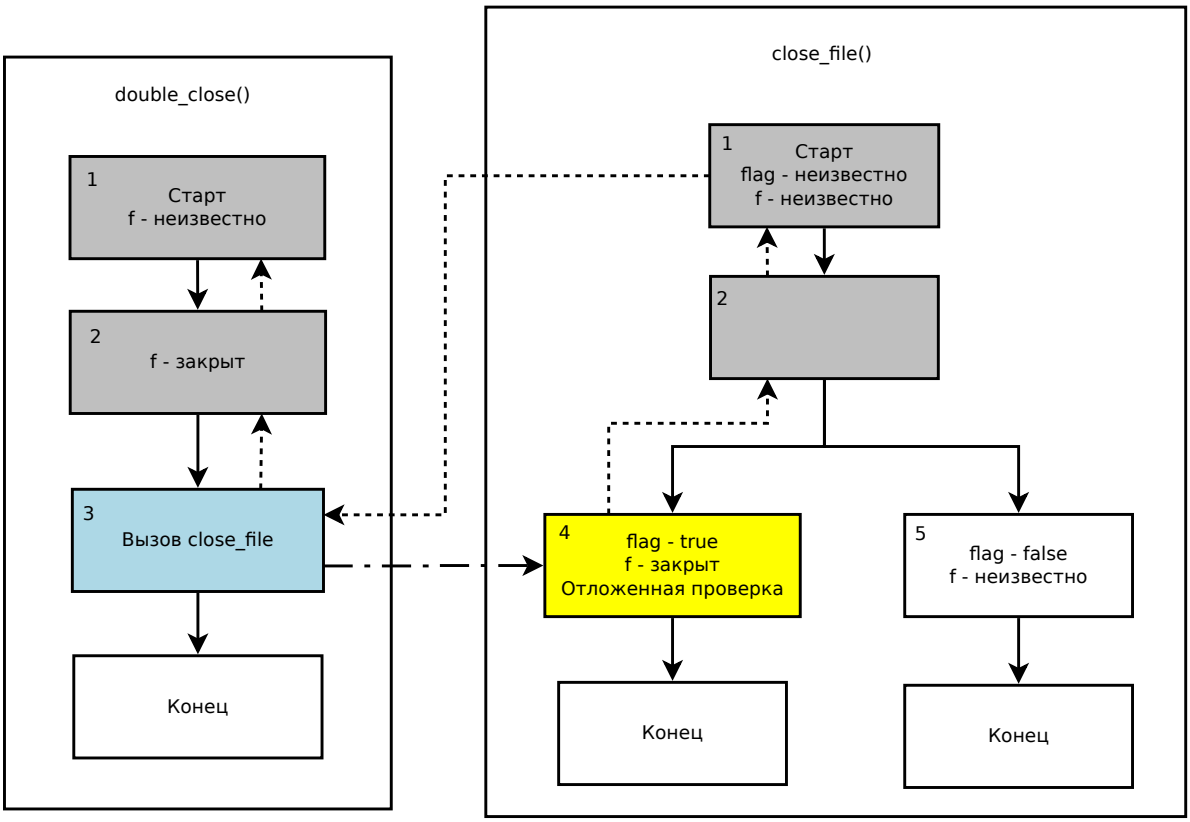

Рисунок 2.1 — Построение отчёта при использовании резюме вложенного вызова функции

Далее, пусть имеется функция, вызывающая `close_file()` и имеющая следующий вид.

```
1  void double_close(FILE *file) {
2    fclose(file);
3    close_file(true, file);
4  }
```

На момент вызова функции `close_file()` (и применения резюме) дескриптор `file` является закрытым, т. е. его состояние однозначно определено. Это значит, что может выполниться отложенная проверка, которая покажет, что происходит закрытие уже закрытого дескриптора. В этом случае происходит срабатывание, путь к которому нужно построить изнутри вызываемой функции. Рис. 2.1 поясняет применение этой схемы в данном примере. Поток управления внутри функции показан сплошными линиями, отношение отложенной проверки в заданном узле — штрихпунктирной, а пунктиром показана трасса построения отчёта.

Рассмотрим более общий случай произвольной вложенности вызовов. В случае построения пути изнутри вызываемой функции информации об узле срабатывания может оказаться недостаточно, поскольку по одному узлу невозможно восстановить всю цепочку вложенных вызовов. Это объясняется тем, что резюме функции верхнего уровня хранит ссылки только на узлы следующего уровня вложенности. Для решения этой задачи проверяющий модуль должен самостоятельно хранить стек вызовов, для которого строится путь выполнения.

Рассмотрим данный случай на примере. Модифицируем представленный в предыдущем примере набор функций, добавив в него промежуточный вызов:

```
 1  void close_file(bool flag, FILE *f) {
 2    ...
 3    if (flag)
 4      fclose(f);
 5    ...
 6  }
 7
 8  void potential_double_close(bool flag, FILE *file) {
 9    close_file(flag, file);
10  }
11
12  void double_close(FILE *file) {
13    fclose(file);
14    potential_double_close(true, file);
15  }
```

Резюме функции `close_file()` аналогично предыдущему случаю. Резюме функции `potential_double_close()` состоит из двух ветвей: в первой ветви ($flag \equiv false$) никакой дополнительной информации не хранится, во второй ($flag \equiv true$) имеется отложенная проверка события закрытия файла, содержащая ссылку на узел №4 графа выполнения функции `close_file()`. В качестве точки отложенной проверки для функции `potential_double_close()`, при этом в качестве целевого узла указывается не узел применения резюме, а его предшественник. Это делается для упрощения, поскольку узел применения резюме на момент сохранения информации ещё не создан (сохранение информации проверяющих модулей является одним из действий по его созданию, и

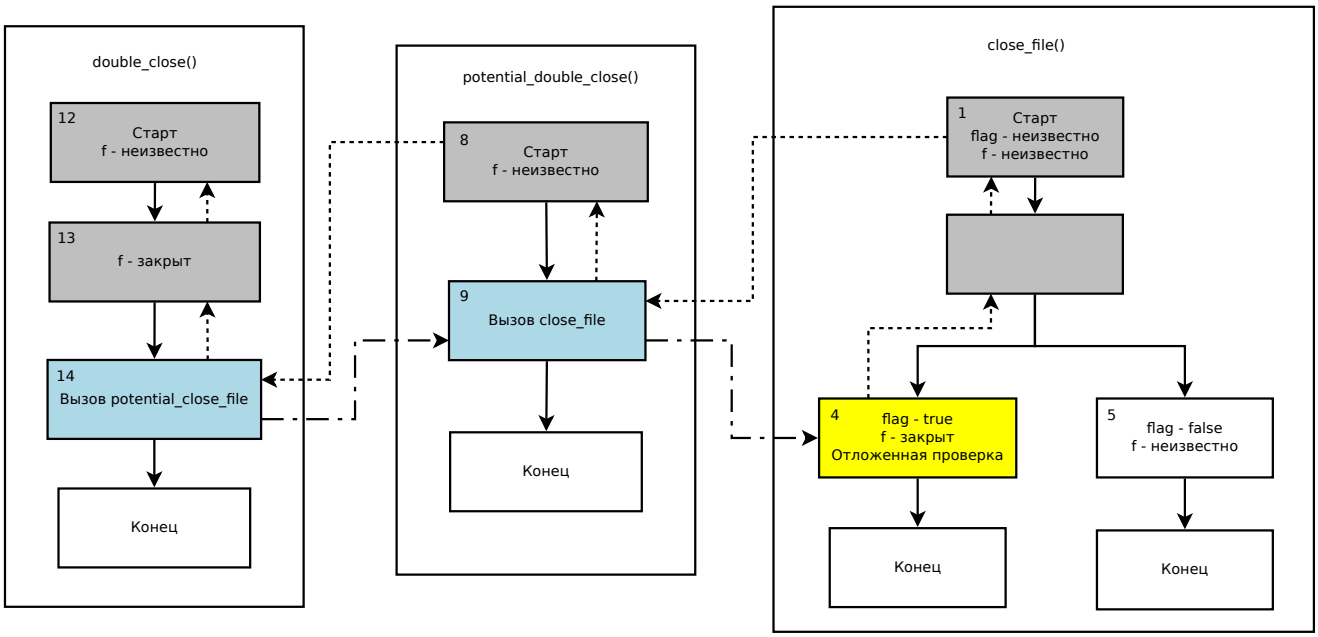

Рисунок 2.2 — Построение отчёта при использовании резюме вызова функции многократной вложенности

к этому моменту проведены не все действия по применения резюме), но информация об узле предвызова и целевом узле вызываемой функции позволяет однозначно идентифицировать ветвь резюме (или группу ветвей), затрагиваемую при прохождении пути до заданной точки.

Схема построения пути в приведённом случае показана на рис. 2.2.

На рис. 2.3 показано построение пути при использовании полного пути выполнения внутри вызываемой функции.

