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

Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (национальный исследовательский университет)»

На правах рукописи

Андрианов Павел Сергеевич

Анализ корректности синхронизации компонентов ядра операционных систем

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

Диссертация на соискание учёной степени кандидата физико-математических наук

Научный руководитель:
кандидат физико-математических наук
Хорошилов Алексей Владимирович

Москва — 2021

Оглавление

Стр.

Стр.

Стр.

Стр.

Стр.

Введение

Информационные технологии являются важной составляющей инфраструктуры современного общества. Они позволяют автоматизировать различные процессы жизнедеятельности человека и обеспечить возможность коммуникации. Таким образом, задача повышения производительности компьютерных систем становится чрезвычайно важной. Развитие технологий многоядерности и многопоточности являются важнейшими направлениями решения данной задачи. Например, в ядре операционной системы может параллельно выполняться большое число (несколько десятков) совершенно различных активностей: обработчики прерываний, системные вызовы от пользовательских программ, драйверы внешних устройств, внутренние службы ядра, например, планировщик. Только за счет использования многопоточности современные системы могут обеспечивать необходимые показатели производительности.

В дополнение к ошибкам, которые встречаются во всех видах программного обеспечения, многопоточные программы могут содержать специфические ошибки, связанные с параллельным выполнением: например, состояния «гонки» и состояния взаимной блокировки. В общем случае состоянием гонки называют ситуацию, при которой поведение программы зависит от порядка или времени выполнения некоторых неконтролируемых событий. Важное уточнение заключается в том, что такое выполнение не всегда является ошибкой. Проблемы возникают тогда, когда разработчик не предусматривает некоторое из возможных поведений программы. Часто рассматривают более узкий класс – «состояния гонки по данным». Эта ситуация возникает при одновременном доступе к данным из разных потоков (процессов). Одновременное чтение данных из нескольких потоков не может привести к недетерминированным результатам, и состояния гонки по данным становятся опасными в случае, если имеет место хотя бы один доступ на запись в разделяемую область памяти. В этом случае результирующее значение переменной зависит от порядка выполнения инструкций, а в параллельно выполняемом коде, в общем случае, последовательность выполнения инструкций не определена.

Состояния взаимных блокировок являются вторым большим классом ошибок в многопоточных программах. Они возникают при некорректном использовании блокирующих механизмов синхронизации. В этом случае некоторые потоки

системы находятся в ожидании некоторого разблокирующего действия от других потоков и не могут продолжить свое выполнение.

Оба класса ошибок так или иначе связаны с некорректной синхронизацией параллельно выполняющихся потоков. Поиск и исправление таких ошибок, связанных с параллельным исполнением кода, затруднен из-за необходимости анализировать все возможные сценарии взаимодействия потоков, а также из-за случайного характера их проявления, так как ошибка может проявляться очень редко. Это связано с тем, что для проявления ошибки необходимы некоторая конкретная последовательность и порядок действий различных потоков. В системном программном обеспечении могут использоваться не только обыкновенные примитивы синхронизации, но и некоторые специальные, например, запреты прерываний. Для упрощения поиска таких ошибок рарабатываются различные автоматизированные подходы. Но для анализа системного программного обеспечения, например, операционных систем, требуются специализированные инструменты, которые будут учитывать их особенности.

Обычно выделяют два класса подходов к анализу программ: статические, которые проводят анализ исходного кода программы, и динамические, которые анализируют поведение программы в процессе ее выполнения. Каждый имеет свои достоинства и недостатки, например, методы динамического анализа обычно характеризуются высоким уровнем истинных предупреждений, в то время как статические методы позволяют покрыть больший объем кода. На данный момент не существует универсального подхода для анализа программ.

В большинстве прикладных пользовательских программ может быть достаточно провести тщательное тестирование, возможно, с помощью инструментов динамического анализа, чтобы проверить основные сценарии поведения программы. В случае же критически важного ПО, в т.ч. системного программного обеспечения, цена пропущенной ошибки может быть слишком велика. Кроме того, некоторые сценарии поведения программы слишком сложно воспроизвести при реальном выполнении, что затрудняет использование динамического анализа. Поэтому применения только динамического анализа недостаточно, необходимо применять и методы статического анализа.

Методы статической верификации способны обеспечить доказательство отсутствия ошибок в некоторых заранее заданных предположениях, но при этом традиционно испытывают сложности при масштабировании на большие объемы исходного кода. Как уже упоминалось ранее, к специфике системного ПО отно-

сятся большой объем сильно связанного кода, большое количество активностей, которые могут выполняться параллельно, использование низкоуровневых операций, например, адресной арифметики, специальные примитивы синхронизации и др. Таким образом, разработка масштабируемого метода статической верификации для поиска состояний гонки с учетом специфики системного ПО является актуальной задачей.

Цели и задачи. Целью данной работы является разработка метода поиска состояний гонок для анализа корректности синхронизации, который будет масштабироваться на большие объемы кода, будет обладать приемлемым уровнем ложных предупреждений и будет учитывать специфику ядра операционных систем.

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

1. Разработать метод поиска состояний гонки, на основе подхода с раздельным анализом потоков.
2. Разработать алгоритм построения окружения потока, который будет обладать конфигурируемостью и масштабироваться на большие объемы исходного кода, и доказать его корректность.
3. Модернизировать алгоритм адаптивного статического анализа (англ. Configurable Program Analysis, CPA) с целью обеспечения возможности выполнять верификацию программого обеспчения с раздельным анализом потоков и доказать его корректность.
4. Реализовать разработанные алгоритмы.
5. Провести эксперименты на приложениях и ядрах операционных систем и сравнить результаты с другими инструментами статической верификации.

Научная новизна:

1. Метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.
2. Алгоритм построения окружения потока, который позволяет гибко настраивать уровень абстракции над взаимодействием потоков, и доказательство его корректности.
3. Новый алгоритм, который является обобщением существующего алгоритма статической верификации программ при помощи метода CPA, рас-

ширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказательство его корректности.

Теоретическая и практическая значимость работы. Развитие метода адаптивного статического анализа для обеспечения возможности его использования вместе с подходом раздельного анализа потоков является важным теоретическим вкладом в развитие программной инженерии. Разработка нового алгоритма построения окружения потока является важным развитием подхода с раздельным анализом потоков для обеспечения возможности применения к большому объему исходного кода. Практическим результатом данного исследования является множество найденных ошибок как в операционных системах реального времени, так и в драйверах устройств операционной системы Linux.

Методология и методы исследования. При проведении работы были использованы как теоретические методы исследования (анализ, формализация, абстрагирование), так и практические (эксперименты, сравнение).

Положения, выносимые на защиту:

1. Метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.
2. Алгоритм построения окружения потока и верификации многопоточных программ с помощью подхода с раздельным анализом потоков и доказательство его корректности.
3. Обобщение алгоритма статической верификации программ при помощи метода CPA, расширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказательство его корректности.

Апробация результатов. Основные результаты работы докладывались на:

– Международная научно-практическая конференция «Инструменты и методы анализа программ» (TMPA: Tools and Methods for Program Analysis), Кострома, 2014 г.
– Международный семинар разработчиков CPAchecker, Москва, 2015 г.
– Летняя научная школа компании Microsoft (Microsoft Summer School), Кэмбридж, Англия, 2015 г.
– Научно-практическая Открытая конференция ИСП РАН, Москва, 2016 г.

– Научно-исследовательский семинар лаборатории «Software and Computational Systems Lab» Университета Пассау, Германия, 2016 г.
– Международная научно-практическая конференция «Инструменты и методы анализа программ» (TMPA: Tools and Methods for Program Analysis), Москва, 2017 г.
– Международный семинар разработчиков CPAchecker, Падерборн, Германия, 2017 г.
– Международный семинар разработчиков CPAchecker, Москва, Россия, 2018 г.
– Соревнования по статической верификации SV-COMP, Прага, Чехия, 2019 г.
– Международный семинар разработчиков CPAchecker, Фрауенинзель, Германия, 2019 г.
– Семинар «Математические вопросы информатики» Мехмат МГУ, Москва, Россия, 2019 г.

Публикации. Основные результаты по теме диссертации изложены в 8 печатных изданиях [1–8], 6 из которых изданы в журналах, рекомендованных ВАК [1–6], из них 3 находится в базах Scopus и Web of Science [2; 3; 6], 2 — в тезисах докладов [7; 8].

В статье [4] автором описана основная идея метода (глава 3) и его реализация (глава 5). В статье [5] автором написаны разделы, посвященные общей идее метода (глава 3), его реализации (глава 4), процессу уточнения (глава 5) и анализу потоков (глава 6). В статье [6] автором написаны разделы, посвященные разработанному методу и его реализации (главы 3–6). В статье [7] автором написаны разделы, в которых описывается ключевые особенности метода (главы 3–5). В статье [8] автором написаны разделы, посвященные разработанному методу (главы 3–5). В статье [3] автором написаны разделы, описывающие теорию масштабируемого метода (главы 3–8).

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

Объем и структура работы. Диссертация состоит из введения, четырех глав, заключения и двух приложений. Полный объём диссертации составляет 217 страниц, включая 23 рисунка и 28 таблиц. Список литературы содержит 100 наименований.

Глава 1. Обзор: анализ многопоточных программ и обеспечение корректности синхронизации компонентов ядра операционных систем

1.1. Введение

В настоящее время существует множество различных техник и подходов для поиска ошибок в программном обеспечении. Однако, универсального метода, позволяющего решать задачи любой сложности, создать невозможно. Поэтому во многих областях одновременно применяются различные методы верификации. Сбалансированный набор методов позволяет нивелировать ограничения и недостатки отдельных элементов такого набора.

Можно выделить следующие классы методов верификации [9]:

– экспертиза;
– статические методы;
– динамические методы;
– формальные методы.

Следует сразу отметить, что данная классификация является достаточно условной, и наиболее успешно показывают себя методы, объединяющие в себе несколько различных подходов.

Экспертиза исходного кода позволяет выявлять практически любые ошибки, однако ее эффективность сильно зависит от опыта и компетенции экспертов. Ее особенностью является то, что она практически не автоматизируется, а значит, ее эффективность применения к большому объему исходного кода значительно снижается.

Методы автоматического поиска ошибок традиционно разделяют на два основных класса: статические и динамические. Теоретически оба эти подхода способны обнаруживать ошибки полностью автоматически, то есть без участия человека, однако на практике участие эксперта значительно повышает эффективность подходов.

Методы динамического анализа анализируют программное обеспечение в процессе его выполнения. Исторически такие методы берут свое начало из тестирования и мониторинга. В процессе развития статических методов они начинают комбинироваться с динамическими и возникают статико-динамические подходы.

Статические методы поиска ошибок, в отличие от динамического анализа, проверяют исходный код программы без его реального выполнения. Традиционно такие методы разделяют на те, которые берут свое начало из компиляторных технологий и отличаются высокой скоростью работы, и на те, которые основываются на формальных методах. Мы будем называть первый класс методами статического анализа, а второй – методами статической верификации.

Формальные методы верификации отличаются строгим математическим доказательством корректности программ, однако требуют значительного участия эксперта в доказательстве. Статическая верификация позволяет снизить затраты труда экспертов за счет применения соответствующих инструментов.

Выделим те критерии инструментов верификации, которые являются важными для решения поставленной задачи по проверке корректной синхронизации системного программного обеспечения. Основным объектом исследования является системное программное обеспечение, которое применяется в критичных областях, где цена ошибки очень высока, поэтому одной из основных характеристик должна стать точность метода: он не должен в общем случае пропускать ошибки. Возможно применение отдельных эвристик в конкретном случае, но общий метод должен иметь возможность гарантировать корректность проверяемой программы. Второй важной характеристикой метода верификации должны стать возможности по гибкой настройке баланса между количеством ложных предупреждений и скоростью работы. С этим же связана масштабируемость: должна быть возможность применения инструмента к реальным программным системам, состоящим из сотен тысяч строк кода. Кроме того, метод должен адекватно работать с такими сложными конструкциями языка Си, которые используются в системном программном обеспечении, как то: адресная арифметика, приведение (англ. casting) одних типов к другим и др. Перечисленные выше требования отсекают возможность использования метода ручной экспертизы и метода формальной верификации, поэтому далее мы сосредотачиваемся только на методах динамического анализа, статического анализа и статической верификации.

Обзор существующих методов поиска гонок построен на основе изучения публикаций по данной теме, в число которых, в частности, включены работы Имперского колледжа Лодона, университета Оксфорда, университета Юты, Microsoft Research, университета Мэриленда, политехнического университета Лозанны и др., и обзоры по данной теме, опубликованные, например, в [10; 11].

1.2. Общие термины

Определим общие термины, которые будут использоваться далее.

Все методы анализа параллельно выполняющихся программ так или иначе оперируют с многопоточной программой, то есть, программой, код которой может выполняться параллельно. Так как во всех расматриваемых методах объектом анализа является одна программа, которая внутри себя использует параллельное выполнение, то обычно используются термины потоки, а не процесы.

Во время выполнения программы может производиться доступ к разделяемым данным, то есть выполняться операции чтения и записи. Доступ характеризуется идентификатором потока, из которого производился доступ, типом доступа (чтение или запись), а также, в некоторых случаях, информацией об используемых примитивах синхронизации.

Многие алгоритмы дают собственное определение ситуаций, которые они трактуют как состояние гонки. Мы не будем перечислять все возможные определения, но отметим два основных алгоритма, которые часто используются при поиске потенциальных состояний гонки: Lockset [12] и Happens-Before [13]. Многие инструменты берут за основу тот или иной алгоритм, а иногда и их комбинацию, а затем применяют различные техники для повышения масштабируемости и точности.

Алгоритм Lockset определяет состояние гонки, как ситуацию, при которой возможен одновременный доступ к разделяемым данным из разных потоков с непересекающимся множеством блокировок. Один из доступов при этом должен быть записью. К недостаткам этого алгоритма относится поддержка только примитивов синхронизации на основе блокировок, что приводит к повышению количества ложных предупреждений об ошибках, если используются какие-то другие механизмы взаимного исключения.

Алгоритм Happens-Before называется так, потому что он строит связи между операциями, которые означают, что некоторая операция может быть выполнена только раньше некоторой другой операции. Если для каких-нибудь операций различных потоков над общими данными не удается построить такую связь, считается, что данные операции могут быть выполнены параллельно, то есть это является признаком состояния гонки.

В некоторых алгоритмах используется понятия *сериализуемости* и *линеаризуемости*. Сериализуемость означает, что результат выполнения некоторых параллельных действий (транзакций) был эквивалентен результату выполнения тех же самых действий при некотором последовательном выполнении. Линеаризуемость последовательности действий означает, что для всех остальных потоков эта последовательность выглядит атомарной.

1.3. Методы динамического анализа многопоточных программ

Методы динамического анализа подразумевают проверку целевой программы на корректность в процессе ее работы. Такие методы активно применяются для поиска типовых ошибок, их основное достоинство заключается в низком проценте ложных срабатываний. Самые простые методы динамического анализа берут свое начало из тестирования и мониторинга. При тестировании на вход программе подаются некоторые воздействия, и проверяется соответствие выходного результата программы некоторому критерию, на основе которого выносится вердикт о корректности или некорректности. В случае мониторинга поведение системы может анализироваться и в ходе обычной эксплуатации. Тем не менее, методы динамического анализа являются наиболее простыми и могут быть достаточно легко автоматизированы. Ошибки, связанные с параллельным выполнением программы, являются недетерминированными, поэтому генерация тестов и мониторинг являются слишком неэффективным занятием, так как даже на корректном тесте ошибка может проявляться с очень низкой вероятностью. Для повышения вероятности обнаружения ошибки были предложены несколько методов, которые будут рассмотрены далее.

1.3.1. Методы динамического анализа многопоточных программ на основе векторных часов

Для повышения шанса обнаружения состояния гонки применяются более сложные методы динамического анализа, основанные на инструментировании ис-

полняемого кода и наблюдении за внутренним состоянием программы в процессе ее выполнения. Такие подходы основываются на идее векторных часов Лампорта [13] для определения связи Happens-Before. Основная идея, лежащая в основе этого подхода заключается в том, что все события в одном потоке являются упорядоченными. Синхронизация между потоками может быть представлена, как синхронизация между часами соответствующих потоков. Тогда, если для некоторых событий можно установить, что одно из них было строго раньше другого (связь Happens-Before), они не могут образовывать состояние гонки. Часы различных потоков объединяются в понятие *векторных часов*, в которых каждая компонента соответствует часам отдельного потока. Стоит отметить, что такой подход позволяет обнаруживать и реальные, и потенциальные состояния гонки, которые не произошли при выполнении, это влечет за собой появление возможных ложных срабатываний. Основными проблемами подхода на основе векторных часов являются большое количество потенциально разделяемой памяти, которую необходимо отслеживать, поэтому различные инструменты предлагают свои варианты решения данной проблемы. Далее, кратко опишем такие оптимизации в различных инструментах.

В статьях [14; 15] описывается следующая оптимизация. Так как запись информации обо всех действиях в программе занимает слишком много времени, то сохраняется только информация о самых последних доступах к переменной. Менее 1% случаев гонок требуют полного сохранения векторных часов, в остальных случаях достаточен более легковесный подход. Используется понятие эпоха - это идентификатор векторных часов и номер потока. Инструмент позволяет адаптивно выбирать, сохранять ли конкретное значение векторных часов для некоторой переменной или только эпоху. Такой подход позволяет уменьшить накладные расходы, не потеряв точность. Требуемое для анализа время, в среднем, в 4-5 раз больше, чем для работы исходной программы. Объем требуемой памяти возрастает на 20-30%.

В статьях [16; 17] описан инструмент LiteRace, который использует оптимизацию, основанную на гипотезе «холодных регионов»: предполагается, что состояния гонки, скорее всего, проявятся в редко исполняемых участках кода (т.н. «холодных»), так как на часто исполняемых они уже исправлены. Таким образом, за «горячими» участками слежение почти не ведется, а доступы к памяти из холодных участков отслеживаются полностью. При этом «горячие» участки кода вычисляются отдельно для каждого потока. В среднем, накладные расходы со-

ставляют 28% времени работы. Это в 25 раз меньше, чем если бы обрабатывались все обращения к памяти.

Статьи [18; 19] представляет еще один инструмент - PACER, который использует следующую оптимизацию: доступы к памяти учитываются только в период выборки (англ. sampling period). Инструмент Pacer дает гарантии того, что существующая гонка будет найдена с вероятностью равной уровню выборки (англ. sampling rate). Pacer случайно включает и выключает слежение за динамическими операциями, настраивая период выборки. Однако, такой метод имеет два важных недостатка:

1. можно пропустить happens-before дуги, потеря которых приведет к ложному предупреждению;
2. при уровне выборки r число найденных состояний гонок будет лишь r^2.

Для уровня выборки 1-3% накладные расходы составили 52-83%. Основное отличие инструмента от LiteRace заключается в том, что последний детерминирован, а Pacer намеренно использует случайные значения, чтобы найти состояния гонки, которые были пропущены в предыдущих запусках.

Статья [20] представляет улучшение метода, реализованного в инструменте ThreadSanitizer-Valgrind [21], представленного ранее. Основная часть логики метода осталась той же самой, однако вследствие использования компилятора LLVM удалось повысить скорость работы, а также портируемость, с которой были большие проблемы. TSan-LLVM использует инструментацию во время выполнения программы, что приводит к замедлению ее работы в 2-3 раза. Для определения состояния гонки используется и Lockset алгоритм, и Happens-Before, что позволяет повысить точность. Для того, чтобы уменьшить расходы на анализ, применяются выборочное слежение за доступами к памяти (sampling) и гипотеза «холодных регионов». В отличие от LiteRace, который применяет ту же идею, TSan всегда выполняет инструментированный код, но если для данного потока счетчик выполнений данной операции выше уровеня выборки (sampling rate), то информация о доступе к памяти игнорируется. Второе важное отличие заключается в том, что LiteRace считает холодными участками кода целые функции, TSan оперирует на уровне базовых блоков, что уточняет анализ. Основной недостаток инструментации во время выполнения - это то, что могут быть потеряны ошибки, находящиеся в коде, который не компилируется: системные библиотеки или JIT код. Сравнение подхода показало, что, в среднем, применение инструментации позволяет ускорить работу в 2-3 раза по сравнению с оригинальным TSan-Valgrind и в 10 раз по

сравнению с другими инструментами, такими, как Valgrind. Однако, не сравнивалось число найденных ошибок.

1.3.2. Методы статико-динамического анализа многопоточных программ

Многие динамические инструменты используют для оптимизации своей работы результаты простого статического анализа программы. Такой предварительный анализ предоставляет некоторое множество потенциальных обращений к памяти, которые могут участвовать в состоянии гонки. После проведения быстрого предварительного анализа запускается основной динамический анализ, который, выполняя программу, следит только за выделенными областями памяти. Даже при использовании самых простых техник статического анализа удается сократить множество рассматриваемых переменных и, тем самым, значительно повысить эффективность основного метода динамического анализа.

В статье [22] описан такой статико-динамический подход. Кроме общего описания в статье приведено множество оптимизаций для повышения эффективности и точности как статического, так и динамического анализов. В частности, описана интересная оптимизация памяти для динамического анализа. Основной алгоритм Lockset требует сохранять множество доступов к переменной для последующего вычисления множества общих блокировок. Авторы высказывают мысль, что на самом деле все доступы подряд сохранять не обязательно. Нужны лишь множества с минимальными множествами блокировок. То есть, если уже была сохранена информация о доступе, который защищен одной блокировкой, то не нужно хранить информацию о другом доступе, которое было защищено двумя блокировками, если одна из них - та же самая.

В статье [23] авторы добавили третью стадию: точный алгоритм статического анализа, который проверяет потенциальные состояния гонки, которые могли бы возникнуть при другом расписании выполнения потоков. Этот алгоритм записывает формулу пути, которую затем разрешает с помощью SAT-решателя. Состояния гонки, найденные во время динамического анализа являются истинными ошибками, а предупреждения, найденные во время третьей стадии - лишь потенциальными ошибками. Однако, добавление третьей стадии позволяет получить больше информации за один запуск инструмента. Применение для этого точ-

ных методов статической верификации повышает шанс обнаружения истинных ошибок по сравнению с простым методом статического анализа, применяемого на первой стадии.

В статье [24] представлен оригинальный способ поиска гонок в ядре ОС. DataCollider расставляет несколько точек прерывания на случайные доступы к памяти. Когда программа доходит до них, выполнение останавливается, сохраняется значение той ячейки памяти, куда происходит обращение, и спустя какое-то время проверяет, что значение не изменилось. В ином случае фиксируется состояние гонки. Не все состояния гонки ведут к ошибкам. Так, эксперименты с DataCollider показали, что только 10% предупреждений соответствует истинным ошибкам. DataCollider был применен к нескольким драйверам ОС Windows. Было найдено 38 состояний гонки. При этом замедление работы было всего 10-15% процентов.

Развитие идеи DataCollider было продемонстрировано в статье [25]. Инструмент DRDDR был применен к операционной системе Linux, в которой запускались два набора тестов. Инструмент показал очень низкий уровень накладных расходов в 0,1%, но при этом смог обнаружить только несколько безобидных состояний гонки.

Статико-динамический подход имеет определенные перспективы, позволяя сочетать в себе плюсы нескольких подходов. Такие подходы неплохо масштабируются, и способны успешно применяться к системному коду. Однако, даже такие методы демонстрируют достаточно невысокий процент истинных ошибок, что показали результаты DataCollider. Кроме того, они также не могут гарантировать отсутствие ошибок в программе. При этом возникают традиционные проблемы, связанные с настройкой тестового окружения, что в случае операционной системы приводит к определенным требованиям к оборудованию. Одним из возможных направлений развития таких подходов является использование виртуальных машин, для которых задача настройки тестового окружения значительно упрощается. Такие методы сейчас активно развиваются, однако, основной их целью по-прежнему остается проверка некоторого множества уже найденных предупреждений.

1.3.3. Узкоспециализированные методы динамического анализа многопоточных программ. Специфика анализа ядер операционных систем.

На практике часто оказывается, что под каждую задачу, связанную с анализом системного программного обеспечения, эффективнее разработать собственный инструмент. Так, в статьях [26; 27] представлен опыт поиска ошибок в сервере MySQL. Сначала были проанализированы часть исправлений, сделанных в процессе разработки от версии 3.x до версии 6.x. Результаты показали, что ошибки, связанные с параллельным выполнением становятся все более и более распространенными. Состояние взаимной блокировки вызывали 40% ошибок. 28% всех ошибок приводили к падению системы, а 15% - к предоставлению неверных данных пользователю. 15% всех багов приводили к ошибке не сразу, т.н. “скрытые” баги, причем 92% из них впоследствии проявляют себя тем, что выдают пользователю невереные данные - это семантические баги. Далее авторы рассказывают о своем опыте поиска гонок с помощью инструмента PIKE. Метод поиска гонок основан на том, что несмотря на внутренний параллелизм, сервер ведет себя так, как если бы все запросы выполнялись последовательно. Каждое параллельное выполнение сравнивается со всеми возможными его сериализациями. Если ни одна из них не соответствует реальной трассе, фиксируется ошибка. Для того, чтобы найти скрытые ошибки, которые повреждают структуру данных, а не логику выполнения, необходимо применить описанный метод не в конце выполнения, а для каждого внутреннего состояния программы. Инструмент PIKE пытается обойти эти ограничения с помощью алгоритма случайного планирования. При старте программы планировщик выбирает случайный приоритет для каждого потока. В каждый момент времени активен поток с наибольшим приоритетом. Приоритет случайным образом меняется в некоторых случайных точках программы.

Повысить эффективность динамических подходов возможно за счет сужения класса выявляемых ошибок. В статьях [28–30] описывается подход к динамическому поиску конкретного класса ошибок в многопоточных программах - нарушениям атомарности операций. Чтобы уменьшить объем анализируемых данных, все возможное множество вариантов переключения контекста ограничивается теми, которые соответствуют несериализуемым взаимодействиям (англ. unserializable interleavings). Для таких взаимодействий не существует никакого последовательного эквивалента выполнения участвующих операций. Инструмент

запускался на таких промышленных программных системах, как MySQL, Apache, Mozilla и др. Затраченное время на анализ оказалось в 10-1000 раз меньше, чем у других подходов. Однако, следует помнить, что такая эффективность достигается за счет сужения класса обнаруживаемых ошибок.

В статье [31] был предложен модульный подход для построения предикатной абстракции. Подход был успешно применен к различным примерам как из набора соревнований sv-comp, так и на примерах, построенных на основе драйверов операционной системы Linux. Этот подход был применен для анализа низкоуровневого кода, в котором присутствуют прерывания (even-driven systems) [32].

Основные особенности методов динамического анализа заключаются в следующем:

1. Инструменты динамического анализа ориентированы, в основном, на пользовательские приложения.
2. Вероятностный характер алгоритмов не позволяет гарантировать отсутствия ошибок.
3. Масштабируемость зависит больше не от общего объема кода, а от числа точек инструментации, то есть того кода, в котором производится поиск состояний гонки.

Данные особенности позволяют заключить, что методы динамического анализа не являются достаточным способом поиска состояний гонки в программном обеспечении, которое предъявляет повышенные требования к надежности. Кроме того, эффективность таких методов существенно снижается при анализе сложного программного обеспечения.

1.4. Методы статического анализа многопоточных программ

В данном разделе под статическим анализом понимаются такие методы, которые анализируют исходный код программы без ее реального выполнения и характеризуются высокой скоростью работы за счет применения различных фильтров и эвристик. Такие методы традиционно применяются на больших программных системах. Они отличаются более высоким уровнем покрытия кода, чем динамический анализ, но также не могут претендовать на формальное доказательство отсутствия ошибок. В основном, статический анализ применяется для поиска ти-

повых ошибок, таких как некорректная работа с памятью. Поиск состояний гонки является достаточно трудоемкой задачей по сравнению с поиском таких ошибок, поэтому большая часть статических анализаторов не предоставляет возможность поиска ошибок, связанных с некорректным взаимодействием потоков. Тем не менее, существуют различные экспериментальные инструменты статического анализа, позволяющие находить состояния гонки.

1.4.1. Методы статического анализа для поиска состояний гонки

Статья [33] представляет инструмент Locksmith для статического поиска гонок. Locksmith реализован на основе библиотеки CIL [34], которая используется для разбора исходного кода. Инструмент последовательно и независимо друг от друга проводит несколько стадий анализа, собирая информацию о потоках данных, потоке управления, разделяемых данных и используемых примитивах синхронизации. Такое разделение анализов положительно сказывается на скорости, однако инструмент сильно теряет в точности анализа. Например, множество разделяемых данных вычисляется для всей программы сразу. То есть, если некоторая память сначала является локальной, а после некоторых действий становится разделяемой, то эта память будет считаться разделяемой в каждой точке программы. Другой важной особенностью является то, что инструмент предполагает работу только с полностью доступным кодом. Это необходимо для того, чтобы можно было определить область памяти, на которую указывает каждый из указателей.

Как и все инструменты статического анализа, Locksmith работает очень быстро, характерное время работы составляет 1с на 1000 исходного строк кода. Но при этом находит достаточно много ложных предупреждений, так, для драйверов ОС Linux истинных ошибок было 9%. Большинство ложных срабатываний связано с неточноностью анализа разделяемых данных.

Инструмент не позволяет настраивать используемые примитивы синхронизации, функции работы с разделяемыми структурами данных и точность самого анализа. Это значительно снижает возможность его применения к новому коду. Кроме того, он требует полного определения всех объектов, в том числе блокировок, что не всегда возможно обеспечить при применении инструмента к отдельным модулям или библиотекам. Таким образом, можно заключить, что ин-

струмент, хотя и является достаточно масштабируемым, не способен обеспечить необходимую точность и конфигурируемость анализа.

Инструмент Relay [35] был применен к большому объему системного программного обеспечения. Он также основан на алгоритме Lockset. Relay вычисляет некоторое «суммарное действие» функции (англ. summary) в терминах изменений множества захваченных примитивов синхронизации и доступов к памяти относительно точки входа в функцию. Модель потоков является достаточно простой: считается, что все функции могут быть выполнены параллельно друг с другом. Определение множества возможных алиасов происходит с помощью анализа Стинсгаарда [36]. Авторы применили инструмент к модулям операционной системы Linux, при этом получили более 5000 предупреждений. Такое количество предупреждений проанализировать вручную невозможно, поэтому авторы выбрали некоторое подмножество, проанализировали его, а затем разработали набор неточных фильтров, которые сокращают количество предупреждений.

Похожим образом действуют авторы IteRace [37], инструмента для поиска ошибок синхронизации в Java программах. Сначала определяются алиасы с помощью анализа Андерсена, затем для каждой области памяти проверяются синхранизационные примитивы, что позволяет вычислить множество потенциальных состояний гонки. Затем применяются различные оптимизации, которые позволяют сократить количество ложных сообщений об ошибке. В их числе как оптимизации в процессе анализа (рассмотрение только двух потоков, выделение областей кода, которые могут выполняться только в однопоточном режиме), так и оптимизации поиска состояний гонки после проведения анализа кода. Результаты позволяют заключить, что данный метод является практически применимым, однако применяемые оптимизации являются специфичными для Java-программ.

Еще один похожий инструмент представлен в статье [38]. В нем авторы также с помощью простого анализа находят множество доступов к данным, а затем применяют некоторые фильтры, которые позволяют исключить ложные сообщения о состояних гонки. Самой интересной идеей авторов является расширение анализа указателей Андерсена на многопоточный случай. Для апробации авторы взяли инструмент TSan и применили его к небольшому набору пользовательских программ. А затем применили его же с подсказками от своего инструмента. В статье [39] тот же инструмент сравнивается относительно довольно старого анализа алиасов. В некоторых случаях достигался десятикратный выигрыш и по времени, и по памяти, однако большая часть тестов являлась небольшими программами,

которые решались за десятки секунд даже без использования предложенной оптимизации.

А в статье [40] было предложено использовать векторные часы в статическом анализе, при этом сам фреймворк остался тем же. Анализ алиасов снова сравнивался с некоторым старым анализом, а результаты поиска гонок сравнивались с результатами инструмента Locksmith. Время работы анализа алиасов сильно зависит от типа задачи, на некоторых примерах предложенный метод позволил сократить количество затрачиваемой памяти на два порядка, а на некоторых показал такой же результат, как и старый вариант анализа. Такие же результаты были получены и при сравнении методов поиска состояний гонки: для некоторых задач ускорение получается достаточно существенным, а для некоторых – время работы не изменяется. При этом стоит отметить, что авторам удалось значительно сократить количество ложных предупреждений за счет увеличения точности анализа

Существуют определенные попытки построить эффективный способ поиска состояний гонки на основе методов статической верификации, применив некоторую эвристику, которая не позволит гарантировать отсутствие ошибок. Например, в статье [41] описывается инструмент, который рассматривает не все возможные чередования операций нескольких потоков, а чередования блоков операций. Программа разбивается на блоки несколько раз случайным образом, а затем полученные измененные программы анализируются параллельно и независимо друг от друга. Такой подход демонстрирует скорость на несколько порядков выше, чем исходный подход, рассматривающий все чередования, однако, в некоторых случаях только несколько процентов (1%-3%) полученных трансформированных программ содержали ошибку, которая была в исходной программе.

Важной проблемой, которая возникает при анализе реального программного обеспечения, является определение равенства областей памяти. Для этого обычно используется анализ алиасов (синонимов), однако, в общем виде он является достаточно сложным алгоритмом. Разработка эффективной его реализации является сложной задачей без применения его к поиску состояний гонки. Поэтому в некоторых статьях, представляющих методы поиска состояний гонки, акцент смещен в сторону описания методов анализа алиасов, а затем на полученном множестве алиасов применяется простой алгоритм Lockset для определения потенциальных состояний гонки. Такие подходы не являются эффективными с точки зрения поиска состояний гонки, так как точный анализ алиасов требует значительных ресурсов: как времени, так и памяти.

В статье [42] для быстрого определения множества алиасов рассматривается идея применения разбиения Стинсгаарда [36]. Указатели могут быть алиасами указателей только из своего подмножества. Так как разбиение Стинсгаарда обычно невелико, то определять точное равенство указателей, чтобы доказать корректность алиасов, приходится нечасто. Кроме того, анализ необходимо выполнять для всех указателей, хотя на самом деле разделяемыми являются только некоторые из них, поэтому классический анализ алиасов становится неэффективным.

В статье [43] предложено два варианта анализа алиасов, которые основаны на анализе регионов. Цель анализа регионов - показать, что выделенная на куче память корректно разделена на непересекающиеся блоки. Первый подход основан на сложном анализе размера структур данных. Этот подход очень плохо масштабируется на большие программы. Другой подход работает с динамически выделенными объектами, как с блоками памяти, связанными с абстрактными состояниями. В этом случае возникают проблемы с анализом областей памяти, выделенных в одной точке программы. В статье описан анализ регионов достаточно быстрый и точный для программ, работающих с непересекающимися областями памяти.

При применении инструмента к модулям ОС Linux авторы столкнулись с проблемами обработки структур, содержащих массивы, с арифметикой указателей, преобразованием типов (кастированием). Были попытки расширить возможности анализа, например, предположением, что указатель может указывать не только на начало блока памяти. Инструмент запускался на 9 драйверах. Число разделяемых переменных оказалось невелико.

Как и ожидалось, для применения анализа алиасов к сложному программному обеспечению приходится использовать различные эвристики. В этом случае снижается ценность самого анализа алиасов. Кроме того, в больших программных системах, в частности, в ядре операционной системы, количество указателей будет слишком большим, а значит, сам анализ алиасов будет требовать неприемлемое количество времени. Таким образом, можно сказать, что точный анализ алиасов неприменим к большим программным системам.

1.4.2. Узкоспециализированные методы статического анализа. Специфика анализа ядер операционных систем.

Высокую эффективность показывают инструменты статического анализа, которые нацеливаются на поиск очень узкого класса ошибок. Очень интересна работа [44], которая посвящена поиску неявных типов синхронизаций. В статье проведен анализ нескольких больших программных продуктов на предмет наличия в них неявнных или специальных (ad hoc) видов синхронизации, например, таких как ожидание в цикле пока некоторая переменная не примет определенное значение. Важно заметить, что если в программе определяется некоторая функция, которая используется для синхронизации, она не считается неявной. В процессе исследования были получены следующие результаты.

- Каждая изученная параллельная программа использует неявную синхронизацию.
- В основном, все неявные типы синхронизации представляют собой бесконечные циклы, выход из которых возможен при специальном условии. Но несмотря на это, такие неявные синхронизации тяжело выявлять.
- Во многих случаях неявный способ синхронизации приводил к ошибкам. Обычно это выражалось во взаимной блокировке или бесконечному ожиданию в цикле.
- Из-за того, что многие инструменты не могут определить неявную синхронизацию, пропускается большое число ошибок.

Для помощи разработчикам был создан инструмент, который аннотирует синхронизационные циклы. Инструмент SyncFinder определяет условие выхода из цикла и с помощью решателя доказывает, что это условие нарушается только после действий другого потока. В результате каждый найденный синхронизационный цикл аннотируется. Такой результат может быть использован в двух аспектах. Во-первых, разработчики могут вручную просмотреть каждый цикл и понять, является ли он ошибкой или нет. На проведенных тестах инструмент верно нашел 96% неявных способов синхронизации. При этом число ложных сообщений об ошибках было всего 6%. Во-вторых, другие инструменты поиска гонок могут использовать данные аннотации для более точного анализа. Например, авторы взяли некоторый инструмент поиска гонок на базе Valgrind и показали, что число ложных предупреждений сокращается на 43-86%.