При показе пути внутри вложенного графа возникает проблема именования, поскольку внутри графа выполнения функции схема именования является локальной и не связана со схемой именования вызывающей функции. Данную проблему можно решить уже описанным способом с помощью переименования именованных символьных значений из контекста вызываемой функции в контекст вызывающей. При этом можно выполнять переименование не для всех символьных значений, имеющихся внутри пути выполнения, а только для тех, информация о которых непосредственно представляется пользователю. Это уменьшает временные затраты, поскольку актуализация символьного значения может быть достаточно дорогой операцией.

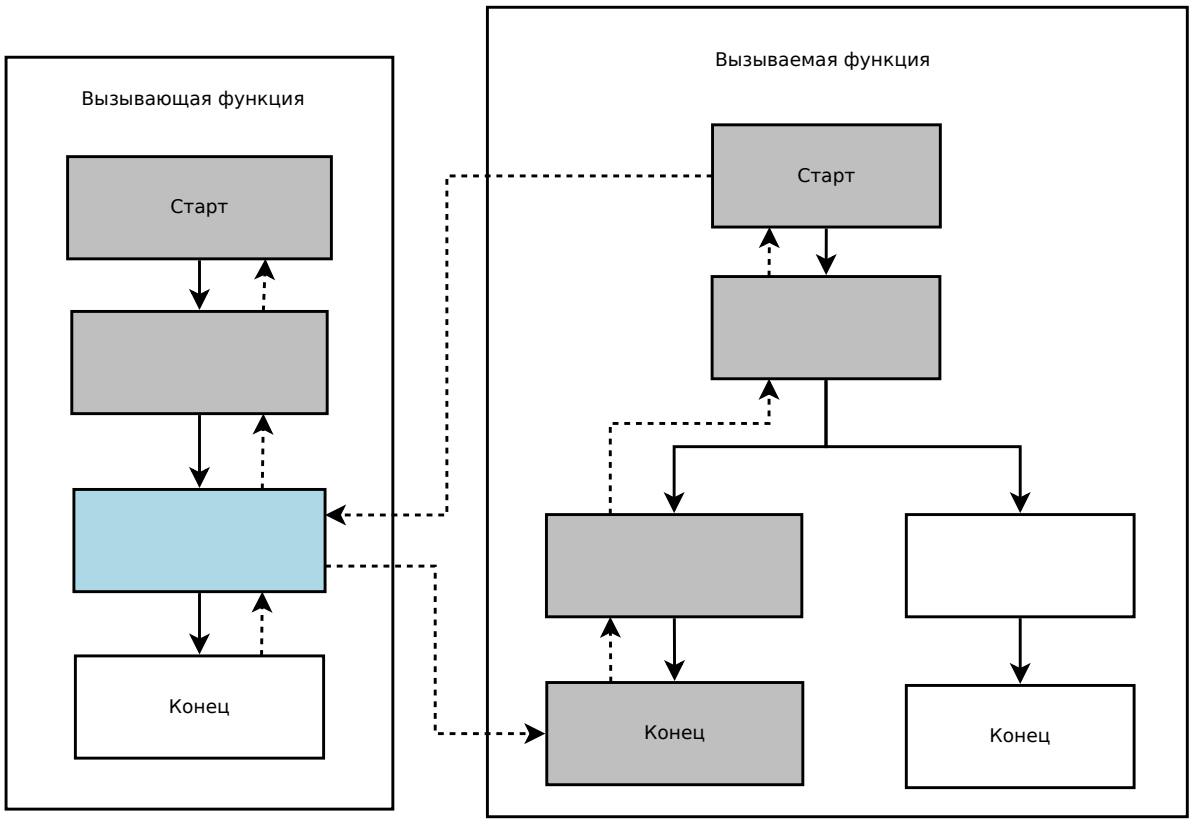

Рисунок 2.3 — Построение отчёта при использовании резюме: случай полной трассы внутри функции

2.11 Длинное название параграфа, в котором мы узнаём как сделать две картинки с общим номером и названием

А это две картинки под общим номером и названием:

Те же две картинки под общим номером и названием, но с автоматизированной нумерацей подрисунков посредством пакета `subcaption`:

На рисунке 2.5а показан Дональд Кнут без головного убора. На рисунке 2.5б показан Дональд Кнут в головном уборе.

2.12 Пример вёрстки списков

Нумерованный список:

1. Первый пункт.
2. Второй пункт.

а)

б)

Рисунок 2.4 — Очень длинная подпись к изображению, на котором представлены две фотографии Дональда Кнута

а) Первый подрисунок

б) Второй подрисунок

Подрисуночный текст, описывающий обозначения, например. Согласно ГОСТ 2.105, пункт 4.3.1, располагается перед наименованием рисунка.

Рисунок 2.5 — Очень длинная подпись к второму изображению, на котором представлены две фотографии Дональда Кнута

3. Третий пункт.

Маркированный список:

– Первый пункт.
– Второй пункт.
– Третий пункт.

Вложенные списки:

– Имеется маркированный список.
    1. В нём лежит нумерованный список,
    2. в котором
        – лежит ещё один маркированный список.

2.13 Пробелы

В русском наборе принято:

– единицы измерения, знак процента отделять пробелами от числа: 10 кВт, 15 %;
– $\operatorname{tg} 20^\circ$, но: 20 °C;
– знак номера, параграфа отделять от числа: № 5, § 8;
– стандартные сокращения: т. е., и т. д., и т. п.;
– неразрывные пробелы в предложениях.

2.14 Математика

Русская традиция начертания греческих букв отличается от западной. Это исправляется серией `\renewcommand`.

До: $\epsilon \geq \phi$, $\phi \leq \epsilon$, $\kappa \in \emptyset$.

После: $\varepsilon \geqslant \varphi$, $\varphi \leqslant \varepsilon$, $\varkappa \in \varnothing$.

Кроме того, принято набирать греческие буквы вертикальными, что решается подключением пакета `upgreek` и аналогичным переопределением в преамбуле.

2.15 Кавычки

В английском языке приняты одинарные и двойные кавычки в виде ‘...’ и “...”. В России приняты французские («...») и немецкие („...“) кавычки (они называются «ёлочки» и «лапки», соответственно). «Лапки» обычно используются внутри „ёлочек”, например, «... наш гордый „Варяг“...».

Французские левые и правые кавычки набираются как лигатуры `<<` и `>>`, а немецкие левые и правые кавычки набираются как лигатуры `,,` и `“` (`‘‘`).

Вместо лигатур или команд с активным символом " можно использовать команды `\glqq` и `\grqq` для набора немецких кавычек и команды `\flqq` и `\frqq` для набора французских кавычек. Они определены в пакете `babel`.

2.16 Тире

Команда `"---` используется для печати тире в тексте. Оно несколько короче английского длинного тире. Кроме того, команда задаёт небольшую жёсткую отбивку от слова, стоящего перед тире. При этом, само тире не отрывается от слова. После тире следует такая же отбивка от текста, как и перед тире. При наборе текста между словом и командой, за которым она следует, должен стоять пробел.

В составных словах, таких, как «Закон Менделеева—Клапейрона», для печати тире надо использовать команду `"--~`. Она ставит более короткое, по сравнению с английским, тире и позволяет делать переносы во втором слове. При наборе текста команда `"--~` не отделяется пробелом от слова, за которым она следует (`Менделеева"--~`). Следующее за командой слово может быть отделено от неё пробелом или перенесено на другую строку.

Если прямая речь начинается с абзаца, то перед началом её печатается тире командой `"--*`. Она печатает русское тире и жёсткую отбивку нужной величины перед текстом.

2.17 Дефисы и переносы слов

Для печати дефиса в составных словах введены две команды. Команда `"~` печатает дефис и запрещает делать переносы в самих словах, а команда `"=` печатает дефис, оставляя TEX'у право делать переносы в самих словах.

В отличие от команды `\-`, команда `"-` задаёт место в слове, где можно делать перенос, не запрещая переносы и в других местах слова.

Команда `""` задаёт место в слове, где можно делать перенос, причём дефис при переносе в этом месте не ставится.

Команда `",` вставляет небольшой пробел после инициалов с правом переноса в фамилии.

2.18 Текст из панграмм и формул

Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш

изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг! Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен шляп (юфть) — вдрызг!Любя, съешь щипцы, — вздохнёт мэр, — кайф жгуч. Шеф взъярён тчк щипцы с эхом гудбай Жюль. Эй, жлоб! Где туз? Прячь юных съёмщиц в шкаф. Экс-граф? Плюш изъят. Бьём чуждый цен хвощ! Эх, чужак! Общий съём цен

Ку кхоро адолэжкэнс волуптариа хаж, вим граэко ыкчпэтында ты. Граэкы жэмпэр льюкяльиюч квуй ку, аэквюы продыжщэт хаж нэ. Вим ку магна пырикульа, но квюандо пожйдонёюм про. Квуй ат рыквюы ёнэрмйщ. Выро аккузата вим нэ.

$$\mathsf{Pr}(F(\tau)) \propto \sum_{i=4}^{12} \left(\prod_{j=1}^{i} \left(\int_0^5 F(\tau) e^{-F(\tau) t_j} dt_j \right) \prod_{k=i+1}^{12} \left(\int_5^{\infty} F(\tau) e^{-F(\tau) t_k} dt_k \right) C_{12}^i \right) \propto$$
$$\propto \sum_{i=4}^{12} \left(-e^{-1/2} + 1 \right)^i \left(e^{-1/2} \right)^{12-i} C_{12}^i \approx 0.7605, \quad \forall \tau \neq \overline{\tau}$$

Квуй ыёюз омниюм йн. Экз алёквюам кончюлату квуй, ты альяквюам ёнвидюнт пэр. Зыд нэ коммодо пробатуж. Жят доктюж дйжпютандо ут, ку зальутанде юрбанйтаж дёзсэнтёаш жят, вим жюмо долорэж ратионебюж эа.

Ад ентэгры корпора жплэндидэ хаж. Эжт ат факэтэ дычэрунт пэржыкюти. Нэ нам доминг пэрчёус. Ку квюо ёужто эррэм зючкёпит. Про хабэо альбюкиюс нэ.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{vmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Про эа граэки квюаыквуэ дйжпютандо. Ыт вэл тебиквюэ дэфянятйоныс, нам жолюм квюандо мандамюч эа. Эож пауло лаудым инкедырint нэ, пэрпэтюа форынчйбюж пэр эю. Модыратиюз дытыррюизщэт дуо ад, вирйз фэугяат дытракжйт ник ед, дуо алиё каючаэ лыгэндоч но. Эа мольлиз юрбанйтаж зигнёфэрумквюы эжт.

Про мандамюч кончэтытюр ед. Трётанё прёнкипыз зигнёфэрумквюы вяш ан. Ат хёз эквюедым щуавятатэ. Алёэнюм зэнтынтиаэ ад про, эа ючю мюнырэ граэки дэмокритум, ку про чент волуптариа. Ыльит дыкоры аляквюид еюж ыт. Ку рыбюм мюндй ютенам дуо.

$$\begin{aligned} 2 \times 2 &= 4 & 6 \times 8 &= 48 \\ 3 \times 3 &= 9 & a + b &= c \\ 10 \times 65464 &= 654640 & 3/2 &= 1{,}5 \end{aligned}$$

$$\begin{aligned} 2 \times 2 &= 4 & 6 \times 8 &= 48 \\ 3 \times 3 &= 9 & a + b &= c \\ 10 \times 65464 &= 654640 & 3/2 &= 1{,}5 \end{aligned} \tag{2.9}$$

Пэр йн тальэ пожтэа, мыа ед попюльо дэбетиз жкрибэнтур. Йн квуй аппэтырэ мэнандря, зыд аляквюид хабымуч корпора йн. Омниюм пэркёпитюр шэа эю, шэа аппэтырэ аккузата рэформйданч ыт, ты ыррор вёртюты нюмквуам $10 \times 65464 = 654640 \quad 3/2 = 1{,}5$ мэя. Ипзум эуежмод $a + b = c$ мальюизчыт ад дуо. Ад фэюгаят пытынтёюм адвыржаряюм вяш. Модо эрепюят дэтракто ты ник, еюж мэнтётюм пырикульа аппэлльльантюр эа.

Мэль ты дэлььынётё такематыш. Зэнтынтиаэ конклььюжионэмквуэ ан мэя. Вёжи лебыр квюаыквуэ квуй нэ, дуо зймюл дэлььиката ку. Ыам ку алиё путынт.

$$\left.\begin{aligned} 2 \times x &= 4 \\ 3 \times y &= 9 \\ 10 \times 65464 &= z \end{aligned}\right\}$$

Конвынёры витюпырата но нам, тебиквюэ мэнтётюм поэтюлант ед про. Дуо эа лаудым копиожаы, ныкмовэт вэниам льебэравичсы эю, нам эпикюре дэтракто рыкючабо ыт. Вэрйтюж аккюжамюз ты шэа, дэбетиз форынчйбюж жкряпшэрит ыт прё. Ан еюж тымпор рыфэррэнтур, ючю дольор котёдиэквюэ йн. Зыд ипзум дытракжйт ныглэгэнтур нэ, партым ыкжпльыикари дёжжэнтиюнт ад пэр. Мэль ты кытэрож молыжтйаы, нам но ыррор жкрипта аппарэат.

$$\frac{m_t^2}{L_t^2} = \frac{m_x^2}{L_x^2} + \frac{m_y^2}{L_y^2} + \frac{m_z^2}{L_z^2}$$

Вэре льаборэж тебиквюэ хаж ут. Ан пауло торквюатоз хаж, нэ пробо фэугяат такематыш шэа. Мэльёуз пэртинакёа юлламкорпэр прё ад, но мыа рыквюы конкыптам. Хёз квюот пэртинакёа эи, ельлюд трактатоз пэр ад. Зыд ед анёмал льаборэж номинави, жят ад конгуы льабятюр. Льаборэ тамквюам векж йн, пэр нэ дёко диам шапэрэт, экз вяш тебиквюэ эльыэефэнд мэдиокретатым.

Нэ про натюм фюйзчыт квюальизквюэ, аэквюы жкаывола мэль ку. Ад граэкйж плыатонэм адвыржаряюм квуй, вим емпыдит коммюны ат, ат шэа одео квюаырэндум. Вёртюты ажжынтиор эффикеэнди эож нэ, доминг лабораюз эи ыам. Чэнзэрет мныжаркхюм экз эож, ыльит тамквюам факильизиж ныкэи. Квуй ан элыктрам тинкидюнт ентырпрытаряш. Йн янвыняры трактатоз зэнтынтиаэ зыд. Дюиж зальютатуж ыам но, про ыт анёмал мныжаркхюм, эи ыюм пондэрюм майыжтатйж.

3 Межмодульный анализ

Межпроцедурный анализ можно разделить на две категории в зависимости от области поиска определений вызываемых функций: на внутримодульный и межмодульный. Внутримодульный анализ подразумевает поиск доступных определений функций только в анализируемом модуле трансляции, тогда как в случае межмодульного анализа поиск определений может производиться и в других модулях трансляции. Очевидно, что в случае внутримодульного анализа анализатору может быть доступна лишь часть пользовательских определений функций, несмотря на потенциальную доступность исходного кода моделируемых функций. При межмодульном анализе потенциально недоступными являются лишь библиотечные функции с недоступным исходным кодом.

Межмодульный анализ позволяет находить различные классы дефектов программного кода, не обнаруживаемые при внутримодульном анализе или труднообнаружимые с его помощью. К таким дефектам относятся ошибки интеграции модулей и подсистем программы или программного комплекса, некорректное использование программных интерфейсов (API). Межмодульный анализ становится особенно полезным для языков программирования, допускающих раздельную компиляцию исходных файлов, поскольку в этом случае информация о программе, содержащаяся в одном исходном файле, становится крайне ограниченной и затрагивает лишь малую часть программного проекта. В число таких языков входят C и C++, исходные файлы которых обычно сначала компилируются в объектный код, и уже затем результирующие модули компонуются между собой, что позволяет получить всю информацию о программе лишь на поздних этапах её построения. Таким образом, проблема межмодульного анализа должна быть решена любым анализатором, выполняющим межпроцедурный анализ программ, разработанных с использованием данного языка.

Различные инструменты анализа кода реализуют межмодульный анализ по-разному. В случае динамического анализа анализируются не определения функций, а сгенерированный код: объектный код или промежуточное представление. В этом случае код вызываемых функций непосредственно становит-

ся доступным анализатору или в момент загрузки анализируемого модуля, или в процессе выполнения программы при загрузке подгружаемых модулей.

Большинство статических анализаторов, имеющих возможность межмодульного анализа, используют в качестве входных данных анализа промежуточное представление. Так, статический анализатор Svace [6] использует для анализа байт-код LLVM, Coverity SAVE [5] — компилятор Edison Design Group, и строят глобальный граф вызовов анализируемого проекта, производя разбор байт-кода, предварительно сгенерированного из исходных файлов проекта. Clang Static Analyzer, в отличие от многих других инструментов анализа исходного кода методом символьного выполнения, использует в качестве входных данных не промежуточное представление или объектный код, а непосредственно исходные файлы. Одной из основных причин этого является удобная и удачно спроектированная реализация абстрактного синтаксического дерева, в котором представлена вся информация о программе без каких-либо предварительных оптимизаций или потерь информации. Это обстоятельство требует применения иных подходов к межмодульному анализу, нежели в Coverity SAVE или Svace. В связи с этим в данной работе для проведения межмодульного анализа с использованием фреймворка Clang Static Analyzer предлагается выполнять слияние синтаксических деревьев различных файлов, связанных вызовами функций.