В работе [45] авторы концентрируются на поиске неправильного использования глобальных переменных, которое может привести к ошибкам. Было проведено экспериментальное исследование, выделено три категории неправильного использования и создан инструмент для поиска таких случаев. Кроме того инструмент пытается помочь разработчикам и автоматически пытается преобразовать код таким образом, чтобы вместо глобальных переменных использовались локальные параметры.

В работах [46; 47] представлен подход к поиску состойний гонки в ядрах операционных систем на примере FreeRTOS. Однако, большую часть статей занимает описание построения окружения для ядра, то есть, описание того, что может выполняться параллельно. Дело в том, что в ядре имеется много активностей, которые выполняются после наступления некоторого события. Для статического анализа необходимо искусственно создать некоторую *main*-функцию, в которой будут описаны те активности, которые могут выполняться параллельно. Например, задана взаимосвязь между обработчиками прерываний, отключением прерываний и функции приостановки работы (suspend). Эта взаимосвязь не выражается с помощью стандартных POSIX API, поэтому и нужна специфическая модель окружения и анализ, учитывающий особенности ядра. Сам анализ многопоточных программ является расширением анализа явных значений на множества (вариация анализа многоугольников). А значит, снова возникает проблема потери точности анализа при любых операциях с указателями.

Многие рассмотренные методы ограничиваются лишь корректным вычислением алиасов, и задача поиска гонок отходит на второй план. Эффективные методы, которые применяются при решении практических задач, не подходят для доказательства корректности и позволяют быстро обнаруживать потенциальные ошибки. При этом системное программное обеспечение накладывает определенные требования, поэтому на практике часто применяются специальные решения, которые нацеливают общий метод Lockset на конкретный исходный код. Таким образом, основной задачей становится не разработка эффективного метода, а адаптация существующего подхода.

1.5. Методы статической верификации многопоточных программ

Статическая верификация отличается от легковесных методов статического анализа более формальным подходом к проверке корректности программы. За счет этого возможно получения доказательства отсутствия определенного класса ошибок в некоторых заранее заданных предположениях. Основная идея методов статической верификации программ заключается в автоматическом построении модели программы по ее исходному коду, а затем в проверке этой модели на соответствие некоторой спецификации. Для этого возможно применение различных методов, таких как абстракция, уточнение абстракции по контрпримерам (англ. CEGAR), ограничиваемая проверка моделей. Так или иначе строится конечный путь к ошибке, затем этот путь трансформируется в логическую формулу, которая проверяется на выполнимость с помощью специального компонента: решателя (англ. solver). Это позволяет исключить многие ложные предупреждения, не потеряв при этом реальные ошибки, в отличие от методов статического анализа.

Основными плюсом данного подхода является строгость доказательства и, как следствие, низкий процент ложных сообщений об ошибках. Минусом подхода является высокие требования к ресурсам.

1.5.1. Методы статического верификации на основе чередований потоков

Первыми, как наиболее простые, стали развиваться методы, рассматривающие все возможные чередования потоков. Для применения таких методов к многопоточным программам необходимо упорядочить последовательность операций во всех потоках, чтобы иметь возможность построить логическую формулу пути. Для этого необходимо определить точки, в которых происходит переключение выполнения операций одного потока на операции другого потока. Наблюдение, что ошибки проявляются уже при небольшом числе переключений контекста, позволило ограничить число переключений некоторым небольшим фиксированным K. В условиях ограниченного количества переключений (англ. context bounded switches) применяются различные техники и оптимизации для сокращения пространства состояний.

Многие методы моделируют взаимодействие между потоками только с помощью операций над глобальными переменными. В этом случае переключение потока имеет смысл рассматривать в случае, если в текущем состоянии происходит доступ к глобальной переменной. Такая идея используется, например, в статье [48]. В ней же описаны еще несколько оптимизаций.

«Ленивый» подход предполагает, что дерево достижимости обходится в глубину, собираются все полученные ограничения для пути и для каждого узла дерева вызывается обычная процедура для ограниченной проверки модели. Подход с записью расписания (schedule recording) добавляет к формуле ограничения на планировщик, что позволяет объединить все возможные варианты исполнения программы в одну формулу и отдать ее решателю. Расширяющийся подход (under-approximation and widening approach) основан на том, что сначала строится формула для частного случая взаимодействия процессов, и она проверяется решателем. Если найдена ошибка, значит, более общая модель уже не нужна. Иначе, постепенно отбрасываются ограничения на планировщик, тем самым покрывается все больше вариантов параллельной работы.

В уже упомянутой статье [48] основным результатом, на котором акцентируется внимание является то, что предложенные оптимизации позволяют анализировать программы с 10 переключениями контекста, в то время, как похожие инструменты (в статье упоминаются CHESS [49] и SATABS [50]), способны анализировать лишь 5 переключений контекста. При этом тестовый набор составлялся авторами статьи и включал в себя 21 рукописную программу.

В статье [51] авторы представляют инструмент для проверки моделей для применения к низкоуровневому системному коду. Сначала производится трансляция исходной Си программы в программу на языке Boogie [52] с помощью инструмента HAVOC [53]. Это делается для того, чтобы избавиться от таких конструкций, как динамическое выделение памяти, арифметика указателей, преобразование типов. Далее, параллельная программа преобразуется в последовательную, и для нее уже строятся формулы, которые проверяются с помощью SMT решателя. Для того чтобы повысить масштабируемость метода, применяется слайсинг по полям структур. Такая идея базируется на предположении, что для проверки конкретного свойства необходимо наблюдение за очень небольшим количеством полей. Построение множества отслеживаемых полей производится с помощью метода CEGAR. Для моделирования переключения контекста для карты памяти глобальных переменных создаются K копий. И переключение контекста модели-

руется переключением работы с одной карты памяти на другую. Инструмент запускался на четырех реальных драйверах ОС Windows. Время работы составляло порядка часа на программу из 600 операторов (locations).

В продолжении этой статьи [54] представлено довольно объемное экспериментальное исследование применения инструмента на реальном исходном коде. К сожалению, авторы не указали вид программы, на которой проводились эксперименты. Было проведено значительное количество запусков инструмента с различными вариантами преобразований в последовательную программу, с различным количеством активных потоков, количеством переключений контекста и различным количеством операторов в каждом из потоков. Общий вывод оказался следующим: при количестве потоков больше трех или количестве переключений контекста больше четырех инструмент неприменим, так как время анализа превосходит несколько часов. При этом количество операторов в программе измеряется всего несколькими десятками.

В статье [55] авторы представляют идею абстракции по программному счетчику. Вводится понятие эквивалентности глобальных состояний относительно перестановки локальных состояний и для каждого локального состояния вводятся счетчики. При переходе из одного состояния в другое увеличивается счетчик полученного и уменьшается для исходного. Для того чтобы еще более повысить скорость работы применяется объединение символических состояний. Оно может произойти, если два состояния отличаются только значением глобальных переменных или локальным состоянием в одном из потоков. Такая оптимизация эффективно себя показывает при анализе программ, состоящих из большого количества одинаковых потоков. В случае, если функции, которые выполняются в отдельных потоках, являются различными, такой подход оказывается бесполезным.

В статье [56] рассматриваются идеи уменьшения пространства состояний на основе редукции частичных порядков и применении слайсинга. Основной идеей редукции является выявление независимых блоков с целью исключения их детального анализа с учетом всех возможных чередований потоков. Зависимость операций определяется по работе с разделяемыми данными, при этом разработан специальный анализ алиасов с использованием обновляющихся последовательностей. Слайсинг применяется для уменьшения количества рассматриваемых операторов. Некоторые операторы программы могут быть опущенны из-за ограничений на примитивы синхронизации или из-за недостижимости соответствующих частей кода.

Оценка инструмента проводилась на 9 драйверах Linux с известными гонками. Ожидаемо оказалось, что время работы инструмента является приемлемым только для примеров с небольшим количеством (порядка десятка) разделяемых переменных. Трассы, построенные на основе полученных предупреждений, подавались на вход BMC, который пытался их доказать. В отдельных случаях, время доказательства занимало 10 часов.

В статье [57] представлен новый подход к редукции частичных порядков, называемый монотонной редукцией. Он основан на новой характеристике частичного порядка, которая определяется через вычисления данной программы в терминах квазимонотонных последовательностей. Этот подход может быть использован и точными, и символическими методами проверки моделей.

Некоторое развитие подхода к редукции частичных порядков представлены в статье [58]. Экспериментальные данные показывают большое ускорение по сравнению с оригинальным методом. Для специально подобранных тестовых примеров достигнутой ускорение составляет три порядка, но на задачах, которые основаны на реальном коде, оптимизация показала ускорение 1,5-2 раза. Стоит отметить, что, как и в статье [55], такие результаты возможны только на таких программах, потоки в которых выполняют однотипные действия.

В работе [59] авторы предлагают расширение для редукции частичных порядков для анализа низкоуровневых программ, содержащих вложенные прерывания. Как и все оптимизации при попытке рассмотреть чередования, этот подход также страдает недостаточной масштабируемостью. Даже на небольших примерах, он не может доказать корректность программы, уходя в таймаут.

Как мы видим, очень много исследователей развивают методы статической верификации, основанные на переборе возможных чередований операций нескольких потоков. Несмотря на обилие различных техник и оптимизаций, характерное время работы инструмента анализа составляет порядка 1с на 1 строку кода. Поэтому такие методы не могут быть применены на практике.

1.5.2. Методы статической верификации на основе трансляции

Уже упоминались подходы, которые применяют трансляцию параллельной программы в последовательную для последующей верификации. Следу-

ет отметить статьи, посвященные отдельно этой проблеме трансляции (англ. sequentilization) с последующей ее верификацией существующими инструментами анализа последовательных программ. Все существующие трансляторы используют следующие ограничения:

1. число активных потоков ограничено;
2. взаимодействие между потоками возможно только с помощью глобальных переменных;
3. в каждый момент времени только один поток может быть активен;

Известны несколько методов трансляции [54; 60–62]: от простого дублирования исполняемого кода до использования сложных и оптимизированных карт памяти, переключение между которыми происходит в случайный момент. Для верификации новой последовательной программы применяются две основные парагдимы. Одна из них - это логическая проверка моделей, при которой формулы строятся для конечного пути, а проверяется их выполнимость. Второй вариант - это верификация условий (verification-condition, VC), при которой для построения формул требуется пред- и постусловия и формальные правила преобразования. Эксперименты показали, что эти две парагдимы сильно отличаются, и для них нужны различные способы трансляции. Все попытки применения подхода трансяции к последовательной программе в общем случае показывают, что он плохо масштабируется и на реальных программах пока не может быть использован.

Нужно заметить, что трансляция параллельной программы может производиться не в последовательную, а в некоторую промежуточную модель. Эта модель может быть использована, например, для генерации тестов как это сделано в [63]. В ней представлен фраймворк который автоматически транслирует программу на языке Си в модель на языке Promela [64]. Полученная модель верифицируется с помощью инструмента SPIN [64]. При наличие ошибки полученный контрпример может быть специальным образом преобразован к исполняемому тесту.

В статье [65] авторы рассматривают проблему трансляции программ с динамическим созданием потоков, т.н. асинхроннными вызовами. Число создаваемых потоков может быть потенциально бесконечным, хотя число переключений каждого потока является ограниченным. Авторы доказывают несколько теорем из которых следует алгоритмическая разрешимость задачи за конечное время. Результаты данной статьи являются только теоретическими, и предложенный подход не был реализован, что позволило бы оценить его практическую применимость.

В статье [66] авторы рассматривают проблему проверки низкоуровневого системного кода, содержащего обработчики прерывания. Основная проблема заключается в том, что обработчики прерываний в реальной системе выполняются с высоким приоритетом, и их выполнение не может быть прервано переключением на другой поток. То есть, прерывания выполняются атомарно, если не приходит другое прерывание с более высоким приоритетом. В статье предложено два подхода к верификации таких программ: на основе трансляции в последовательную и на основе моделирования прерываний, как обычных потоков. В качестве основного инструмента верификации использовался CBMC. Было проведено достаточно серьезное сравнение с другими инструментами.

Одним из успешных инструментов является WHOOP [67], который использует технику трансляции и фактически не учитывает взаимодействие потоков. Он использует только алгоритм Lockset и не имеет возможности расширить подход другими видами анализа. Авторы предлагают применять этот инструмент в связке с более точным инструментом верификации CORALL.

1.5.3. Методы статической верификации на основе раздельного рассмотрения потоков

Все рассмотренные выше подходы так или иначе учитывают взаимодействие потоков друг с другом. На реальных программных системах это приводит к комбинаторному взрыву числа возможных вариантов выполнения нескольких потоков. Для решения этой проблемы был предложен подход с абстракцией от окружения потока, т.н. подход с раздельным рассмотрением потоков (англ. thread-modular approach). В самом простом варианте этот подход был описан еще в [68]. Затем этот подход был дополнен предикатной абстракцией в [69]. В этой статье пока рассматривался только один поток в нескольких копиях, то есть окружение формировалось самим же потоком, а примитивы синхронизации вообще не использовались.

Расширение подхода с раздельным рассмотрением потоков и абстракцией было представлено в [70], а затем реализовано в инструменте TAR [71]. В этом подходе была предложена идея эффектов от потока, которые вычислялись на ос-

нове действий потока. Эти эффекты затем применялись к другим потокам. Примитивы синхронизации были реализованы, как обыкновенные переменные.

В статье [72] представлено применение подхода с раздельным рассмотрением потоков для анализа динамически выделяемой памяти (англ. Shape analysis). В подходе предполагалось итеративное построение инвариантов памяти для доказательства корректной работы с памятью в многопоточной программе. Авторы реализовали некоторый прототип инструмента и провели его экспериментальную оценку на специально подготовленных задачах из 50 – 300 строк кода.

Похожие идеи были представлены в статьях [73–75], которые описывают инструмент Threader. Основная идея метода заключается в поиске некоторого инварианта для программы, из которого следует доказываемое свойство. В процессе анализа строится абстракция, при этом для окружения используется аппроксимация сверху, основанная на Хорновских дизъюнктах. Если в какой-то момент возникнет состояние, которое не противоречит проверяемому свойству, абстракция уточняется. При этом уточняются как состояния этой абстракции, так и эффекты, то есть, взаимодействие потоков. Threader не ограничивает число возможных переключений контекста.

Для оценки результатов использовались примеры, являющиеся упрощенными моделями некоторых реальных программ, таких, как драйверы ОС Windows и Linux. В основном, размер тестов варьировался в районе 100 строк, при этом время работы инструмента на некоторых примерах составляло несколько минут, а на некоторых превышало ограничение в 15 минут. В статье [76] авторы предоставляют результаты запуска инструмента на некотором общедоступном наборе тестов, который используется для ежегодного соревнования инструментов статической верификации. В качестве тестовых примеров там используются, в основном, небольшие рукописные программы. Большинство инструментов, принимающих там участие, не способны справится с обычным реальным кодом. Threader запускался на 32 тестах из категории «Параллельность» и набрал 43 балла из 49 возможных. Правильный вердикт был выдан на 28 тестах. Таким образом, на небольших программах Threader позволяет получить правильный вердикт в 87% случаев.

1.5.4. Другие методы статической верификации

В статье [77] авторы подходят к проблеме поиска доказательства для многопоточной программы с другой стороны. Для каждого из потоков записывается свой инвариант в терминах глобальных переменных и доказывается его корректность, а из конъюнкции всех инвариантов следует выполнимость необходимого свойства для целой программы. Главная проблема заключается в том, что не для всякой программы существуют инварианты в терминах одного потока, то есть локального доказательства. В некоторых случаях зависимости будут более сложными, что приведет к появлению инвариантов, связывающих несколько потоков, что будет означать, что локального доказательства не существует. В таких случаях поиск локального доказательства может быть бесконечным. Основное достижение данной статьи в том, что был предложен конечный метод для поиска локального доказательства. Во многих тестах (со слов авторов) их подход показал хорошие результаты.

Для помощи инструменту верификации можно предоставлять некоторые подсказки - аннотации. Вовлечение человека в процесс верификации несомненно повышает качество и скорость, однако требует определенных затрат времени на написание таких аннотаций. В статье [78] представлен верифицирующий Си компилятор (VCC), который интегрирован в Microsoft Visual Studio. Это полностью автоматическая система, которая может верифицировать аннотированные программы. Аннотации представляют собой инварианты, пред- и постусловия. Из аннотаций генерируются формулы логики первого порядка, которые разрешаются с помощью решателя.

Похожая идея демонстрируется в статье [79], в которой представлен фреймворк для аннотации Java программ. Аннотации представляют пред- и постусловия для некоторых блоков кода, которые проверяются по ходу выполнения программы. Для оценки работы инструмента использовалось два набора тестов: Java Grande Forum (JGF) и Parallel Java (PJ) Library, в сумме 13 тестов 1000-4000 строк кода. В среднем потребовалось около 10 строк аннотаций, чтобы найти те же гонки, что и CalFuzzer.

Очень интересная идея была описана в статье [80]. В ней представлен подход к верификации многопоточных программ, основанный на понятии разрешения. Так, поток имеет право на доступ к некоторой ячейке памяти, если у него

есть на это разрешение, которое представляет собой вероятность от 0 до 100% включительно. Значение 100% означает эксклюзивный доступ на запись, любое другое число от 0 до 100 - доступ на чтение. Для каждой функции записывается предусловие, в котором указывается, какое значение для разрешения требуется для ее выполнения. Отдельно нужно заметить, что рассматриваются объектно-ориентированные программы, в которых синхронизация между потоками осуществляется с помощью мониторов. Мониторы представляют собой блокировки, которые используются для защиты от одновременного доступа некоторые области памяти. Во время создания нового объекта или захвата монитора для него разрешение устанавливается на 100%. При доступе к объекту строится формула из ограничений и подается на вход решателю. Данные об экспериментах не были опубликованы.

Рассмотренные подходы проверки моделей позволяют обеспечить доказательство корректности программы в определенных ограничениях, однако все они испытывают определенные проблемы при применении к реальному программному обеспечению. Подход с раздельным рассмотрением потоков является перспективным решением проблемы комбинаторного взрыва числа состояний программы. Кроме того, существующие подходы с раздельным рассмотрением потоков используют неэффективные базовые варианты анализа, что не позволяет их применять к большим программным системам. Например, во всех реализациях все примитивы синхронизации были записаны, как обычные переменные, и для отслеживания их значений использовался тот же анализ, что и для отслеживания значений переменных.

1.6. Основные выводы. Анализ многопоточных программ в ядрах операционных систем.

Проведенный обзор современных методов анализа многопоточных программ позволяет сделать следующие выводы.

Основные усилия сейчас сосредоточены на анализе пользовательских приложений. Для решения практических задач разрабатываются инструменты специально нацеленные на конкретные классы ошибок или на определенный про-

граммный код. Это является верным для всех типов подходов: как статических, так и динамических.

Методы динамического анализа нацелены, в первую очередь, на повышение процента истинных срабатываний, возможно, жертвуя при этом некоторыми реальными ошибками. Основными проблемами инструментов динамического анализа является увеличение накладных расходов на анализ, в том числе, замедление работы целевой программы или увеличение числа используемой памяти. На решение этих проблем направлены основные усилия. Таким образом, основными направлениями развития инструментов динамического анализа сейчас являются разработка различных оптимизаций, позволяющих более эффективно использовать предоставленные ресурсы и повышающих вероятность обнаружить реальную ошибку.

Общецелевые методы статического анализа нацелены на высокую скорость анализа. В случае применения таких методов к большому объему сложного кода будет получен огромное количество предупреждений. Анализ этих предупреждений вручную может затянуться на неопределенное время. Непрактичность таких методов подтверждается тем, что почти все инструменты данного класса не используются для решения прикладных задач, а являются чисто исследовательскими реализациями. Исключениями становятся инструменты созданные для решения очень узкого класса задач, например, поиска специфичных ошибок в отдельном программном продукте.

В результате обзора различных техник статического анализа можно сделать вывод, что почти все инструменты так или иначе используют неточные фильтры и эвристики, в качестве основы своего метода. Таким образом, они принципиально не позволяют гарантировать отсутствие ошибки. Такие инструменты, как Relay [35], которые позволяют отключать свои фильтры, выдают огромное количество ложных предупреждений, так как не учитывают специфику системного кода, взаимодействия потоков и условия, встречающиеся на пути.

Таким образом, и методы динамического анализа, и общецелевые методы статического анализа не подходят для качественной верификации системного программного обеспечения, так как они не могут дать гарантию того, что программа корректна.

Методы статической верификации позволяют провести верификацию с высоким уровнем надежности, однако требуют значительного объема ресурсов. В настоящее время существует большое число инструментов, реализующих методы

на основе автоматической проверки моделей, однако все они являются исследовательскими инструментами, а их применение ограничивается небольшим искусственным набором тестовых программ.

Методы статической верификации, основанные на переборе различных чередований (англ. interleavings), не способны обеспечить достаточной скорости анализа. Наиболее подходящим подходом является подход с раздельным рассмотрением потоков, который позволяет анализировать потоки по-отдельности.

Многие из исследователей, которые пытались применить свои инструменты к системному программному обеспечению, сталкивались с проблемами анализа сложных низкоуровневых конструкций, например, адресной арифметики. Другой важной особенностью анализа реального программного обеспечения является его принципиальная недоопределенность. В системном программном обеспечении используются различные источники входных данных, которые не доступны на этапе анализа, например, внешние устройства, действия пользователя или прикладные программы. Многие из академических инструментов работают в предположении, что анализируемая программа замкнута, то есть весь используемый код доступен, вся используемая память явно инициализирована. На практике часто оказывается, что тела некоторых функций находятся в библиотеках и код их не доступен, эти же функции могут возвращать некоторую память, которая неизвестно как инициализирована. Эту неопределенность необходимо уметь гибко настраивать.

Многие из инструментов стараются предположить как можно худшее, стараясь не пропустить реальные ошибки. Но обычно это приводит лишь к огромному множеству ложных срабатываний. На основе знаний от разработчиков можно выдвинуть некоторые разумные предположения к неопределенным функциям, которые позволят значительно сократить количество найденных ложных предупреждений. Однако, такие предположения должны быть четко зафиксированы, чтобы полученное доказательство корректности не было ими скомпрометировано.

Еще одним важным различием подходов является определение ошибки. Так, методы динамического и статического анализа, по сути, ищут несинхронизированные доступы к памяти. Методы статической верификации проверяют многопоточную программу на соответствие некоторой спецификации. Это означает, что пользователь сам должен задать, что является ошибкой. Примером такой спецификации может быть требование, чтобы значения ни одной разделяемой переменной не могло быть изменено сразу после его чтения или записи. Это условие

является одной из попыток формализовать условия отсутствия состояний гонки. Однако, выразительности простого языка спецификаций, который обычно реализовывается в инструментах статической верификации, оказывается недостаточно для полного описания тех неформальных требований к программе, которые обычно подразумеваются под словами «отсутствие состояний гонки». Например, оба этих условия не позволяют формализовать требование к отсутствию таких ошибок, как высокоуровневые гонки. Также становится невозможным доказывать корректность различных алгоритмов, которые не требуют синхронизации (lock-free). Таким образом, для проверки сложных требований необходимо иметь возможность написания некоторых пользовательских инвариантов, которые требуется доказать.

Таким образом, можно заключить, что в настоящее время отсутствуют такие методы анализа больших объемов системного кода, в том числе, операционных систем, которые могут обеспечить высокий уровень надежности. Данная работа посвящена описанию разработанного метода анализа корректности программ, в том числе, поиска состояний гонки, который может применяться к реальным программным системам и позволяет гибко настраивать баланс между точностью и скоростью работы.

Глава 2. Метод анализа корректности синхронизации многопоточной программы с раздельным рассмотрением потоков

2.1. Основные определения

В данной главе описывается метод анализа корректности синхронизации многопоточной программы с раздельным рассмотрением потоков. Описание метода опирается на понятия, используемые в статической верификации. Основными из них являются понятия *состояние программы*, *достижимости* и *абстракции*. Описанию этих и других необходимых понятий посвящена данная секция.

Будем рассматривать простой императивный язык программирования, в котором поддерживаются операторы присваивания (assignment), проверки условия (assumption), захват и освобождение примитивов синхронизации (acquire/release) и создание потоков (thread_create).

Параллельная программа представляется автоматом потока управления (англ. Control Flow Automaton, CFA), который состоит из множества вершин L и множества соединяющих их ребер G . Множество L – это множество всех точек программы (англ. program location), множество $G \subseteq L \times Ops \times L$, где Ops – это множество всех операций программы. Так, отдельная дуга $g \in G, g = (l_1, op, l_2)$ моделирует операцию op, которая выполняется в программе, когда управление переходит из точки l_1 в точку l_2. Операция создания нового потока создает новый поток с идентификатором из множества T и новый поток начинает свое выполнение из некоторой точки программы из множества L. Множество всех переменных программы, которые встречаются в операциях $assignment$ и $assumption$ обозначается как X. Будем считать, что переменные принимают только значения лежащие в $\mathbb{Z}$. Подмножество X, которое содержит все локальные переменные программы, обозначается как X^{local}, а подмножество, содержащее все глобальные переменные, – X^{global}. Операции захвата и освобождения примитивов синхронизации определены над множеством специальных переменных $S : S \cap X = \varnothing$. Далее мы будем называть примитив синхронизации, связанный с переменной $s \in S$, блокировкой s. Переменные из множества S принимают значения из $\mathcal{T} = T \cup \{\bot_T\}$. При этом, $s \mapsto t, t \in T$ означает, что блокировка s захвачена потоком t, а $s \mapsto \bot_T$ означает, что эта блокировка не захвачена.

Конкретным состоянием параллельной программы называется четверка (c_{pc}, c_l, c_g, c_s), где

1. Отображение $c_{pc} : T \to L$ является частичной функцией из идентификаторов потоков во множество точек программы. Каждое такое состояние задает точку в программе, в которой находится каждый из потоков.
2. Отображение $c_l : T \to C^{local}$ является частичной функций из множества идентификаторов потоков во множество присваиваний локальным переменным их значений. $C^{local} : X^{local} \to \mathbb{Z}$ – отображение, которое задает каждой переменной из X^{local} ее значение из $\mathbb{Z}$.
3. $c_g : X^{global} \to \mathbb{Z}$ задает значения глобальных переменных программы.
4. $c_s : S \to \mathcal{T}$ задает значения блокировок.

Множество всех возможных конкретных состояний программы обозначим через C.

Отображения c_{pc} и c_l представляют локальные части конкретного состояния для каждого потока, а c_g и c_s представляют глобальную часть конкретного состояния, не зависящую от потока. Обозначим dom область определения частичной функции, например, $dom(c_{pc}) = \{t \mid \exists (t, l) \in c_{pc} \wedge t \in T \wedge l \in L\}$. Будем использовать обозначение $\mathbf{.}$ для замены параметра, который является несущественным. Например, предыдущее определение может быть записано следующим образом $dom(c_{pc}) = \{t \mid \exists (t, \mathbf{.}) \in c_{pc} \wedge t \in T\}$.

Каждое состояние $c = (c_{pc}, c_l, c_g, c_s) \in C$ должно удовлетворять условию $dom(c_{pc}) = dom(c_l)$, которое означает, что области определения локальных частей состояния находятся в консистентном состоянии. Будем использовать обозначение $dom(c)$, как область значения локальных частей состояния, то есть $dom(c) = dom(c_{pc}) = dom(c_l)$.

Отношение переходов $TR \subseteq C \times G \times T \times C$ определяет преобразование конкретных состояний под действием операторов программы. Переход (c_1, g, t, c_2), где $c_1, c_2 \in C$ моделирует переход программы из состояния c_1 в состояние c_2 при выполнении потоком $t \in T$ операции на дуге $g \in G$. Для наглядности будем использовать обозначение $c_1 \xrightarrow{g,t} c_2$ вместо $(c_1, g, t, c_2) \in TR$. Для полноты будем считать, что существует специальный ε-переход из каждого состояния в себя: $\forall c \in C, t \in T : c \xrightarrow{\varepsilon,t} c$.

Определим множество конкретных переходов $\mathcal{T} = C \times G \times T$. Конкретный переход $\tau \in \mathcal{T}$ – это тройка $\tau = (c, g, t)$. Будем писать $\tau_1 \longrightarrow \tau_2$, если $\exists c_3 \in C : c_1 \xrightarrow{g_1,t_1} c_2 \xrightarrow{g_2,t_2} c_3$.

Для всех $c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$, $g = (l, \cdot, l') \in G$, $t \in T$ любой переход $c \xrightarrow{g,t} c'$ должен удовлетворять следующим требованиям:

1. Переход начинается в состоянии c: $t \in dom(c_{pc}) \wedge c_{pc}(t) = l$.
2. При выполнении операции поток t переходит в точку l': $c'_{pc}(t) = l'$.
3. Каждый переход $c \xrightarrow{g,t} c'$ может изменить только локальную часть состояния, которая соответствует потоку t и, возможно, глобальную часть состояния, то есть локальные части других потоков остаются неизменны. Формально $\forall t' \in T : (t' \neq t) \wedge (t \in dom(c_{pc}) \cap dom(c'_{pc})) \Rightarrow c'_{pc}(t') = c_{pc}(t') \wedge c'_l(t') = c_l(t')$.

Полное отношение переходов $\longrightarrow$ определяется как объединение по всем переходам: $\longrightarrow = \bigcup\limits_{g \in G, t \in T} \xrightarrow{g,t}$. Будем обозначать $c \xrightarrow{g,t} c'$, если $(c, g, t, c') \in \longrightarrow$, и $c \xrightarrow{g} c'$, если $\exists t \in T : c \xrightarrow{g,t} c'$, и $c \longrightarrow c'$, если $\exists g \in G : c \xrightarrow{g} c'$. Конкретное состояние c_n будем называть *достижимым* из некоторого подмножества $r \subseteq C$ и обозначать $c_n \in Reach_{\longrightarrow}(r)$, если существует последовательность конкретных состояний $\langle c_0, c_1, ..., c_n \rangle$ такая, что $c_0 \in r$ и $\forall i : 1 \leqslant i \leqslant n : c_{i-1} \longrightarrow c_i$.

Обозначим через $eval(c, t, expr)$ значение выражения $expr$ над переменными из $X^{local} \cup X^{global}$ и со значениями этих переменных из конкретного состояния $c \in C$ для потока с идентификатором $t \in T$.

Далее определим семантику операций в программе: оператора присваивания (assignment), оператора проверки условия (assumption), операторов захвата и освобождения блокировки (acquire/release), а также оператора порождения потока (thread_create). Для каждого из этих операторов необходимо задать правила преобразования конкретных состояний программы. Далее мы не будем отдельно подчеркивать требования 1–3, которые выполняются для любого перехода.

2.1.1. Оператор проверки условия

Рассмотрим дугу CFA, которая содержит оператор проверки условия $g = (l, assume(expr), l') \in G$, $l, l' \in L$. Переход $c \xrightarrow{g,t} c'$, $c, c' \in C, t \in T$ существует в том и только в том случае, если

– $c = c'$ – переход не меняет состояние программы;

– $eval(c, t, expr) \neq 0$ – значение проверяемого выражение не равно нулю.

2.1.2. Оператор присваивания

Рассмотрим дугу CFA, которая содержит оператор проверки условия $g = (l, assign(x, expr), l') \in G$, $l, l' \in L$. Переход $c \xrightarrow{g,t} c'$, $c, c' \in C, t \in T$, $c = (c_{pc}, c_l, c_g, c_s)$ существует в том и только в том случае, если

- $dom(c) = dom(c')$ – переход не меняет множество активных потоков;
- Переход присваивает переменной x значение выражения $expr$, то есть
  - Если $x \in X^{local}$

$$\forall x' \in X^{local} : c'_l(t')(x') = \begin{cases} eval(c, t, expr) & \text{, если } x' = x \wedge t' = t \\ c_l(t')(x') & \text{, если } x' \neq x \vee t \neq t' \end{cases}$$

    $c'_g = c_g$.
  - Если $x \in X^{global}$

$$\forall x' \in X^{global} : c'_g(x') = \begin{cases} eval(c, t, expr) & \text{, если } x' = x \\ c_g(x') & \text{, если } x' \neq x \end{cases}$$

    $c'_l = c_l$.
- $c'_s = c_s$ – блокировки не меняют свои значения.

2.1.3. Операции с примитивами синхронизации

Предполагаем, что операция $acquire(s)$ в потоке $t \in T$, где $s \in S$ – блокировка, имеет следующую семантику: если $s \mapsto \bot_T$ в предыдущем состоянии, тогда $s \mapsto t$ в следующем состоянии, и эта операция (сравнение и изменение состояния блокировки) производится атомарно за один переход.

Формально, для дуги $g = (l, acquire(s), l') \in G$ и $t \in T, l, l' \in L$ существует переход $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ тогда и только тогда, когда

- $dom(c) = dom(c')$ – переход не меняет множество активных потоков;
- $c'_l = c_l$ – переход не меняет значения локальных переменных;
- $c'_g = c_g$ – переход не меняет значения глобальных переменных;

– $c_s(s) = \bot_T \wedge c'_s(s) = t \wedge \forall s' \in S : s' \neq s \Rightarrow c'_s(s') = c_s(s')$ – никакие другие блокировки не меняют свое значение.

Предполагаем, что операция $release(s)$ в потоке $t \in T$ имеет следующую семантику: если в начальном состоянии блокировка $s \in S$ была захвачена тем же потоком, то есть $s \mapsto t$, то в следующем состоянии эта блокировка освобождается, то есть $s \mapsto \bot_T$. Операция проходит атомарно.

Формально, для дуги $g = (l, release(s), l') \in G$ и $t \in T$, $l, l' \in L$ существует переход $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ тогда и только тогда, когда

– $dom(c) = dom(c')$ – переход не меняет множество активных потоков;
– $c'_l = c_l$ – переход не меняет значения локальных переменных;
– $c'_g = c_g$ – переход не меняет значения глобальных переменных;
– $c_s(s) = t \wedge c'_s(s) = \bot_T \wedge \forall s' \in S : s' \neq s \Rightarrow c'_s(s') = c_s(s')$ – никакие другие блокировки не меняют свое значение

2.1.4. Создание потоков

Определим семантику $thread\_create(l_\nu)$ таким образом: текущий поток при выполнении этого оператора переходит в следующую точку программы после $thread\_create$, а новый поток создается с новым идентификатором $\nu \in T$, который добавляется в локальную часть состояния. Созданный поток начинает свое выполнение из точки программы $l_\nu \in L$.

Заметим, что такое определение позволяет поддерживать анализ программ с неограниченным количеством потоков, так как $thread\_create$ может встречаться в цикле.

Для дуги $g = (l, thread\_create(l_\nu), l')$ существует переход $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$, $\nu \in T$ тогда и только тогда, когда

– $\nu \notin dom(c) \wedge dom(c') = dom(c) \cup \{\nu\}$ – поток с идентификатором ν добавляется во множество потоков;
– $c'_{pc}(t) = l'$ – счетчик команд в родительском потоке переходит на следующую команду l';
– $c'_{pc}(\nu) = l_\nu$ – счетчик команд в дочернем потоке устанавливается в начальную позицию l_ν;

– $c'_l(\nu) = \varnothing$ – локальные переменные созданного потока не инициализированы;
– $c'_l(t) = c_l(t) \wedge c'_g = c_g \wedge c'_s = c_s$ – все остальные части конкретного состояния остаются без изменений.

2.1.5. Ошибка в программе

Существует несколько различных вариантов определения корректности программы, в частности, определения, что является ошибкой. Наиболее общим вариантом является использования понятия достижимости. В этом случае при верификации программа считается корректной, если удается доказать, что все ошибочные состояния недостижимы. Если ошибочное состояние представить как точку в программе, где стоит, например, оператор assert, то корректной можно назвать программу, в которой все логические условия, записанные в качестве параметра оператора assert, не нарушаются.

Такое определение ошибки является достаточно общим, так как позволяет сформулировать различные свойства программы, которые требуется верифицировать. Еще одной важной особенностью является то, что такое определение с одинаковым успехом может применяться как к последовательным программам, так и к параллельным.

Состояние гонки обычно определяются с помощью такой конструкции, как последовательность

```
variable = expression;
assert(variable == expression);
```

Эта конструкция моделирует проверку, не может ли измениться значение переменной $variable$ в данной точке программы. Если инструменту удается доказать, что условие оператора assert не нарушается, это будет означать, что не существует потока, который мог бы обновить значение переменной. И наоборот, если условие нарушается, это означает, что значение переменной было изменено в параллельном потоке.

Основным минусом такого подхода является то, что при поиске состояний гонки в реальных программах не известно, какая именно переменная может моди-

фицироваться из нескольких потоков, а значит, придется расставлять assert после каждой записи в переменную, проверяя, может ли измениться ее значение в другом потоке или нет. Кроме того, такой подход не позволяет обнаруживать состояния гонки, в котором один из доступов является чтением. Тем не менее, многие инструменты статической верификации используют его, как наиболее простой. Данная теория также поддерживает такое определение, однако, для практического поиска состояний гонки нам потребуется более практичное определение.

Определение 1. *Определим **состояние гонки** как конкретное состояние c такое, что $\exists c_1, c_2 \in C$, $g_1, g_2 \in G$, $t_1, t_2 \in T$, $t_1 \neq t_2 : c \xrightarrow{g_1,t_1} c_1 \wedge c \xrightarrow{g_2,t_2} c_2$ и операции g_1, g_2 выполняются над одной и той же глобальной переменной $x \in X^{global}$ и хотя бы одна операция является присваиванием (то есть $assign(x, expr)$).*

Однако, на практике нас интересуют только состояния гонки, достижимые при реальном выполнении программы, то есть, $c_1, c_2 \in Reach(\{c_0\})$.

2.2. Адаптивный статический анализ с абстрактными переходами

В классической теории [81; 82], абстрактное состояние описывает некоторую совокупность конкретных состояний программы. Рассматривая абстрактные состояния программы вместо конкретных, инструмент сокращает пространство поиска, что позволяет повысить эффективность анализа. В предлагаемом расширении теории абстрактное состояние является частичным, то есть оно может содержать лишь часть информации о конкретных состояниях программы. Для того, чтобы получить соответствующие ему конкретные состояния необходимо взять другие частичные абстрактные состояния. Это является следствием того, что полное конкретное состояние может быть получено только из нескольких частичных состояний, описывающих потоки по-отдельности. Таким образом, функция конкретизации, которая предоставляет соответствие между абстрактными состояниями и конкретными, в расширенной теории отличается от классической: она определяется на множестве абстрактных элементов.

Частичными также являются и абстрактные переходы. Каждый абстрактный переход теперь содержит лишь ту часть информации о конкретном переходе, которая соответствует частичному абстрактному состоянию. Чтобы получить ин-

формацию о полном абстрактном переходе, необходимо взять несколько частичных переходов, в общем случае некоторое количество k.

Определим формально *адаптивный статический анализ с абстрактными переходами* для параллельных программ $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$. Он состоит из абстрактного домена D, множество точности Π, отношения переходов $\rightsquigarrow$, оператора слияния $merge$, оператора останова $stop$, оператора уточненения $prec$. Несмотря на то, что сами операторы остались теми же, что и в классической версии теории, они претерпели некоторые изменения. Для того чтобы различать операторы программы и операторы CPA, будем использовать термин *cpa-операторы* для обозначения последних.

Рассмотрим подробно cpa-операторы, которые задают используемый CPA.

– *Абстрактный домен* $D = (\mathcal{T}, \mathcal{E}, [\![\cdot]\!])$ определяется множеством $\mathcal{T}$ конкретных переходов ($\mathcal{T} \subseteq C \times G \times T$), полурешеткой $\mathcal{E}$ абстрактных переходов и функцией конкретизации $[\![\cdot]\!]$. Полурешетка $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ состоит из (возможно, бесконечного) множества E элементов абстрактного домена, верхнего элемента $\top \in E$, нижнего элемента $\bot \in E$, частичного порядка $\sqsubseteq \subseteq E \times E$ и функции $\sqcup : E \times E \to E$ (cpa-оператор объединения состояний). Функция $\sqcup$ возвращает наименьший элемент решетки, который больше, чем каждый из двух ее аргументов, а символы $\top$ и $\bot$ обозначают наибольший и наименьший элемент решетки E соответственно. Функция конкретизации $[\![\cdot]\!] : 2^E \to 2^{\mathcal{T}}$ отображает каждое множество абстрактных переходов $R \subseteq E$ на множество его значений, то есть на множество конкретных состояний программы, которое оно представляет. Основным отличием от классической функции конкретизации – это определение на множестве абстрактных элементов. Это объясняется тем, что суммарное знание для множества частичных состояний может быть больше, чем знание для одного частичного состояния, то есть,

$$\forall R \subseteq E : [\![R]\!] \supseteq \bigcup_{e \in R} [\![\{e\}]\!]$$

Для корректности анализа программы абстрактный домен должен удовлетворять следующим требованиям:

$$\forall R \subseteq \overline{R} \subseteq E : [\![R]\!] \subseteq [\![\overline{R}]\!] \tag{2.1}$$

$$\forall e, e' \in E, R \subseteq E : e \sqsubseteq e' \implies [\![R \cup e]\!] \subseteq [\![R \cup e']\!] \tag{2.2}$$

– Множество *точности* Π определяет уровень детализации абстрактного домена. Анализ использует элементы из Π, чтобы определять различный уровень точности для различных абстрактных элементов. Пара (e, π) называется абстрактным элементом e с точностью π. Cpa-операторы на абстрактном домене параметризуются уровнем точности.
Для $R \subseteq E \times \Pi$ обозначаем $[\![R]\!] = [\![\bigcup_{(e,\pi)\in R}\{e\}]\!]$.
– Отношение переходов $\rightsquigarrow$: $E \times \Pi \times 2^{E\times\Pi} \times E$ определяет для каждого частичного перехода e с точностью π и множества достигнутых частичных переходов $\widehat{R}$ следующие возможные абстрактные переходы e'. Будем писать $(e, \pi) \overset{\widehat{R}}{\rightsquigarrow} e'$, если $(e, \pi, \widehat{R}, e') \in \rightsquigarrow$.
Обозначим $Suc(e, R) = \{e' \mid e \overset{R}{\rightsquigarrow} e'\}$, то есть состояния достижимые из e, а объединение по всем состояниям из R: $Suc(R) = \bigcup_{e\in R} Suc(e, R)$.
Далее нам понадобится требование на $Suc(e, R)$:

$$\begin{aligned} &\forall R_1, R_2, \widehat{R} \subseteq E, e \in E : \\ &[\![R_1]\!] \subseteq [\![R_2]\!] \implies [\![Suc(e, R_1) \cup \widehat{R}]\!] \subseteq [\![Suc(e, R_2) \cup \widehat{R}]\!] \end{aligned} \tag{2.3}$$

Обозначим $Reach^k$ как

$$\begin{aligned} &\forall R \subseteq E : Reach^0(R) = R \\ &\forall k \geqslant 1 : Reach^{k+1}(R) = Suc(Reach^k(R)) \cup Reach^k(R) \end{aligned} \tag{2.4}$$

Далее мы будем писать $Reach(R) = Reach^1(R)$.
Требование к отношению переходов в классическом CPA [82] является слишком строгим. В некоторых случаях, в частности, при анализе многопоточных программ, более эффективным является аппроксимировать переходы из $[\![\widehat{R}]\!]$ в несколько шагов:

– применение переходов из частичных состояний по отдельности;
– применение некоторых шагов позже, например, переходов по окружению.

Требованием к отношению переходов является аппроксимация сверху множества конкретных переходов:

$$\begin{aligned} &\exists k \geqslant 1 : \forall R \subseteq E \times \Pi : \\ &[\![Reach^k(R)]\!] \supseteq \bigcup_{\tau\in[\![R]\!]} \{\tau' \mid \tau \longrightarrow \tau'\} \end{aligned} \tag{2.5}$$

Таким образом, условие 2.5 ослабляет требования на cpa-оператор $transfer$ по сравнению с классической теорией CPA. Оно означает,

что анализ может получить все конкретные переходы не за один шаг абстрактного перехода, а после k шагов. Для анализа каждого потока по-отдельности мы далее увидем, что $k = 2$. При $k = 1$ условие 2.5 вырождается в классическое требование на cpa-оператор $transfer$.

– Cpa-оператор слияния состояний $merge : E \times E \times \Pi \to E$ ослабляет второй параметр, используя информацию от первого параметра, и возвращает новое абстрактное состояние с точностью, которая передавалась, как третий параметр. Cpa-оператор *merge* должен удовлетворять следующим требованиям:

$$\forall e, e' \in E, \pi \in \Pi : \quad e' \sqsubseteq merge(e, e', \pi) \tag{2.6}$$

– Cpa-оператор останова $stop : E \times 2^{E \times \Pi} \times \Pi \to \mathbb{B}$ проверяет, является ли абстрактный переход, передаваемый, как первый параметр с точностью, передаваемой как третий параметр, покрытым множеством абстрактных переходов, которые передаются вторым параметром. Cpa-оператор останова должен удовлетворять следующему требованию:

$$\begin{aligned} &\forall e \in E, R \subseteq E, \pi \in \Pi : \\ &stop(e, R, \pi) \implies \forall \hat{R} \subseteq E : [\![\{e\} \cup \hat{R}]\!] \subseteq [\![R \cup \hat{R}]\!] \end{aligned} \tag{2.7}$$

– Функция настройки точности $prec : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ вычисляет новое абстрактное состояние и новую точность для данного абстрактного состояния и множества асбтрактных состояний. Эта функция может производить ослабление абстрактного состояния. Она должна удовлетворять следующему требованию:

$$\begin{aligned} &\forall e, e' \in E, \pi, \pi' \in \Pi, R \subseteq E \times \Pi : \\ &(e', \pi') = prec(e, \pi, R) \implies e \sqsubseteq e' \end{aligned} \tag{2.8}$$

В целом, множество точности Π, cpa-оператор останова $stop$, cpa-оператор объединения $merge$, cpa-оператор настройки точности $prec$ остаются такими же, как и в классической теории CPA.

2.3. Алгоритм вычисления достижимых переходов

Алгоритм 1 представляет основной алгоритм, который вычисляет множество достижимых абстрактных переходов. Он также не претерпел никаких изменений относительно классической теории CPA за исключением расширения сра-оператора $transfer$.

Описание алгоритма состоит из входных данных $Data$, описания результата $result$ и, собственно, алгоритма.

Data: адаптивный статический анализ $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$, начальный абстрактный переход e_0 с точностью $\pi_0 \in \Pi$, множество $reached$ элементов из $E \times \Pi$, множество $waitlist$ элементов из $2^{E \times \Pi}$
Result: множество достижимых состояний $reached$

```
waitlist := {(e_0, π_0)};
reached := {(e_0, π_0)};
while waitlist ≠ ∅ do
    pop (e, π) from waitlist;
    for e' : (e, π) ⇝^reached e' do
        (ê, π̂) = prec(e', π, reached);
        for (e'', π'') ∈ reached do
            e_new = merge(ê, e'', π̂);
            if e_new ≠ e'' then
                waitlist := waitlist \ {(e'', π'')} ∪ {(e_new, π'')};
                reached := reached \ {(e'', π'')} ∪ {(e_new, π'')};
            end
        end
        if !stop(ê, reached, π̂) then
            waitlist := waitlist ∪ {(ê, π̂)};
            reached := reached ∪ {(ê, π̂)};
        end
    end
end
```

Algorithm 1: Алгоритм $CPA(\mathbb{D}, e_0, \pi_0)$

Теорема 1. *(Soundness) Для заданного адаптивного статического анализа с частичными состояниями* $\mathbb{D}$ *и начального абстрактного состояния* e_0 *с точностью* π_0*, алгоритм* CPA *вычисляет множество абстрактных состояний, которое аппроксимирует сверху множество достижимых конкретных состояний:*

$$[\![CPA_{PS}(\mathbb{D}, e_0, \pi_0)]\!] \supseteq Reach_{\longrightarrow}([\![\{e_0\}]\!])$$

Доказательство теоремы 1 приведено в Приложении A.1.

2.4. Конфигурация CPA

Представление анализа в терминах cpa-операторов было разработано в качестве универсального способа для описания различных техник верификации программы. Например, классические методы проверки моделей и методы анализа потоков данных могут быть записаны единообразно, что позволит более детально сравнить эти методы, а также оценить их эффективность в одинаковых условиях.

Другим важным следствием универсального представления становится возможность переиспользовать существующие CPA и строить новые типы анализа на их основе, в том числе, путем комбинации нескольких CPA. При этом существующие CPA могут быть дополнены новыми вариантами cpa-операторов, которые позволят более точно настроить их параметры под конкретную задачу.

Настройка CPA, или его конфигурация, позволяет получить необходимый баланс между скоростью анализа и его точностью. Полная конфигурация алгоритма анализа заключается в подготовке набора используемых CPA, а также в выборе нужных вариантов их cpa-операторов.

Набор различных CPA обычно представляется в виде дерева, то есть, один CPA может быть вложен в другой. Cpa-операторы корневого CPA используются в алгоритме 1. А вложенность CPA друг в друга означает, что при использовании cpa-операторов верхнего CPA, применяются cpa-операторы и внутреннего. При этом внутренний CPA должен гарантировать выполнение требований на свои cpa-операторы с учетом возможного влияния внутреннего CPA. Такой способ позволяет использовать несколько различных CPA за один запуск алгоритма, что и означает композицию различных вариантов анализа.

Подход с раздельным рассмотрением потоков будет реализовывать как раз такую схему. Соответствующий CPA (будем называть его ThreadModularCPA), который реализует всю функциональность предложенного подхода, будет внешним CPA для алгоритма построения множества достижимых состояний (алгоритм 1), то есть, его cpa-операторы будут использоваться алгоритмом. В ThreadModularCPA могут быть вложены другие CPA, на которые накладываются дополнительные требования. Далее будет рассмотрен пример вложенного CPA – CompositeCPA, который реализует возможность параллельной композиции различных CPA. В CompositeCPA может быть вложено несколько различных CPA, которые работают независимо и параллельно, то есть, соответствующие cpa-операторы вызываются независимо друг от друга.

Далее будет представлено описание ThreadModularCPA, CompositeCPA и примеры CPA, которые реализуют различные варианты анализа: анализ предикатов (PredicateCPA), анализ примитивов синхронизации (LockCPA), анализ потоков (ThreadCPA) и др.

2.5. Адаптивный статический анализ с раздельным рассмотрением потоков

2.5.1. Общая схема метода

Рассмотрим простую программу, в которой всего два потока (рис. 2.1).

```
volatile int g = 0;          volatile int d = 0;
Thread1 {                    Thread2 {
1:  g = 1;                   4:  if (d == 1) {
2:  d = 1;                   5:    g = 2;
3:  …                        6:  }
}                            }
```

Рисунок 2.1 — Пример небольшой программы

Это некоторый модельный пример, в котором используется конструкция неявной синхронизации между потоками: первый поток инициализирует некоторые данные (в данном случае, глобальную переменную g), а затем выставляет флаг, что данные готовы. Второй поток может использовать эти данные только

после выставления флага, поэтому в этом примере нет состояния гонки для переменной g. Классические методы проверки моделей перебирают все возможные варианты чередования двух потоков.

С точки зрения инструмента статической верификации, необходимо рассмотреть полное множество состояний программы, которые возникают при всех возможных чередованиях (рис. 2.2).

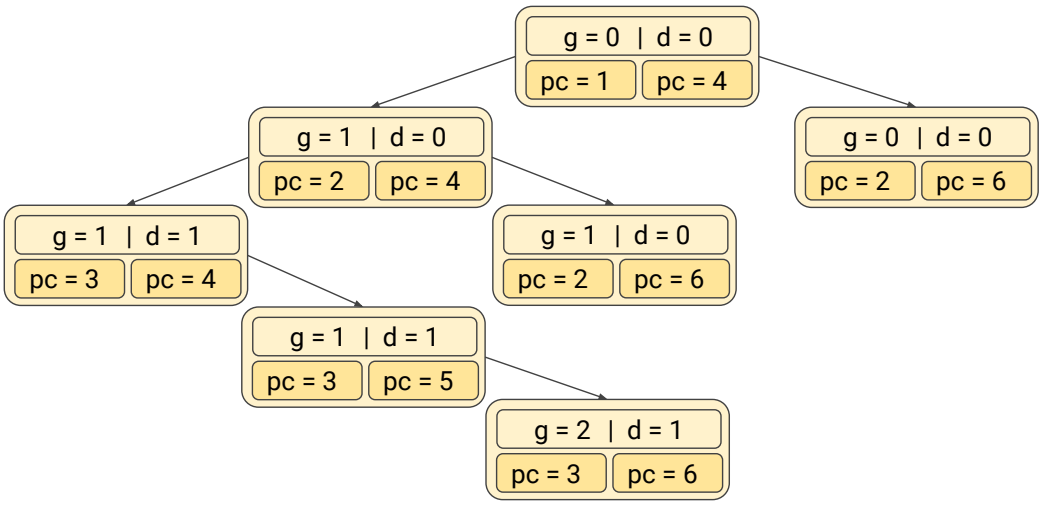

Рисунок 2.2 — Построение множества чередований

Даже в простом примере и при различных оптимизациях общее число состояний растет с катастрофической скоростью. Происходит так называемый «комбинаторный взрыв» числа состояний, что приводит к исчерпанию ресурсов. Таким образом, классические методы проверки моделей не могут обеспечить доказательства корректности программы.

Простые методы статического анализа пытаются вычислить аппроксимацию сверху возможных действий одного потока на другой, т.н. эффект потока. Однако, они не способны прослеживать сложные зависимости между переменными. Например, зависимости между глобальными переменными, которые, в свою очередь, могут быть модифицированы в других потоках. В общем случае, это требует вычисления некоторой неподвижной точки, что является нежелательным при статическом анализе, так как значительно возрастают требования к ресурсам. В итоге, в таких сложных случаях считается, что глобальные переменные могут принимать любые значения. А это, в свою очередь, снижает точность анализа.

Предлагаемый подход базируется на известной идее раздельного анализа потоков (англ. thread-modular approach). Потоки в этом случае анализируются по-отдельности, одновременно с этим строится общее для всех потоков окружение, которое аппроксимирует сверху влияние других потоков. Это окружение форми-

руется на основе анализа всех потоков, так как каждый поток являются частью окружения для других потоков. Для каждого потока определяется, как и в каких условиях он может модифицировать разделяемые данные, использовать примитивы синхронизации и выполнять иные действия, влияющие на другие потоки. Точность анализа потока зависит от того, как точно будет сформировано окружение. Однако, остается вопрос как эффективно вычислять и представлять окружение.

При анализе последовательных программ успешной техникой, позволяющей уменьшить число рассматриваемых состояний программы, является абстракция. Она позволяет абстрагироваться от несущественных деталей программы и рассматривать обобщенные (абстрактные) состояния, которые могут соответствовать целому множеству реальных (конкретных) состояний программы. Это позволяет значительно сократить пространство состояний. Ключевой идеей предлагаемого подхода является расширение абстракции не только на состояния программы, но и на операции, то есть, переходы потока. Настраивая уровень абстракции, можно получать варианты анализа, которые будут ближе к статическому анализу многопоточных программ, или к классической статической верификации.

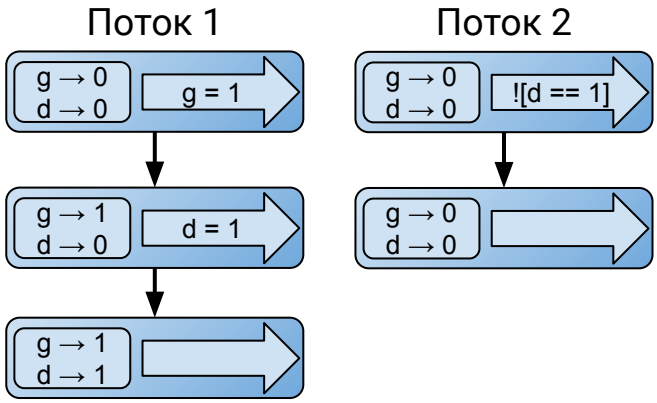

Рисунок 2.3 — Построение абстрактных переходов двух потоков

Рисунок 2.3 показывает часть абстрактного графа достижимости (англ. Abstract Reachability Graph, ARG) для первого и второго потока без влияния друг на друга. Представленный анализ основан на простом анализе явных значений, который отслеживает только явные значения переменных. Переход содержит в себе абстрактное состояние и абстрактную операцию. Первое абстрактное состояние содержит информацию только о значении глобальной переменной x. Новая информация о значении переменной y появляется в дочерних элементах, после того как выполнен переход, соответствующей инициализации переменной.

Теперь необходимо учесть влияние потоков друг на друга, то есть сформировать окружение. Будем называть проекцией операции потока описание ее эффекта, видимого для других потоков. Например, любые модификации локальных переменных потока не влияют на другие потоки, то есть их проекция является пустой операцией. Модификация глобальной переменной является значимой для всех потоков, поэтому ее проекция должна совпадать с самой операцией, либо аппроксимировать ее сверху, например, теряя информацию о точном присваиваемом значении. При этом в проекции может быть не только информация о самом действии, но и об условии на его применение к другому потоку. Например, на рисунке 2.4 первый поток, присваивая $g = 1$ меняет значение переменной g с нуля на единицу. Можно представить проекцию этой операции таким образом: если значение переменной x равно нулю, то оно может быть изменено на единицу. Иными словами, проекция состоит из двух частей: условия ее применения ($[g == 0]$) и непосредственно действия ($g \to 1$).

При анализе некоторого потока одновременно строится его представление для остальных в качестве окружения. Оно состоит из набора проекций операций этого потока. Далее каждая из этих проекций должна быть применена ко всем возможным (с учетом условий внутри проекций) состояниям других потоков. Что, в свою очередь, может породить новые, еще не исследованные состояния, а значит, и проекции.

После построения переходов в потоках независимо друг от друга (рисунок 2.3), для всех переходов вычисляются проекции. Для второго потока, например, это действие, которое меняет значение переменной x значение с нуля на тройку. Остальные действия второго потока не модифицируют глобальные переменные и не порождают значимых проекций. Затем эта проекция применяется к каждому состоянию первого потока. На рисунке 2.4 приведен результат применения к первому переходу. Также эту проекцию можно применить и ко второму, однако, никаких новых путей это не породит. К третьему переходу первого потока применить данную проекцию нельзя, так как значение переменной x не удовлетворяет условию проекции. Применение проекции к первому переходу порождает новый путь выполнения, который в свою очередь может породить новые проекции.

На рисунке 2.4 представлен вариант с точными проекциями, которые рассматривают переходы другого потока так, как они есть. На рисунке представлены не все возможные проекции и порожденные ими переходы. Например, отсутствует проекция перехода $g = 2$ второго потока.

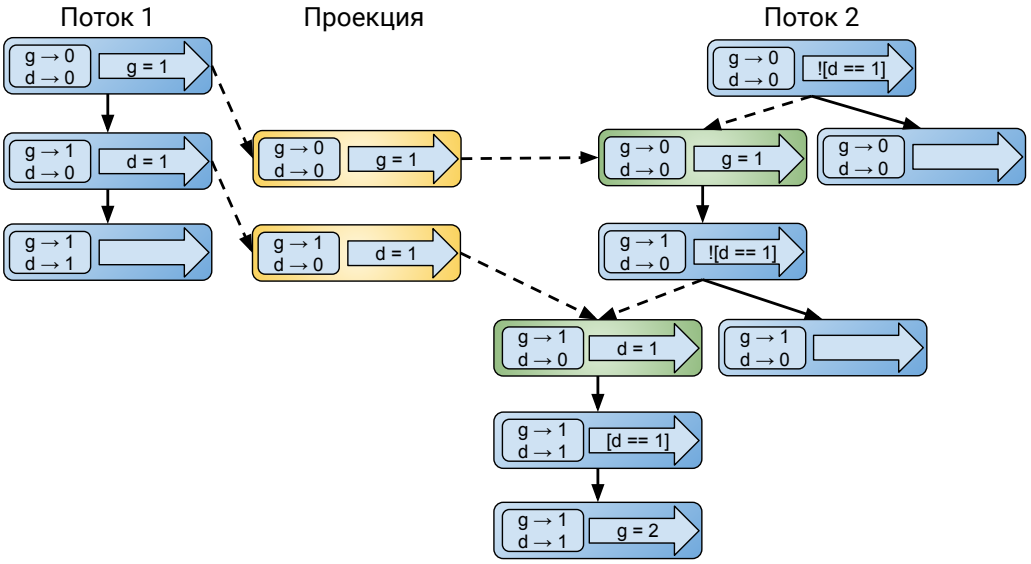

Рисунок 2.4 — Построение абстрактных переходов двух потоков

Для проверки возможности состояния гонки нам необходимо найти два перехода, которые модифицируют одну переменную: $g = 1$ в первом потоке и $g = 2$ во втором. Далее, необходимо проверить, являются ли два абстрактных состояния совместными, то есть, могут ли они быть частью одного глобального состояния. В данном случае, в частичных состояниях потока значения глобальных переменных имеют разные значения, а значит, они не могут быть частью одного глобального состояния, то есть, указанные два перехода не могут быть выполнены одновременно. Отсюда следует, что состояние гонки отсутствует.

Предлагаемый подход предоставляет гибкие варианты конфигурации для решения каждой конкретной задачи. Как было показано на примерах, проекции действий потока могут быть представлены более точной абстракцией или, наоборот, слишком общей. Проекции нескольких операций могут быть объединены в одну или быть рассмотрены по-отдельности. Это позволяет выбирать необходимый баланс между точностью и скоростью.

Рассмотрим другой вариант построения абстрактных переходов на рисунке 2.5.

Здесь используется более абстрактное представление проекции, при котором несколько воздействий потока объединяются в одну проекцию (эффект от окружения). При этом обычно теряется некоторая информация. В частности, в данном случае была потеряна информация о точном значении переменной g, и поэтому данный объединенный эффект может применяться при любых ее значениях. Это позволяет сократить число состояний для анализа. Пример показывает, как анализ рассматривает эффекты влияния одного потока на другой, что гаранти-

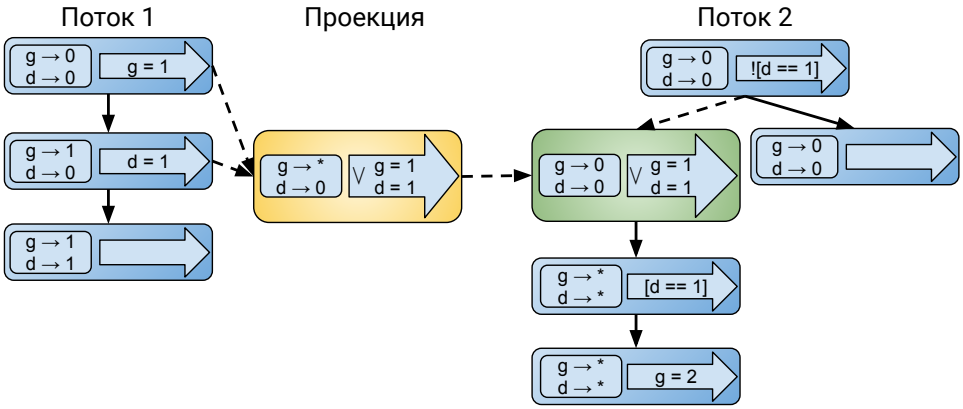

Рисунок 2.5 — Построение абстрактных переходов двух потоков

рует корректность подхода. Более того, он показывает гибкость подхода, который позволяет варьировать уровень абстракции, например, на стадии построения проекции, выбирая уровень абстракции каждого перехода в окружении.

2.5.2. Формальное описание внутреннего CPA

CPA, реализующий логику анализа с раздельным рассмотрением потоков, требует некоторые дополнительные возможности от вложенных CPA. Поэтому прежде, чем формально описывать ThreadModularCPA, опишем расширенные требования ко вложенным CPA.

Определение CPA, который может быть использован внутри анализа с раздельным рассмотрением потоков, расширяется дополнительным набором cpa-операторов: $compatible_I$, $\cdot|_p$, $compose_I$. Таким образом, полный набор cpa-операторов для определения CPA теперь выглядит следующим образом: $\mathbb{I} = (D_I, \Pi_I, \rightsquigarrow_I\ , merge_I, stop_I, prec_I, compatible_I, \cdot|_p, compose_I)$. Кроме того, усиливаются требования к основным cpa-операторам.

Абстрактный домен $D_I = (\mathcal{T}_I, \mathcal{E}_I, \bigoplus_I)$ включает в себя множество конкретных переходов $\mathcal{T}_I$, полурешетку абстрактных переходов $\mathcal{E}_I$, а вместо функции конкретизации, в отличие от обычного CPA, используется cpa-оператор композиции $\bigoplus_I$. Это необходимо из-за того, что для подхода с раздельным анализом потоков используется одна общая схема вычисления конкретных состояний, которая основана на этом cpa-операторе композиции. Поэтому от вложенных CPA требу-

ется только определить сра-оператор композиции, который объединяет несколько частичных переходов во множество конкретных.

Как было уже сказано, состояния и переходы являются частичными, поэтому они могут не соответствовать напрямую конкретным состояниям и переходам. Чтобы получить полный переход, нужно взять композицию множества частичных переходов, которые соответствуют всем доступным потокам. Совместные частичные переходы могут быть объединены в полный конкретный переход с помощью сра-оператора композиции $\bigoplus$: $E \times T \times 2^{E \times T} \to 2^{\mathcal{T}}$. Он возвращает множество конкретных переходов, которое соответствует данным частичным переходам.

Основное требование к сра-оператору $\bigoplus_I$ состоит в том, что он должен соответствовать полурешетке. Так, если один из абстрактных переходов меньше, чем другой, то композиция с тем же множеством не должна получить большее множество конкретных переходов.

$$\begin{gathered}\forall e \sqsubseteq e' \in E, e_1, \dots e_n \in E, t_0, t_1, \dots, t_n \in T, t_i \neq t_j : \\ \bigoplus_I \left(\begin{pmatrix} e \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \subseteq \\ \bigoplus_I \left(\begin{pmatrix} e' \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)\end{gathered} \tag{2.9}$$

И симметричное требование для второго аргумента:

$$\begin{gathered}\forall e \sqsubseteq e' \in E, e_1, \dots e_n \in E, t_0, t_1, \dots, t_n \in T, t_i \neq t_j : \\ \bigoplus_I \left(\begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \left\{ \begin{pmatrix} e \\ t_0 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \subseteq \\ \bigoplus_I \left(\begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \left\{ \begin{pmatrix} e' \\ t_0 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)\end{gathered} \tag{2.10}$$

Сра-оператор проверки совместности $compatible_I : E \times E \to \{true, false\}$ проверяет, могут ли два частичных перехода начинаться из общего полного родительского состояния. Этот сра-оператор позволяет исключить из анализа те переходы, которые никак не могут выполняться параллельно друг с другом, а значит, и влиять друг на друга.

Сра-оператор проекции $\cdot|_p : E \to E$ проецирует переход в потоке на окружение, то есть, оставляет только изменение глобального состояния программы. При этом точность проекции зависит от точности анализа. Например, проекция

может содержать конкретные присваиваемые значение глобальным переменным или лишь информацию, что некоторые глобальные переменные могли измениться неопределенным образом.

$compose_I : E \times E \to E$ объединяет два абстрактных перехода в один. Он применяет абстрактную дугу из одного перехода к абстрактному состоянию другого перехода.

В дальнейшем мы будем использовать cpa-оператор $apply_I$, как комбинацию трех cpa-операторов: $\cdot|_p$, $compose_I$ и $compatible_I$::

$$\forall e, e' \in E : apply(e, e') = \begin{cases} compose_I(e, e'|_p), & \text{если } compatible_I(e, e'|_p) \\ \bot, & \text{иначе} \end{cases} \tag{2.11}$$

Таким образом, cpa-оператор $apply$ означает, что переходы могут быть объединены, только если они совместны. Результатом применения cpa-оператора является новый переход, который будем называть переходом в окружении, так как он представляет собой эффект окружения.

Cpa-оператор $apply$ связан с cpa-оператором $transfer_I$, поэтому отдельного условия на него нет. Тем не менее, в дальнейшем будет описано явное условие 2.13, связывающее cpa-операторы $apply$ и $transfer_I$.

2.5.3. Алгоритм построения окружения потока в терминах CPA

Определим специальный CPA, который реализует логику отдельного анализа с раздельным рассмотрением потоков, в частности алгоритм построения окружения потока: $\mathbb{TM} = (D_{TM}, \Pi_{TM}, \rightsquigarrow_{TM}, merge_{TM}, stop_{TM}, prec_{TM})$, который основан на внутреннем CPA $\mathbb{I} = (D_I, \Pi_I, \rightsquigarrow_I, merge_I, stop_I, prec_I, compatible_I, \cdot|_p, compose_I)$.

1. Абстрактный домен $D_{TM} = (\mathcal{T}, \mathcal{E}, [\![\cdot]\!]_{TM})$.
   Полурешетка $\mathcal{E} = \mathcal{E}_I$ эквивалентна полурешетке внутреннего анализа. Функция конкретизации $[\![\cdot]\!]$ выражается через cpa-оператор композиции:

$$\forall R \subseteq E : [\![R]\!]_{TM} =$$
$$\bigcup_{k=1}^{\infty} \bigcup_{\substack{e_0, e_1, \ldots, e_k \in R \\ t_0, t_1, \ldots, t_k \in T}} \bigoplus\nolimits_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) \tag{2.12}$$

Таким образом, множество конкретных состояний получается композицией всех возможных подмножеств из частичных состояний. Такое определение необходимо для потенциально бесконечных конкретных состояний, которые успешно поддерживаются с помощью конечных абстрактных состояний. Покажем, что при выполнении 2.9, 2.10 требования 2.1, 2.2 выполнены.

Доказательство. Предположим, что $R \subseteq R' \subseteq E$

$$[\![R]\!]_{TM} = \bigcup_{k} \bigcup_{\substack{e_0, \ldots, e_k \in R \\ t_0, \ldots, t_k \in T}} \bigoplus\nolimits_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) \subseteq$$
$$\subseteq \bigcup_{k} \bigcup_{\substack{e_0, \ldots, e_k \in \overline{R} \\ t_0, \ldots, t_k \in T}} \bigoplus\nolimits_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) = [\![R']\!]_{TM}$$

Теперь предположим, что $e \sqsubseteq e' \in E, R \subseteq E$

$$[\![R \cup \{e\}]\!]_{TM} =$$
$$\bigcup_{k} \bigcup_{\substack{e_0, \ldots, e_k \in R \cup \{e\} \\ t_0, \ldots, t_k \in T}} \bigoplus\nolimits_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) \subseteq$$
$$\subseteq (req.\ 2.9, 2.10) \subseteq$$
$$\subseteq \bigcup_{k} \bigcup_{\substack{e_0, \ldots, e_k \in R \cup \{e'\} \\ t_0, \ldots, t_k \in T}} \bigoplus\nolimits_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) =$$
$$= [\![R \cup \{e'\}]\!]_{TM}$$

□

2. Отношение переходов определяет следующие переходы, после чего применяет все достигнутые переходы, как переходы в окружении, к новым переходам, а новые переходы, как переходы в окружении, – к уже достижимым. Отношение перехода $transfer$ представлено на алгоритме 2.

Data: начальный переход e_0 с точностью $\pi_0 \in \Pi$
Result: множество следующих переходов $result$
$result := \varnothing$;
for *each* $\widehat{e} : e_0 \overset{R}{\rightsquigarrow}_I \widehat{e}$ **do**
  $result := result \cup \{\widehat{e}\}$;
  for *each* $e' \in reached$ **do**
    $result := result \cup \{apply(e', \widehat{e})\}$;
    $result := result \cup \{apply(\widehat{e}, e')\}$;
  end
end
return result

Algorithm 2: $transfer_{TM}(e_0, \pi_0, reached)$

Нужно доказать, что такой cpa-оператор $transfer$ удовлетворяет условию 2.5. Для этого нам потребуется более сложное условие, связывающее cpa-операторы внутреннего CPA 2.13:

$$\forall \tau, \tau' \in \mathcal{T}, R \subseteq E : \tau \longrightarrow \tau', \forall e_0, e_1, \ldots, e_n \in R : e_i = (s_i, q_i)$$
$$\left(\begin{array}{l} \exists t_0, t_1, \ldots, t_n \in T, t_i \neq t_j, t_0 = t \\ \tau \in \bigoplus_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \end{array} \right) \Rightarrow$$
$$\left[\begin{array}{ll} \vee q_0 & \neq thread\_create \wedge \\ & \exists e'_0, e'_1, \ldots, e'_n \in E, \forall 1 \leqslant i \leqslant n : e_i \overset{R}{\rightsquigarrow} e'_i : \\ & \exists 1 \leqslant k \leqslant n : t_k = t' \wedge \\ & \tau' \in \bigoplus_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\}) \\ \vee q_0 & = thread\_create \wedge \\ & \exists e'_0, e'_1, \ldots, e'_{n+1} \in E, \forall 1 \leqslant i \leqslant n : e_i \overset{R}{\rightsquigarrow} e'_i \wedge e_0 \overset{R}{\rightsquigarrow} e'_{n+1} : \\ & \exists 1 \leqslant k \leqslant n+1 : t_k = t' \wedge \\ & \tau' \in \bigoplus_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\}) \end{array} \right. \quad (2.13)$$

Доказательство. Действительно, рассмотрим случайный переход $\tau \longrightarrow \tau'$, $\tau \in [\![R]\!]$. Нужно показать, что $\exists k : \tau' \in Reach^k(R)$. Так как $\tau \in [\![R]\!]$ по определению $[\![R]\!]$ 2.12 это означает, что $\exists t, t_1, \ldots, t_n \in T, t_i \neq t_j, e_0, e_1, \ldots, e_n \in R$:

$$\tau \in \bigoplus_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right).$$

По условию на внутренние сра-операторы 2.13 это означает, что найдутся такие элементы $e'_0, e'_1, \ldots, e'_n \in E$, которые могут быть получены за два шага:

а) применение сра-оператора $transfer$ и сра-оператора $apply$ для перехода в текущем потоке;

б) применение сра-оператора $transfer$ к переходам в окружении, которые были получены на предудыщем шаге с помощью сра-оператора $apply$.