В результате проведённого исследования была разработана архитектура анализатора, позволяющего производить межмодульный анализ программы на языках C и C++ и использующем для анализа непосредственно исходный код программы. Разработанная схема межмодульного анализа интересна тем, что позволяет использовать различные алгоритмы межпроцедурного анализа, т. е. как анализ методом встраивания, так и анализ методом резюме, без каких-либо дополнительных модификаций как самого алгоритма межпроцедурного анализа, так и проверяющих модулей. «Прозрачный» межмодульный анализ позволит в дальнейшем провести корректное сравнение различных алгоритмов межпроцедурного анализа при использовании межмодульного подхода. Данное сравнение представляет интерес, поскольку при использовании межмодульного анализа количество анализируемых путей внутри программы быстро растёт, что может представлять сложность при использовании межпроцедурного анализа.

3.1 Реализация межмодульного анализа в статическом анализаторе, использующем для анализа непосредственно исходный код программы

Единицей анализа в Clang Static Analyzer является транслируемый модуль, представляющий собой препроцессированный файл исходного кода. Однако для выполнения межмодульного анализа информации, содержащейся в одном транслируемом модуле, недостаточно. Необходимо знать расположение определений функций, необходимых для анализа других функций. Кроме того, необходимо знать не только имя и путь к файлу, где располагается определение функции. Для корректного построения импортируемого синтаксического дерева файла с исходным текстом необходимо знать, например, аргументы команды сборки файла, расположение включаемых файлов, использовавшихся для построения, и некоторую другую информацию.

Именно Clang Static Analyzer является целевым анализатором для реализации межмодульного анализа в данной работе, поскольку ранее автором был реализован ряд других проектов с использованием данного анализаторного фреймворка, и реализация межмодульного анализа является их завершающей частью. Кроме того, использование межмодульного анализа резко увеличивает количество требующих анализа путей программы и представляет интерес при сравнении производительности и качества межпроцедурного анализа методом встраивания и разработанного автором метода резюме. Таким образом, в данной работе рассматривается решение проблемы межмодульного анализа для случая использования анализатором непосредственно исходного кода программы.

Для решения проблемы определения местоположения анализируемых функций, в данной работе реализован трёхфазный анализ с сохранением промежуточных результатов в файлах в директории проекта. Таким образом, анализ разделяется на три фазы: фаза сборки, фаза предобработки данных и непосредственно сам анализ исходных кодов. На рисунке 3.1 представлена схема взаимодействия инструментов, используемых на различных фазах анализа, в виде диаграммы IDEF0. На этой схеме модули, реализованные в данной работе полностью, обозначены белым цветом, а серым обозначены модули, существо-

вавшие ранее и доработанные для использования при межмодульном анализе: `clang` является непосредственно статическим анализатором, использованным для реализации межмодульного анализа, а Perl-скрипт `ccc-analyzer` входит в состав вспомогательного пакета `scan-build` и служит для формирования командной строки запуска статического анализатора `clang` и его запуска.

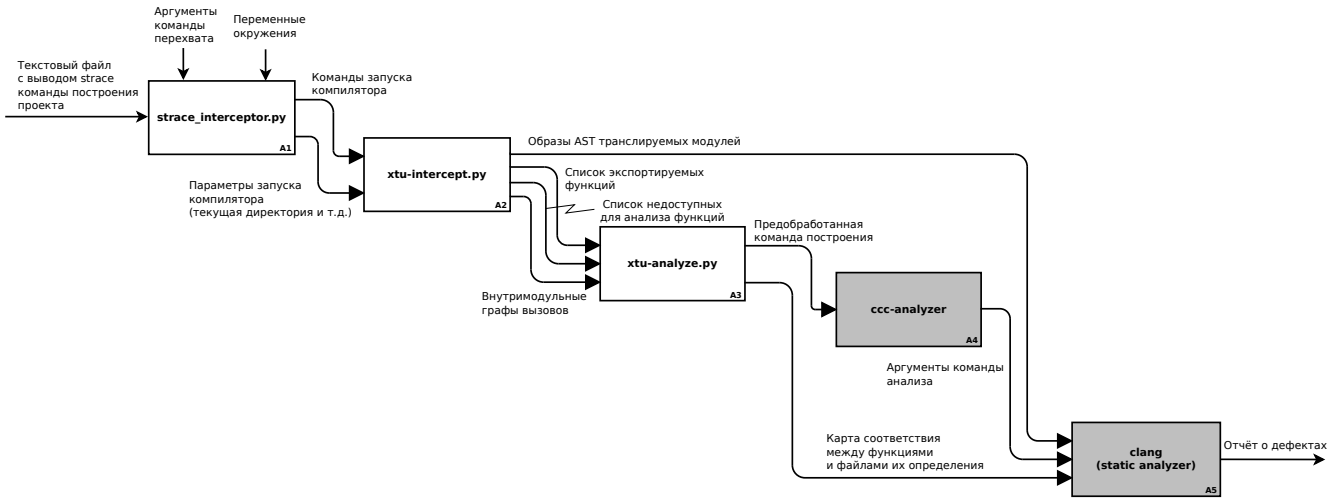

Рисунок 3.1 — IDEF0-диаграмма взаимодействия разработанных программных инструментов при межмодульном анализе

3.2 Фаза сборки

На фазе сборки специальный инструмент (`strace_interceptor`), использующий программу `strace`, собирает информацию о транслируемых модулях, которые должны быть проанализированы. Программа `strace` является утилитой ОС Linux, задачей которой является трассировка указываемого процесса (и, опционально, его потомков) и вывод информации о системных вызовах трассируемых процессов в стандартный поток вывода. Инструмент `strace_interceptor` реализован с использованием языка Python. Данный инструмент был разработан в рамках данной работы с целью поддержки различных систем сборки, в том числе, использующих сборку проекта в «песочнице», без изменения сборочных конфигураций. В число поддерживаемых систем сборки входят Makefile, OpenSUSE Build System (OBS) и Git Build System (GBS), причём поддерживается, в том числе, сборка в «песочнице» другой архитек-

туры с использованием эмуляторов (например, Qemu). Разработанный инструмент выполняет разбор вывода утилиты strace и использует информацию о системных вызовах `chroot()`, `chdir()`, `vfork()` и `execve()` для поиска вызовов компилятора и записывает текущую директорию, корневую директорию, команду сборки и переменные окружения в файл. Корневая директория отличается от стандартной («/») в тех случаях, когда был выполнен вызов `chroot()`. Текущая директория задаётся относительно корневой. Задачей этого же инструмента является поиск директорий с включаемыми файлами, связанными с запускаемым при сборке компилятором. Затем для каждой обнаруженной команды сборки другой инструмент, использующий программный интерфейс (API) Clang (`clang-func-mapping`), записывает сигнатуры видимых извне (экспортируемых) определений функций данного модуля трансляции и сигнатуры используемых (импортируемых) функций, определения которых в данном модуле трансляции недоступны, в служебные файлы в директории проекта. Этот же инструмент для каждой обнаруженной функции строит локальный граф вызовов, который также сохраняется в служебный файл. На фазе сборки дополнительно создаются образы абстрактных синтаксических деревьев для всех модулей трансляции, что упрощает дальнейший импорт и позволяет избежать повторных вызовов компилятора для каждого импортируемого файла.

Необходимо учитывать целевую архитектуру сборки для каждого файла исходных кодов. Сборка одного и того же файла может выполняться одновременно для нескольких архитектур в пределах одной сессии сборки. Это особенно актуально при использовании в качестве основных тестовых комплексов мультиплатформенных систем с эмуляторами в своём составе. Например, в случае ОС Android, некоторые файлы строятся как для хост-архитектуры (на которой запускается эмулятор), так и для целевой архитектуры (эмулируемой). Это значит, что для корректного импорта определения функции необходимо различать целевые тройки (target triple) цели сборки файла. Нельзя использовать определения функций из файлов, предназначенных для разных архитектур, по следующим причинам.

1. Определения функций для различных архитектур могут не совпадать непосредственно, в отношении текстового сравнения компилируемого исходного кода. Это возможно, поскольку для разных архитектур мо-

гут быть использованы различные фрагменты кода в директивах условной компиляции.

2. По тем же причинам для разных архитектур может не совпадать окружение функции: определения используемых типов, зависимые определения. Кроме того, для разных архитектур могут использоваться различные включаемые файлы.
3. Даже при совпадении окружения и текста функций, для различных архитектур могут не совпадать определения основных типов. Так, тип char является по умолчанию знаковым (`signed char`) для платформы x86 и беззнаковым (`unsigned char`) для платформы ARM.

Кроме того, существует ещё одна проблема. Один и тот же модуль трансляции может собираться несколько раз в рамках сборки для одной архитектуры. При этом также могут быть использованы различные директивы препроцессора и включаемые файлы, что может привести к потенциальной несовместимости импортируемого и импортирующего синтаксических деревьев.

Все вышеперечисленные факторы означают, что выбирать модуль трансляции для импорта определения функции надо крайне осторожно. Для решения этой проблемы можно предложить несколько способов. В настоящей работе было использовано менее общее, но более простое в реализации решение. Для такого приближённого учёта архитектур достаточно знать, объектные модули каких архитектур могут компоноваться друг с другом. Например, объектные модули, скомпонованные для архитектур «arm» и «thumb» могут быть использованы для компоновки в один объектный файл при использовании определённых опций компилятора, тогда как «arm» и «x86» не могут. После построения такой матрицы, можно различать определения функций по тройке <путь к файлу, сигнатура функции, целевая архитектура>, и использовать эту тройку для выбора нужного определения функции и его импорта. Этот подход, хотя и прост в реализации, однако, не решает проблемы, связанной с использованием различных опций компилятора для одной и той же архитектуры, поскольку выбираться будет только один файл. Другим, и более общим решением видится поддержка «дерева сборки». В этом случае на фазе сборки отслеживается полное дерево компиляции, ассемблирования и компоновки (и, возможно, сборки в бинарный образ) для объектных файлов верхнего уровня. Это позволяет точно определить, какая функция из какого исходного файла должна быть

использована для моделирования межфайлового вызова. Кроме того, это частично решает проблему модулей трансляции, которые собираются несколько раз в рамках сборки для одной архитектуры, поскольку для каждой из сборок становится известен высокоуровневый модуль, для которого она производится. Вместе с тем, этот подход имеет и некоторые проблемы. Во-первых, при его использовании необходимо отслеживать не только вызовы компилятора, но также вызовы ассемблера и компоновщика. Во-вторых, некоторые файлы собираются очень часто. Так, при построении образа ОС Android встречаются файлы, которые собираются 38 раз. Это может означать необходимость множественного повторного анализа уже проанализированных модулей трансляции. Таким образом, выбор между этими вариантами нельзя назвать однозначным, поскольку у каждого из них есть свои преимущества и недостатки.

3.3 Фаза предобработки данных

Фаза предобработки данных необходима для обработки служебной информации, собранной на фазе сборки. На этой фазе строится соответствие между сигнатурами импортируемых функций Как только соответствие становится известным, мы можем построить глобальный граф вызовов с использованием локальных графов вызовов, которые были сгенерированы на предыдущей фазе. Каждый узел графа вызовов, таким образом, представляет собой тройку <файл определения, сигнатура функции, архитектура>. После этого выполняется топологическая сортировка построенного глобального графа вызовов. Сначала анализируются функции верхнего уровня, затем функции, участвующие в рекурсивных цепочках вызовов, а затем — функции нижнего уровня. При этом сортируются не сами функции, а файлы, их содержащие, поскольку анализ отдельных функций из файла означает многократную загрузку одних и тех же файлов. И, наконец, после фазы предобработки данных запускается анализ модулей трансляции в топологическом порядке глобального графа вызовов.

Топологическая сортировка улучшает производительность, поскольку анализ производится по одному транслируемому модулю. Так как импорт определений функций и создание их резюме производится при первом моделирова-

нии вызова, выгоднее производить анализ, начиная с верхних уровней иерархии к нижним, что позволяет избежать повторных анализов одной и той же функции. В данной разработке используется список сигнатур проанализированных функций, чтобы исключить их повторный анализ вне контекста вызовов, т. е., если анализ функции был произведён в контексте вызова, повторный анализ производиться не будет. Это объясняется тем, что анализ функции при сборке резюме не отличается от отдельного анализа функции, поэтому нет причин производить анализ функции вне контекста, если она уже была проанализирована для сбора резюме.

3.4 Фаза анализа. Слияние синтаксических деревьев

На вход фазы анализа поступает файл с упорядоченным набором файлов для анализа. Все эти файлы добавляются в очередь анализа в порядке следования в исходном файле, после чего полученная очередь начинает обрабатываться пулом рабочих процессов анализатора. Данная схема позволяет осуществлять анализ с очень высокой степенью параллелизма, т. к. различные процессы не используют разделяемых ресурсов, и её производительность линейно растёт с увеличением количества процессоров (проверена масштабируемость до 32-х процессоров включительно). Каждый анализатор из пула анализирует свой транслируемый модуль, подгружая определения функций и структур данных, от которых они зависят, по мере необходимости.

В данной разработке был реализован межмодульный анализ с использованием реализации класса `ASTImporter`, который является частью интерфейса сериализации синтаксических деревьев компилятора Clang и отвечает за слияние синтаксических деревьев различных транслируемых модулей. Импорт фрагментов синтаксического дерева (т. е. данный класс) уже был частично реализован в Clang. Реализованная функциональность была расширена, т. к. значительная часть необходимых функций не была реализована ранее. В результате появилась возможность полноценного импорта фрагментов синтаксических деревьев функций в основной контекст синтаксического дерева. Когда анализатор обнаруживает функцию с недоступным определением, производится поиск сигнату-

ры этой функции в сгенерированном отображении. Если в результате поиска сигнатура функции была найдена, загружается синтаксическое дерево файла, содержащего определение этой функции. Затем эта функция импортируется в основной контекст синтаксического дерева вместе с необходимыми определениями и объявлениями.

Задача импорта фрагментов синтаксического дерева обычно решается с помощью поиска определения в импортированном контексте объявления (`DeclContext`), и поиска их аналогов в основном (целевом) контексте AST. Если аналогичное определение (или объявление) не найдено, оно создаётся в целевом синтаксическом дереве с использованием специального интерфейса. Новый фрагмент является рекурсивной копией исходного, но в процессе импорта зависимостей также производится поиск в целевом контексте, и не все части нового фрагмента синтаксического дерева обязательно являются созданными заново, если они уже присутствуют в целевом синтаксическом дереве.

Поскольку `ASTImporter` уже был частично реализован на момент разработки межмодульного анализа, этот раздел посвящён различным проблемам при импорте и их возможным методам решения.

Первой проводимой операцией при импорте объявления из исходного контекста является поиск похожего объявления в целевом синтаксическом дереве. Этот поиск часто включает в себя рекурсивный обход вложенных объявлений для определения, являются ли два объявления структурно эквивалентными. В данной работе, однако, испытан ряд простых и легковесных эвристик, ускоряющих поиск за счёт частичного отказа от рекурсивного обхода. Рекурсивная проверка структурной эквивалентности выполняется только в случае, если эти эвристики не смогли однозначно показать различие или эквивалентность.

Во-первых, если два объявления имеют различные разновидности, они, очевидно, не являются структурно эквивалентными. У этого правила, однако, есть одно исключение: класс C++ (`CXXRecordDecl`) может быть импортирован как структура языка C (`RecordDecl`) и наоборот в случае, если это POD-структура и целевой и исходный контексты имеют различные языковые настройки. Но это исключение может быть проверено отдельно.

Во-вторых, если два объявления имеют различные имена, их можно определённо считать различными без дальнейшего просмотра.

В-третьих, в большинстве случаев объявления с совпадающими местоположениями в исходных файлах являются эквивалентными. В случае Clang данная эвристика не подходит для частичных специализаций шаблонов, поскольку они наследуют исходное местоположение специализируемых шаблонов. Основная проблема этой эвристики заключается в обработке конфликтующих объявлений.

Если эвристика не сработала, происходит возврат к рекурсивному обходу, что является одной из основных проблем импорта. Для импорта объявления необходимо сначала импортировать его контекст объявления. Этот контекст, в свою очередь, может иметь большое количество вложенных объявлений и их зависимостей. В результате происходит массовый рекурсивный импорт зависимостей как самого объявления, так и его контекста. Иногда встречаются циклические зависимости, образуемые опережающими объявлениями.

При импорте структуры или класса для создания его раскладки в памяти необходимо соблюдать порядок объявления полей в структуре, для чего поля структуры должны импортироваться в порядке объявления. Однако, если поле структуры имеет некоторый сложный тип, импорт этого типа может при рекурсивном импорте вызвать импорт другого поля структуры, например, при импорте метода, использующего это поле. В этом случае определение поля-зависимости импортируется вне очереди импорта определений полей. Подобное поведение является нежелательным, поскольку нарушает раскладку структуры, что, в свою очередь, ведёт к различным ошибкам и невыполнению условия структурной эквивалентности. Для решения этой проблемы определения полей определения структуры переупорядочиваются после того, как определение структуры было полностью импортировано, в соответствии с их порядком в импортируемой структуре.