□

И для этих элементов $e'_0, e'_1, \ldots, e'_n$ будет выполнено $\tau' \in \bigoplus_I ((e'_i, t_i), \{(e'_0, t_0), \ldots, (e'_n, t_n)\})$. То есть, показано, что $\exists k = 2 : \tau' \in Reach^k(R)$.

3. $\Pi_{TM} = \Pi_I$
4. $merge_{TM} = merge_I$.
5. $stop_{TM} = stop_I$.
6. $prec_{TM} = prec_I$.

Для сра-операторов $merge$, $stop$, $prec$ требования для внутреннего анализа совпадают с требованиями 2.6, 2.7, 2.8.

2.5.4. Использование явного вида переходов

Описанный в предыдущих разделах вариант анализа является достаточно общим. В различных CPA, применяемых на практике, вид абстрактного перехода зачастую может быть явно разделен на абстрактное состояние и на абстрактную дугу. В этом случае, можно упростить описание этого анализа, используя явное разделение перехода на две части, каждая из которых отображается на свои подмножества конкретных элементов. В этом разделе опишем общие свойства для

такого варианта CPA, чтобы в дальнейшем не повторять их для каждого варианта анализа.

Итак, предположим, что полурешетка конкретных переходов $\mathcal{E}_I$ состоит из двух частей: $\mathcal{E}_I = \mathcal{E}_I^S \times \mathcal{E}_I^T$. Это означает, что множество элементов также состоит из двух частей $E_I = E_I^S \times E_I^T$, $\top_I = \{\top_I^S, \top_I^T\}$, $\bot_I = \{\bot_I^S, \bot_I^T\}$, а все cpa-операторы получаются такой же композицией:
$\forall e, e' \in E_I, e = (s, q), e \sqsubseteq e' \iff (s \sqsubseteq s' \wedge q \sqsubseteq q')$,
$\forall e_1, e_2, e \in E_I, e = (s, q), e = e_1 \sqcup e_2 \iff (s = s_1 \sqcup s_2 \wedge q = q_1 \sqcup q_2)$.

Таким образом, имеют место две независимые решетки: над состояниями и над дугами.

Кроме того, множество абстрактных дуг почти всегда включает в себя конкретные дуги $g \in G$. То есть, $E_I^T = G \cup \mathcal{G}$, где $\mathcal{G}$ и есть множество абстрактных дуг (включая $\bot_I^T$). При этом, для абстрактных дуг можно представить аналог конкретизации для состояний: $||\cdot||_I : E_I^T \to 2^G$. Для корректности, должно быть обеспечены следующие условия:

$$\forall e, e' \in E_I^E : e \sqsubseteq e' \implies ||e||_I \subseteq ||e'||_I \tag{2.14}$$

$$\forall q \in \mathcal{G}, g \in G : g \in ||q|| \implies g \sqsubseteq_I^E q \tag{2.15}$$

Для cpa-оператора композиции $\bigoplus_I$ почти всегда требуется вспомогательный предикат для определение совместности множества переходов $check_C : 2^{E_I} \to \mathbb{B}$, то есть могут ли указанные частичные переходы составить один глобальный переход. Не путать с cpa-оператором $compatible_I$, который проверяет возможность того, что два различных перехода могут *начинаться* из одного состояния. Этот cpa-оператор $check_C$ может проверять соответствие перехода состоянию, то есть для некоторых типов анализа он не может быть разбит на два cpa-оператора, определенных на множестве состояний и на множестве переходов. Тем не менее, cpa-оператор $\bigoplus_I$ может быть представлен в виде композиции:

$$\forall e_1,\dots,e_n \in E_I, e_i = (s_i,q_i):$$
$$\bigoplus_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) =$$
$$= \begin{cases} \left\{ (c,g,t_0) \in \mathcal{T} \;\middle|\; \begin{array}{c} c \in \bigoplus_I^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right) \\ g \in ||q_0||_I \cap ||q_1||_I \cap \dots \cap ||q_n||_I \} \end{array} \right\}, \\ \quad \text{if } check_C(e_0, \{e_1,\dots,e_n\}) \\ \varnothing, \text{ otherwise} \end{cases} \tag{2.16}$$

При этом $\bigoplus_I^S$ должен удовлетворять похожим условиям(2.9, 2.10):

$$\forall s \sqsubseteq s' \in E_I^S, s_1,\dots s_n \in E_I^S, t_0,t_1,\dots,t_n \in T, t_i \neq t_j:$$
$$\bigoplus_I^S \left(\begin{pmatrix} e \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \subseteq$$
$$\bigoplus_I^S \left(\begin{pmatrix} e' \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \tag{2.17}$$

И симметричное требование для второго аргумента:

$$\forall s \sqsubseteq s' \in E_I^S, s_1,\dots s_n \in E_I^S, t_0,t_1,\dots,t_n \in T, t_i \neq t_j:$$
$$\bigoplus_I^S \left(\begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \left\{ \begin{pmatrix} e \\ t_0 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \subseteq$$
$$\bigoplus_I^S \left(\begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \left\{ \begin{pmatrix} e' \\ t_0 \end{pmatrix}, \dots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \tag{2.18}$$

Из условий 2.14, 2.15, 2.17, 2.18 следуют условия 2.9, 2.10, так как множества конкретных состояний и абстрактных дуг только расширяются.

Нужно отметить, что разбить cpa-оператор $\bigoplus_I^S$ по аналогии с cpa-оператором $\bigoplus_I^E$ невозможно, так как уже существуют варианты анализа, которые используют иформацию о других частичных состояниях. Хотя таких видов анализа очень мало.

Как уже было упомянуто, вспомогательный cpa-оператор $check_C$ в общем случае не может быть разбит на две части, однако, часто он может быть представлен через cpa-оператор проекции следующим образом:

$$\begin{aligned}&\forall e_1,\ldots,e_n \in E_I, e_i = (s_i,q_i) : \\ &check_C(e_0,\{e_1,\ldots,e_n\}) = \\ &check_C^S(s_0,\{s_1,\ldots,s_n\}) \wedge (\forall 1 \leqslant i \leqslant n, e_0|_p = (\overline{s},\overline{q}), \overline{q} \sqsubseteq q_i)\end{aligned} \tag{2.19}$$

Представление 2.19 означает, что анализ не отслеживает соответствие состояний переходам. В этом случае, состояния должны быть совместными, а от абстрактных дуг требуется, чтобы переходы в окружении соответствовали исходному переходу в потоке.

Если переход разбивается на две части, то сра-оператор $compose_I$ становится тривиальным:

$$\forall e, e' \in E_I, e = (s,q), e' = (s',q') : compose_I(e,e') = \tilde{e} = (s,q') \tag{2.20}$$

Сра-оператор $compatible_I$ проверяет возможность параллельного выполнения различных переходов из частичных состояний, поэтому обычно имеет место такое выражение:

$$\forall e, e' \in E_I, e = (s,q), e' = (s',q') : compatible_I(e,e') = check_C^S(s,s') \tag{2.21}$$

2.5.5. Анализ, инвариантный к эффектам окружения

Анализ $\mathbb{I}$ будем называть инвариантным к эффектам окружения, если $\forall e, e' \in E, R \subseteq E, \widehat{e} = apply_I(e,e') : \widehat{e} = \bot \vee \widehat{e} \overset{R}{\rightsquigarrow} e$. То есть, никакие переходы, полученные применением эффектов окружения (проекций) не могут изменить текущего состояния (перехода). В этом случае применение эффектов окружения является бессмысленным.

Нужно заметить, что такой анализ не сводится к классическому анализу, так как по-прежнему $\forall R \subseteq E_C : [\![R]\!]_R \neq \bigcup_{e \in R} [\![e]\!]_C$. Кроме того, часто может быть определен нетривиальный сра-оператор $compatible^I$ внутри сра-оператора $apply_I$. Это означает, что несмотря на то, что эффектны окружения этого вида анализа не могут изменить его состояние, сам он способен влиять на применение эффектов окружения на другие типы анализов, которые не являются инвариантными к эффектам окружения.

Если все используемые при анализе программы виды анализа инвариантны к эффектам окружения, это позволяет значительно повысить скорость работы, за счет применения только переходов в потоке. При этом, как правило, теряется точность анализа, так как анализ полностью абстрагируется от поведения других потоков.

2.6. Анализ с раздельным рассмотрением потоков без абстракции

В этом разделе покажем, что в представленную теорию укладывается классический алгоритм проверки моделей, описанный в [68].

Определим анализ с раздельным рассмотрением потоков с эффектами окружения, как $\mathbb{Q} = (D_Q, \Pi_Q, \rightsquigarrow_Q, merge_Q, stop_Q, prec_Q, compatible_Q, \cdot|_p, compose_Q)$.

Также как и в алгоритме [68] применение анализа возможно только к программам с ограниченным количеством точек создания потоков. Далее предполагаем, что программа имеет ограниченное количество потоков, которые отличаются точками в программе, обозначающих начало потока, например, для $thread\_create(pc_\nu)$ будет всегда создан поток с идентификатором pc_ν.

1. $D_Q = (\mathcal{T}_Q, \mathcal{E}_Q, \bigoplus_Q)$.

   $\mathcal{T}_Q = C \times G \times T$ – все конкретные переходы программы.

   $\mathcal{E}_Q = (E_Q, \top^E_Q, \bot^E_Q, \sqsubseteq^E_Q, \sqcup^E_Q)$ определен над $E_Q = \mathcal{R} \times T \times (G \cup \mathcal{G})$, где $\mathcal{R}$ множество всех проекций конкретных состояний на некоторый поток: $\mathcal{R} \subseteq T \times L \times C^{local} \times c_g \times c_s$, а $\mathcal{G}$ содержит эффекты окружения на глобальные части состояния $\mathcal{G} \subseteq c_g \times c_s \times c_g \times c_s$.

   Отметим, что кодирование элементов анализа было максимально приближено к оригинальному, описанному в статье [68]. И в отдельных случаях кодирование может быть избыточно.

   Для переходов определим дополнительный cpa-оператор, проверяющий, могут ли указанные частичные переходы образовывать глобальный переход:

$$\begin{aligned}
&\forall e_0,\ldots,e_n \in E_Q, e_i = (s_i,t_i,q_i), s_i = (\widehat{t_i},pc_i,l_i,gl_i,cs_i), t_i,\widehat{t_i} \in T \\
&check_C(e_0,\{e_1,\ldots,e_n\}) = \\
&\forall 1 \leqslant i \leqslant n : t_i = t_0 \wedge \widehat{t_i} \neq \widehat{t_k} \wedge gl_i = gl_k \wedge cs_i = cs_k \wedge \\
&e_0|_p = (s_p,t_p,q_p), q_p = q_i
\end{aligned} \tag{2.22}$$

Для совместных переходов $e_1,\ldots,e_n$ можно определить $\widehat{gl} = gl_i = gl_k$ и $\widehat{s} = s_i = s_k$. Сра-оператор композиции $\bigoplus_Q$ можно определить с помощью сра-оператора 2.22.

$$\begin{aligned}
&\forall e_0,\ldots,e_n \in E_Q, t_i \in T, e_i = (s_i,t_i,q_i), s_i = (\widehat{t_i},pc_i,l_i,gl_i,cs_i) \\
&\bigoplus\nolimits_Q \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) = \\
&\begin{cases}
\left\{ \left(\begin{array}{c|c} (c,g,t_0) \in \mathcal{T} & \{t_0 \to pc_0,\ldots,t_j \to pc_j\}, \\ c = (c_{pc},c_l,c_g,c_s) & \{t_0 \to l_0,\ldots,t_j \to l_j\}, \\ & \hat{gl},\hat{s} \\ & g \in \|q_0\| \end{array} \right) \right\} \\
\quad \text{, if } check_C(\{e_0,\ldots,e_j\}) \\
\varnothing, \text{ otherwise}
\end{cases}
\end{aligned} \tag{2.23}$$

$\sqsubseteq_Q, \sqcup_Q$ определены как равенство соответствующих элементов, то есть $e_1 \sqsubseteq_Q^E e_2 \iff e_1 = e_2$, а $e_1 \neq e_2 \iff e_1 \sqcup_Q e_2 = \top^E$. Таким образом, основные требования 2.9, 2.10 на $\bigoplus_Q$ выполнены.

2. $\Pi_Q = \{\varnothing\}$ содержит один элемент, так как анализ не использует абстракцию.
3. Отношение переходов $\rightsquigarrow_Q$ содержит переход $e \overset{R}{\rightsquigarrow}_Q e'$, $e = (s,t,q)$ если
   - $q \in G$. Пусть $s = (t,pc,l,gl,cs)$ и существует соответствующий переход на проекциях
     - $(\{t \to pc\},\{t \to l\},gl,s) \xrightarrow{g,t} (\{t \to pc'\},\{t \to l'\},gl',cs'))$, где $g \neq thread\_create$, тогда следующее состояние $s' = (t,pc',l',gl',cs')$.
     - или в случае $q = thread\_create(pc_\nu)$, $\nu = pc_\nu$, $(\{t \to pc\},\{t \to l\},gl,cs) \xrightarrow{thread\_create(pc_\nu),t} (\{t \to pc',\nu \to pc_\nu\},\{t \to l,\nu \to l\},gl,cs))$, тогда следующее

состояние либо $s' = (t,pc',gl,s)$ (родительский поток), либо $s' = (\nu,pc_\nu,gl,s)\}$ (новый поток).

– $q = (gl, cs, gl'',cs'') \in \mathcal{G}$, $s = (t, pc, l, gl, cs)$. Тогда следующее состояние $s' = (t, pc, l, gl'', cs'')$.

Доказательство условия 2.13, связывающее сра-операторы $transfer$, $\bigoplus$ и $apply$, приведено в приложении А.2.

4. $merge_Q(e_1,e_2,R) = e_2$. Данный сра-оператор очевидно удовлетворяет условию 2.6.
5. $stop_Q(e,R,\pi) = \exists e' \in R : e \sqsubseteq e'$. Данный сра-оператор очевидно удовлетворяет условию 2.7.
6. $prec_Q(e,\pi) = (e,\pi)$ (точность и состояние никогда не изменяются). Данный сра-оператор очевидно удовлетворяет условию 2.8.
7. $compose_Q(e,e') = \tilde{e} = (s, t, g')$.
8. $e|_p = e' = (s, t, (gl, s, gl', s'))$.
9. $compatible_Q : E_Q \times E_Q \to \mathbb{B}$ определяется, как равенство глобальных частей состояний, $compatible_Q(e,i)$ для $e = (t,pc, l, gl,s)$ и $e' = (t',gl',s',gl'',s'')$ есть $\forall \tau, \tau' \in E_Q, \tau_i = (e_i, t_i, q_i), e_i = (t_i,pc_i, l_i, gl_i,s_i)$ $compatible_Q(\tau, \tau') = (gl = gl' \wedge s = s')$

2.7. Композиция различных видов анализа

Композиция различных типов анализа принципиально важна для объединения различных техник анализа программы в одном алгоритме. CompositeCPA содержит в себе различные внутренние CPA, в которых переходы выполняются параллельно. Таким образом, вычисляется абстракция сразу для нескольких типов анализа, что позволяет значительно увеличить точность.

Пусть имеется несколько различных видов анализа с раздельным рассмотрением потоков: $\Delta_1, \ldots, \Delta_n$. Композиция различных видов анализа может быть представлена, как отдельный анализ $\mathscr{C} = (D_\mathscr{C}, \Pi_\mathscr{C}, \rightsquigarrow_\mathscr{C}, merge_\mathscr{C}, stop_\mathscr{C}, prec_\mathscr{C}, \cdot|_p, compose_\mathscr{C}, compatible_\mathscr{C})$. Каждый Δ_i реализует отдельный вид анализа. $\Delta_i = (D_{\Delta_i}, \Pi_{\Delta_i}, \rightsquigarrow_{\Delta_i}, merge^E_{\Delta_i}, stop^E_{\Delta_i}, prec^E_{\Delta_i}, \cdot|_{\Delta_i}, compose_{\Delta_i}, compatible_{\Delta_i})$. Тогда элементы $\mathscr{C}$ выражаются через элементы Δ_i следующим образом.

– $D_\mathscr{C} = D_{\Delta_1} \times \cdots \times D_{\Delta_n}$

Это означает, что $\mathcal{T} = C \times G \times T$, $\mathcal{E}_{\mathscr{C}} = \mathcal{E}_{\Delta_1} \times \cdots \times \mathcal{E}_{\Delta_n}$.

$$\bigoplus_{\mathscr{C}} \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right) = \bigoplus_{\Delta_1} \left(\begin{pmatrix} e_0^1 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1^1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m^1 \\ t_m \end{pmatrix} \right\} \right) \cap \ldots \cap \ldots \cap \bigoplus_{\Delta_n} \left(\begin{pmatrix} e_0^n \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1^n \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m^n \\ t_m \end{pmatrix} \right\} \right) \tag{2.24}$$

Из выполнимости условий 2.9, 2.10 для вложенных Δ_i следует выполнимость условий для $\mathscr{C}$, так как пересечение более широких множеств не может быть меньше, чем пересечение исходных множеств.

– $\Pi_{\mathscr{C}} = \Pi_{\Delta_1} \times \cdots \times \Pi_{\Delta_n}$
– Внутренние элементы Δ работают с графом потока управления, с двумя дополнительными операциями: [1]
    1. tc_{parent} представляет действие $thread\_create$ в родительском потоке, а
    2. tc_{child} представляет действие $thread\_create$ в дочернем потоке.

  Для отношения переходов в композиции $e \overset{R}{\rightsquigarrow}_{\mathscr{C}} e'$, где $e = (e_1, \ldots, e_n)$, $e_i = (s_i, q_i)$, $q_i \in G$, $q_i = (l, op, l')$.
    – если $op = thread\_create(l_v)$, то $\forall 0 \leqslant j \leqslant n$ рассматриваются два перехода в родительском и в дочернем потоке
        1. $(s_j, (l, tc_{parent}(l_v), l')) \overset{R}{\rightsquigarrow}_{\Delta_j} (e'_j, \pi')$,
        2. $(s_j, (l, tc_{child}(l_v), l')) \overset{R}{\rightsquigarrow}_{\Delta_j} (e'_j, \pi')$.
    – иначе, $e_j \overset{R}{\rightsquigarrow}_{\Delta_j} (e'_j, \pi')$.

  Комбинация полученных e_j в один переход CompositeCPA напрямую $e' = (e'_1, \ldots, e'_n)$ является корректным, однако недостаточно точным для использования на практике. Один из внутренних CPA может получить более точную абстракцию, но за счет декартова произведения с другими анализами, он потеряет эту информацию. В частности, это важно для определения следующей дуги. Один из внутренних CPA может определить, что следующий переход возможен только по одной-единственной дуге, а другой внутренний CPA, не имея этой информации, очертит более широкое

[1] Заметим, что в реализации CPAchecker уже присутствуют две дуги в ГПУ для каждого вызова функции: function summary и function entry. Поэтому вызов функции $thread\_create$ не требует изменений в ГПУ с точки зрения реализации.

результирующее множество возможных следующих дуг. В это ситуации было бы логичнее рассмотреть только одну дугу даже второму анализу. Для этого определяется cpa-оператор усиления: $\downarrow$: $E_{\Delta_1} \times \cdots \times E_{\Delta_n} \to E_{\mathscr{C}}$. Как видно из определения, этот cpa-оператор зависит от конкретных видов анализа (CPA). Предполагая, что внутренние CPA, используют явный вид переходов, описанный в 2.5.4, опишем общий вид этого cpa-оператора для усиления информации о дугах.

$$\begin{aligned}
&\forall e \in E_{\mathscr{C}}, e = (e_1, \ldots, e_n), \forall 1 \leqslant i \leqslant n : e_i = (s_i, q_i) \\
&\downarrow (e_1, \ldots, e_n) = \\
&\begin{cases}
((s_1, g), \ldots, (s_n, g)), \text{если } \exists g \in G : \|q_1\| \cap \cdots \cap \|q_n\| = \{g\} \\
\bot, \text{если } : \|q_1\| \cap \cdots \cap \|q_n\| = \varnothing \\
(e_1, \ldots, e_n), \text{иначе}
\end{cases}
\end{aligned} \tag{2.25}$$

Такое определение является самым примитивным cpa-оператором $\downarrow$, так как оно не учитывает возможности усиления ни абстрактных состояний, ни абстрактных дуг. Такой cpa-оператор передает информацию другим cpa-операторам только о том, что существует единственная конкретная дуга, соответствующая абстрактным дугам всех внутренних CPA. Тем не менее, возможны другие, более точные, реализации этого cpa-оператора. Таким образом, полученные выше e'_j переходы внутренних CPA комбинируются в один переход CompositeCPA с помощью cpa-оператора $\downarrow$: $e' = \downarrow (e'_1, \ldots, e'_n)$. Для краткости далее будем обозначать $\downarrow (e'_1, \ldots, e'_n) = \downarrow e'$. Основным требованием на cpa-оператор $\downarrow$ является то, что он не должен терять конкретные переходы:

$$\begin{aligned}
&\forall e_0, e_1, \ldots, e_n \in E_{\mathscr{C}}, t_0, \ldots, t_n \in T \\
&\bigoplus\nolimits_{\mathscr{C}} \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} apply_{\mathscr{C}}(e_1, e_0) \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} apply_{\mathscr{C}}(e_n, e_0) \\ t_n \end{pmatrix} \right\} \right) = \\
&\bigoplus\nolimits_{\mathscr{C}} \left(\begin{pmatrix} \downarrow e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} apply_{\mathscr{C}}(\downarrow e_1, \downarrow e_0) \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} apply_{\mathscr{C}}(\downarrow e_n, \downarrow e_0) \\ t_n \end{pmatrix} \right\} \right)
\end{aligned} \tag{2.26}$$

Доказательство того, что требование 2.26 выполнено для простого cpa-оператора $\downarrow$ (определение 2.25), приведено в приложении A.3.

Так как композитный анализ разбивает операцию $thread\_create$ на две дуги, требование 2.13 трансформируется в

$$\forall \tau, \tau' \in \mathcal{T}, R \subseteq E : \tau \longrightarrow \tau', \forall e_0, e_1, \ldots, e_n \in R : e_i = (s_i, q_i)$$

$$\left(\begin{array}{l} \exists t_0, t_1, \ldots, t_n \in T, t_i \neq t_j, t_0 = t \\ \tau \in \bigoplus_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right) \end{array} \right) \Rightarrow$$

$$\left[\begin{array}{ll} \vee q_0 & \neq thread\_create \wedge \\ & \exists e'_0, e'_1, \ldots, e'_n \in E, \forall 1 \leqslant i \leqslant n : e_i \overset{R}{\rightsquigarrow} e'_i : \\ & \exists 1 \leqslant k \leqslant n : t_k = t' \wedge \\ & \tau' \in \bigoplus_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\}) \\ \vee q_0 & = thread\_create \wedge \\ & \exists e'_0, e'_1, \ldots, e'_{n+1} \in E, \forall 0 \leqslant i \leqslant n : \\ & \left\{ \begin{array}{l} (s_0, (l, tc_{parent}, l')) \overset{R}{\rightsquigarrow} e'_0 \\ (s_0, (l, tc_{child}, l')) \overset{R}{\rightsquigarrow} e'_{n+1} \\ e_i \rightsquigarrow e'_i, \text{ если } i \neq 0 \end{array} \right. \\ & \exists 0 \leqslant k \leqslant n+1 : t_k = t' \wedge \\ & \tau' \in \bigoplus_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\}) \end{array} \right. \tag{2.27}$$

– Сра-оператор слияния $merge$ может определяться различными способами, в зависимости от требований к анализу.

Самый простой способ: использовать сра-оператор $merge$ каждого из внутренних видов анализа.

$merge_{\mathscr{C}}(e_1,e_2,\pi) = (merge_{\Delta_1}(e_1^1, e_2^1,\pi), \ldots, merge_{\Delta_n}(e_1^n, e_2^n,\pi))$. В этом случае объединение переходов каждого из внутренних вариантов анализа производится независимо друг от друга. Проверим условие 2.6. Воспользуемся тем, что внутренние CPA удовлетворяют условию 2.6, то есть $\forall 1 \leqslant i \leqslant n : e_2^i \sqsubseteq merge_{\Delta_i}(e_1^i, e_2^i,\pi)$. По определению решетки $\mathscr{C}$ это означает, что $(e_2^1, \ldots, e_2^n) = e_2 \sqsubseteq merge_{\mathscr{C}}(e_1,e_2,\pi)$.

Такой простой вариант объединения состояний не очень эффективен, например, состояния различных потоков не всегда имеет смысл обединять. Для этого возможно объединение состояний, если состояния некоторого вида анализа равны.

$$merge_{\mathscr{C}}(e_1,e_2,\pi) = \begin{cases} (e_1^1, merge_{\Delta_1}(e_1^2, e_2^2,\pi), \ldots, merge_{\Delta_n}(e_1^n, e_2^n,\pi)), \\ \qquad \text{если } e_1^1 = e_2^1 \\ e_2, \text{ иначе} \end{cases}$$

В таком примере состояния объединяются только при совпадении (равенстве) состояний первого анализа ($i = 1$). Очевидно, что такой вариант cpa-оператора $merge$ тоже удовлетворяет условию 2.6, так как для $i = 1 : e_2^1 \sqsubseteq e_2^1$.

Стоит отметить, что такой способ слияния не равносилен первому варианту при некотором cpa-операторе $merge_{\Delta_1}$. При первом варианте, если хотя бы один анализ объединил свои состояния, возникнет новое состояние CompositeCPA. Во втором варианте новое состояние возникнет только при некотором условии (в данном случае $e_1^1 = e_2^1$), что не может быть выражено в терминах cpa-операторов $merge_{\Delta_i}$ первого способа.

– $stop_{\mathscr{C}}(e,R,\pi) = \forall 0 \leqslant j \leqslant n : stop_{\Delta_j}(e_j, R_j, \pi)$, где $R_j = \{e_j \mid e \in R \wedge e = (\ldots, e_j, \ldots)\}$

  Проверим условие 2.7 для $\mathscr{C}$. По условию 2.7 для внутренних CPA выполнено $\forall 1 \leqslant i \leqslant n, \widehat{R_i} \subseteq E_i : [\![\widehat{R_i} \cup \{e_i\}]\!]_{TM} \subseteq [\![R_i \cup \widehat{R_i}]\!]_{TM}$. Рассмотрим множество $\widehat{R} = \widehat{R_1} \times \cdots \times \widehat{R_n} \subseteq E$. $[\![\widehat{R} \cup \{e\}]\!]_{TM} = [\![\widehat{R_1} \cup \{e_1\}]\!]_{TM} \cap \cdots \cap [\![\widehat{R_n} \cup \{e_n\}]\!]_{TM} \subseteq [\![R_1 \cup \widehat{R_1}]\!]_{TM} \cap \cdots \cap [\![R_n \cup \widehat{R_n}]\!]_{TM} = [\![R \cup \widehat{R}]\!]_{TM}$.

– $prec_{\mathscr{C}}(e,\pi) = (prec_{\Delta_1}(e_1, \pi_1), \ldots, prec_{\Delta_n}(e_n, \pi_n))$. Условие 2.8 выполнено для композитного анализа, если оно выполнено для каждого из внутренних видов анализа Δ_j, так как по определению $e \sqsubseteq e' \iff \forall j : e_j \sqsubseteq e'_j$.

– $compatible_{\mathscr{C}}(e_1, e_2) = compatible_{\Delta_1}(e_1^1, e_2^1) \wedge \cdots \wedge compatible_{\Delta_n}(e_1^n, e_2^n)$.

– $composite_{\mathscr{C}}(e_1, e_2) = (composite_{\Delta_1}(e_1^1, e_2^1), \ldots, composite_{\Delta_n}(e_1^n, e_2^n))$.

– $e|_{p\mathscr{C}} = (e_1|_{p\Delta_1}, \ldots, e_n|_{p\Delta_n})$.

Доказательство того, что требование 2.13 выполнено для $\mathscr{C}$, если требование 2.27 выполнены для всех его внутренних элементов, приведено в приложении А.4.

2.8. Простой анализ потоков

Определим анализ потоков, инвариантный к переходам по окружению, $\mathbb{T} = (D_T, \Pi_T, \rightsquigarrow_T, merge_T, stop_T, prec_T, compatible_T, composite_T, \cdot|_p)$, который будет отслеживать идентификаторы потоков.

Анализ потоков содержит те же ограничения, что и анализ, представленный в статье [68], и его применение ограничено на программы с фиксированным количеством создаваемых потоков. Данный анализ использует в качестве идентификаторов потока точку старта нового потока, то для $thread\_create(pc_\nu)$ всегда создается поток с идентификатором pc_ν. Так, применение анализа потоков ограничивается программами, в которых в каждый момент времени может существовать только один экземпляр каждого потока. Предполагаем, что программа имеет ограниченное количество потоков, определяемых точками создания этих потоков. Заметим, что остальные виды анализа не ограничены количеством создаваемых потоков.

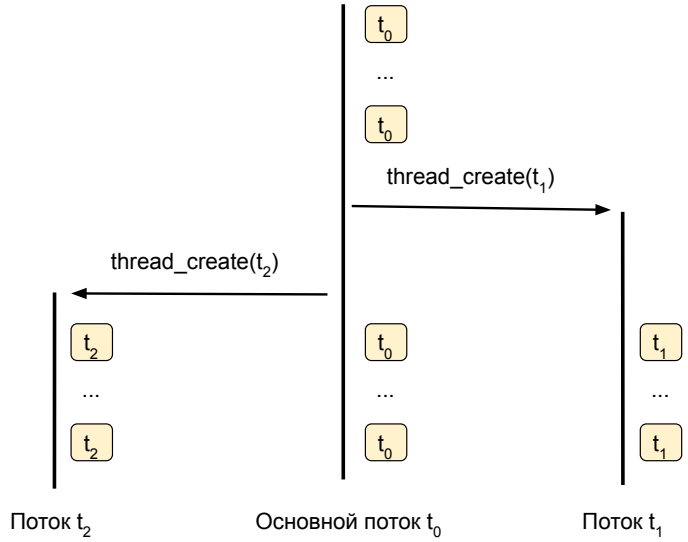

Рисунок 2.6 — Иллюстрация состояний ThreadCPA

На рисунке 2.6 приведена иллюстрация работы анализа потоков. На нем изображен основной поток t_0, который создает два дополнительных в процессе своей работы: t_1 и t_2. Состояния ThreadCPA содержат только информацию о том потоке, которому принадлежит данное состояние. Состояния, принадлежащие разным потокам, являются совместными. Очевидно, такой способ является очень неточным, так как не позволяет отличить, например, состояния до и после

создания потока. Более точные варианты анализа потока будут описаны далее, а пока формально дадим определение ThreadCPA.

Воспользуемся общим видом анализа с явным видом переходов, описанном в подразделе 2.5.4.

– Множество абстрактных состояний $E_T^E = T \cup \{\bot^E, \top^E\}$, $\bot^E \sqsubseteq t \sqsubseteq^E \top^E$ и $t \neq t' \Rightarrow t \not\sqsubseteq^E t'$ для всех элементов $t, t' \in T$ (что означает $\bot^E \sqcup^E t = t$, $\top^E \sqcup t = \top^E$, $t \sqcup^E t' = \top^E$ для всех элементов t,$t' \in T$, $t \neq t'$).

  В данном анализе требуется расширенный cpa-оператор проверки совместности:

$$\begin{gathered}\forall t_0, \ldots, t_j \in T : \\ check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n,t_n)\}) = \forall i \neq j : t_i \neq t_j \wedge s_i \sqsubseteq t_i\end{gathered}$$

  Cpa-оператор композиции для состояний строит множество таких конкретных состояний, которые лишь содержат требуемые идентификаторы потока, не накладывая больше никаких ограничений.

$$\begin{gathered}\forall t_1, \ldots, t_n \in T : \\ \bigoplus\nolimits_T^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right) = \\ \begin{cases} \{c \in C \mid dom(c) = \{t_0, \ldots, t_n\}\}, \\ \qquad \text{если } check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n,t_n)\}) \\ \varnothing, \text{ иначе} \end{cases}\end{gathered} \tag{2.28}$$

  Такое определение cpa-оператора $\bigoplus_T^S$ очевидно удовлетворяет условиям 2.17, 2.18, так как, фактически, результат не зависит от состояний s_i, которые влияют только на cpa-оператор $check_C$.

  Множество абстрактных дуг содержит только тождественную дугу, которая не меняет абстрактное состояние, и верхний и нижний элементы решетки: $\mathcal{G} = \{\bot_T^T, \varepsilon, \top_T^T\}$. Cpa-оператор конкретизации для дуг является тривиальным: $\|\bot_T^T\| = \varnothing$, $\|\varepsilon\| = \|\top_T^T\| = G$.

– Множество точности не используется и состоит из одного элемента: $\Pi_T = \{\{\varnothing\}\}$.

– Отношение переходов $\rightsquigarrow_T$ определяет переход $e \rightsquigarrow_T (e', \pi)$, $g = (\cdot, op, \cdot)$, если

– $op \neq tc_{child}$ и $e' = (t, \top)$, то есть текущий поток не меняется.
– $op = tc_{child}(l_\nu)$, тогда $e' = (l_\nu, \top)$, то есть текущий поток становится равным созданному.

Переход $\top \overset{\tau,g}{\rightsquigarrow}_T (\top, \pi)$ определен для всех $g \in G$, однако на практике переход будет усилен (определение 2.25) в CompositeCPA.

– Cpa-оператор слияния не объединяет абстрактные состояния: $merge_T(e, e', \pi) = e'$. Условие 2.6 очевидно выполнено.
– Cpa-оператор останова для абстрактных переходов проверяет наличие абстрактного перехода во множестве достижимых состояний: $stop_T(e, R, \pi) = (e \in R)$. Проверим условие 2.7. Возьмем $\widehat{R} \subseteq E_T$: $[\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup e \cup R]\!]_{TM} = [\![\widehat{R} \cup R]\!]_{TM}$.
– Точность переходов никогда не меняется: $prec_T(e, \pi, R) = (e, \pi)$. Условие 2.8 выполнено, так как $e \sqsubseteq e$.
– $\forall e_1, e_2 \in E_T, e_i = (s_i, q_i) : compatible_T(e_1,e_2) = s_1 \neq s_2$.
– Переходы являются *инвариантными к окружению*, то есть ни один поток не может изменить идентификатор другого потока. Это означает, что проекцией является тождественный переход: $(s, q)|_p = (s, \varepsilon)$.
– Cpa-оператор $compose$ является обычным 2.21.

Доказательство того, что определенные таким образом cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27, приведено в приложении A.5.

2.9. Анализ потоков с эффектами окружения

Определим анализ потоков с эффектами окружения. Этот CPA будет строить полный граф вызовов потоков, что позволит более точно определять совместные состояния, чем в описанном ранее простом анализе потоков (раздел 2.8). Анализ потоков содержит те же ограничения, что и описанный ранее ThreadCPA.

На рисунке 2.7 приведена иллюстрация работы анализа потоков с эффектами окружения. На нем изображен основной поток t_0, который создает два дополнительных в процессе своей работы: t_1 и t_2. Для данного варианта ThreadCPA абстрактные состояния содержат не только информацию о текущем потоке, но и обо всем множестве активных потоков. Состояния являются совместными только если они принадлежат разным потокам и их множество активных потоков сов-

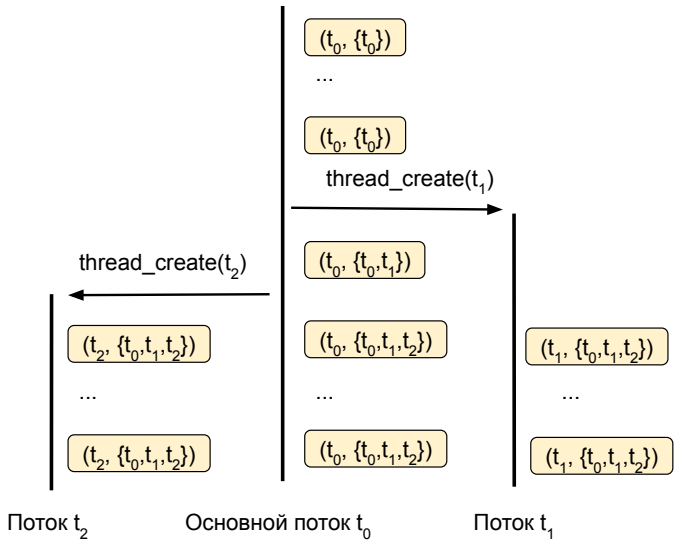

Рисунок 2.7 — Иллюстрация состояний ThreadCPA

падает. Такой вариант анализа потоков является более точным, чем предыдущий вариант, так как он позволяет отсеивать состояния до и после создания потока. Однако, информация обо всех потоках может быть получена только с помощью переходов в окружении.

Перейдем к формальному описанию ThreadCPA: $\mathbb{T}_1 = (D_{T_1}, \Pi_{T_1}, \rightsquigarrow_{T_1}, merge_{T_1}, stop_{T_1}, prec_{T_1}, compatible_{T_1}, composite_{T_1}, \cdot|_p)$. Для этого воспользуемся общим видом анализа с явным видом переходов, описанном в подразделе 2.5.4.

– Множество абстрактных состояний $E_T^S = T \times 2^T \cup \{\bot_T^S, \top_T^S\}$.
$\bot^E \sqsubseteq s \sqsubseteq^E \top^E$ и $s \neq s' \Rightarrow s \not\sqsubseteq_T^S s'$ для всех элементов $s, s' \in E_T^S$ (что означает $\bot_T^S \sqcup_T^S s = s$, $\top_T^S \sqcup t = \top_T^S$, $t \sqcup_T^S t' = \top_T^S$ для всех элементов s,$s' \in E_T^S$, $s \neq s'$).
В данном анализе требуется расширенный cpa-оператор проверки совместности:

$$\forall t_0, \ldots, t_n \in T, s_i = (\tilde{t}_i, C_i) :$$
$$check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n, t_n)\}) =$$
$$\forall 0 \leqslant i \leqslant n : C_i = \{t_0, \ldots, t_n\} \wedge \tilde{t}_i = t_i$$

Этот cpa-оператор проверяет, что все частичные состояния содержат одинаковое множество созданных потоков C, каждый поток $\tilde{t}_i$ является созданным с точки зрения других потоков и среди всех состояний ни одна пара не принадлежит одному и тому же потоку.
Cpa-оператор композиции для состояний

$\forall s_1, \ldots, s_n \in T$:

$$\bigoplus_T^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right) =$$

$$\begin{cases} \{c \in C \mid dom(c) = \{t_0, \ldots, t_n\} = C_i\}, \\ \qquad \text{если } check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n, t_n)\}) \\ \varnothing, \text{ иначе} \end{cases} \tag{2.29}$$

Такое определение сра-оператора $\bigoplus$ очевидно удовлетворяет условиям 2.17, 2.18, так как используется плоская решетка.

Множество абстрактных дуг содержит тождественную дугу, которая не меняет абстрактное состояние, абстрактные дуги, соответсвующие созданию потока, а также верхний и нижний элементы решетки: $\mathcal{G} = \{\bot_T^T, \varepsilon, \top_T^T\} \cup \{create\} \times T$. Сра-оператор конкретизации для дуг является тривиальным: $\|\bot_T^T\| = \varnothing$, $\|\top_T^T\| = G$, $\|\varepsilon\| = G \setminus \{thread\_create(\cdot)\}$, $\|(create, t)\| = thread\_create(t)$.

– Множество точности не используется и состоит из одного элемента: $\Pi_T = \{\{\varnothing\}\}$.
– Отношение переходов $\rightsquigarrow_T$ определяет переход $e \rightsquigarrow_T (e', \pi)$, где $e = (s, q)$ и $e' = (s', \top)$, если
  – $q \in G, q = (\cdot, op, \cdot)$
    * $op \neq tc_{child}$, $op \neq tc_{parent}$ и $s' = s$, то есть состояние не меняется.
    * $op = tc_{parent}(l_\nu)$ $s' = (t, C \cup \{l_\nu\})$, то есть текущий поток не меняется, но новый поток добавляется во множество созданных потоков.
    * $op = tc_{child}(l_\nu)$ $s' = (l_\nu, C \cup \{l_\nu\})$, то есть новый поток становится текущим в состоянии и добавляется во множество созданных потоков.
  – $q = (create, t)$, тогда $s' = (t, C \cup \{t\})$.

  Переход $\top \overset{\tau, g}{\rightsquigarrow}_T (\top, \pi)$ определен для всех $g \in G$, однако на практике переход будет усилен (определение 2.25) в CompositeCPA.
– Сра-оператор слияния не объединяет абстрактные состояния: $merge_T(e, e', \pi) = e'$. Условие 2.6 очевидно выполнено.
– Сра-оператор останова для абстрактных переходов проверяет наличие абстрактного перехода во множестве достижимых состояний:

$stop_T(e, R, \pi) = (e \in R)$. Проверим условие 2.7. Возьмем $\widehat{R} \subseteq E_T$: $[\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup e \cup R]\!]_{TM} = [\![\widehat{R} \cup R]\!]_{TM}$.

– Точность переходов никогда не меняется: $prec_T(e, \pi, R) = (e, \pi)$. Условие 2.8 выполнено, так как $e \sqsubseteq e$.
– $\forall e_1, e_2 \in E_T, e_i = (s_i, q_i), s_i = (t_i, C_i) : compatible_T(e_1,e_2) = t_1 \in C_2 \wedge t_2 \in C_1 \wedge t_1 \neq t_2 \wedge C_1 = C_2$.
– Проекцией перехода является тождественный переход, кроме перехода $thread\_create$: $(s, q)|_p = \begin{cases} (s, (create, t)), \text{если} q \in G, q = (\cdot, tc_{child}(t), \cdot) \vee q \\ (s, \varepsilon), \text{иначе} \end{cases}$
– Cpa-оператор $compose$ является обычным 2.21.

Доказательство того, что определенные таким образом cpa-операторы $transfer$ и $apply$ удовлетворяют условию, приведено в разделе Приложения A.10.

2.10. Расширенный анализ потоков, инвариантный к переходам в окружении

Определим анализ расширенный анализ потоков. Этот CPA будет строить некоторое множество создаваемых потоков таким образом, что позволит более точно определять совместные состояния, чем в описанном ранее простом анализе потоков (раздел 2.8). Однако, он не будет использовать переходы в окружении, что позволит ему оставаться инвариантным к переходам в окружении. Анализ потоков содержит те же ограничения, что и описанный ранее ThreadCPA.

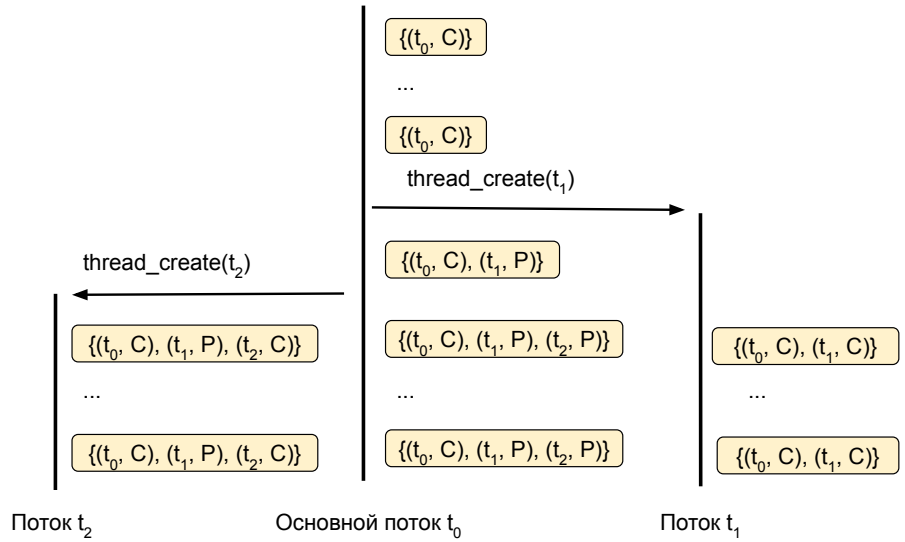

Рисунок 2.8 — Иллюстрация состояний ThreadCPA

На рисунке 2.8 приведена иллюстрация работы анализа потоков с эффектами окружения. На нем изображен основной поток t_0, который создает два дополнительных в процессе своей работы: t_1 и t_2. Для данного варианта ThreadCPA абстрактные состояния содержат информацию о том, какие потоки были созданы на пути к этому состоянию. Кроме того, хранится информация о том, находимся ли мы внутри созданного потока (C – child), или остались внутри родительского (P – parent). Несмотря на то, что множество созданных потоков может быть неполным (поток t_1 не имеет инвормации про поток t_2, так как был создан раньше), такой анализ позволяет обеспечить такую же точность как и вариант с ThreadCPA с переходами в окружении. Основная идея проверки совместности состоит в том, что анализ, по сути, проверяет, может ли быть у двух абстрактных состояний одна общая точка ветвления (создание потока). В предположении, что начальный поток был строго один, это позволяет обеспечить требуемую точность.

Перейдем к формальному описанию расширенного варианта ThreadCPA: $\mathbb{T}_2 = (D_{T_2}, \Pi_{T_2}, \rightsquigarrow_{T_2}, merge_{T_2}, stop_{T_2}, prec_{T_2}, compatible_{T_2}, composite_{T_2}, \cdot|_p)$. Для этого воспользуемся общим видом анализа с явным видом переходов, описанном в подразделе 2.5.4.

– Множество абстрактных состояний $E_T^S = 2^{T \times \{Parent, Child\}}$.
$\bot_T^S = \varnothing$, $\top_T^S = T \times \{Parent, Child\}$. $\forall s,s' \in E_T^S, s \sqsubseteq s' \iff s \subseteq s'$.
$\forall s,s' \in E_T^S, s \sqcup s' = s \cup s'$.
В данном анализе требуется расширенный cpa-оператор проверки совместности:

$\forall t_0, \ldots, t_n \in T, s_0, \ldots, s_n \in E_T^S :$
$check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n,t_n)\}) =$
$\forall 0 \leqslant i, j \leqslant n :$
$\exists t \in T : (t, Parent) \in s_i \wedge (t, Child) \in s_j \vee (t, Parent) \in s_j \wedge (t, Child) \in s_i$

Cpa-оператор композиции для состояний

$\forall s_1, \ldots, s_n \in T$:

$$\bigoplus_T^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right) =$$

$$\begin{cases} \{c \in C \mid dom(c) = \{t_0, \ldots, t_n\}\}, \\ \qquad \text{если } check_C((s_0, t_0), \{(s_1, t_1), \ldots, (s_n, t_n)\}) \\ \varnothing, \text{ иначе} \end{cases} \tag{2.30}$$

Такое определение cpa-оператора $\bigoplus$ очевидно удовлетворяет условиям 2.17, 2.18, так как информация из состояния используется только для определения совместности.

Множество абстрактных дуг содержит только тождественную дугу, которая не меняет абстрактное состояние, и верхний и нижний элементы решетки: $\mathcal{G} = \{\bot_T^T, \varepsilon, \top_T^T\}$. Cpa-оператор конкретизации для дуг является тривиальным: $\|\bot_T^T\| = \varnothing$, $\|\varepsilon\| = \|\top_T^T\| = G$.

– Множество точности не используется и состоит из одного элемента: $\Pi_T = \{\{\varnothing\}\}$.
– Отношение переходов $\rightsquigarrow_T$ определяет переход $e \rightsquigarrow_T (e', \pi)$, $g = (\cdot, op, \cdot)$ и $e' = (s', \top)$, если
  – $op \neq tc_{child}$, $op \neq tc_{parent}$ и $s' = s$, то есть состояние не меняется.
  – $op = tc_{child}(l_\nu)$, тогда $s' = s \cup (l_\nu, Child)$.
  – $op = tc_{parent}(l_\nu)$, тогда $s' = s \cup (l_\nu, Parent)$.

  Переход $\top \overset{\tau, g}{\rightsquigarrow}_T (\top, \pi)$ определен для всех $g \in G$, однако на практике переход будет усилен (определение 2.25) в CompositeCPA.
– Cpa-оператор слияния не объединяет абстрактные состояния: $merge_T(e, e', \pi) = e'$. Условие 2.6 очевидно выполнено.
– Cpa-оператор останова для абстрактных переходов проверяет наличие абстрактного перехода во множестве достижимых состояний: $stop_T(e, R, \pi) = (e \in R)$. Проверим условие 2.7. Возьмем $\widehat{R} \subseteq E_T$: $[\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup e \cup R]\!]_{TM} = [\![\widehat{R} \cup R]\!]_{TM}$.
– Точность переходов никогда не меняется: $prec_T(e, \pi, R) = (e, \pi)$. Условие 2.8 выполнено, так как $e \sqsubseteq e$.
– $\forall e_1, e_2 \in E_T, e_i = (s_i, q_i)$: $compatible_T(e_1, e_2) = check_C(\{e_1, e_2\})$.
– Переходы являются *инвариантными к окружению*, то есть ни один поток не может изменить идентификатор другого потока. Это означает, что проекцией является тождественный переход: $(s, q)|_p = (s, \varepsilon)$.

– Cpa-оператор $compose$ является обычным 2.21.

Доказательство того, что определенные таким образом cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27, приведено в приложении A.11.

2.11. Анализ точек программы

Определим анализ точек программы (LocationCPA), инвариантный к переходам по окружению, $\mathbb{L} = (D_L, \Pi_L, \rightsquigarrow_L, merge_L, stop_L, prec_L, compatible_L, \cdot|_p, compose_L)$, который отвечает за синтаксическую достижимость точек программы. Классический вариант этого анализа описан в [82]. Расширим его компоненты для возможности его применения в анализе с раздельным рассмотрением потоков.

```
1: int main {
2:   int y = 3;
3:  int x = y + 1;
4:   ...
}
```

Рисунок 2.9 — Пример исходного кода

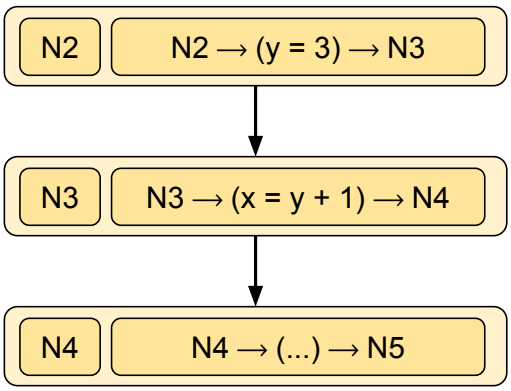

Рисунок 2.10 — Пример переходов LocationCPA

На рисунке 2.10 представлен пример переходов для LocationCPA. Переходы содержат в себе абстрактное состояние и абстрактную дугу. Абстрактное состояние обычно соответсвует узлу ГПУ, который можно условно считать соответсвующим строке исходного кода. Но в общем случае состояние LocationCPA может соответствовать некоторому множеству узлов ГПУ. Отличительной особенностью данного CPA является то, что именно он имеет полную информацию о возможных следующих дугах, поэтому при применении cpa-оператора $strengthen$ в CompositeCPA дуга LocationCPA будет браться за основу. С точки зрения подхода с раздельным рассмотрением потоков данный CPA не несет никакой нагрузки, так как переходы в различных потоках могут выполняться параллельно независимо от того, в какой точке программы находятся эти потоки. Таким образом, LocationCPA

является важным служебным CPA, который обеспечивает обход ГПУ, но не влияет прямо на точность анализа.

Теперь дадим формальное определение cpa-операторам LocationCPA. Воспользуемся общим видом анализа с явным видом переходов, описанном в подразделе 2.5.4.

1. Множество абстрактных состояний E_L^S состоит из абстрактных точек программы, которые отображаются на конкретные узлы ГПУ с помощью функции $loc : E_L^S \to 2^L$. $\top_L^S$ означает, что анализ не может определить конкретную точку программы, формально, $loc(\top_L^S) = L$. В общем случае анализ может работать с абстрактными точками программы, которые выражают несколько конкретных точек программы, но такой вариант является слишком общим и пока не нашел применения на практике, поэтому дальше мы будем рассматривать упрощенную вариацию этого анализа, в котором рассматриваются только одиночные точки программы: $\forall s \in E_L^S : s = \top_L^S \vee s = \bot_L^S \vee loc(s) = l \in L$. Определенная таким образом $\mathcal{E}_L^S$ является плоской решеткой, что означает, что две различные точки программы являются несравнимыми: $l \neq l' \Rightarrow l \not\sqsubseteq^E l'$ для всех элементов $l, l' \in L$ (отсюда следует $\bot^E \sqcup^E l = l$, $\top^E \sqcup l = \top^E$, $l \sqcup^E l' = \top^E$ для всех элементов l,$l' \in L$, $l \neq l'$), и
   В данном анализе cpa-оператор является тривиальным, так как потоки могут находиться в любых точках программы независимо друг от друга:

$$\forall s_0, \ldots, s_j \in T : check_C(\{s_0, \ldots, s_n,\}) \equiv true$$

   Cpa-оператор композиции для состояний

$$\begin{gathered}\forall s_1, \ldots, s_n \in E_L^S : \\ \bigoplus\nolimits_L^S \left(\begin{pmatrix} s_1 \\ t_1 \end{pmatrix} , \ldots , \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right) = \\ \left\{ c \in C \;\middle|\; \begin{array}{c} \forall 1 \leqslant i \leqslant n \\ c_{pc}(t_i) = l_i \in loc(s_i) \\ dom(c_l) = \{t_1, \ldots, t_n\} \end{array} \right\}\end{gathered} \tag{2.31}$$

   Такое определение cpa-оператора $\bigoplus_L^S$ очевидно удовлетворяет условиям 2.17, 2.18.

Множество абстрактных дуг содержит только тождественную дугу, которая не меняет абстрактное состояние, и верхний и нижний элементы решетки: $\mathcal{G} = \{\bot_T^T, \varepsilon, \top_T^T\}$. Сра-оператор конкретизации для дуг является тривиальным: $\|\bot_T^T\| = \varnothing$, $\|\varepsilon\| = \|\top_T^T\| = G$. Состояния этого анализа *инвариантны к окружению*, то есть ни один поток не может изменить точку в программе, на которой находится другой поток.

2. Множество точности содержит только один элемент $\Pi_L = \{\{\varnothing\}\}$, так как не подразумевается применение абстракции.
3. Отношение переходов $\rightsquigarrow_L$ содержит переход $e \overset{R}{\rightsquigarrow}_L e'$, где $e = (s, q)$, если
   - $q \in G, q = (l_1, op, l_2)$, $l_1 \in loc(s)$ и
     - $op \neq tc_{child}$ и $loc(s') = l_2$ (следующее состояние в ГПУ без учета семантики операции op), $g' = (l_2, op, \cdot)$ - все операции в ГПУ, которые соответствуют заданной точке программы.
     - $op = tc_{child}(l_\nu)$ и $l_2 = l_\nu$, $g' = (l_\nu, op, \cdot)$ - все операции в ГПУ, которые соответствуют заданной точке программы.
   - $q = \varepsilon$, $s' = s$, $g' = (l_2, op, \cdot)$ - все операции в ГПУ, которые соответствуют заданной точке программы.
4. Сра-оператор слияния не объединяет элементы: $merge_L(e, e', \pi) = e'$. Очевидно, он удовлетворяет условию 2.6, так как $e' \sqsubseteq e'$.
5. Сра-оператор останова рассматривает состояния индивидуально: $stop_L(e, R, \pi) = (e \in R)$. Очевидно, он удовлетворяет условию 2.7, так как

$$\begin{aligned} &\forall e \in E, R \subseteq E, \pi \in \Pi : \\ &(e \in R) \implies \forall \widehat{R} \subseteq E : \{e\} \cup \widehat{R} \subseteq R \cup \widehat{R} \implies \\ &\implies (eq.2.1) \forall \widehat{R} \subseteq E : [\![\{e\} \cup \widehat{R}]\!]_{TM} \subseteq [\![R' \cup \widehat{R}]\!] \end{aligned} \tag{2.32}$$

6. Точность перехода никогда не изменяется: $prec_L(e, \pi, R) = (e, \pi)$. Очевидно, он удовлетворяет условию 2.8, так как $e \sqsubseteq e$.
7. Переходы являются *инвариантными к окружению*, то есть ни один поток не может изменить положение другого потока. Это означает, что проекцией является тождественный переход: $(s, q)|_p = (s, \varepsilon)$.
8. Сра-оператор $compose$ является обычным 2.21.
9. $\forall e_1, e_2 \in E_L, e_i = (s_i, q_i) : compatible_L(e_1, e_2) \equiv true$.

Доказательство того, что определенные таким образом cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27, приведено в приложении A.6.

2.12. Анализ предикатов

В этом разделе описан известный анализ предикатов (Predicate Analysis) [83] с абстрактными переходами. Переходы анализа предикатов состоят из двух частей: абстрактного состояния и абстрактной дуги, которая может быть выражена либо обычной CFA дугой, либо логической формулой, кодирующей выполняемую операцию. Кроме того, локальные переменные в этих формулах должны быть переименованы для избежания коллизии имен для разных потоков.

```
1: int main {
2:   int y = 3;
3:  int x = y + 1;
4:   if (x > 0) {
5:     ...
6:   }
}
```

Рисунок 2.11 — Пример исходного кода

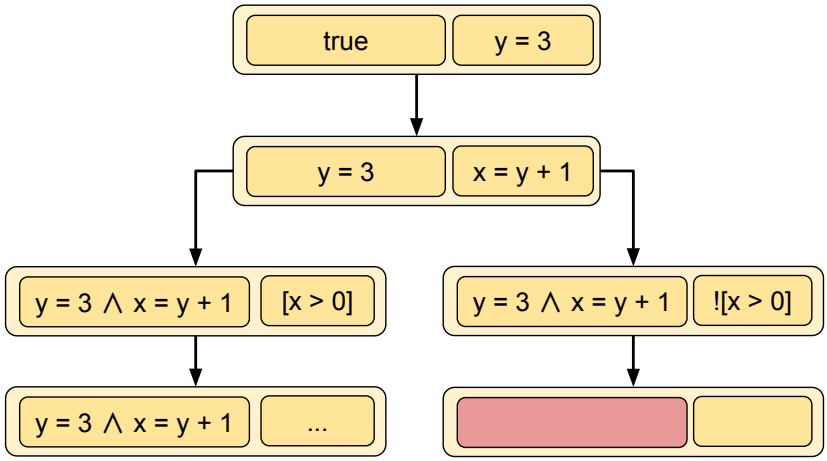

Рисунок 2.12 — Пример переходов PredicateCPA

Рисунок 2.12 демонстрирует пример переходов анализа предикатов. В абстрактном состоянии накапливается логическая формула на основе абстрактных дуг. Эти формулы проверяются на разрешимость, и если оказывается, что построенная формула неразрешима, это означает, что данное состояние недостижимо при реальном выполнении. Данный рисунок демонстрирует лишь общий вид анализа предикатов, и на нем не отображены переходы в окружении, построение предикатной абстракции и т.п.

Перейдем к формальному описанию PredicateCPA. Пусть $\mathscr{P}$ – это множество формул над переменными программы в теории без кванторов $\mathscr{T}$. Пусть $\mathcal{P} \subseteq \mathscr{P}$ – это множество предикатов. Формула φ является логической комбинацией предикатов из $\mathscr{P}$.

Пусть $v : X \to \mathbb{Z}$ является отображением из переменной в ее значение. Определим $v \models \varphi$, где v называется моделью φ.

Определим переименование переменных $\theta : X \to X'$, которое применимо к формулам $\theta(\varphi)$ и их моделям $\theta(v)$. Обозначим

$$\theta_{X,i} = \begin{cases} x \mapsto x\#i, & \text{если } x \in X \\ x \mapsto x, & \text{иначе.} \end{cases}$$

и

$$\theta^{-1}_{X,i} = \begin{cases} x\#i \mapsto x, & \text{если } x \in X \\ x \mapsto x, & \text{иначе.} \end{cases}$$

Определим $(\varphi)^\pi$ – логическую предикатную абстракцию формулы φ.

Определим $SP_{op}(\varphi)$ – сильнейшее постусловие φ и операции op. Для сра-оператора $assign : SP_{x=exp}(\varphi) = \varphi[x \to \hat{x}] \wedge (x = exp)$. Для сра-оператора $assume : SP_{[cond]}(\varphi) = \varphi \wedge cond$. Для остальных сра-операторов $SP_{op}(\varphi) = \varphi$.

Определим анализ предикатов (Predicate Analysis) $\mathbb{P} = (D_P, \Pi_P, \rightsquigarrow_P, merge_P, stop_P, prec_P, compatible_P, \cdot|_p, compose_P)$, который отслеживает значение предикатов над переменными программы.

Воспользуемся общим видом анализа с явным видом переходов, описанном в подразделе 2.5.4.

1. Множество абстрактных состояний $E^S_P = \mathscr{P}$, таким образом, состояние является формулой первого порядка. Верхний элемент решетки является тождественно истинной формулой $\top^E = true$, а нижний элемент – тождественно ложной $\bot^E = false$. Частичный порядок $\sqsubseteq^E \subseteq E \times E$ определяется как $e \sqsubseteq^E e' \Leftrightarrow e \implies e'$. Объединение элементов $\sqcup^E : E \times E \to E$ определяет ближайший верхний по решетке элемент в соответствии с частичным порядком.

$$\begin{aligned} &\forall s_1, \ldots, s_n \in E^S_P \\ &C_{check}(\{s_1, \ldots, s_n\}) = \\ &\exists v : v \models \theta_{X^{local},1}(s_1) \wedge \cdots \wedge \theta_{X^{local},n}(s_n) \end{aligned} \tag{2.33}$$

Сра-оператор C_{check} проверяет, имеют ли частичные состояния совместными, то есть, имеющими общую глобальную часть (X^{global}). Локальные переменные каждого состояния переименовываются, чтобы избежать путаницы с одинаковыми переменными разных потоков.

Для совместных состояний $s_1, \ldots, s_n \in E^S_P$ и для каждого решения $v \models \theta_{X^{local},1}(s_1) \wedge \cdots \wedge \theta_{X^{local},n}(s_n)$ можно определить функцию $\hat{v}_g =$

$v_{|X^{global}}$, представляющую общую глобальную часть состояния и функции $\hat{v}_1 = \theta^{-1}_{X^{local},1}(v)_{|X^{local}}, \ldots, \hat{v}_n = \theta^{-1}_{X^{local},n}(v)_{|X^{local}}$, представляющие локальные части.

Cpa-оператор композиции для состояний

$$\begin{gathered}\forall s_0, \ldots, s_n \in E^S_P, t_0, \ldots, t_n \in T, t_i \neq t_j : \\ \bigoplus\nolimits^S_P \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right) = \\ \begin{cases} \left\{ c \in C \;\middle|\; \begin{array}{c} \forall 0 \leqslant i \leqslant n \\ c_l(t_i) = \hat{v}_i, \\ c_g = \hat{v}_g \end{array} \right\}, \text{if } C_{check}(\{s_0, \ldots, s_n\}) \\ \varnothing, \text{otherwise} \end{cases}\end{gathered} \tag{2.34}$$

Абстрактная дуга $q \in E^T_P$ – это действие, которое может быть выражено или формулой, или обычной CFA дугой: $E^T_P = G \cup \mathcal{G}$. Где $\mathcal{G} = E^S_P$, то есть эффект окружения - это логическая формула, описывающая изменение состояния. Однако, функция частичного порядка $\sqsubseteq^T_P$ определена совершенно по-другому: $\forall q_1, q_2 \in \mathcal{G} : q_1 \sqsubseteq^T_P q_2 = (\forall \varphi \in E^S_P, q_1 \wedge \varphi \implies q_2 \wedge \varphi)$. Дело в том, что тождественный эффект окружения $\varphi \equiv true$, который был бы верхним элементом в прежней решетке, означает, что никаких изменений глобальных переменных не произошло. Соответственно, верхним элементом решетки абстрактных дуг должен стать элемент, означающий, что изменились значения всех переменных неопределенным образом, то есть, после применения которого результирующая формула будет $\varphi' \equiv true$. Отсюда такая необходимость изменения функции частичного порядка.

2. Множество точности $\Pi_P = 2^{\mathscr{P}}$ отвечает за точность абстрактного состояния и содержит в себе множество предикатов.
3. Отношение переходов $e \overset{R}{\rightsquigarrow}_P e'$, $e = (s, g)$, $e' = (s', \top_P)$. Так как анализ предикатов не отслеживает следующие релевантные дуги, он возвращает все возможные.

   Для $g \in G$ существует переход с $g = (\cdot, op, \cdot)$, если $s' = SP_{op}(s)$.

   Для $g = \varphi \in \mathscr{P}$, $s' = s[x \to \hat{x}] \wedge \varphi$. То есть, по сути, применение эффекта окружения к состоянию эквивалентно применению $SP_{op}(s)$ для

присваивания при том исключении, что операция op уже задана логической формулой.

4. Сра-оператор объединения merge может иметь несколько модификаций, например, $merge_{Join}$ объединяет обе части перехода:

$$\forall e, e' \in E, \pi \in \Pi_P, e = (s,g) :$$
$$merge_{Join}(e, e', \pi) = \begin{cases} (s \vee s', g), \text{if } g = g', g \in G \\ e', \text{if } g \in G \wedge g' \in \mathscr{P} \vee g \in \mathscr{P} \wedge g' \in G \\ (s \vee s', g \vee g'), \text{if } g, g' \in \mathscr{P} \end{cases} \tag{2.35}$$

Сра-оператор $merge_{Eq}$ объединяет только абстрактные дуги при равных (или покрытых) состояниях.

Сра-оператор $merge_{Sep}$ не объединяет элементы: $merge_{Sep}(e, e', \pi) = e'$.

Требование 2.6 для всех модификаций очевидно выполнено, так как $e \sqsubseteq e'$ по условиям сра-оператора $\sqcup_P$ решетки.

5. Сра-оператор останова проверяет, покрыт ли переход e другим переходом из множества достижимых состояний: $stop_P(e, R, \pi) = \exists e' \in R : (e \sqsubseteq e')$.

   Проверим, что требование 2.7 выполнено.

$$\forall e \in E, R, \widehat{R} \subseteq E, \pi \in \Pi :$$
$$e \sqsubseteq e' \implies (eq.\ 2.2) \implies [\![\widehat{R} \cup R \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R \cup e']\!]_{TM} \implies$$
$$\implies (eq.\ 2.1) \implies [\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R]\!]_{TM}$$

6. Функция настройки точности $prec_P$ вычисляет предикатную абстракцию над предикатами в точности $prec_P(e, \pi, R) = e^\pi = (s^\pi, q)$. Условие 2.8 выполнено, так как $e \sqsubseteq e^\pi$.
7. Определим сра-оператор $compatible_P$.

$$\forall e_1, e_2 \in E_P, e = (s, q)$$
$$compatible_P(e_1, e_2) = \exists v : v \models s_1 \wedge s_2 \tag{2.36}$$

8. Сра-оператор проекции строит логическую формулу по выражению также, как это делается при обычном переходе, но переименовывает все встречающиеся в ней локальные переменные для избежания проблем с несколькими одинаковыми потоками.

$$\forall e \in E_P, e = (s,q) :$$
$$e|_p = \begin{cases} e, \text{ if } q \notin G \\ (\theta_{X^{local},env}(s), \theta_{X^{local},env}(SP_{op}(true))), \text{ otherwise} \end{cases} \quad (2.37)$$

9. Cpa-оператор *compose* является обычным 2.21.

Доказательство того, что определение отношения переходов удовлетворяет условию 2.27, приведено в разделе Приложения А.7.

2.13. Анализ примитивов синхронизации

Определим анализ примитивов синхронизации (Lock Analysis) $\mathbb{S} = ((D_S, \Pi_S, \rightsquigarrow_S, merge_S, stop_S, prec_S^E, compatible_S, \cdot|_p, compose_S)$, который отслеживает множество захваченных блокировок (переменных синхронизации) для каждого потока.

```
int g = 0;
int main {
  lock();
  g = g + 1;
  unlock();
  ...
}
```

Рисунок 2.13 — Пример исходного кода

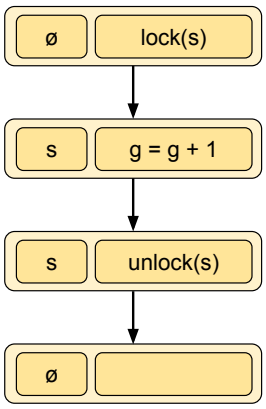

Рисунок 2.14 — Пример переходов LockCPA

Рисунок 2.14 демонстрирует пример переходов анализа примитивов синхронизации. В абстрактном состоянии накапливается информация о захваченных блокировках.

Перейдем к формальному описанию LockCPA. Для этого снова воспользуемся явным видом перехода, описанным в разделе 2.5.4. Он состоит из следующих компонент.

1. Множество абстрактных состояний $E_S^S = 2^S \cup \{\bot_S^S\}$ состоит из множества всех подмножеств всех переменных синхронизации. При этом

$\top^S_S = \varnothing$. $s \subseteq s' \Rightarrow s \sqsupseteq^S_S s'$ для всех элементов $s, s' \subseteq S$ (откуда следует $\bot^E \sqcup^E s = s$, $\top^E \sqcup s = \top^E$, $s \sqcup^E s' = s \cap s'$ для всех элементов s,$s' \subseteq S$, $s \neq s'$).

В данном анализе cpa-оператор является тривиальным, так как потоки могут находиться в любых точках программы независимо друг от друга: $\forall s_0, \ldots, s_n \in E^S_S : C_{Check}(\{s_0, \ldots, s_n\}) = \forall 0 \leqslant i,j \leqslant n : (s_i \neq \bot^S_S \wedge s_j \neq \bot^S_S \wedge s_i \cap s_j = \varnothing$.

$$\forall s_1, \ldots, s_n \in E^S_S :$$
$$\bigoplus{}^S_S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s'_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) =$$
$$= \begin{cases} \{c \in C \mid \hat{s} \in s_i \implies c_s(\hat{s}) = t_i\}, \text{ если } C_{Check}(\{s_0, \ldots, s_n\}) \\ \varnothing, \text{ иначе} \end{cases}$$

Такое определение cpa-оператора $\bigoplus^S_S$ очевидно удовлетворяет условиям 2.17, 2.18, так как уменьшение количества переменных блокировок в состоянии только ослабляют условия, а значит, позволяют получить большее количество конкретных состояний.

Множество абстрактных дуг содержит только тождественную дугу, которая не меняет абстрактное состояние, и верхний и нижний элементы решетки: $\mathcal{G} = \{\bot^T_S, \varepsilon, \top^T_S\}$. Cpa-оператор конкретизации для дуг является тривиальным: $\|\bot^T_S\| = \varnothing$, $\|\varepsilon\| = \|\top^T_S\| = G$. Состояния этого анализа *инвариантны к окружению*, то есть ни один поток не может изменить точку в программе, на которой находится другой поток.

2. Множество точности содержит только один элемент $\Pi_S = \{\{\varnothing\}\}$, так как не подразумевается применение абстракции.
3. Отношение переходов $\rightsquigarrow_S$ содержит переход $e \overset{R}{\rightsquigarrow}_S e'$, $e = (s,q)$, $e' = (s', \top^T_S)$, если
   - $q \in G$, $q = (\cdot,op,\cdot')$ и
     - $op = acquire(\hat{s})$ и $\hat{s} \notin s \wedge s' = s \cup \{\hat{s}\}$,
     - $op = release(\hat{s})$ и $s' = s \setminus \{\hat{s}\}$,
     - $op = tc_{child}(l_\nu)$ и $s' = \varnothing$.
     - $op \neq acquire$ и $op \neq release$ и $op \neq tc_{child}$ и $s' = s$.
   - $q = \varepsilon$ и $s' = s$.

4. Cpa-оператор слияния не объединяет абстрактные состояния: $merge_S(e, e', \pi) = e'$. Условие 2.6 очевидно выполнено.
5. Cpa-оператор останова проверяет, существует ли состояние, которое содержит меньше захваченных блокировок: $stop_S(e, R, \pi) = (\exists e' \in R \wedge e \sqsubseteq e')$. Проверим, что требование 2.7 выполнено.

   Доказательство.

   $$\forall e \in E, R, \widehat{R} \subseteq E, \pi \in \Pi :$$
   $$e \sqsubseteq e' \implies (eq.\ 2.2) \implies [\![\widehat{R} \cup R \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R \cup e']\!]_{TM} \implies$$
   $$\implies (eq.\ 2.1) \implies [\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R]\!]_{TM}$$

   □

6. Точность анализа никогда не изменяется: $prec_S(e, \pi, R) = (e, \pi)$.
7. $\forall e_1, e_2 \in E_S^S : compatible_S(e_1,e_2) = (e_1 \cap e_2 = \varnothing)$
8. Cpa-оператор $compose$ является обычным 2.21.
9. Переходы являются *инвариантными к окружению*, то есть ни один поток не может изменить положение другого потока. Это означает, что проекцией является тождественный переход: $(s, q)|_p = (s, \varepsilon)$.

Доказательство того, что определение отношения переходов удовлетворяет условию 2.27, приведено в разделе Приложения A.8.

2.14. Анализ явных значений

Определим анализ явных значений (Value Analysis) $\mathbb{V} = ((D_V, \Pi_V, \rightsquigarrow_V, merge_V, stop_V, prec_V, compatible_V, \cdot|_p, compose_V)$, который отслеживает явные значения переменных. В отличие от анализа предикатов ValueCPA хранит только значения переменных, а значит, не может отлеживать более сложные зависимости между переменными.

Рисунок 2.16 демонстрирует пример переходов анализа явных значений. В абстрактном состоянии содержится отображение из переменных в их значения.