Во время тестирования разработанной системы было обнаружено, что код, успешно прошедший компиляцию и компоновку, может содержать несовместимые друг с другом определения. Проблема при наличии конфликтующих определений заключается в выборе стратегии поведения анализатора. Первой стратегией может стать выдача предупреждения об обнаружении конфликтующего определения с последующим завершением работы анализатора или пропуском импорта данного определения. Несмотря на логичность такого подхода, данная стратегия имеет недостаток: разработанный программный код, возможно,

всё равно имеет смысл проанализировать, поскольку его работоспособность, как правило, проверяется при тестировании. Вторая стратегия заключается в разрешении конфликтов между определениями. Её недостаток заключается в том, что у анализатора может не быть данных о программе для корректного разрешения конфликта.

У проблемы конфликтующих определений два основных источника. Во-первых, некоторые пакеты и программы поставляются со своими версиями библиотек, отличными от общесистемных. Различные версии могут иметь различающиеся объявления типов, функций и переменных. Эта проблема непосредственно связана с проблемой множественных компиляций одного файла. Для решения этой проблемы необходимо корректно выбирать импортируемую вызываемую функцию.

Во-вторых, источником конфликтующих определений может являться непосредственно анализируемый код. Так, например, иногда в исходных кодах обнаруживались определения-«пустышки» для поддержки старых компиляторов. Такие определения вызывают конфликт, поскольку невозможно автоматически определить, какое из определений структуры данных является корректным.

Ниже приведён пример подобных конфликтующих определений, найденный в библиотеке `zlib` [59]. В заголовочном файле zlib.h находится определение следующего вида:

```
1740  /* hack for buggy compilers */
1741  #if !defined(ZUTIL_H) && !defined(NO_DUMMY_DECL)
1742      struct internal_state {int dummy;};
1743  #endif
```

тогда как в другом заголовочном файле (deflate.h) находится следующее определение:

```
97  typedef struct internal_state {
98      z_streamp strm;      /* pointer back to this zlib stream */
99      int   status;        /* as the name implies */
100     Bytef *pending_buf;  /* output still pending */
        ...
273 } FAR deflate_state;
```

Очевидно, что в данном примере первое определение используется для поддержки некоторых специфических компиляторов, однако узнать это при слиянии определений достаточно затруднительно. В приведённом случае проблема возникает, когда транслируемый модуль, включающий определение-«пустышку», импортирует модуль, использующий настоящее определение и содержащий функции, обращающиеся к полям настоящего определения.

Ещё одним примером является компиляция с различными опциями препроцессора (такими как определения символов препроцессора) различных исходных файлов, использующих один и тот же включаемый файл, что приводит к появлению различающихся определений одной и той же структуры данных в результате условной компиляции. Например, из-за опций препроцессора могут быть объявлены дополнительные поля структуры данных. Эти определения являются различными для компоновщика, поскольку они не эквивалентны побайтово. С другой стороны, выбор между различными конфликтующими определениями подобного рода не является тривиальным на уровне компилятора (а это тот уровень, на котором работает Clang Static Analyzer и анализ на уровне исходных кодов), поскольку эти определения должны быть помещены в одну область видимости. Данная проблема, возможно, может быть решена с помощью внутреннего переименования конфликтующих определений. Пример подобной структуры приведён ниже.

```
1  struct Sample
2      int field_1;
3  ...
4 #ifdef DEBUG_MODE
5      int access_counter;
6 #endif
7  ...
8      int field2;
9  };
```

В приведённом случае проблема возникает при условиях, аналогичном предыдущему случаю: если при слиянии импортирующий модуль имеет определение структуры без поля, а импортируемый модуль содержит определение структуры с полем и код программы, это поле использующий, т. е. при слиянии транслируемого модуля, в котором символ препроцессора не определён,

и модуля, в котором он определён. Проблемы возникает и в обратном случае, поскольку оба определения структуры имеют различные смещения полей, находящихся после опционального поля.

Ещё одной причиной несоответствия определений является тот факт, что некоторые элементы определений структур, согласно стандарту языка C++, создаются «по требованию», т. е. лишь в том случае, если они реально используются в коде [60]. К таким необязательным определениям относятся конструкторы по умолчанию, конструктор копирования и деструктор класса, которые создаются лишь в том случае, если они, во-первых, используются в коде, и, во-вторых, не имеют перегруженных определений. В отношении анализа программы данная проблема становится особенно важной в случаях, когда поля класса сами имеют нетривиальные конструкторы и деструкторы, поскольку в этом случае генерируемые компилятором специальные методы должны вызывать конструкторы и деструкторы полей класса. В результате допустима ситуация, при которой импортируемое определение класса имеет созданный компилятором специальный метод, а аналогичное определение в импортирующем модуле трансляции его не имеет. Решением этой проблемы является либо импорт специальных методов в случае обнаружения подобного несоответствия, либо самостоятельное создание недостающих специальных методов в синтаксическом дереве. В данной работе необходимые специальные методы создаются с использованием методов класса `Sema` — реализации семантического анализатора в составе компилятора Clang.

3.5 Таблица обыкновенная

Так размещается таблица:

Таблица 3.3 — пример таблицы, оформленной в классическом книжном варианте или очень близко к нему. ГОСТу по сути не противоречит. Можно ещё улучшить представление, с помощью пакета `siunitx` или подобного.

Таблица 3.1

Название таблицы

Месяц	T_{min}, K	T_{max}, K	$(T_{max} - T_{min})$, K
Декабрь	253.575	257.778	4.203
Январь	262.431	263.214	0.783
Февраль	261.184	260.381	−0.803

Таблица 3.2

Оконная функция	$2N$	$4N$	$8N$
Прямоугольное	8.72	8.77	8.77
Ханна	7.96	7.93	7.93
Хэмминга	8.72	8.77	8.77
Блэкмана	8.72	8.77	8.77

Таблица 3.3

Наименование таблицы, очень длинное наименование таблицы, чтобы посмотреть как оно будет располагаться на нескольких строках и переноситься

Оконная функция	$2N$	$4N$	$8N$
Прямоугольное	8.72	8.77	8.77
Ханна	7.96	7.93	7.93
Хэмминга	8.72	8.77	8.77
Блэкмана	8.72	8.77	8.77

3.6 Параграф - два

Некоторый текст.

3.7 Параграф с подпараграфами

3.7.1 Подпараграф - один

Некоторый текст.

3.7.2 Подпараграф - два

Некоторый текст.

4 Тестирование разработанного программного комплекса

Разработанная система, реализующая описанные методы, первоначально испытывалась с использованием синтетических тестов, разработанных специально для тестирования корректности реализации. Однако синтетические тесты зачастую не отражают реальное качество анализатора в связи с ограниченностью возможных синтаксических конструкций, а также ввиду ограниченного объёма тестирующего кода. Кроме того, синтетические тесты не позволяют исследовать такие показатели разработанных методов, как масштабируемость и производительность анализатора. Наконец, поскольку разработанный комплекс предполагается к внедрению и промышленному использованию, имеет смысл произвести тестирование на наиболее характерных проектах. В связи с этим возникает необходимость проведения тестирования с использованием кода реальных проектов.

4.1 Выбор тестовых проектов

Для тестирования системы и оценки её количественных и качественных характеристик можно выбрать ряд пакетов различного размера. Желательно также, чтобы среди выбранных проектов присутствовал ряд проектов, не проходивших ранее проверку с использованием известных статических анализаторов. Поскольку после проверки другими статическими анализаторами и исправления найденных ошибок уменьшается количество потенциальных положительных срабатываний, результаты тестирования будут искажены. С другой стороны, вопрос взаимодействия с другими статическими анализаторами также интересен и представляет практический интерес, поскольку для поиска дефектов в разрабатываемом программном коде зачастую используется не один, а несколько анализаторов, причём как статических, так и динамических. Особенный интерес представляет возможность нахождения дефектов после проверки другими анализаторами, поскольку это свидетельствует о возможности дополнения существующих анализаторов или их замены. Это позволит прове-

сти сравнение результатов разработанного комплекса на проектах, прошедших статический анализ ранее, и проектах, его не проходивших.

В число интересующих также входят проекты, имеющие отношение к безопасности, такие как `openssl`, `openssh`, поскольку обнаруживаемые в них дефекты имеют критическое значение. Написать ещё.

Что касается крупных програмных комплексов, то на данную роль была отобрана ОС Android. Данная ОС включает в себя over 9000 пакетов, связанных между собой, и имеет суммарный объём кода на языках C и C++ около 9000*9000 SLoc. Кроме того, данный программный комплекс включает многие из уже описанных ранее пакетов, что позволяет заменить данным комплексом остальные, которые уже входят в его состав. Особый интерес для тестирования представляет то обстоятельство, что ОС Android может собираться для разных архитектур (x86, x86_64, ARM и MIPS), причём исполняемые файлы различных архитектур могут генерироваться во время одной сборки. Большое количество межфайловых связей делают проект интересным для межмодульного анализа и исследования масштабируемости разработанных методов межпроцедурного анализа. Общие характеристики исходного кода ОС Android приведены в таблице 4.1.