Перейдем к формальному описанию ValueCPA. Для этого снова воспользуемся явным видом перехода, описанным в разделе 2.5.4. Он состоит из следующих компонент.

```
1: int main {
2:    int y = 3;
3:  int x = y + 1;
4:   if (x > 0) {
5:     ...
6:   }
}
```

Рисунок 2.15 — Пример исходного кода

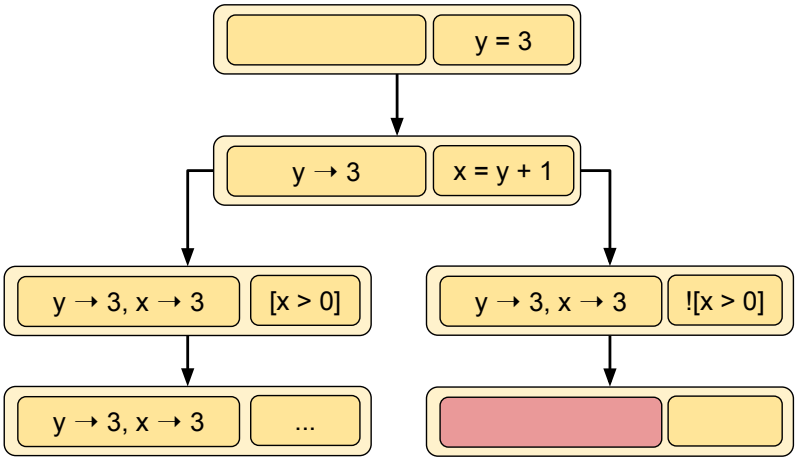

Рисунок 2.16 — Пример переходов ValueCPA

1. Абстрактное состояние этого анализа является отображением из имен переменных в их значение: $\forall s \in E_V^S, s : X \to \mathcal{Z}$, где $\mathcal{Z} = \mathbb{Z} \cup \{\bot_Z, \top_Z\}$. Таким образом, множество абстрактных состояний E_V^S является плоской решеткой над целыми числами. Верхний элемент решетки $\top_V^S = \{v \mid \forall x \in X : v(x) = \top_Z\}$ является отображением, в котором каждая переменная имеет любое значение. А нижний элемент решетки $\bot_V^S = \{v \mid \exists x \in X : v(x) = \bot_Z\}$ является отображением, в котором хотя бы одна переменная не может иметь никакого явного значения. Такое состояние является недостижимым при реальном выполнении программы. Порядок является тривиальным: любые два невырожденных состояния (неравные $\top_V^S$ или $\bot_V^S$) являются несравнимыми.

   Cpa-оператор C_{Check} проверяет, существует ли общее отображение глобальных переменных: $\forall s_0, \ldots, s_n \in E_V^S : C_{Check}(\{s_0, \ldots, s_n\}) = \forall x \in X^g, \exists z \in \mathcal{Z} : \forall 0 \leqslant i \leqslant n : z = s_i(x) \vee s_i(x) = \top_Z$. Обозначим такое общее отображение через $\hat{v}_g$.

$$\forall s_1, \ldots, s_n \in E_V^S :$$

$$\bigoplus_S^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s'_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) =$$

$$= \begin{cases} \{c \in C \mid \forall x \in X^g : \hat{v}_g(x) \neq \top_Z \implies c_g(x) = v_g(x)\} \\ \qquad \text{, если } C_{Check}(\{s_0, \ldots, s_n\}) \\ \varnothing, \text{ иначе} \end{cases}$$

Такое определение cpa-оператора $\bigoplus_S^S$ очевидно удовлетворяет условиям 2.17, 2.18, так как большее состояние ($\sqsubseteq_V^S$) означает только $\top_V^S$, что, очевидно, только расширяет множество конкретных состояний.

Множество абстрактных дуг содержит множество обычных CFA дуг и переходы в окружении, которые определяются изменением глобальных переменных: $E_V^T = 2^{X \to \mathcal{Z}} \cup G$. Тождественный переход $\varepsilon = \varnothing$ является пустым отображением, при котором ни одна переменная не меняет своего значения. Cpa-оператор конкретизации для дуг $\|\cdot\|$ сопоставляет каждому отображению множество дуг, которое может иметь такое отображение. Например, $\|x \to a\| = \{g \mid g = (\cdot, assign(x, a), \cdot)\}$.

2. Точность анализа явных значений определяется отслеживаемыми переменными, таким образом множество точности содержит подмножества из всех переменных программы: $\Pi_V = 2^X$.
3. Отношение переходов $\rightsquigarrow_V$ содержит переход $e \overset{R}{\rightsquigarrow}_V e'$, с $e = (s,q)$, $e' = (s', \top_V^T)$, если
   - $q \in G, q = (\cdot, assume(expr), \cdot')$

$$\forall x \in X : s'(x) = \begin{cases} \bot_Z, \text{ если } \nexists a \in \mathbb{Z}.(x \to a) : (expr \neq 0)_{/s} \\ a, \text{ если } \exists! a \in \mathbb{Z}.(x \to a) : \\ \qquad\qquad (expr \neq 0)_{/s} \vee s(x) = c \\ \top_Z, \end{cases}$$

     Здесь $(expr)_{/s}$ означает интерпретацию выражения $expr$ над переменными из X для абстрактного присваивания s. А выражение $(x \to a) : (expr \neq 0)_{/s}$ означает, что значение a у переменной x удовлетворяет интерпретации.
   - $q \in G, q = (\cdot, assign(w, expr), \cdot')$

$$\forall x \in X : s'(x) = \begin{cases} expr_{/s}, \text{ если } x = w \\ s(x), \end{cases}$$

   - $q \in G$ в остальных случаях состояние не меняется: $s' = s$.
   - $q : X \to \mathcal{Z}$, что означает, что мы имеем переход, который меняет определенные переменные. В этом случае, следующее состояние

$$\forall x \in X : s'(x) = \begin{cases} q(x), \text{ если } x \in dom(q) \\ s(x), \end{cases}$$

4. Cpa-оператор слияния не объединяет абстрактные элементы: $merge_V(e, e', \pi) = e'$. Условие 2.6 очевидно выполнено.
5. Cpa-оператор останова рассматривает уникальные состояния: $stop_S(e, R, \pi) = (\exists e' \in R \wedge e \sqsubseteq e')$. Проверим, что требование 2.7 выполнено.

   Доказательство.

   $$\forall e \in E, R, \widehat{R} \subseteq E, \pi \in \Pi :$$
   $$e \sqsubseteq e' \implies (eq.\ 2.2) \implies [\![\widehat{R} \cup R \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R \cup e']\!]_{TM} \implies$$
   $$\implies (eq.\ 2.1) \implies [\![\widehat{R} \cup e]\!]_{TM} \subseteq [\![\widehat{R} \cup R]\!]_{TM}$$

   □

6. Функция настройки точности вычисляет новое абстрактное состояние и точность, ограничивая присваивания только теми переменными, которые содержатся в точности: $prec_V(e, \pi, R) = (e_\pi, \pi)$.
7. $\forall e_1, e_2 \in E_V : compatible_V(e_1,e_2) = C_{check}(\{s_1, s_2\})$.
8. Cpa-оператор $compose$ является обычным 2.21.
9. Переход в окружении может затрагивать только глобальные переменные: $\forall e \in E_V, e = (s,q) : e|_p = (s^{global},q^{global})$. Здесь отображение s^{global} означает только ту часть, которая относится к глобальным переменным.

Доказательство того, что определение отношения переходов удовлетворяет условию 2.27, приведено в разделе Приложения A.9.

2.15. Метод поиска состояния гонки

Метод поиска состояния гонки заключается в построении множества достижимых абстрактных переходов с помощью алгоритма CPA 1, а затем в поиске пары абстрактных переходов, начальные состояния которых являются совместными, а сами переходы производят доступ к разделяемой области памяти, при этом хотя бы один из доступов является записью.

Построение множества достижимых состояний производится с помощью комбинации описаных ранее CPA. Возможны различные конфигурации CPA, в зависимости от поставленной задачи, например, такая: анализ точек программы

$\mathcal{L}$, анализ потоков $\mathbb{T}$, анализ примитивов синхронизации $\mathbb{S}$ и анализ предикатов $\mathbb{P}$. Все эти анализы объединяются с помощью композитного анализа $\mathbb{C}$. Кроме различного множества используемых CPA возможно более точная настройка каждого CPA в отдельности, например, изменением варианта cpa-оператора $merge$.

Тем не менее, было показано, что cpa-операторы каждого из этих видов анализа удовлетворяют условиям 2.5, 2.6, 2.7, 2.8, а значит, позволяют утверждать по теореме 1, что множество вычисленных абстрактных состояний аппроксимирует сверху множество конкретных состояний программы. Таким образом, если в числе абстрактных состояний программы присутствует ошибочное, значит, программа не соответствует поставленной спецификации.

Покажем, что для программы, содержащей состояний гонки в смысле определения 1, будет найдено состояние гонки на абстрактных состояниях. Пусть у нас есть множество достижимых состояний $R : \exists c, c_1, c_2 \in [\![R]\!] \subseteq C, g_1, g_2 \in G, \widehat{t}, \overline{t} \in T, \widehat{t} \neq \overline{t} : c \xrightarrow{g_1,\widehat{t}} c_1, c \xrightarrow{g_2,\overline{t}} c_2$, то есть присутствует состояние гонки. По теореме 1 это означает, что

$$
\begin{aligned}
&\exists e_0, e_1, \ldots, e_n, \tilde{e}_0, \tilde{e}_1, \ldots, \tilde{e}_n \in R,\\
&t_0, \ldots, t_n, \tilde{t}_0, \ldots, \tilde{t}_n \in T,\\
&\tau_1 \in \bigoplus \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right),\\
&\tau_1 \in \bigoplus \left(\begin{pmatrix} \tilde{e}_0 \\ \tilde{t}_0 \end{pmatrix}, \left\{ \begin{pmatrix} \tilde{e}_1 \\ \tilde{t}_1 \end{pmatrix}, \ldots, \begin{pmatrix} \tilde{e}_n \\ \tilde{t}_n \end{pmatrix} \right\} \right),
\end{aligned}
$$

По требованию 2.13 переход e_0 должен быть применим ($apply$) ко всем переходам $\tilde{e}_i$, а значит и к $\tilde{e}_0$. Таким образом, получаем, что переходы e_0 и $\tilde{e}_0$ должны быть совместны. При этом требования на абстракные операции q_1, q_2 должны быть теми же, что и в определении состояния гонки 1: доступ к одной области памяти, и хотя бы одна операция записи. Итак, можно сформулировать определение состояния гонки на абстрактных состояниях.

Определение 2. *Программа, для которой получено множество достижимых абстрактных состояний R, содержит **состояние гонки**, если $\exists e_1, e_2 \in R$, $e_i = (s_i, q_i)$, $q_1, q_2 \in G$, $compatible(e_1, e_2)$ и операции g_1, g_2 выполняются над одной и той же глобальной переменной $x \in X^{global}$ и хотя бы одна операция является присваиванием (то есть $assign(x, expr)$).*

Определение 2 является ключевым в данном методе. Именно оно определяет, какие именно состояния будут трактоваться, как ошибочные. Особенное внимание следует обратить на использование оператора compatible, который изначально появился при описании алгоритма построения окружения. Этот оператор позволяет накладывать дополнительные ограничения на рассматриваемые пары абстрактных переходов, что повышает точность анализа.

Итак, метод поиска состояний гонки состоит из двух основных частей.

– Построение множества достижимых состояний с использованием подхода с раздельным анализом потоков.
– Поиск таких пар абстрактных переходов, которые удовлетворяют определению 2.

Глава 3. Реализация метода анализа корректности многопоточтных программ для применения для компонентов ядра операционных систем

3.1. Устройство инфраструктуры CPAchecker

Для реализации был выбран фреймворк CPAchecker, поэтому далее будет кратко представлена его инфраструктура. CPAchecker является международным открытым проектом, разработка которого началась в 2007 году после публикации статьи [81]. Основным разработчиком является Дирк Бейер (Dirk Beyer) и его команда. В данный момент он преподает в Мюнхенском университете имени Людвига и Максимилиана. Сообщество разработчиком инструмента включает в себя участников из различных стран, например, Германия, Чехия, Россия, Италия, Аргентина, Австрия и др.

Фреймворк CPAchecker содержит в себе всю необходимую инфраструктуру для реализации различных подходов статической верификации. Классический набор компонентов для верификации программного обеспечения является следующим: парсер, который преобразует исходный код программы во внутреннее представление, алгоритм, задающий правила работы с cpa-операторами, сам набор CPA и автомат, который определяет нарушение спецификации. В основном, фреймворк используется для верификации программ на языке Си, однако имеет возможность верификации Java программ, а также потенциальную возможность расширения подхода на другие языки.

В качестве парсера используется сторонний компонет Eclipse CDT Parser[1], который строит абстрактное синтаксическое дерево (англ. Abstract Syntax Tree, AST). Далее, AST преобразуется в CFA, в этот момент выполняются такие преобразования, как добавление дуг ведущих внутрь функции и обратно в точку ее вызова, что необходимо для обеспечения межпроцедурности анализа. Также, в этот момент собираются различные метаданные, которые затем могут быть использованы в процессе основного анализа: информация о переменных, циклах, зависимостях и т.п.

Исходный код программы, подаваемый на вход может быть разбит на несколько файлов, но, тем не менее, этот набор должен компилироваться. Что-

[1] https://eclipse.org/cdt

бы избежать сложных конструкций языка Си и упростить код, часто применяется инструмент CIL [34], который, в том числе, может объединять несколько файлов в один.

Алгоритм задает правила работы с cpa-операторами. Классический CPAAlgorithm был представлен в алгоритме 1. Кроме него могут использоваться и другие, например, BMCAlgorithm, PDRAlgorithm, которые реализуют подходы Bounded Model Checking (BMC) [84] и Property Directed Reachability (PDR) [85] соответственно. Алгоритмы могут задавать комбинацию различных подходов, например, CEGARAlgorithm использует внутри вложенный алгоритм основного анализа, а при его завершении начинет итерацию уточнения. RestartAlgorithm позволяет задавать последовательную комбинацию различных алгоритмов.

Любой алгоритм так или иначе требует определения CPA, с которыми он будет работать. Используемый набор CPA определяет возможности проводимого анализа и его ограничения. Почти во всех конфигурациях используются служебные CPA: LocationCPA и CallstackCPA. Различные примеры CPA были описаны ранее, и здесь нет смысла повторяться.

Для описания ошибочной ситуации используется автоматная модель. С помощью простого языка задается наблюдательный автомат, который совершает переходы в соответствии с графом потока управления одновременно с основным процессом анализа. Как только этот автомат доходит до некоторого терминального (ошибочного) состояния, считается, что найдена ошибка в программе. Несмотря на то, что формат описания ошибки позволяет определять достаточно сложные условия, наиболее часто используется задание специальной функции, трактуемой, как ошибка. В этом случае, автомат выглядит как

```
CONTROL AUTOMATON UNREACH

INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH {__VERIFIER_error($?)}->ERROR("unreach-call");

END AUTOMATON
```

Таким образом, вызов функции __VERIFIER_error, рассматривается, как ошибка. В процессе анализа автомат представляется в виде одного из CPA.

Такая спецификация используется для решения задачи достижимости, при которой проверяется, достижимо ли некоторое состояние программы, в данном случае, вызов функции __VERIFIER_error. CPAchecker поддерживает и решение других задач, например, проверки корректного использования памяти или завершимости программы. Такие задачи не описываются в виде спецификаций и могут быть решены только при специальной конфигурации Algorithm-CPA.

Реализация инструмента CPALockator для поиска состояний гонки позволило проводить поиск ошибок данного типа.

3.2. Общий вид инструмента для верификации многопоточных программ

Как уже было описано, инструмент CPAchecker позволяет проверять программное обеспечение на соответствие различным типам спецификации. Достижимость ошибочного состояния, корректная работа с памятью, завершаемость могут проверяться для многопоточных программ также, как и для последовательного случая. Далее мы будем подробно рассматривать два основных варианта: проверку достижимости ошибочного состояния и поиск состояний гонки.

Основной набор CPA для каждого из этих вариантов остается неизменным: ThreadModularCPA (раздел 3.3), ARGCPA (раздел 3.5), CompositeCPA (раздел 3.10), LocationCPA (раздел 3.7), CallstackCPA, LockCPA (раздел 3.6), ThreadCPA (раздел 3.8), PredicateCPA (раздел 3.9).

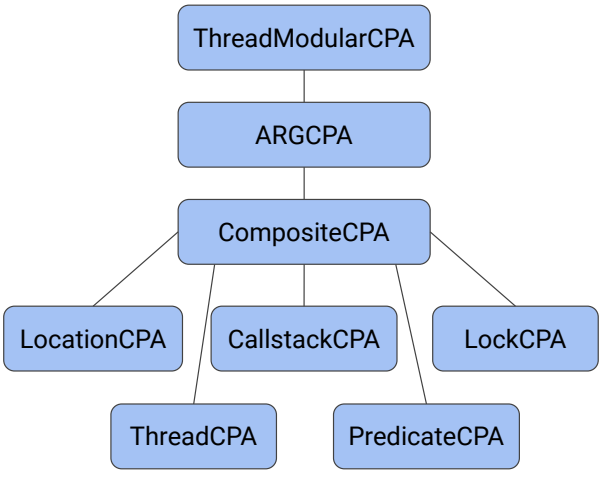

Рисунок 3.1 — Вариант комбинации CPA

На рисунке 3.1 показана типовая комбинация CPA для подхода с раздельным рассмотрением потоков. CallstackCPA является служебным и отвечает за поддержку межпроцедурного анализа. Его реализация не изменилась по сравнению с классической версией, поэтому он не будет описан далее. При поиске состояния гонки добавляется некоторый служебный UsageCPA (раздел 3.11), который позволяет упростить работу с доступами к сложным типам данных. При решении некоторых практических задач на первый план может ставиться скорость анализа. В этом случае может быть применена оптимизация BAM (раздел 3.4), которая позволяет существенно ускорить проведение анализа, но накладывает дополнительные ограничения на используемые CPA. Далее реализация всех упомянутых CPA будет описана подробно.

Для определения состояния гонки необходимо было для каждой пары достигнутых состояний проверить наличие обращения к одинаковой разделяемой памяти. Такой простой алгоритм был совершенно не эффективен для практического применения. Количество абстрактных состояний может превышать десятки миллионов. Поэтому необходимы очень эффективные алгоритмы хранения и поиска пары состояний образующих состояние гонки. Такие оптимизации описаны в разделе 3.13.3.

Процесс уточнения построенной абстракции может занимать достаточно много времени даже при решении задачи достижимости. При поиске состояний гонки может быть необходимо уточнить абстракцию сразу для нескольких обнаруженных состояний гонки. Уточнение абстракции последовательно становится очень неэффективным. В разделе 3.14 описан процесс последовательного уточнения абстракции при поиске состояния гонки и применяемые оптимизации.

Важной задачей, на которую часто не обращают внимания во многих академических исследованиях, является понятное и наглядное представление результатов верификации. Это особенно важно для описания ошибок, связанных с параллельным выполнением нескольких потоков, так как при этом не достаточно указать только строку или переменную, в которой возможно наличие ошибки. Необходимо представить полную трассу выполнения потоков, выделив некоторые важные события, например, создание потоков, захват примитивов синхронизации и одновременные доступы к разделяемой памяти. В подразделе 3.15 будет описан формат выходных данных, который был основан на существующем формате GraphML для представления результатов верификации.

3.3. Реализация ThreadModularCPA

Основное отличие реализации ThreadModularCPA от теоретического описания (раздел 2.5) заключается в том, что в реализации применение переходов происходит до cpa-оператора $transfer$, а не после. Это приводит к тому, что все основные cpa-операторы – $merge$, $stop$, $prec$ – выполняются над проекциями, а не над примененными переходами. Проекций значительно меньше, а очень многие проекции могут быть отброшены как покрытые, что позволяет значительно сократить количество элементов, к которым применяются эти cpa-операторы. С точки зрения представленной теории такая операция не совсем корректна, так как композиция ($\oplus$) проекции с любым другим переходом не даст ни одного конкретного перехода. В дальнейшем возможно определение расширенного cpa-оператора $\oplus$, который позволит работать и с проекциями. Однако, это не является темой данной работы, и поэтому данная оптимизация была представлена только в реализации.

Некоторые незначительные оптимизации связаны с сокращением количества проверок совместности состояния и проекции. Для этого определяются те состояния, к которым не имеет смысла применять проекции. Такими состояниями могут быть либо инвариантные ко всем возможным проекциям, либо те, к которым в принципе невозможно применить ни одну проекцию. Примером первого варианта могут быть состояния, в которых уже считается, что разделяемые данные принимают любые значения. Примером второго варианта являются состояния, в которых известно, что активен только один поток.

3.4. Реализация BAMCPA

3.4.1. Краткое описание

Сохранение результатов анализа абстрактных блоков (англ. Block Abstraction Memoization, BAM) является очень эффективной оптимизацией, которая основана на кэшировании результатов. Абстрактными блоками, на границах которых производится кэширование результатов, могут быть как функции, так и

тела циклов. При входе в абстрактный блок запоминается начальное состояние, а при выходе из него – конечное. И при следующем входе в этот абстрактный блок с тем же абстрактным состоянием, второй раз построение всего множества достижимых состояний не производится, а сразу выдается конечное состояние, взятое из кэша.

Для увеличения частоты попадания в кэш используются дополнительные операции $reduce/expand$, которые возвращают существенную для каждого внутреннего CPA часть абстрактного состояния. Таким образом, можно избавиться от отличающихся, но несущественных деталей, которые мешали бы попаданию в кэш. Обратная операция $expand$ возвращает утраченные детали в конце абстрактного блока на основе исходного состояния, в котором они присутствуют.

```
int g = 0;
 int f(int a) {
   return a + 1;
 }
 int main() {
    int l = 0;
    g = f(g);
    l = f(l);
    ...
}
```

Рисунок 3.2 — Пример исходного кода

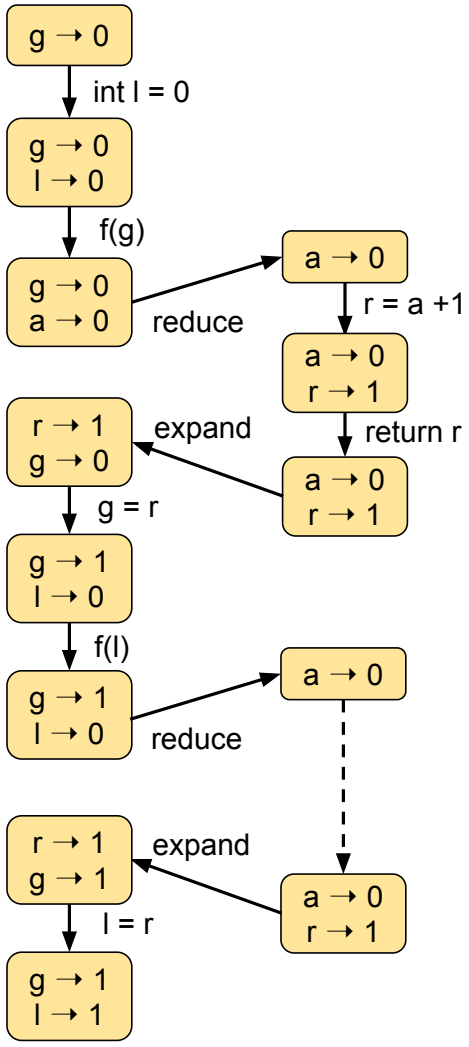

Рисунок 3.3 — Пример переходов BAMCPA

На рисунке 3.3 показан пример работы BAMCPA. При первом входе в функцию f применяется операция $reduce$, которая определяет, что переменная g не используется во внутреннем блоке (функции), и удаляет эту информацию из состо-

яния. Таким образом, анализ функции производится с более абстракным начальным состоянием. А при выходе из блока применяется обратная операция $expand$, которая возвращает прежнюю точность. При втором заходе в функцию снова применяется операция $expand$, которая удаляет изменившееся значение переменной g, что позволяет получить точно такое же начальное состояние, а значит, попасть в кэш. Финальное состояние берется из кэша, к нему применяется операция $expand$, и анализ подолжается.

3.4.2. Недостатки оптимизации BAM

BAM позволяет значительно сократить время работы инструмента при частых и однообразных вызовах функций. Однако, такая оптимизация имеет и недостатки, которые существенным образом влияют на весь процесс анализа программы.

1. Множество достижимых состояний (reached set) становится неполным. С учетом операции $reduce$ в нем могут отсутствовать те состояния, которые были бы достижимы при анализе программы без оптимизации BAM. Хотя операция $reduce$ и должна сохранить всю существенную для анализа информацию, отсутствующие детали могут не позволить использовать упрощенные состояния напрямую в некотором пост-анализе.
2. Множество достижимых состояний становится распределенным, то есть для каждого абстрактного блока создается своя копия множества достижимых состояний, в котором находятся все достижимые в данном блоке состояния, с учетом упрощения cpa-оператором $reduce$. Такая копия необходима для удобства работы с кэшем.
3. Вычисление пути в графе достижимых состояний значительно усложняется из-за перечисленных выше пунктов.
4. Все операции модификации множества достижимых состояний, в первую очередь при уточнении, становятся значительно сложнее, так как при любой модификации вложенного множества достижимых состояний необходимо проверить, как эта модификация затрагивает все остальные множества достижимых состояний.

Эти сложности и недостатки были не очень существенными для классического анализа последовательной программы. В случае подхода с раздельным рассмотрением потоков такие недостатки могут привести к некорректному результату. Действительно, при вычислении возможных эффектов окружения на некоторое состояние, необходимо иметь доступ ко всему множеству достижимости, а в случае BAM это доступное множество ограничивается одним блоком. Таким образом, применение BAM в текущем виде в подходе с раздельным рассмотрением потоков становится невозможным. Одним из возможных направлений развития инструмента может стать реализация BAM в новой концепции, которая позволит сохранить монолитность множества достижимых состояний, что позволит расширить область ее применения на случай подхода с раздельным рассмотрением потоков.

В некотором частном случае применение BAM возможно. Это может произойти, если все вложенные CPA являются инвариантными к переходам в окружении, то есть применение проекций ни в каком случае не даст новых состояний. В таком случае алгоритм анализа сводится к классическому, и ограничения BAM не смогут привести к некорректному результату.

3.4.3. Особенности оптимизации BAM при поиске состояния гонки

Еще одна проблема, связанная с BAM, возникает при поиске состояний гонки. Она связана с восстановлением пути, приводящего к ошибочному состоянию. В исходном варианте оптимизации BAM состояние было только одно – ошибочное, соответственно, ошибка могла быть зафиксирована, если анализ обнаружил путь к этому конкретному состоянию. В этот момент можно завершить построение абстракции и перейти к восстановлению ошибочного пути и его уточнению. При решении задачи достижимости найденная ошибка выдается как только она была найдена, и поэтому стек абстрактных блоков, приводящих к ошибке, восстанавливается единственным образом. Если бы это было не так, то уже бы существовало как минимум два пути к ошибке, проходящих через различные абстрактные блоки. Так как процесс анализа остается последовательным, то в этом случае данное ошибочное состояние было бы уже обнаружено при анализе первого абстрактного блока.

При поиске состояний гонки приходится сначала строить все достижимые состояния, а затем искать те пары, которые образуют состояние гонки. Это означает, что при восстановлении пути может возникнуть неопределенность, если одна из функций на стеке вызовов может быть вызвана из нескольких мест. Такая неопределенность не может привести к некорректным результатам, однако она может повлиять на детерминированную работу инструмента, что значительно снижает удобство использования.

Для решения этой проблемы был переработан механизм восстановления пути, для того чтобы восстановление возможных путей проводилась строго детерминировано. При визуализации результатов нет необходимости показывать все возможные пути к некоторому доступу к памяти, однако в процессе уточнения может быть необходимо проверить все варианты. Подробнее про процесс уточнения результатов будет рассказано в соответствующем разделе.

3.5. Реализация ARGCPA

ARGCPA отвечает за построение абстрактного графа достижимости (англ. Abstract Reachability Graph, ARG). Его состояния содержат в себе информацию о связях с другими состояниями. Имеются следующие типы связей:

– parent-child. Связь соответствует cpa-оператору $transfer$, то есть дочерний элемент был получен из родительского путем применения cpa-оператора $transfer$ к последнему. У каждого дочернего элемента может быть множество родительских, например, из-за применения cpa-оператора $merge$, а также у каждого родительского может быть несколько дочерних.
– covers-covered by. Связь соответствует cpa-оператору $stop$, то есть если cpa-оператор вернул true, то новый элемент считается покрытым старым. Следует отметить, что несмотря на теоретическую возможность покрытия множеством состояний, все реализации cpa-оператора $stop$ используют покрытие только одним состоянием. Поэтому каждое абстрактное состояние может покрывать множество других, но быть поктыто только одним состоянием.

– mergedInto. Связь соответствует cpa-оператору $merge$ и означает, что один элемент был результатом вызова cpa-оператора $merge$ для другого состояния.

Данные типы состояний используются в разных целях: для визуализации, для построения контрпримера, для вычисления поддерева при уточнении. Для подхода с раздельным рассмотрением потоков были добавлены две дополнительные связи:

– projectedTo-projectedFrom. Связь соответствует cpa-оператору $project$ и нужна, чтобы определить, из какого исходного перехода была получена та или иная проекция. Эта связь является связью типа *много-много*, так как каждая проекция может быть получена из нескольких переходов за счет cpa-оператора $merge$, а состояние может быть спроецировано несколько раз на различных итерациях уточнения. При этом, скорее всего, вторая проекция будет покрыта первоначальной.
– appliedTo-appliedFrom. Связь соответствует cpa-оператору $apply$ и нужна, чтобы определить из каких состояний и проекций был получен тот или иной переход в окружении. Эта связь является связью типа *один-много*, так как каждая переход в окружении может быть получен из единственных проекции и перехода в потоке, но каждая проекция и переход может быть применен к множеству других переходов.

Алгоритм восстановления пути выглядит следующим образом. Для каждого перехода в потоке известен родительский переход (связь *parent-child*). С ее помощью для ошибочного состояния можно восстановить последовательность переходов, которые приводят к нему. При использовании cpa-оператора $merge$ процесс усложняется тем, что родительских состояний может быть несколько. Тем не менее, стандартным алгоритмом обхода графа всегда можно восстановить путь из начального состояния до ошибочного. В случае, если в процессе восстановления пути встречается переход в окружении, необходимо использовать дополнительные связи. Однако, возникает выбор, каким образом использовать информацию о проекциях.

– Полностью игнорировать информацию о пути к проекции, и восстановить локальный путь в потоке. При этом данный путь будет содержать эффекты окружения, то есть переходы в окружении.
– Дополнительно к предыдущему пункту восстановить путь до проекции и добавить его, как некоторый подпуть основного пути. Данный вариант

поможет проверить достижимость пути до проекции, которая сама по себе может быть недостижима.

– Восстановить путь с чередованиями между основным путем до ошибочного состояния в потоке и путем в окружении до проекции. Данный вариант позволит проверить совместную достижимость путей, что является наиболее точным вариантом. Однако, такой вариант является наиболее сложным, так как точная информация о возможных чередованиях была потеряна при абстрагировании от взаимодействия потоков. Это означает, что необходимо предпринимать дополнительные усилия, чтобы определить возможное поведение потоков, то есть, появляется некоторая дополнительная стадия анализа.

В данный момент реализован первый вариант восстановления пути, так как реализация уточнения позволяет проверять только локальные пути. Кроме того, представление пути из дуг ГПУ не позволяет выражать более сложные варианты переходов из окружения. Тем не менее, остальные варианты уточнения являются перспективными направлениями для дальнейшего развития инструмента и позволят проводить более эффективное уточнение абстракции.

3.6. Реализация LockCPA

Теоретически описанный LockCPA (раздел 2.13) имеет ряд допущений:

– захватываемый объект (блокировка) должен быть явно указан в cpa-операторе $acquire/release$;
– никакие другие cpa-операторы, кроме $acquire/release$, не работают с примитивами синхронизации;
– отсутствие рекурсивного захвата блокировки;

Однако, такие допущения не всегда выполняются при анализе реального кода.

3.6.1. Указатели на объекты блокировок

В реальных же программах блокировки часто передаются в функцию захвата или освобождения по указателям. При этом указатели могут храниться в полях различных структур. Это значительно усложняет задачу определения реального объекта блокировки, с помощью которого производится синхронизация. Во многих работах, которые посвящены вопросу применения инструментов статической верификации к реальному программному обеспечению, применяют анализ указателей для вычисления объектов, на которые могут указывать те указатели, которые используются при захвате блокировки. Однако, как и всякий анализ указателей, такой подход имеет ряд недостатков, главными из которых являются его эффективность и точность. Дело в том, что анализ указателей Андерсена (и его различные вариации) может применяться только если указатели должным образом инициализируются. В случае же если инициализация указателя не была проведена, и этому указателю не была поставлена в соответствие некоторая память, такой анализ будет некорректным. В нашем случае инициализация многих указателей может отсутствовать, так как не весь исходный код может доступен.

Все это приводит к тому, что определить точное соответствие указателей в общем случае становится невозможным. Приходится делать некоторое разумное предположение о том, что захват блокировки и ее освобождение, скорее всего, будет производится одинаковым образом. Например, если был использован указатель на объект при захвате, то он же будет использован и при освобождении. А если при захвате использовался указатель на поле структуры, то и при освобождении будет передан указатель на поле с тем же именем структуры того же типа. Таким образом, становится возможно, в соответствии с теорией, по имени переменной, быть может указателю, явно идентифицировать тот объект, с которым производится работа.

3.6.2. Неявные операции работы с примитивами синхронизации

В пользовательских программах используется выделенный интерфейс для работы с примитивами синхронизации, который предоставляет операционная си-

стема. В POSIX это функции pthread_mutex_lock/pthread_mutex_unlock и некоторые дополнительные. Однако, компоненты операционной системы внутри себя могут использовать различные примитивы синхронизации напрямую. Например, проверять, включено ли планирование, которое также может выступать в роли особой (глобальной) блокировки. При этом такая проверка может быть, как через специальный интерфейсный макрос/функции, например, $dispatchEnable()$, так и через явное обращение к переменной $dispatchEnabled == 0$[2]. Кроме того, отключение планирование (захват блокировки) может также проводиться с помощью интерфейсного макроса/функции $dispatchEnabled()$ или напрямую $dispatchEnabled = 1$. Хотя явное присваивание в служебные переменные без выделения интерфейсных макросов/функций является плохим стилем программирования, такой код иногда встречается на практике, и поэтому нуждается в поддержке. Таким образом, реализация анализа примитивов синхронизации была расширена для обработки таких случаев, как присваивание в переменную и проверки ее значения.

3.6.3. Рекурсивный захват блокировки

Еще одной небольшой особенностью реализации является поддержка рекурсивного захвата блокировки. Некоторые блокировки допускают захват себя несколько раз в одном потоке. В этом случае как только число вызовов функций освобождения будет равно числу ее захватов, она считается окончательно освобожденной. В описанной теории такой случай не поддерживается, так как захваченные блокировки моделируются множеством. В реализации достаточно просто поддержать рекурсивный захват, заменив множество на мультимножество. Однако, в этом случае возможны ситуации с неконтролируемом ростом числа захваченных блокировок. Это возможно как из-за ошибки в самом исходном коде, например, из-за захвата блокировки в цикле, так и из-за неточности самого анализа, например, рассмотрение недостижимого пути с отсутствующей парной операцией освобождения блокировки. То есть, при реальном выполнении программы такой путь был бы невозможен, и для каждой операции захвата была бы соответствую-

[2]Пример кода является вымышленным, совпадение с какой-либо известной операционной системой является случайным

щая операция освобождения. Но из-за неточности анализа анализа рассматриваются дополнительные пути, на которых отсутствует операция освобождения блокировки. Классическим примером такой ситуации является захват и освобождение блокировки под одним и тем же условием. После выполнения такого участка кода при реальном выполнении ситуация, при которой эта блокировка останется захваченной, невозможна. Однако, анализ может не определить, что условия являются тождественными, и будет рассматривать такой путь, при котором захват блокировки был произведен, а освобождения не было. Такая ситуация чревата тем, что количество абстрактных состояний после анализа данного участка кода может удвоиться, то есть, после выхода из функции (или блока кода внутри функции) вместо единственного достижимого состояния без захваченных блокировок, будут рассматриваться состояния как без блокировок, так и с захваченной блокировкой. И хотя такие шаблоны встречаются нечасто, все-таки обычно блокировки захватываются без сложных условий, необходимо иметь возможность давать анализу некоторые подсказки.

Такими подсказками стали аннотации функций. Пользователь может добавить информацию о том, как именно следует обрабатывать конкретную функцию. Поддерживаются следующие типы аннотаций:

- функция захватывает конкретную блокировку;
- функция освобождает конкретную блокировку;
- функция полностью сбрасывает конкретную блокировку, включая все рекурсивные захваты;
- при выходе из функции все изменения блокировок сбрасываются, то есть, считается, что на выходе из функции захвачены те и только те блокировки, которые были захвачены на входе в функцию.