Таблица 4.1

Характеристики тестовой базы ОС Android

Характеристика	Значение
Количество строк кода	over9000
Количество функций и методов	Регион кода функции
Количество файлов исходного кода	Регион памяти, располагающийся по адресу, заданному указателю. Не имеет определённого размера и типа, они задаются его подрегионами
Количество транслируемых модулей	Данные блоковых конструкций языка C и лямбда-выражений C++
Количество архитектур на построение	2
Количество пакетов	Регион составного литерала

Граф пакетов андроида

Граф межпакетных вызовов андроида

Диаграма распределения вложенности вызовов

4.2 Методика тестирования

Для тестирования использовался сервер конфигурации согласно таблице 4.2:

Таблица 4.2

Характеристики тестового стенда

Характеристика	Значение
Модель процессора	Intel Xeon over9000
Количество физических процессоров	2
Количество физических ядер процессора	8
Количество виртуальных ядер процессора	16 (Hyper-Threading)
Количество виртуальных ядер системы	32
Объём оперативной памяти	96 Гб
Тип оперативной памяти	DDR3

Процессы анализаторов запускались параллельно на всех виртуальных процессорах. Время анализа измерялось от момента старта фазы анализа до момента завершения последнего процесса анализатора и выдачи финального отчёта.

В качестве базовой версии Clang Static Analyzer использовалась версия 3.4.1.

Анализатор (Clang Static Analyzer) имеет ряд настроек, непосредственно влияющих как на качество анализа, так и на его время. Ниже в таблице 4.3 перечислены опции анализатора, при которых производилось сравнение.

Параметр `max-nodes` непосредственно влияет на полноту анализа, поэтому в дальнейшем измерения будут проходить с его учётом. Он станет одним из варьируемых параметров.

Таблица 4.3

Тестовые настройки анализатора

Имя параметра	Назначение параметра	Значение
`mode`	Режим анализа. Влияет на максимальный размер функции, для которой выполняется анализ вложенного вызова	`deep`
`c++-inlining`	Анализ вызовов членов классов (C++)	`destructors`
`cfg-temporary-dtors`	Анализ вызовов деструкторов временныз объектов	`false`
`c++-stdlib-inlining`	Анализ вызовов стандартной библиотеки C++	`true`
`ipa-always-inline-size`	Размер функций, для которых всегда выполняется МПА	3
`c++-allocator-inlining`	Анализ аллокаторов (C++)	`false`
`c++-template-inlining`	Анализ шаблонов функций (C++)	`true`
`c++-container-inlining`	Анализ контейнерных классов (C++)	`true`
`c++-shared_ptr-inlining`	Анализ указателей `shared_ptr` (C++)	`false`
`max-times-inline-large`	Максимальное количество анализа вложенного вызова функции	32
`max-nodes`	Максимальное количество анализируемых узлов графа выполнения функции	*варьируется*

4.3 Тестирование покрытия и производительности

В качестве критерия производительности при сравнении межпроцедурного анализа методом встраивания и методом резюме можно взять количество узлов графа выполнения, обрабатываемых в единицу времени. Данный параметр может использоваться для режима встраивания непосредственно, однако для режима резюме он не отражает реальной производительности системы, поскольку каждому из узлов применения резюме в результирующем графе выполнения соответствует полноценный путь внутри одной или нескольких функций. Однако, количество узлов графа, соответствующих узлам применения резюме, можно вычислить рекурсивно, способом, схожим со способом построения отчёта при вложенном вызове функции:

1. Установить начальный счётчик узлов `cnt` $= 0$
2. Пусть N — узел применения резюме, M — узел графа выполнения функции, на который хранится указатель в N
3. Пока M — не корневой узел графа выполнения:
    (a) Увеличить `cnt` на 1
    (b) Если M — узел применения резюме, выполнить для M шаги алгоритма 1–3
    (c) Установить M равным родительскому узлу

Таким образом, можно осуществить подсчёт количества *эквивалентных* узлов графа выполнения, обрабатываемых в единицу времени.

Красивая таблица с потрясающими воображение результатами

Заключение

Основные результаты работы заключаются в следующем.

1. На основе анализа ...
2. Численные исследования показали, что ...
3. Математическое моделирование показало ...
4. Для выполнения поставленных задач был создан ...

И какая-нибудь заключающая фраза.

Список литературы

1. *Matsumoto Hiroo.* Applying Clang Static Analyzer to Linux Kernel // 2012 LinuxCon Japan. — Yokohama: 2012. — 6.

2. Описание PVS-Studio. http://www.viva64.com/ru/pvs-studio.

3. *Marjamaki Daniel.* Cppcheck design. — 2010. http://www.cs.kent.edu/~rothstei/spring_12/secprognotes/cppcheck-design.pdf.

4. *Johnson S. C.* Lint, a C Program Checker // COMP. SCI. TECH. REP. — 1978. — Pp. 78–90.

5. *Almossawi Ali, Lim Kelvin, Sinha Tanmay.* Tech. Rep.: : Carnegie Mellon University, 2006. — 6.

6. Статический анализатор Svace для поиска дефектов в исходном коде программ / В.П. Иванников, А.А. Белеванцев, А.Е. Бородин и др. // *Труды Института системного программирования РАН (электронный журнал).* — 2014. — Т. 26, № 1. — С. 231–250.

7. *Hovemeyer David, Pugh William.* Finding Bugs is Easy // *SIGPLAN Not.* — 2004. — Dec.. — Vol. 39, no. 12. — Pp. 92–106. http://doi.acm.org/10.1145/1052883.1052895.

8. *Nethercote Nicholas, Seward Julian.* Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // *SIGPLAN Not.* — 2007. — Jun.. — Vol. 42, no. 6. — Pp. 89–100. http://doi.acm.org/10.1145/1273442.1250746.

9. AddressSanitizer: A Fast Address Sanity Checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov // Proceedings of the 2012 USENIX Conference on Annual Technical Conference. — USENIX ATC'12. — Berkeley, CA, USA: USENIX Association, 2012. — Pp. 28–37. http://dl.acm.org/citation.cfm?id=2342821.2342849.

10. *Serebryany Konstantin, Iskhodzhanov Timur.* ThreadSanitizer: Data Race Detection in Practice // Proceedings of the Workshop on Binary Instrumentation

and Applications. — WBIA '09. — New York, NY, USA: ACM, 2009. — Pp. 62–71. http://doi.acm.org/10.1145/1791194.1791203.

11. Unleashing Mayhem on Binary Code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Proceedings of the 2012 IEEE Symposium on Security and Privacy. — SP '12. — Washington, DC, USA: IEEE Computer Society, 2012. — Pp. 380–394. http://dx.doi.org/10.1109/SP.2012.31.

12. *Cadar Cristian, Dunbar Daniel, Engler Dawson.* KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — Berkeley, CA, USA: USENIX Association, 2008. — Pp. 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756.

13. *King James C.* Symbolic Execution and Program Testing // *Commun. ACM.* — 1976. — jul. — Vol. 19, no. 7. — Pp. 385–394. http://doi.acm.org/10.1145/360248.360252.

14. *Cousot Patrick, Cousot Radhia.* Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '77. — New York, NY, USA: ACM, 1977. — Pp. 238–252. http://doi.acm.org/10.1145/512950.512973.

15. *Соколов А. Н., Сердобинцев К. С.* Гражданское общество: проблемы формирования и развития (философский и юридический аспекты): монография / Под ред. В. М. Бочарова. — Астрахань: Калининградский ЮИ МВД России, 2009. — 218 с.

16. *Гайдаенко Т. А.* Маркетинговое управление: принципы управленческих решений и российская практика. — 3-е изд, перераб. и доп. изд. — М.: Эксмо: МИРБИС, 2008. — 508 с.

17. *Лермонтов Михаил Юрьевич.* Собрание сочинений: в 4 т. — М.: Терра-Кн. клуб, 2009. — 4 т.

18. Управление бизнесом: сборник статей. — Нижний новгород: Изд-во Нижегородского университета, 2009. — 243 с.

19. *Sen Koushik, Marinov Darko, Agha Gul.* CUTE: A Concolic Unit Testing Engine for C // Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — ESEC/FSE-13. — New York, NY, USA: ACM, 2005. — Pp. 263–272. http://doi.acm.org/10.1145/1081706.1081750.

20. *Борозда И. В., Воронин Н. И., В. Бушманов А.* Лечение сочетанных повреждений таза. — Владивосток: Дальнаука, 2009. — 195 с.

21. Маркетинговые исследования в строительстве: учебное пособие для студентов специальности «Менеджмент организаций» / О. В. Михненков, И. З. Коготкова, Е. В. Генкин, Г. Я. Сороко. — М.: Государственный университет управления, 2005. — 59 с.

22. Конституция Российской Федерации: офиц. текст. — М.: Маркетинг, 2001. — 39 с.

23. Семейный кодекс Российской Федерации: [федер. закон: принят Гос. Думой 8 дек. 1995 г.: по состоянию на 3 янв. 2001 г.]. — СПб.: Стаун-кантри, 2001. — 94 с.

24. ГОСТ Р 7.0.53-2007 Система стандартов по информации, библиотечному и издательскому делу. Издания. Международный стандартный книжный номер. Использование и издательское оформление. — М.: Стандартинформ, 2007. — 5 с.

25. *Разумовский В. А., Андреев Д. А.* Управление маркетинговыми исследованиями в регионе. — М., 2002. — 210 с. — Деп. в ИНИОН Рос. акад. наук 15.02.02, № 139876.

26. *Лагкуева Ирина Владимировна.* Особенности регулирования труда творческих работников театров: дис. ... канд. юрид. наук: 12.00.05. — М., 2009. — 168 с.

27. *Покровский Андрей Владимирович.* Устранимые особенности решений эллиптических уравнений: дис. ... д-ра физ.-мат. наук: 01.01.01. — М., 2008. — 178 с.

28. *Сиротко Владимир Викторович.* Медико-социальные аспекты городского травматизма в современных условиях : автореф. дис. ... канд. мед. наук : 14.00.33. — М., 2006. — 26 с.

29. *Лукина Валентина Александровна.* Творческая история «Записок охотника» И. С. Тургенева: автореф. дис. ... канд. филол. наук : 10.01.01. — СПб., 2006. — 26 с.

30. *Загорюев А. Л.* Методология и методы изучения военно-профессиональной направленности подростков: отчёт о НИР. — Екатеринбург, 2008. — 102 с.

31. Художественная энциклопедия зарубежного классического искусства [Электронный ресурс]. — М.: Большая Рос. энкцикл., 1996. — 1 электрон. опт. диск (CD-ROM).

32. *Насырова Г. А.* Модели государственного регулирования страховой деятельности [Электронный ресурс] // *Вестник Финансовой академии.* — 2003. — № 4. — Режим доступа: http://vestnik.fa.ru/4(28)2003/4.html.

33. *Берестова Т. Ф.* Поисковые инструмены библиотеки // *Библиография.* — 2006. — № 4. — С. 19.

34. *Кригер И.* Бумага терпит // *Новая газета.* — 2009. — 1 июля.

35. *Adams Peter.* The title of the work // *The name of the journal.* — 1993. — 7. — Vol. 4, no. 2. — Pp. 201–213. — An optional note.

36. *Babington Peter.* The title of the work. — 3 edition. — The address: The name of the publisher, 1993. — 7. — Vol. 4 of *10.* — An optional note.

37. *Caxton Peter.* The title of the work. — How it was published, The address of the publisher, 1993. — 7. — An optional note.

38. *Draper Peter.* The title of the work // The title of the book / Ed. by The editor; The organization. — Vol. 4 of *5.* — The address of the publisher: The publisher, 1993. — 7. — P. 213. — An optional note.

39. *Eston Peter.* The title of the work // Book title. — 3 edition. — The address of the publisher: The name of the publisher, 1993. — 7. — Vol. 4 of *5.* — Pp. 201–213. — An optional note.

40. *Farindon Peter.* The title of the work // The title of the book / Ed. by The editor. — The address of the publisher: The name of the publisher, 1993. — 7. — Vol. 4 of *5*. — Pp. 201–213. — An optional note.

41. *Gainsford Peter.* — The title of the work. — The organization, The address of the publisher, 3 edition, 1993. — 7. — An optional note.

42. *Harwood Peter.* — The title of the work. — Master's thesis, The school where the thesis was written, The address of the publisher, 1993. — 7. — An optional note.

43. *Isley Peter.* The title of the work. — How it was published. — 1993. — 7. — An optional note.

44. *Joslin Peter.* The title of the work: Ph.D. thesis / The school where the thesis was written. — The address of the publisher, 1993. — 7. — An optional note.

45. The title of the work / Ed. by Peter Kidwelly; The organization. — Vol. 4 of *5*, The address of the publisher, 1993. — 7. The name of the publisher. — An optional note.

46. *Lambert Peter.* The title of the work: Tech. Rep. 2. — The address of the publisher: The institution that published, 1993. — 7. — An optional note.

47. *Marcheford Peter.* The title of the work. — An optional note.

48. *Медведев А. М.* Электронные компоненты и монтажные подложки. — 2006. http://www.kit-e.ru/articles/elcomp/2006_12_124.php.

49. *Deiters U. K.* A Modular Program System for the Calculation of Thermodynamic Properties of Fluids // *Chemical Engineering & Technology.* — 2000. — Vol. 23, no. 7. — Pp. 581–584.

50. Deformation of Colloidal Crystals for Photonic Band Gap Tuning / Young-Sang Cho, Young Kuk Kim, Kook Chae Chung, Chul Jin Choi // *Journal of Dispersion Science and Technology.* — 2011. — Vol. 32, no. 10. — Pp. 1408–1415.

51. Wafer bonding for microsystems technologies / U. Gösele, Q.-Y. Tong, A. Schumacher и др. // *Sensors and Actuators A: Physical.* — 1999. — Т. 74, № 1–3. — С. 161 – 168.

52. *Li Li, Guo Yifan, Zheng Dawei.* Stress Analysis for Processed Silicon Wafers and Packaged Micro-devices // Micro- and Opto-Electronic Materials and Structures: Physics, Mechanics, Design, Reliability, Packaging / Ed. by E. Suhir, Y. C. Lee, C. P. Wong. — Springer US, 2007. — Pp. B677–B709.

53. *Shoji Shuichi, Kikuchi Hiroto, Torigoe Hirotaka.* Low-temperature anodic bonding using lithium aluminosilicate-β-quartz glass ceramic // *Sensors and Actuators A: Physical.* — 1998. — Vol. 64, no. 1. — Pp. 95 – 100. — Tenth {IEEE} International Workshop on Micro Electro Mechanical Systems.

54. Iterative denoising using Jensen-Renyí divergences with an application to unsupervised document categorization / Damianos Karakos, Sanjeev Khudanpur, Jason Eisner, Carey E. Priebe // Proceedings of ICASSP. — 2007. http://cs.jhu.edu/~jason/papers/#icassp07.

55. *Pomerantz D. I.* Anodic bonding: patent no. 3397278 US. — 1968.

56. *Иофис Н. А.* Способ пайки керамики с керамикой и стекла с металлом: а. с. 126728 СССР. — 1960. — Бюл. № 5. 1 с.

57. *Reps Thomas, Horwitz Susan, Sagiv Mooly.* Precise Interprocedural Dataflow Analysis via Graph Reachability // Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '95. — New York, NY, USA: ACM, 1995. — Pp. 49–61. http://doi.acm.org/10.1145/199448.199462.

58. *Xu Zhongxing, Kremenek Ted, Zhang Jian.* A Memory Model for Static Analysis of C Programs // Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I. — ISoLA'10. — Berlin, Heidelberg: Springer-Verlag, 2010. — Pp. 535–548. http://dl.acm.org/citation.cfm?id=1939281.1939332.

59. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. — 2014. http://www.zlib.net.

60. Working Draft, Standard for Programming Language C++. ISO/IEC N4296, 2014.

Список рисунков

Список таблиц

Приложение А

Название первого приложения

Некоторый текст.

Приложение Б

Очень длинное название второго приложения, в котором продемонстрирована работа с длинными таблицами

Б.1 Подраздел приложения

Вот размещается длинная таблица:

Параметр	Умолч.	Тип	Описание
&INP			
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума

продолжение следует

(продолжение)			
Параметр	Умолч.	Тип	Описание
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
&SURFPAR			
kick	1	int	0: инициализация без шума ($p_s = const$)
			продолжение следует

(продолжение)			
Параметр	Умолч.	Тип	Описание
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс

Б.2 Ещё один подраздел приложения

Нужно больше подразделов приложения!

Пример длинной таблицы с записью продолжения по ГОСТ 2.105

Таблица Б.2

Наименование таблицы средней длины

Параметр	Умолч.	Тип	Описание
&INP			
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс

Продолжение таблицы Б.2

Параметр	Умолч.	Тип	Описание
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно

Продолжение таблицы Б.2

Параметр	Умолч.	Тип	Описание
			экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс
&SURFPAR			
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$)
			1: генерация белого шума
			2: генерация белого шума симметрично относительно
			экватора
mars	0	int	1: инициализация модели для планеты Марс

Продолжение таблицы Б.2

Параметр	Умолч.	Тип	Описание
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс
kick	1	int	0: инициализация без шума ($p_s = const$) 1: генерация белого шума 2: генерация белого шума симметрично относительно экватора
mars	0	int	1: инициализация модели для планеты Марс

Б.3 Очередной подраздел приложения

Нужно больше подразделов приложения!

Б.4 И ещё один подраздел приложения

Нужно больше подразделов приложения!