3.6.4. Эффект блокировки

Еще одно отличие от теории – использование *эффектов блокировок* – было сделано больше для удобства, чем для эффективности. С точки зрения теории сра-оператор $transfer$ должен предоставить следующее (возможно, измененное) состояние. В реализации этот сра-оператор для каждой CFA дуги извлекает некоторое действие, которое она может совершить над состоянием. И затем, это действие

применяется к состоянию. Выделения действия в отдельный интерфейс является не обязательной даже с точки зрения эффективности, так как даже на больших примерах время работы LockCPA находится в рамках статистической погрешности (меньше 0,1% от общего времени работы). Однако, выделение каждого изменения в отдельный класс упрощает структуру кода. Так, имеются следующие варианты действия над состоянием:

- захват блокировки;
- освобождение блокировки;
- установка конкретного значения счетчика рекурсивного захвата блокировки;
- восстановление заданного значения счетчика рекурсивного захвата для некоторой блокировки;
- восстановление заданного значения счетчика рекурсивного захвата для всех блокировок;
- проверка значения счетчика рекурсивного захвата для заданной блокировки;

3.6.5. Оптимизация BAM

Отдельно следует отметить оптимизации, сделанные в рамках подхода BAM, который описан в подразделе 3.4. Как было описано, BAM позволяет кэшировать результаты, если некоторая часть состояния ($reduced\ state$) совпадает с уже пройденным. При этом, чем менее уникальным состоянием будет $reduced\ state$, тем эффективнее будет работать кэширование, то есть тем чаще будут возникать попадания в кеш. С другой стороны, необходимо иметь возможность восстановить исходное состояние по $reduced\ state$ в конце блока. Очевидно, что если внутри блока не происходит никакого изменения состояния, то есть, внутри блока нет операций с примитивами синхронизации, то исходное состояние в конце блока равно исходному состоянию в начале блока, и в этом случае можно полностью удалять все захваченные блокировки. В этом случае кэш будет работать максимально эффективно с учетом работы других CPA. При этом, конечно, остаются технические моменты, связанные с вычислением состояний гонки по завершению анализа. Действительно, во множестве достижимых состояний сохраняются

$reduced\ state$, в которых могут отсутствовать те блокировки, которые используются для обеспечения взаимного исключения параллельной работы потоков. В этом случае будет необходимо восстановить все утраченные блокировки после построения абстракции, о чем будет подробно рассказано в соответствующем разделе.

Одной из достаточно тривиальных оптимизаций для блоков, в которых производится работа с примитивами синхронизации, является удаление только тех блокировок, которые не затрагиваются этими операциями. Например, если известно, что в некотором блоке производится захват и/или освобождение блокировки $lock_1$, то из состояния можно удалить все остальные блокировки, а на выходе построить новое полное состояние, взяв изменение счетчика рекурсивных захватов $lock_1$ и добавив к нему значения счетчиков для других блокировок, взятые из исходного состояния. Такая оптимизация требует некоторого преданализа, результатом которого будет множество используемых блокировок для каждого блока.

Другой возможной оптимизацией является сокращение счетчика рекурсивных захватов. Действительно, даже в случае, если внутри функции происходит захват/освобождение блокировки один раз, не имеет значения, сколько раз эта блокировка была захвачена до этого. Таким образом, при входе в функцию можно заменить значение счетчика рекурсивных захватов на единицу, а при выходе, соответственно, применить полученную разницу к исходному состоянию. В случае применения этой оптимизации становится невозможным различить применение операции $release$ и $reset$, которая полностью сбрасывает счетчик рекурсивных захватов. Аналогичные проблемы вызывает операция установки конкретного значения счетчика рекурсивных захватов, хотя это исключительно редкая операция. Более того, если в блоке используются более одной операции $acquire$/$release$ подряд, это становится аналогичным использованию уже описанных операторов. Однако, все эти случаи встречаются достаточно редко, обычный сценарий использования - это захват одной блокировки в начале функции и освобождение ее в конце. Отсюда следует, что указанная оптимизация может применяться только в тех блоках, в которых не используются такие запрещенные конструкции.

3.6.6. Возможные вариации LockCPA

Одним из возможных вариантов настройки LockCPA является реализация различных cpa-операторов $merge$ по аналогии с PredicateCPA. $merge_{Sep}$ будет рассматривать все состояния по-отдельности, а $merge_{Join}$ – объединять их в соответствии с решеткой, то есть, пересекать множества захваченных блокировок, строя т.н. множество $must$-блокировок. Из общих соображений вариант $merge_{Join}$ должен уменьшить количество абстрактных состояний в случае, если возможны несколько путей, на которых захватываются различные блокировки, через одну точку программы. При этом возможно снижение точности.

Другим вариантом настройки LockCPA является использование уточнения. Уже описанный вариант анализа отслеживает все указанные блокировки с самого начала. Однако, эти блокировки могут быть ненужными при построении абстракции, а значит, такая точность с самого начала может быть лишней. Поэтому возможно применение метода CEGAR, с помощью которого будут определяться те блокировки, котороые являются необходимыми для исключения ложных сообщений об ошибке. Основной вопрос заключается в том, будет ли выигрыш времени из-за построения неточной абстракции перекрыт проигрышем на некоторое количество дополнительных уточнений. Ответы будут даны далее в разделе экспериментов.

3.7. Реализация LocationCPA

Анализ точек программы соответствует описанному в теоретической части LocationCPA: раздел 2.11. В целом, данный анализ не претерпел существенных изменений по сравнению с классической реализацией. Основной задачей было представить его работу в терминах переходов, то есть ввести понятие абстрактной дуги. В классическом варианте анализа, LocationCPA оперирует дугами CFA, но для подхода с раздельным рассмотрением потоков пришлось дополнить это множество следующими специальными типами дуг:

– пустая дуга;
– любая дуга;

– отсутствие дуги.

Пустая дуга означает, что данный переход не меняет состояние LocationCPA и применяется в переходах в окружении. Любая дуга означает, что данный переход может соответствовать любой CFA дуге и применяется в проекциях. Технически, можно было бы обойтись одним вариантом для обозначения обоих случаев, но для удобства и отличия перехода в окружении от проекции были использованы различные служебные абстрактные дуги. Отсутствие дуги требуется для обозначения ситуации, в которых отсутствует следующий переход.

3.8. Реализация ThreadCPA

Анализ потоков соответствует описанному в теоретической части ThreadCPA: разделы 2.8, 2.9 и 2.10. В зависимости от конфигурации запуска включается вариант ThreadCPA, основанный на той или иной версии теории.

3.8.1. Ограничения реализации

Важным отличием реализации данного анализа от теории является отсутствие жестко заданных идентификаторов потока. В теории идентификатором потока является точка входа в функцию потока. Соответственно, в этом случае становится невозможно поддерживать создание нескольких одинаковых потоков. Поэтому требуется некоторый явный идентификатор потока, который должен быть уникален.

В реальных языках используются различные интерфесы для создания потоков: POSIX, MPI, OpenMP и дл. Для системного программного обеспечения ближе оказывается интерфейс POSIX, в котором указывается некоторая переменная, в которой будет находится системный идентификатор потока, функция потока, ее аргумент, а также некоторые атрибуты. В этом случае идентификатором для анализа может выступать имя переменной, в которой должен находиться системный идетификатор потока. Ограничением такого подхода является различные нестандартные ситуации, в которых происходит создание нового потока, с записью его

идентификатора в переменную, в которой уже находится идентификатор другого исполняемого потока. Такие ситуации не являются некорректными с точки зрения стандарта POSIX, но являются нетипичными для реального программного обеспечения.

Другим ограничением данного способа идентификации потоков является невозможность передачи идентификаторов через присваивания другим переменным. Так как для анализа потоков идентификатором потока является имя переменной, то присваивание системного идентификатора потока в другую переменную останется незамеченным. Такие ограничения могут быть существенными для некоторого системного программного обеспечения, однако при решении конкретных прикладных задач автору не требовалась необходимость для расширения возможностей ThreadCPA. Тем не менее, в платформе CPAchecker уже реализованы несколько CPA, которые соответствуют различным видам анализа алиасов (синонимов). Они могут быть использованы для более точного вычисления идентификатора потока. Таким образом, данное ограничение хотя и может оказаться существенным для специфического программного обеспечения, само по себе не является критичным, и в случае необходимости может быть устранено с помощью существующих компонентов CPAchecker.

Еще одним ограничением эффективной релизации является неполная поддержка операций типа $thread\_join$. В описанной теории данные операции не были описаны, хотя возможно расширение языка и формального описания ThreadCPA для поддержки таких случаев. В данной реализации предполагается, что ожидать завершения потока может только тот поток, который его создал. В общем случае, это, конечно, может быть не так, однако, ситуация с передачей идентификатора дочернего потока от одного родительского потока другому для ожидания является необычной, и не встречалася в конкретных практических программах. Традиционная схема работы является следующей: один родительский поток создает несколько дочерних рабочих потоков (worker), раздает им задачи, а затем ожидает их завершения. Следует особенно подчеркнуть, что данное ограничение справедливо только для эффективной реализации, которая является инвариантной к эффектам окружения. При использовании менее эффективной реализации, в которой используются переходы окружения для построения полного дерева потоков, нет никакой разницы, какой поток выполняет операцию $thread\_join$.

3.8.2. Используемые оптимизации

Одной из важных оптимизаций является обработка ситуации, при которой производится создание потока с уже существующим идентификатором. Как уже было описано в предыдущем подразделе, такие ситуации являются нестандартными при реальном выполнении программы, однако они могут возникать внутри анализа из-за его неточностей. То есть, путь, в котором производится создание нескольких одинаковых потоков, является недостижимым при реальном выполнении. В соответствии с подходом CEGAR можно было бы для каждого такого случая запустить процесс уточнения абстракции, а затем перестроить абстракцию заново с новым уровнем точности. В целях повышения эффективности в процессе анализа такие ситуации не приводят к уточнению абстракции, а обрабатываются одним из двух следующих способов.

– После создания второго экземпляра (инстанса) потока, производится создание т.н. *самопараллельного потока*. Это означает, что теряется информация о точном количестве созданных потоков, и считается, что все операторы этого потока могут быть выполнены параллельно друг с другом. В этом случае становится бессмысленным дальнейшее создание потока, так как полученные состояния уже являются аппроксимацией сверху, и соответствуют любому количеству созданных потоков. Эта идея похожа на абстракцию счетчиков (англ. counter abstraction), при которой теряется информация о количестве активных потоков. Для классических методов проверки моделей такая оптимизация приводит к значительной потере точности, но для подхода с раздельным рассмотрением потоков информация о возможных зависимостях между потоками уже потеряна. Поэтому применение такой оптимизации при данном подходе не должна сильно снизить точность анализа.
– Игнорирование создания второго и последующего потоков. Такой подход, в общем случае, может приводить к пропуску ошибок. Однако, в некоторых предположениях она позволяет получить корректные результаты. Например, если имеется дополнительная информация, что поток всегда создается в единственном экземпляре. Такая ситуация возникает, в ситуации, когда потоки являются искусственными, то есть не относятся к

исходному коду программы, а лишь определяют те части кода, которые могут выполняться параллельно.

3.9. Реализация PredicateCPA

3.9.1. Обзор

Анализ предикатов в целом соответствует описанному PredicateCPA в разделе 2.12. Однако, его реализация содержит несколько особенностей, о которых не упоминалось в теоретическом описании, но о которых необходимо сказать в данном разделе, так как они оказывают серьезное влияние при реализации подхода с раздельным рассмотрением потоков.

Уточнение абстракции по контрпримерам (англ. Counterexample-guided abstraction refinement, CEGAR) [86], [87] – это подход для итеративного поиска точности анализа, который является достаточно строгим, чтобы доказать корректность программы, но достаточно грубы, чтобы быть эффективным. Анализ начинается обычно с пустой точности (пустого множества фактов, например, предикатов). Построенная начальная абстракция программы является чрезмерной. Если в этой абстракции было обнаружено некоторое ошибочное абстрактное состояние, то восстанавливается конкретный путь программы, который приводит к этому состоянию, и проверяется на достижимость. Если такой путь является достижимым в исходной программе, то ошибка считается найденной, и анализ завершается. В противном случае путь является недостижимым из-за неточной абстракции, и множество точности дополняется новыми фактами (уточняется) таким образом, чтобы исключить этот путь ошибки из абстракции. Для предикатного анализа новые предикаты могут быть построены на основе вычисления интерполятора Крейга [88]. Эти шаги повторяются до тех пор, пока не будет найден конкретный путь ошибки или не будет доказана безопасность абстрактной модели (и, следовательно, программы).

Для улучшения про производительности применяется ленивая абстракция [89]. После шага уточнения алгоритма CEGAR абстракция перестраивается не полностью, а только те ее части, в которых необходимы новые предикаты. Кроме

того, новые предикаты не будут использоваться глобально для всех путей выполнения, а только в той части абстракции, для которых они актуальны.

Настраиваемое кодирование блоков (англ. Adjustable Block Encoding, ABE) [83] позволяет улучшить производительность предикатной абстракции за счет сокращения числа вычислений абстракции и числа итераций уточнения. Новая абстракция вычисляется не для каждого нового абстрактного состояния, а в конце некоторого множества (блока) абстрактных состояний. С ABE абстрактные состояния являются кортежами из абстрактной формулы и конкретной формулы пути. Формула пути любого абстрактного состояния всегда представляет собой набор конкретных путей от входа блока до этого абстрактного состояния. При создании нового абстрактного состояния вычисляется новая формула пути, которая является сильнейшим постусловием для предыдущей формулы пути и текущего ребра. Абстрактная формула копируется из предыдущего абстрактного состояния. Только в конце блока вычисляется новая абстрактная формула, которая является абстракцией конъюнкции старой абстрактной формулы и формулы пути. В этот момент формула пути сбрасывается до $true$. ABE не только сокращает количество вычислений абстракции, но также уменьшает количество проверок покрытия (которые выполняются только на концах блока) и размер ARG (из-за слияния абстрактных состояний). Последнее сильно влияет на количество итераций уточнения, так как во время уточнения интерполянты вычисляются только для абстрактных состояний на концах блока, поскольку только для этих абстрактных состояний необходимы предикаты.

3.9.2. Настраиваемое кодирование блоков

Как уже было сказано, настраиваемое кодирование блоков (ABE) оказывает существенное влияние на эффективность проводимого анализа. Однако, важной особенностью этой оптимизации является возможность проверки покрытия (сра-оператор $stop$) только в конце блока. Для классического варианта реализации это позволяет уменьшить количество бесполезных проверок, так как количество состояний в конце блока не очень велико. Однако, для подхода с раздельным рассмотрением потоков это может быть не так, потому что к каждому состоянию внутри блока могут быть применены эффекты окружения для получения перехо-

дов в окружении. Если сразу же не отбросить их с помощью cpa-оператора $stop$, произойдет комбинаторный взрыв состояний. А это значит, что классический вариант ABE становится невозможным, и приходится пересчитывать абстракцию после каждого состояния, чтобы получить возможность проверять покрытие состояний.

Таким образом, избежать частого пересчета абстракции, то есть получить некоторую вариацию ABE, станет возможно только если для текущего состояния удастся показать, что к нему невозможно применение эффектов окружения. По общему виду состояния такое возможно определить, только если известно, что активен единственный поток. Для этого необходим нетривиальный анализ потоков. Такой вариант оптимизации позволяет исключить пересчет абстракции для самой начальной части пути, содержащей объявления переменных и типов. Для больших программ эта часть может занимать серьезный объем, и даже такая простая оптимизация позволяет существенно повысить эффективность.

Следующим шагом является более интеллектуальная проверка, имеет ли смысл применять эффект окружения к текущему состоянию. Эта идея восходит к давно известной редукции Липтона [90] и заключается в том, что некоторые операции в потоке являются локальными, а значит, результат действия окружения перед этой операцией и после нее будет одинаковым. Например, не имеет значения, как могут меняться глобальные переменные окружением, если текущий оператор потока записывает явное значение в локальную переменную. А значит, к такому состоянию можно не применять эффекты окружения, отложив их до того момента, когда они станут релевантными операции.

С такой оптимизацией необходимо обращаться очень аккуратно, так как некорректное определение нерелевантных операций приведет к пропуску ошибки из-за того, что в нужный момент не был применен переход в окружении.

3.9.3. Представление эффектов окружения

С точки зрения теории проекция перехода представляет собой пару из переименованных формул, соответствующих состоянию и действию. В реализации абстрактное состояние содержит абстрактную формулу и формулу пути. Обе эти формулы записаны над переменными программы, однако формула пути использу-

ет SSA представление (англ. Single Static Assignment) [91]. В этой форме присваивание в каждую переменную происходит лишь единожды. На практике это достигается использованием SSA-индексов. Соответственно, если применить действие другого потока в виде формулы пути, то используемые в этой формуле индексы старого потока станут некорректными в новом.

Были исследованы два варианта для представления эффектов окружения:

– Сохранение действия эффекта окружения в виде некоторого выражения, по которому каждый раз заново строится формула. Такой способ может быть реализован достаточно просто, однако, он снижает эффективность, так как каждый раз нужно заново построить формулу пути для этого выражения. Заметим, что никакое кэширование здесь не может быть использовано, так как необходимо получить формулу с уникальными индексами.
– Сохранение действия эффекта окружения в виде формулы. Такой способ позволяет проводить анализ более эффективно, так как не требуется перевычислять саму формулу каждый раз. Однако, необходим аккуратный пересчет индексов.

Таким образом, проекция перехода выглядит следующим образом: абстрактная формула, представляющая исходное состояние, и множество формул, которые описывают возможное действие. Абстрактная формула будет использоваться для проверки совместности состояний. А множество атомарных формул представляют собой различные дизъюнкты, которые возникают при объединении нескольких эффектов. Вместо такого множества можно было бы хранить одну большую формулу, но тогда это не позволило бы выбирать только релевантные дизъюнкты для каждого конкретного состояния. Например, если эффект окружения кодирует изменение переменной, про которую нет информации в текущем состоянии, то нет смысла применять такой эффект, так как получившееся состояние будет ниже по решетке, чем исходное и сразу же будет отброшено, как покрытое. Таким образом, это множество позволяет применять только релевантные эффекты для каждого конкретного перехода.

Еще одной оптимизацией для увеличения скорости анализа является использование неизвестных значений при присваивании. Во многих случаях становится неважным, как именно окружение может изменить разделяемые данные, и важно лишь то, что значение меняется. А работа с точными значениями переменных требует дополнительных ресурсов. В таких случаях эффект окружения

для каждого присваивания $x = 1$ может быть представлен в виде $x = *$, означающий, что значение переменной x было изменено недетерминированным образом. В качестве предельного случая может быть использован единственный эффект окружения $* = *$, который означает, что все разделяемые данные недетерминированно изменили свое значение.

Однако, получить результат, равносильный результату анализа с эффектом окружения $* = *$ можно значительно эффективнее. Для этого необходимо построить такую модификацию предикатного анализа, которая была бы инвариантна к эффектам окружения. Эффекты окружения ThreadCPA и LockCPA содержат только условие применимости, то есть не могут изменить состояние. А значит, весь анализ становится инвариантным к эффектам окружения. В этом случае (см. раздел 2.5.5) можно исключить ThreadModularCPA из конфигурации инструмента, что позволит существенно ускорить анализ. При этом следует отметить, что в некоторых случаях результаты у этих двух вариаций анализа будут отличаться. Это может произойти, если эффект окружения $* = *$ будет дополнен информацией от других CPA, например, ThreadCPA, который сумеет доказать, что данный эффект $* = *$ сбрасывает значения переменных только, если запущен некоторый поток.

3.9.4. Уточнение абстракции

Процесс уточнения предикатной абстракции не описывался в теоретическом разделе, так как пока он принципиально не отличается от классического варианта уточнения, то есть не содержит серьезных изменений, специфичных для подхода с раздельным рассмотрением потоков. Тем не менее, использование классического варианта уточнения является ограничением, которое не позволяет считать подход полным, так как некоторые ложные пути исполнения не могут быть удалены из абстракции в рамках классического варианта.

Как уже было сказано, при обнаружении ошибки запускается процедура уточнения CEGAR, первым шагом которой является восстановление некоторого пути от начального состояния до ошибочного. Стоит отметить, что при использовании ABE такой путь может быть не один из-за того, что промежуточные абстрактные состояния могли быть объединены с помощью cpa-оператора $merge$.

Восстановление пути происходит с учетом сохраненных связей ARG, что является технической задачей для классического анализа последовательной программы. В случае многопоточной программы глобальный путь должен содержать действия каждого потока. При этом, таких путей может быть снова некоторое множество. Однако, в подходе с раздельным рассмотрением потоков невозможно восстановить возможные варианты чередования, так как эффекты окружения уже не имеют никакой информации о них. Таким образом, с помощью классической процедуры восстановления пути становится возможным только получить локальный путь в отдельном потоке с точками воздействия эффектов окружения, то есть потенциальными переключениями на другие потоки. Это является серьезным ограничением, так как позволяет уточнять абстракцию в рамках только одного потока, и не позволяет ставить предикаты на эффекты окружения. Поэтому одним из дальнейших направлений развития подхода является реализация возможности глобального уточнения, то есть восстановление пути с чередованиями в нескольких потоках.

Отдельным вопросом остается представление эффекта окружения в пути. В классическом варианте уточнения путь представляет собой последовательность CFA дуг. Соответственно, для некоторых эффектов окружения может не существовать таких дуг, которые бы полностью отражали его действие. Более того, в случае, если эффект окружения включает в себя нетривиальные действия нескольких различных CPA, между ними возникнет конфликт, на основе какой информации подбирать подходящие дуги. В данной реализации этот вопрос также не рассматривался, так как на практике всегда применялся анализ предикатов. Напомним, что ни LockCPA, ни ThreadCPA, ни другие служебные CPA не используют никакого действия на другие потоки, а служат лишь для повышения точности cpa-оператора $compatible$. Однако, при использовании ValueCPA и PredicateCPA параллельно друг с другом возможна ситуация, при которой эффект окружения будет построен на основе как перехода ValueCPA, так и перехода PredicateCPA. Кроме того, возможны применения различных cpa-операторов $merge$, что приведет к полному рассогласованию связи перехода в окружении с исходными CFA дугами. Таким образом, для представления полного пути с чередованиями будет необходимо разработать представление переходов, которое будет основано не на CFA дугах, а на более общих принципах, что также является одним из возможных направлений дальнейшего исследования.

Локальный путь в потоке тоже может содержать противоречия, и он может быть уточнен по классической схеме: для полученного пути строится логическая

формула, она загружаются в специальный компонент *решатель* (англ. solver), который возвращает новые предикаты. Мы здесь не будем подробно касаться того, как именно могут извлекаться новые предикаты: на основе интерполянтов, инвариантов, слабейших предусловий или эвристик.

Отметим лишь, что можно использовать различные схемы построения формул для пути для решателя.

- Классический вариант – это извлечь уж построенную формулу пути из абстрактного состояния. Это должна быть именно формула пути, а не абстрактная формула, которая может быть менее точная.
- Перестроить логическую формулу заново. Здесь открывается простор для различных оптимизаций, например, не учитывать разделяемые данные в формуле. Такой вариант напоминает эффект окружения $* = *$, так как, по сути, игнорирование значений разделяемых переменных в формуле и означает, что они принимают любые значения. Другим вариантом может быть перестроение логической формулы с учетом потенциальных эффектов окружения. Такой способ позволяет сократить время на перестроение абстракции, так как за одну итерацию уточнения позволяет получить расширенное множество новых предикатов.

3.10. Реализация CompositeCPA

CompositeCPA объединяет все вложенные CPA в параллельную композицию. Основное отличие реализации от теоретического описания заключается в том, что этот CPA позволяет работать как с теми CPA, которые реализуют подход с раздельным анализом потоков, так и с теми, кто его не релизует, считая, что абстрактные состояния этих CPA инвариантны к переходам в окружении. Таким образом, если доступна конкретная CFA дуга, переход осуществляется по ней, а если внутри перехода обнаруживается только абстрактная дуга, то переход по ней будет совершать сам анализ. В этом случае остальные CPA, которые не реализуют соответствующий интерфейс, не осуществляют переход, считая, что их состояние не изменяется. Такой способ позволяет использовать различные существующие CPA, которые не реализуют соответствующий интерфейс.

Еще одной важной особенностью данного CPA является возможность реализовать cpa-оператор $merge$ специального вида. Дело в том, что внутренние cpa-операторы $merge$ не могут взаимодействовать друг с другом. Это значит, что каждый конкретный анализ должен самостоятельно решить, как он объединяет состояния. Однако, на практике часто бывает полезно использовать объединение состояний одного CPA при некотором условии на состояния другого CPA. Например, можно объединить все проекции для одного потока. Это означает, что если два состояния ThreadCPA равны между собой, то в этом случае можно вызывать $merge$ у PredicateCPA.

Еще одним важным отличием от теоретического описания является использование более сложного cpa-оператора $\downarrow$ ($strengthen$). Дело в том, что вычисление необходимой CFA дуги для перехода может быть сделано более эффективно и без этого cpa-оператора. Достаточно при выполнении cpa-оператора $transfer$ проверить, какой переход содержит LocationCPA внутри. В случае, если это CFA дуга, считается, что это переход в потоке, и все остальные CPA совершают переход по этой дуге. В случае, если это некоторая заглушка, означающая, что переход пустой, это означает, что совершается переход в окружении, и все остальные CPA совершают переход в соответствии со своим имеющимся переходом. Cpa-оператор $strengthen$ используется для отсеивания других бессмысленных переходов для повышения эффективности анализа.

Выходу из функции соответствует большое количества CFA дуг, которые ведут во все возможные места вызова данной функции. Для того чтобы в процессе анализа возврат из функции соответствовал точке вызова этой функции, CallstackCPA хранит не только стек вызовов функций, но и точку программы, в которой произошел этот вызов. Таким образом, даже если в одной функции имеется несколько вызовов другой функции, то возврат буде осуществлен корректно. В классической реализации CPAchecker из всего возможного множества дуг возврата из функции переход осуществлялся только по одной, и все выдаваемые состояния являлись корректными. Однако, при использовании варианта анализа с переходами необходимо выдать все возможные следующие переходы, то есть в том числе и переходы по некорректным дугам, так как определить, что эти переходы являются некорректными, становится возможно только на следующей итерации анализа. А значит, все некорректные переходы должны пройти полный цикл применения cpa-операторов $prec$, $merge$, $stop$. Это снижает эффективность анализа.

Чтобы этого избежать, применяется cpa-оператор $strengthen$, который позволяет отфильтровать только корректные переходы.

3.11. Реализация UsageCPA

Этот анализ используется только в конфигурации, которая применяется для поиска состояний гонки. Исторически UsageCPA был сделан для того, чтобы он занимался эффективным хранением множества доступов к памяти. Однако, в некоторый момент оказалось эффективнее перенести эту функциональность после основного алгоритма (см. раздел 3.13). Таким образом, в данный момент этот CPA реализует вспомогательные задачи по обработке специальных функций, анализ которых напрямую затрудняет работу инструменту. Есть две основные группы таких функций:

– Функции работы со сложными структурам данных, например, списками.
– Неопределенные функции, которые, тем не менее, имеют побочные эффекты.

Рассмотрим первую группу функций и возьмем для примера функцию, которая удаляет некоторый элемент из списка и возвращает его. Внутри эта функция переставляет некоторым образом ссылки на элементы этого списка таким образом, чтобы он оставался консистентным. Таких списков может быть несколько, которые устроены одинаковым образом, но используются по-разному, например, требуют различных механизмов синхронизации при работе с ними. Для анализа было бы удобнее считать, что при доступе к элементу этого списка, производится не только к этому элементу, а ко всему списку целиком. В таком случае появится возможность определить т.н. *высокоуровневые гонки*, которые проявляются на сложных структурах памяти. Таким образом, для каждой функции из первого множества, в конфигурации определяется пара доступов к памяти: один из них тот, к которому имеется непосредственный доступ, но каждый доступ к этой памяти после вызова указанной функции следует трактовать как второй доступ из этой пары. Рассмотрим пример условной функции *getLast()*, которая возвращает последний элемент списка: *element = getLast(list)*. Для такой функции может быть использована аннотация, которая определяет, что доступ к памяти $element$, ко-

торую возвращает эта функция, должен трактоваться как доступ ко всему списку $list$.

Реальное программное обеспечение часто использует библиотеки, код которых недоступен для анализа. Кроме того, такое программное обеспечение зачастую анализируется не целиком, а разбивается на некоторые модули, соответственно, код одного модуля недоступен при анализе другого. Так или иначе, возникает некоторое множество неопределенных функций. Часть из них можно было бы аннотировать, с какими разделяемыми данными производится работа внутри, что позволило бы получить более точные результаты. Рассмотрим пример условной функции *insertElement()*, которая вставляет элемент в список: *insertElement(element, list)*. Для такой функции может быть использована аннотация, которая определяет, что в этой функции производится доступ к памяти по указателю $list$, и этот доступ является записью, так как список может модифицироваться.

3.12. Построение множества разделяемых данных

При поиске состояний гонки может быть использован преданализ для построения множества разделяемых данных. Это множество будет использовано на следующей стадии анализа, и при вычислении потенциального множества состояний гонки для неразделяемых данных не будут выдаваться предупреждения. На рисунке 3.4 представлена визуальная схема метода.

Анализ разделяемых данных состоит из следующих CPA: BAMCPA, CompositeCPA, LocationCPA, CallstackCPA, ThreadCPA, LocalCPA. То есть, все CPA, кроме LocalCPA используются в основном анализе и уже были описаны ранее, поэтому в данном разделе будет представлено только описание LocalCPA, которая и реализует всю функциональность для определения разделяемых данных.

Данный CPA является инвариантным к переходам в окружении, так как вся возможная информация об указателях консервативно объединяется. То есть, некоторая переменная становится разделяемой в некоторой точке программы, то считается, что это переменная стала разделяемой для всех путей выполнения программы, в том числе и для всех потоков. Поэтому описание LocalCPA будет про-

Рисунок 3.4 — Общая схема метода

изведено в терминах классического CPA, то есть без описания абстрактных переходов и проекций, которые являются лишними для такого простого варианта анализа.

Будем использовать множество статусов $S = \{shared, unknown, unshared\}$, которые обозначают, что соответсвующий указатель является разделяемым, неизвестным и неразделяемым. Абстрактным доменом LocalCPA является множество всех возможных отображений из переменных X в их статусы S: $\forall e \in E, e = \{(x, s) \mid x \in X, s \in S\}$. Соответственно, в состоянии хранится информация о текущем отображении.

Опишем формальные правила преобразования состояний cpa-оператором $transfer$. Отдельно следует отметить, что рассматривается статус именно тех данных, на которые указывает переменная, если она соответствующего типа. Нет необходимости следить за глобальными переменными, так как они в любой момент являются разделяемыми и за локальными переменными, так как к ним нельзя получить доступ из другой функции, а значит, и из другого потока. Таким образом, имеет смысл сохранять информацию о статусе только тех переменных, которые являются указателями. Однако для сбора такой информации требуется знания о локальности самих переменных. Будем обозначать $local$ и $global$ информацию о локальности самих переменных. Не нужно путать их со статусом той памяти, на которую указывает переменная.

Рассмотрим некоторое состояние e и некоторый переход по дуге g, который соответствует некоторому оператору программы. Опишем, как может измениться e'.

1. Объявление локального указателя указателя добавляет информацию, что память, на которую он указывает, неразделяемая: $e' = e \cup \{a \to unshared\}$.
2. Правила обработки присваивания $a = b$:
   - Для переменных, не являющихся указателями, а также для присваиваний переменным некоторых констант, состояние не меняется: $e' = e$.
   - При присваивании локальному указателю другого, статус последнего присваивается первому. $e' = e \cup \{a \to e(b)\}$.
   - При присваивании глобальному указателю разделяемыми становятся данные, на которые указывал присваиваемый указатель. $e' = e \cup \{b \to shared\}$. Данное присваивание следует понимать так, что если в старом состоянии e было какое-либо значение для $e(b)$ оно заменяется на $shared$. Следует напомнить, что в целях оптимизации данные про глобальный указатель a не добавляются в состояние.
   - После взятия адреса у локальной переменной $a = \&b$ указывает на неразделяемые данные. $e' = e \cup \{a \to unshared\}$.
   - После взятия адреса у глобальной переменной $a = \&b$ указывает на разделяемые данные. $e' = e \cup \{a \to shared\}$
3. Условия сами по себе не меняют состояние анализа $e' = e$.
4. Обработка вызовов функций.
   - При вызове функции статус указателя, передающегося, как параметр в функцию, присваивается этому параметру. Кроме этих присваиваний в функцию не передается ничего, так как глобальные указатели не сохраняются. То есть вся информация о статусе переменных вызывающей функции становится недоступна в вызываемой функции. При вызове функции $f(a)$, $e' = \{p \to e(a)\}$, где p – это параметр функции.
   - При передаче в качестве параметра адреса глобальной переменной статус параметра становится shared. При вызове функции $f(\&a)$, $e' = \{p \to shared\}$, где p – это параметр функции.

- При передаче в качестве параметра адреса локальной переменной статус параметра становится unshared. При вызове функции $f(\&a)$, $e' = \{p \to unshared\}$, где p – это параметр функции.
- При вызове функции, возвращающей значение, сначала обрабатывается вызов функции в соответствии с правилами, описанными выше, а затем, согласно п. 2, обрабатывается присваивание возвращенного значения в переменную.

5. Во время возврата из функции состояние возвращается к тому, что было при ее вызове. При наличии добавляется информация о статусе возвращаемого значения.
6. Пустые операции не меняют состояние.

Отдельно следует отметить вычисление статуса для памяти, на которую указывает поле структуры.

- Если соответствующая информация сохранена в состоянии напрямую для $a \to f$, то результатом будет этот статус.
- Если соответствующая информация для $a \to f$ неизвестна, но сохранен статус для объемлющей структуры a, то результатом будет этот статус для структуры a.

Cpa-оператор $merge$ объединяет информацию от нескольких путей анализа, примером может являться две ветви условного оператора. Состояние $e' = merge(e_1, e_2, \pi)$ является таким отображением, для которого выполнено следующее условие: $\forall x \in X, e'(x) = max(e_1(x), e_2(x))$, где $shared > unknown > unshared$. Таким образом, cpa-оператор $merge$ реализует консервативность анализа: если на каком-то пути переменная может стать разделяемой, а на другом нет, она будет считаться разделяемой в дальнейшем анализе.

3.13. Вычисление состояний гонки

3.13.1. Обзор

Вычисление состояний гонки происходит после завершения основного алгоритма анализа, то есть после полного построения множества достижимых со-

стояний. Основной задачей такого постанализа является определение и сохранение информации о доступах к данным. Доступ к памяти представляется некоторым идентификатором, структура которого будет описана в подразделе 3.13.2. Для каждого оператора программы определяются те доступы к памяти, которые могут быть затронуты при его выполнении, определяется тип доступа и некоторая вспомогательная информация, например, номер строки исходного кода. Подробнее структура сохраняемой информации будет представлена в подразделе 3.13.3.

Считается, что следующие типы операторов программы порождают доступ к памяти:

– Присваивание. Для левой части присваивания формируется идентификатор (множество идентификаторов) и добавляется тип доступа на запись, для правой части присваивания формируется идентификатор (множество идентификаторов) и добавляется тип доступа на чтение.
– Условие. Формируется идентификатор (множество идентификаторов) с типом доступа на чтение.
– Вызов функции. Для каждого аргумента формируется идентификатор (множество идентификаторов) с типом доступа на чтение.
– Возврат из функции. Для возвращаемого значения формируется идентификатор (множество идентификаторов) с типом доступа на чтение. В случае, если возвращаемое значение присваивается в некоторую переменную, для этой переменной формируется идентификатор (множество идентификаторов) с типом доступа на запись.

Определения переменной без присваивания, пустые (служебные) операторы не считаются доступами к памяти.

Информация о доступе перед сохранением проходит ряд тривиальных фильтров, которые позволяют избежать хранения бесполезной информации о тех доступах к данным, которые не могут участвовать в состоянии гонки. Следующие доступы к данным отфильтровываются:

– Запись в локальные переменные. Локальные переменные доступны только одному потоку, поэтому их изменение возможно только из этого потока, а значит, доступы к ним не могут образовать состояние гонки.
– Запись в неразделяемую память. Аналогично предыдущему пункту, доступы к неразделяемой памяти не могут образовать состояние гонки. Например, сразу после выделения новой памяти она еще недоступна другим

потокам, и ее можно изменять без использования механизмов синхронизации.

– Запись в служебные переменные, например, те, которые используются в модели окружения для обеспечения корректной последовательности вызовов.

После построения множества доступов к памяти, для каждого идентификатора необходимо определить, существует ли пара доступов, которая образует гонку. Каждая пара доступов проверяется на совместность, то есть на возможность одновременного выполнения. Для этого используется понятие совместности переходов. Совместность означает, что два частичных абстрактных состояния могут быть частью одного глобального состояния. Или, другими словами, один переход может быть применен, как переход в окружении, к другому переходу и наоборот, откуда следует, что эти два перехода могут быть выполнены параллельно. Таким образом, предложенный подход является обобщением подхода Lockset [12], который определяет состояние гонки, как два доступа к некоторой памяти с непересекающимся множеством блокировок. Одним из ограничений подхода Lockset является отсутствие поддержки других типов синхронизации. В расширении подхода для этого используется cpa-оператор $compatible$. Так как проверка совместности использует различные типы анализа, включая анализ примитивов синхронизации, анализ предикатов и другие, такой способ является более точным, чем алгоритм Lockset. Это позволяет, например, добавить поддержку других примитивов синхронизации, реализовав соответствующий CPA. Если обнаружена пара совместных доступов к данным, которые образуют состояние гонки, проверка остальных доступов для данного идентификатора прекращается. Таким образом, для каждого идентификатора может быть выдано не более одного предупреждения о потенциальном состоянии гонки.

3.13.2. Идентификаторы доступа к памяти

Одним из основных вопросов, которые возникают при статическом поиске гонок: как определить, что доступ производится к одной и той же памяти. При описании формальной модели программы использовался простой императивный язык программирования, который поддерживает разделяемые данные, представ-

ленные только глобальными переменными. В реальном программном обеспечении используется большое количество операций с указателями, структурами и более сложными типами данных. Как уже отмечалось в обзоре, многие точные инструменты статической верификации, в частности инструменты, реализующие методы ограничиваемой проверки моделей, вообще не распространяют свои подходы на программы с указателями, ограничиваясь только глобальными переменными. В других инструментах, про которых уже говорилось в при обзоре, применяется анализ синонимов (англ. alias) Андерсена. В этом случае собираются так называемые may-алиасы для каждой переменной, то есть множество областей памяти (адресов), на которые может указывать данный указатель.

Для нашей задачи поиска состояний гонки в системном программном коде не подходит первый способ, так как основная работа с памятью производится по указателям. Поиск алиасов для каждого из нескольких тысяч указателей является слишком трудоемкой задачей. В итоге необходимо использовать некоторую разумную эвристику, которая позволит определить одну и ту же область памяти. В предалагаемом подходе для этого используется модель памяти BnB, которая разделяет всю память на непересекающиеся множества регионов. Далее будем называть их BnB-регионы. Каждый BnB-регион относится к одному типу данных или полю структуры, в случае если у него не брался адрес. Такая модель памяти имеет ряд ограничений. В первую очередь, она не полностью поддерживает адресную арифметику и кастирование, что накладывает некоторые дополнительные условия на применимость подхода. Кроме того, она может приводить к ложным предупреждениям об ошибках, в случае если два указателя одного типа никогда не указывают на одну и ту же память.

Основой предложенной эвристики стало наблюдение, что обычно указателями на один тип данных в системном программном обеспечении работают похожим образом. Так, например, в один и тот же указатель обычно записывают похожие данные, которые защищаются одинаковым способом. В этом случае не обязательно отслеживать с помощью точного анализа алиасов, на какую память конкретно может указывать некоторый указатель. И если к нему обращаются различными способами, например, с использованием примитивов синхронизации и без них, то это уже является достаточно подозрительным местом. При этом встречаются случаи, в которых такая эвристика дает ложные срабатывания, например, если имеются два доступа к различным объектам того же типа.

Отдельно имеет смысл подчеркнуть, что для простых типов BnB-регионы соответствуют типам. А для полей структур, у которых не брался адрес BnB-регион будет отдельный от того типа, к которому принадлежит это поле структуры. Такая оптимизация позволяет сократить число ложных предупреждений об ошибках из-за неравенства двух объектов. Действительно, если у поля структуры никогда не брался адрес, то доступ к этому полю возможен только снова с использованием имени этого поля. При этом не имеет значение, как происходит доступ к родительской структуре. Исключением может стать доступ к памяти с использованием адресной арифметики.

3.13.3. Оптимизации хранения данных

После построения множества достижимых абстрактных переходов программы необходимо вычислить все доступы к разделяемым данным, которые возможны на этих переходах. Эти доступы к данным сохраняются в глобальном контейнере, который затем будет обеспечивать эффективную работу с ними, в том числе поиск.

Описание доступа к данным

Для каждого доступа к данным формируется специальная структура данных, содержащая информацию об этом доступе. Будем использовать обозначение $usage$ для этой структуры данных. $usage = (usage\_core, states)$, где $usage\_core = (node, access, state, path, id)$. Здесь

- $node \in Loc$ – узел CFA, которому соответствует данный доступ;
- $access \in \{READ, WRITE\}$ – тип доступа;
- $state \in E$ – соответствующее абстрактное состояние из анализа, которое используется для восстановления пути;
- $states \in 2^E$ – множество состояний различных анализов, которые должны учитываться при определении состояния гонки;
- $path$ – найденный истинный путь к данному использованию;

– id – идентификатор переменной, описанный в подразделе 3.13.2.

Состояние $state$ является полным состоянием, которое включает в себя все абстрактные состояния вложенных CPA. А множество $states$ включает в себя абстрактные состояния только тех CPA, которые будут использоваться для проверки совместности. Состояния LockCPA, содержащие множество захваченных блокировок, используются для определения потенциальных состояний гонки, поэтому состояния этого анализа присутствуют во множестве $states$. В текущей конфигурации кроме состояний LockCPA используются состояния ThreadCPA.

$path$ содержит путь, ведущий из начального состояния программы к данному доступу. Путь состоит из набора CFA дуг, а не из абстрактных состояний. Это необходимо для эффективного расходования памяти. Дело в том, что из-за специфики сборщика мусора Java, вся память, на которую имеются ссылки, не может быть освобождена. Это приводит к тому, что любое сохраненное абстрактное состояние приводит к тому, что в памяти хранится полный граф достижимости. Сам по себе он занимает достаточно много места, а несколько его экземпляров (с учетом уточнения) могут привести к исчерпанию ресурсов. Поэтому приходится хранить небольшие объекты – CFA дуги, которые, тем не менее, позволят в случае необходимости воспроизвести путь.

Множество всех $usage$ будем обозначать, как $UsageSet$.

Определение доступов к данным

Технически определение возможных доступов к данным является несложной операцией. Для каждого перехода из множества достижимых состояний нужно повторить следующий набор шагов:

1. определить множество доступов к данным на этом переходе;
2. проверить, имеет ли смысл хранить полученный доступ;
3. для каждого релевантного доступа построить соответствующий $usage$;
4. дополнить построенный $usage$ утраченной из-за BAMCPA информацией, чтобы получить полное описание;
5. сохранить полное описание $usage$ для последующего поиска состояния гонки.

Основная проблема в этом алгоритме заключается в восстановлении полного описания. Как уже было описано ранее, для повышения эффективности BAM может удалять некоторую информацию из состояния с помощью операции $reduce$. Для LockCPA эта операция может удалять блокировки, если они не используются в текущем блоке. Тем не менее, эти блокировки являются существенными для поиска состояний гонки. В данный момент только LockCPA модифицирует свое состояние, но в дальнейшем возможно появятся и другие CPA. Поэтому для каждого найденного доступа необходимо восстановить всю утерянную информацию.

Именно поэтому был выделен отдельный элемент $usage\_core$. Для абстрактного перехода исходный $usage$ строится только один раз, а затем он может быть несколько раз модифицирован, в зависимости от того, сколько возможных путей, ведущих через различные абстрактные блоки было обнаружено в процессе анализа. Чтобы не копировать $usage$ целиком, все неизменяемые элементы выделены в отдельный блок, который не копируется при восстановлении.

Эффективный поиск состояний гонки

Одной из задач глобального контейнера является обеспечение быстрого поиска состояний гонки среди всех добавленных доступов. Если бы все найденные $usage$ хранились как обыкновенное множество, то перебор всех возможных пар занимал бы слишком большое время. Однако, условие совместности двух доступов, которое и необходимо проверить для окончательного вердикта, зависит только от информации в $states$. Еще одно требование – это хотя бы один доступ на запись. Таким образом, первой идеей является отказ от лишней информации при поиске состояний гонки. Поэтому на верхнем уровне контейнера для каждого идентификатора переменной содержится множество $point$. $point$ – это срез информации, содержащейся в $usage$, которая может повлиять на наличие состояние гонки: $states$ и $access$. Каждому $point$ может соответствовать несколько реальных $usage$. И в этом случае пространство поиска существенно сокращается. Таким образом, если будет найдено состояние гонки в терминах $point$, любое из соответствующих им $usage$ также будут образовывать состояние гонки.

Следующим шагом является исключение из поиска заведомо бессмысленных случаев. Если имеется $point$ с меньшим количеством блокировок, то его име-

ет смысл рассматривать в первую очередь. Например, $point_1$ содержит внутри $states$ состояние LockCPA, в котором находится только одна блокировка $lock_1$. А $point_2$ содержит две блокировки $lock_1$ и $lock_2$. Какой бы парный $point_3$ мы не взяли, имеет смысл рассматривать с ним только $point_1$, так как если существует состояние гонки с $point_2$, будет иметь место и состояние гонки с $point_1$. При этом возможны ситуации, в которых состояние гонки будет иметь место только в паре с $point_1$.

Конечно, такая оптимизация требует полного покрытия всех состояний из $states$. То есть, если состояния LockCPA покрываются, а состояния ThreadCPA несравнимы, то в этом случае отбросить соответствующий $point$ будет некорректным. Формально говоря, необходимо упорядочить $states$ и рассматривать только верхние элементы. При этом, конечно, таких элементов может быть несколько, так как используется лишь частичный порядок.

Пока мы не интересовались истинностью найденных состояний гонки. То есть, все описанные оптимизации основывались на том, что все скрытые $usage$ и $point$ равноценны оставшимся с точки зрения достижимости. Однако, не все найденные доступы возможны при реальном выполнении с учетом дополнительной информации $states$. Например, может оказаться, что доступ к данной переменной производится только под блокировками, так как соответствующий путь, ведущий к доступу к этой переменной без блокировок, является невыполнимым. В этом случае необходимо перейти к следующему пути, следующему $usage$ или следующему $point$. Подробнее сам процесс уточнения и различные его оптимизации будут представлены в подразделе 3.14.

В случае, если путь является достижимым, становится бессмысленно проверять другие пути, ведущие к данному $usage$. Более того, становится ненужным проверять и другие $usage$, соответствующие тому же $point$. Поэтому полученный достижимый путь сохраняется в поле $path$ у соответствующего $usage$, а соответствующий ему $point$ тоже помечается, как достижимый. Это необходимо для того, чтобы не проверять каждую итерацию уточнения те пути, для которых уже была доказана их достижимость. Действительно, на одной из итераций уточнения может возникнуть ситуация, при которой для одного доступа будет найден достижимый путь, а все остальные доступы, которые могли образовать с ним состояние гонки, будут отброшены, как недостижимые. После этого необходимо заново перестроить уточненную абстракцию, чтобы проверить, появятся ли эти доступы снова. Если доступы будут обнаружены, то к ним уже будут вести другие пути,

которые снова необходимо проверить на достижимость. При этом проверять достижимость первый путь уже будет не нужно, так как он должен существовать и в более точной абстракции. Если для некоторой переменной были найдены два истинных пути, которые образуют состояние гонки, значит, больше нет необходимости собирать информацию для этой переменной. Она помечается специальным образом и больше не участвует в уточнении.

Пример представления данных

Рассмотрим такой фрагмент исходного кода.

```
int global;
...
int dummy_function(){
  int local;
  ...
  global = 1;
  mutex_lock(&mutex);
  local = global;
  mutex_unlock(&mutex);
  ...
  mutex_lock(&mutex);
  local = local + global;
  mutex_unlock(&mutex);
}
```

Часть контейнера для этого фрагмента программы представлена на рисунке 3.5.

Переменной *global* соответствуют две точки использования: запись без использования примитивов синхронизации и чтение при захваченной mutex-блокировке. Причем второй точке использования соответствуют два реальных доступа, расположенных в разных строках исходного кода.

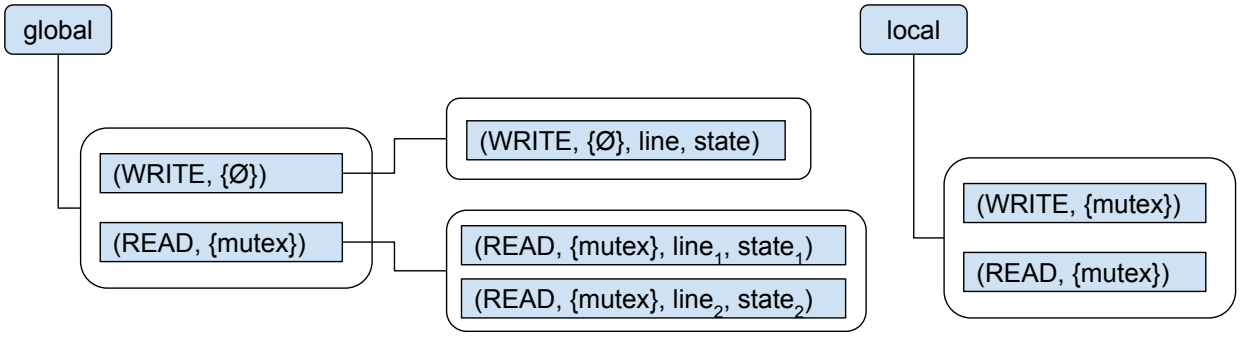

Рисунок 3.5 — Пример хранения данных в контейнере

3.14. Реализация уточнения при поиске состояний гонки

3.14.1. Общее устройство

Уточнение абстракции по контрпримерам используется для того, чтобы исключить недостижимые пути из абстракции. В классическом варианте работы CPAchecker CEGAR используется при решении задачи достижимости. В случае, если найдено ошибочное состояние, начинается процесс уточнения: строится контрпример и проверяется, возможен ли такой сценарий выполнения в исходной программе. Если такой путь является недостижимым из-за неточности абстракции, она уточняется таким образом, чтобы исключить такой путь из абстракции.

При поиске состояний гонки дело усложняется тем, что ошибкой является не одно состояние, а пара. При этом каждый из путей сам по себе не является ошибкой. Кроме того, обычно требуется обнаружить не первое состояние гонки в программе, а все потенциальные состояния гонки. Таким образом, возможны две вариации процедуры уточнения абстракции.

1. Уточнение производится в процессе анализа. При обнаружении пары состояний, составляющих состояние гонки, оба пути проверяются на достижимость. Если найденная ошибка подтверждается, она отмечается, как подтвержденная, и анализ продолжается. Этот вариант уточнения очень похож на классический вариант: при обнаружении ошибки абстракция уточняется до тех пор, пока ошибка либо не подтвердится, либо не опровергнется. Такой подход обладает существенным недостатком: для корректного завершения анализа, требуется построить очень точную

абстракцию. В случае, если анализируемый код содержит сотни тысяч строк кода, построение точной абстракции требует колоссального времени. Более эффективно в этом случае гибко задавать ограничения на ресурсы, чтобы иметь возможность завершить анализ в любой момент, хотя это и повлечет за собой некоторое снижение точности.

2. Уточнение всех путей производится в тот момент, когда абстракция полностью построена. Все обнаруженные состояния гонки проверяются на истинность, и, в случае необходимости, абстракция перестраивается. Существенным недостатком этого подхода является большой объем лишней работы в том случае, если неточность абстракции затрагивает множество состояний гонки. Тогда проверка каждой из них будет давать один и тот же результат. Однако, возможно применение некоторых оптимизаций, которые позволяют сократить время на такую бесполезную работу.

Для того, чтобы обеспечить гибкую настройку, процесс уточнения был разделен на функциональные блоки. Каждый такой блок определяется тем, что он принимает себе на вход от предыдущего блока, и тем, что он выдает на выходе следующему. Результатом работы каждого из абстрактных блоков является вердикт $true$, означающий, что то, что передается на вход этому блоку, содержит состояния гонки, и вердикт $false$, означающий, что состояние гонки обнаружить не удалось. В последнем случае может быть выдана некоторая информация (precision), которая позволит более точно вычислить абстракцию на следующей итерации анализа. В редких случаях возможен вердикт $unknown$, означающий, что однозначный ответ не может быть получен. В большинстве случаев такой вариант реализуется при некорректной работе сторонних компонентов, таких как решателей (англ. solver).

На рисунке 3.6 представлен возможный вариант последовательности функциональных блоков при уточнении.

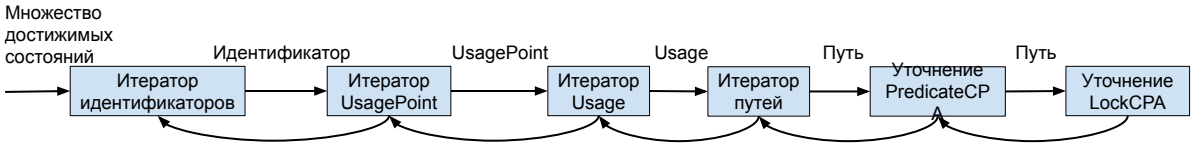

Рисунок 3.6 — Последовательность функциональных блоков при уточнении

Цепочка блоков уточнения состоит из двух частей. Первая часть - служебная, которая используется для подготовки путей к уточнению и обработке полученной от других блоков информации. Эта часть содержит:

1. IdentifierIterator. Блок, принимающий на вход ReachedSet и последовательно перебирающий все возможные переменные, к которым был доступ. Соответственно, следующий за ним блок обязан принимать на вход идентификатор переменной. В случае, если хотя бы для одной переменной был получен вердикт $false$, это означает, что уточнение прошло успешно, и необходимо перестроить абстракцию. Если же для всех новых переменных вердикт был $true$ или $unknown$, это означает, что абстракция построена с достаточным уровнем точности, и все найденные состояния гонки являются истинными с точки зрения анализа.

   Каждый вердикт $false$ сопровождается некоторой информацией о том, как следует уточнить абстракцию. Этот уровень точности сохраняется для каждой переменной отдельно, но при построении абстракции учитывается уровень точности, полученный для каждой из переменных. Однако, в случае если для какой-либо переменной будет доказано, что она участвует в истинном состоянии гонки, то соответствующий этой переменной уровень точности сбрасывается. Это помогает не перестраивать слишком точную абстракцию тогда, когда это уже не нужно.
2. PointIterator. Блок, принимающий на вход идентификатор переменной и перебирающий все возможные пары $point$, которые образуют состояние гонки. Каждая пара $point$ проверяется следующим блоком. Если для некоторой пары $point$ будет получен вердикт $true$, это означает, что ни один следующий блок не смог найти противоречия, а значит, полученное состояние гонки истинное, и в предыдущий блок возвращается результат $true$.
3. UsageIterator. Блок, принимающий на вход пару $point$ и перебирающий все возможные пары $usage$, соответствующие этой паре $point$. Каждая пара $usage$ проверяется следующим блоком. Если для некоторой пары $usage$ будет получен вердикт $true$, это означает, что ни один следующий блок не смог найти противоречия, а значит, полученное состояние гонки истинное и в предыдущий блок возвращается результат $true$.
4. PathIterator. Блок, принимающий на вход пару $usage$ и перебирающий все возможные пары путей в абстракции, соответствующие этой паре $usage$.

Необходимо пояснить, почему в абстракции возможно несколько путей из начального состояния к тому состоянию, которое соответствует данному $usage$. Дело в том, что из-за применении оптимизации BAM (подраздел 3.4), тело функции будет проанализировано один раз, несмотря на то, что вызовов этой функции может быть несколько. В таких случаях для каждого вызова функции возможно несколько точек ее вызова. Если для некоторой пары путей будет получен вердикт $true$, это означает, что ни один следующий блок не смог найти противоречия, а значит, полученное состояние гонки истинное и в предыдущий блок возвращается результат $true$.

Далее идет вторая часть цепочки, которая занимается непосредственно проверкой корректности двух путей. Поэтому каждый из этих блоков принимает на вход и передает в следующий блок пару путей. Если текущий блок считает, что данная пара путей невозможна, в предыдущий блок возвращается результат $false$. Блоки уточнения из этой части не являются обязательными, поэтому допустима любая их комбинация.

1. PredicateRefiner. Основной инструмент для удаления из абстракции локально-недостижимых путей. Использует внутри себя классический алгоритм уточнения для предикатного анализа, но дополнительно применяется некоторый набор оптимизаций для того, чтобы сократить время работы. Например, при переборе всех возможных пар путей может часто возникать проверка отдельного пути на локальную-достижимость. Чтобы избежать лишних проверок, сохраняется результат каждого уточнения для пути, а также тот уровень точности, который нужен для исключения данного пути из абстракции. Таким образом, если возникнет необходимость в уточнении этого же пути, не важно на этой же итерации уточнения или на любой из последующих, будет использован сохраненный результат.

   Возможны различные варианты реализации процесса уточнения. Например, использование только локальных переменных в формуле. Это будет означать, что значения глобальных переменных не будет учитываться, что приведет к потере точности, однако количество итераций уточнение будет значительно снижено. В другом варианте реализации уточнения возможные эффекты окружения могут быть учтены в процессе уточнения, а не на следующей итерации анализа. Возможны и другие, более

сложные варианты, однако в таких случаях необходимо аккуратно соблюдать баланс эффективности. Так, слишком сложный процесс уточнения может слишком редко получать дополнительную точность, а значит, наиболее эффективным алгоритмом становится использование простого уточнения и дополнительной итерации построения абстракции.

2. LockRefiner. Анализ примитивов синхронизации (LockCPA) может отслеживать все состояния блокировки без применения уточнения. Причем обычно каждый захват блокировки делается для того, чтобы исключить некоторое потенциальное состояние гонки. Однако, для больших программных систем не обязательно учитывать все блокировки каждый раз при пересчете абстракции. Если доказано, что данная переменная не может участвовать в состоянии гонки, можно исключить из рассмотрения те захваты блокировки, которыми она защищалась, что позволит сократить время на следующий пересчет абстракции.
3. Фильтр. Фильтром называется такой блок, который, выдавая результат FALSE, не предоставляет уровень точности для того, чтобы исключить полученный путь (пару путей) из абстракции. А это значит, что такая же пара путей будет получена и для следующей абстракции. Такой фильтр, тем не менее, полезен тем, что при переборе всех подходящих путей может быть найдена такая пара, которая будет соответствовать истинному состоянию гонки.

3.14.2. Оптимизации

Основной проблемой при уточнении множества путей в одной абстракции является неизбежное повторение получаемых противоречий. Например, если в абстракции не учитывается некоторое значение переменной в условном операторе, в этом случае все проходящие пути через этот условный оператор будут содержать это противоречие и, следовательно, будут недостижимы по одной и той же причине. Было бы эффективнее избегать уточнения тех путей, которые проходят через точки программы, для которых были установлены новые факты (precision) ранее. В качестве точек программы можно использовать:

– узлы CFA;

– состояния ARG.

В первом случае критерий становится более слабым, что позволяет отложить уточнения большего числа путей. Стоит отдельно заметить, что отложенные пути не выбрасываются из абстракции. Таким образом, даже в случае неверно принятого решения из-за слабости критерия, они не будут полностью выброшены, а появятся снова на следующей итерации уточнения.

Однако, восстановление пути тоже занимает достаточно большое количество времени. Поэтому полностью строить путь для того, чтобы оценить, через какие точки программы он проходит, становится неэффективно. Приходится в процессе восстановления пути проверять, не проходит ли уже промежуточный отрезок пути через какие-нибудь повторяющиеся точки программы.

3.15. Печать и визуализация

3.15.1. Общая схема визуализации

Визуализация найденных состояний гонок является важным моментом для практического применения инструмента. Выдаваемой информации должно быть достаточно для того, чтобы разработчик смог понять, какие именно действия в программе могут привести к состоянию гонки. Таким образом, основными требованиями к визуализации являются:

1. Отображение всех операторов, которые встречаются на пути к каждому из доступов, образующих состояние гонки.
2. Отображение и визуальное выделение двух доступов, образующих состояние гонки.
3. Отображение и визуальное выделение всех операций с примитивами синхронизации.
4. Отображение справочной информации о состоянии гонки: множество захваченных примитивов синхронизации, тип и имя разделяемой переменной.

Основой формата вывода найденных состояний гонки является graphml формат, описанный в [92–95]. Этот формат предполагает преставление пути к

ошибке в виде графа, который описывает путь от начального состояния программы до состояния, соответствующего ошибке. Ребра графа соответствуют дугам графа потока управления.

Путь в графе имеет линейную структуру, хотя формат позволяет описывать различные ветвления, например, циклы, которые удобнее визуализировать, не разворачивая все выполненные итерации. Так как разработанный инструмент реализует метод с раздельным рассмотрением потоков, то мы не может однозначно указать порядок выполнения инструкций (чередование потоков). Лишь те места, где имеется применения эффектов окружения, могут интерпретироваться как возможные переключения потоков. Поэтому в визуализируемом пути выполнения все инструкции каждого потока идут последовательно друг за другом. Главный поток выполняется до точки, в которой происходит разделение путей, ведущих к двум доступам к памяти. Обычно, это происходит при создании некоторого потока. После этого продолжается первый путь, который приводит к первому доступу к памяти. Далее отображаются операторы второго пути, начинающиеся с той точки разветвления. Таким образом для пользователя будет выведено сначала трасса, ведущая к первому доступу, а затем, трасса ко второму доступу.

Пример визуализированной трассы представлен на рисунке 3.7.

Визуализация graphml представления производится инструментом Klever.

3.15.2. Оптимизации при визуализации

Визуализация только истинных ошибок

Как уже было отмечено в соответствующем разделе, реализация процесса уточнения после построения абстракции позволяет завершить анализ в любой момент времени и выдать все найденные предупреждения. При этом полученный результат будет аппроксимацией сверху потенциально возможного результата при бесконечном количестве времени. То есть все потенциальные ошибки будут найдены при первом же построении абстракции, а все остальное время будет тратиться на то, чтобы исключить ложные с точки зрения достижимости пути.

Рисунок 3.7 — Пример визуализированного предупреждения

Таким образом, для удобства анализа экспертом можно выдавать только те ошибки, которые были подтверждены при уточнении. При этом, конечно, будет возможен пропуск реальной ошибки, если она не успела подтвердиться, однако такой режим и не подразумевает доказательство корректности программы.

Фильтрация одинаковых путей

Зачастую работа в критической секции под блокировками содержит несколько операторов. Например, даже простая операция занесения элемента в список требует модификации поля добавляемого элемента и поля элемента в списке. В случае некорректной синхронизации, скорее всего, будут получены предупреждения для обоих доступов к памяти. Однако, причина ошибки у них одна и та же, поэтому нет смысла анализировать эти предупреждения несколько раз.

Для этого используется следующая эвристика. Для каждого доступа к данным формируется идентификатор: функция и множество состояний $states$ из $usage$. Если для некоторой пары доступов, образующей состояние гонки, соответствующая им пара идентификаторов уже была выдана ранее для некоторого другого предупреждения, это означает, что такое состояние гонки является повторяющимся и может быть отфильтровано.

Глава 4. Экспериментальная оценка предложенного метода

4.1. Общая схема проведения экспериментов

Основными целями проведения экспериментов являются:

1. сравнение с ведущими инструментами в области поиска состояний гонки и верификации многопоточных программ;
2. оценка различных конфигураций инструмента.

Сравнение с ведущими инструментами статической верификации будет проводиться на наборе тестовых задач SV-COMP[1]. Категория *ConcurrencySafety* состоит из 1082 задач, большая часть из которых являются небольшими примерами около 100 строк кода Для всех задач требуется доказать достижимость некоторого ошибочного состояния. В основном, используются т.н. выражения *assert*, и требуется доказать, что условия в них не нарушаются. Этот набор задач содержит редко встречающиеся сложные конструкции, например, нетривиальные механизмы синхронизации, в том числе алгоритмы Деккера, Петерсона и др. 7 задач были подготовлены на основе драйверов ОС Linux. Все задачи доступны в официальном репозитории SV-COMP[2].

Сравнение различных конфигураций инструмента будет проводиться на трех множествах задач: набор тестовых задач SV-COMP, набор задач, подготовленных на основе драйверов операционной системы Linux, и задачи, подготовленные на основе ядер закрытых операционных систем реального времени. Верификационные задачи, основанные на драйверах операционной системы Linux, были подготовлены системой Klever, которая предназначена для верификации различного программного обеспечения [96], [97]. Она разделяет большой объем целевого исходного кода на отдельные небольшие верификационные задачи. Для ядра операционной системы Linux верификационная задача соответствует одному модулю. Система Klever автоматически готовит модель окружения модуля, которая включает в себя модель потоков, модель сердцевины ядра и операций над модулем. После подготовки верификационной задачи Klever запускает верификацию через общий интерфейс – BenchExec [98]. Сравнение проводилось на подсистеме

[1] https://sv-comp.sosy-lab.org/2020/

[2] https://github.com/sosy-lab/sv-benchmarks

drivers/net/ ядра операционной системы Linux 4.2.6, для которой Klever подготовил 425 верификационных задач.

Верификационные задачи на основе ядра операционных систем реального времени были подготовлены для двух закрытых операционных систем. Для этого были вручную выделены и закодированы те активности, которые могут выполняться параллельно при реальной работе ОС, в том числе, обработчики прерываний, системные вызовы, системные потоки и т.д. Одна задача на основе ядра ОС РВ занимает более пятисот тысяч строк кода и содержит 497 потоков, а другая – 350 тысяч строк кода и 37 потоков. Так как эти задачи содержат большое количество предупреждений, в том числе истинных, в итоге при любой конфигурации становится невозможным доказать корректность. Таким образом, для таких задач становится бессмысленным сравнивать общий вердикт, поэтому далее будет проводится сравнение количества найденных предупреждений, а также общего времени работы инструмента.

Эксперименты на наборе задач SV-COMP и наборе драйверов проводились с использованием кластера из 191 машины VerifierCloud[3]. В нем были выбраны машины с одним типом процессора Intel Xeon E3-1230 v5, 3.40 GHz. Были использованы ограничения по памяти в 8 Гб и по времени 15 минут. Эксперименты на задачах, основанных на ядрах операционных систем реального времени были проведены на локальных машинах Intel Core i5-2500, 3.30 GHz. Для проведения экспериментов была использована реализация инструмента CPALockator, которая доступна в репозитории https://svn.sosy-lab.org/software/cpachecker/branches/CPALockator-combat-mode, ревизия 32869.

Оценка различных конфигураций анализа будет проводиться для следующих настроек:

- PredicateCPA.
  - Варианты реализации cpa-оператора $merge$:
    - Join;
    - Eq;
    - Sep;
  - Оптимизация ABE (раздел 3.9.2).
  - Оптимизация с присваиванием неопределенных функций (раздел 3.9.3).

[3]https://vcloud.sosy-lab.org/cpachecker/webclient/master/info

- Оптимизация с применением только релевантных эффектов (раздел 3.9.3);
- Игнорирование разделяемых данных в формуле пути (раздел 3.9.3);
- Использование локального уточнения (раздел 3.9.4);

– ThreadCPA.
- Варианты теории, лежащей в основе.
  * Простой вариант инвариантный к эффектам окружения (раздел 2.8).
  * Вариант с эффектами окружения (раздел 2.9).
  * Сложный вариант инвариантный к эффектам окружения (раздел 2.10).
- Варианты обработки повторно создаваемого потока (раздел 3.8.2).
  * Падение.
  * Игнорирование повторно создаваемого потока.
  * Абстракция от повторно создаваемого потока (создание самопараллельного потока).

– LockCPA.
- Варианты реализации cpa-оператора $merge$:
  * Join;
  * Sep;
- Варианты реализации cpa-операторов $reduce/expand$ (раздел 3.6.5):
  * Использование абстракции от счетчика рекурсивных захватов блокировки.
  * Использование абстракции от неиспользуемых примитивов синхронизации.
- Использование уточнения.

– Использование анализа разделяемых данных (раздел 3.12).

– Использование оптимизации BAM (раздел 3.4).

– Использование анализа предикатов (раздел 3.9).

Как уже было сказано, тестовые программы из набора SV-COMP содержат задачу достижимости, а для программ, построенных на основе драйверов и ядер ОС РВ, решается задача поиска состояний гонки. Несмотря на то, что основная

структура инструмента, в том числе дерево CPA, остается неизменным в обоих случаях, тем не менее, общая конфигурация требует изменений. Кроме того, некоторые из оптимизаций накладывают дополнительные ограничения на используемую конфигурацию инструмента. В частности, оптимизация BAM, требует использования CPA, инвариантных к окружению. Другие опции, наоборот, определяют варианты работы с эффектами окружения, то есть, невозможно использовать единственную начальную конфигурацию инструмента для сравнения всех опций. Чтобы отличать различные базовые конфигурации инструмента, далее будем использовать обозначение *CPALockator-Reach* для обозначения конфигурации инструмента для решения задачи достижимости, *CPALockator-TM* – для обозначения конфигурации инструмента с эффектами окружения для решения задачи поиска гонок, и *CPALockator-Inv* – для обозначения конфигурации инструмента, инвариантного к эффектам окружения, для решения задачи поиска гонок. Таким образом влияние части опций будет оценено на нескольких конфигурациях, а там, где это невозможно, будет указан тот вариант конфигурации инструмента, который поддерживает ту или иную опцию.

В следующих разделах будут представлены экспериментальные результаты для оценки влияния той или иной опции. После проведения всех экспериментов будут подведены итоги и сделаны выводы по результатам экспериментов.

4.2. Сравнение различных инструментов статической верификации

Для проведения экспериментов были выбраны следующие инструменты:

- PredicateTM. Реализация подхода с раздельным рассмотрением потоков с использованием анализа предикатов. Включает в себя следующие CPA: ThreadModularCPA, ARGCPA, CompositeCPA, LocationCPA, CallstackCPA, LockCPA, ThreadCPA, PredicateCPA.
- ValueTM. Реализация подхода с раздельным рассмотрением потоков с использованием анализа явных значений. Включает в себя следующие CPA: ThreadModularCPA, ARGCPA, CompositeCPA, LocationCPA, CallstackCPA, LockCPA, ThreadCPA, ValueCPA.

– Threading. Реализация классического варианта анализа с чередованиями потоков [99]. Реализован в том же фреймворке CPAchecker, но с использованием классической версии теории.
– Yogar-CBMC. Победитель соревнования SV-COMP'19, реализует подход ограничиваемой проверки моделей с некоторыми оптимизациями.
– Lazy-CSeq. Серебряный призер соревнования SV-COMP'19, реализует подход секвенциализации программы.

Результаты экспериментов представлены в таблице 4.2. В ней представлена информация о том, сколько у каждого инструмента было выдано сообщений об ошибке (False вердикт), для скольких тестовых примеров было доказано отсутствие ошибок (True вердикт) и для скольких инструмент не смог определить вердикт.

Таблица 1 — Запуск на наборе задач SV-COMP

	CPALockator-Reach		Другие инструменты		
Подход	PredicateTM	ValueTM	Threading	Yogar-CBMC	Lazy-CSeq
Вердикт «ошибка»	1028	241	727	773	811
из них корректных	805	59	727	773	811
из них некорректных	223	182	0	0	0
Вердикт «нет ошибки»	21	20	165	284	256
из них корректных	21	20	165	284	256
из них некорректных	0	0	0	0	0
Анализ не завершен	33	821	190	25	15
Время CPU (с)	23 700	38 200	111 000	7 000	29 000

Общие выводы из полученных результатов являются следующими.

Результаты подтверждают, что подход с раздельным анализом потоков не пропускает ошибок при некоторых заранее известных ограничениях.

Конфигурация ValueTM является достаточно простым и быстрым анализом, но иногда она вынуждена рассматривать все возможные варианты значений переменных, что приводит к исчерпанию ресурсов по времени. В данном эксперименте был использован вариант анализа без уточнения, что привело к значительному повышению количества тестовых примеров, анализ которых был незавершен за отведенный лимит времени. Так как данный анализ является достаточно простым и используется, в основном, на модельных примерах, то реализация полноценного варианта с уточнением не была приоритетной задачей.

Классический анализ Threading является корректным и точным и не выдает ложных вердиктов. Однако, он требует значительного числа ресурсов, это является главным недостатком подхода. Эта конфигурация решила только один из семи сложных задач, основанных на драйверах операционной системы Linux. Подход с раздельным анализом потоков (PredicateTM) решает пять из семи таких задач.

Большая часть новых доказательств корректности, полученных подходом с раздельным рассмотрением потоков, (26 из 27 для PredicateTM) не находились классическим методом (Threading). Это также является одним из важных вкладов данного метода.

Подход с раздельным рассмотрением потоков выдает большое количество ложных предупреждений об ошибке. Большинство из них связаны с неподдерживаемыми атомарными конструкциями, такими как 'compare and swap' *__VERIFIER_atomic_CAS*. Еще в некоторых случаях данный подход не способен определить отношение happens-before между созданием потока (дочерний поток не может работать одновременно с родительским до точки своего создания). Текущие ограничения процедуры уточнения не позволяют определять и расставлять интерполянты на поток, исходный для эффекта окружения. В небольшой части случаев требовалось точное рассмотрение переключений между потоками.

Инструменты, основанные на других подходах, показывают отличные результаты почти на всех тестовых примерах, но ни один из них не справился со сложными задачами, основанными на драйверах ОС Linux. Таким образом, это подтверждает тот факт, что для решения задач, основанных на реальных программных системах, такие инструменты не подходят.

4.3. Сравнение различных вариантов анализа предикатов

4.3.1. Сравнение различных реализаций cpa-оператора merge

Для проведения экспериментов были использованы следующие конфигурации:

– MergeJoin. Реализация cpa-оператора $merge_{Join}$.

– MergeEq. Реализация cpa-оператора $merge_{Eq}$.

– MergeSep. Реализация cpa-оператора $merge_{Sep}$.

Результаты сравнения на множестве задач SV-COMP представлены в таблице 2.

Таблица 2 — Сравнение на наборе задач SV-COMP

	CPALockator-Reach		
Подход	MergeJoin	MergeEq	MergeSep
Вердикт «ошибка»	1028	1028	977
из них корректных	805	805	762
из них некорректных	223	223	215
Вердикт «нет ошибки»	21	21	31
из них корректных	21	21	31
из них некорректных	0	0	0
Анализ не завершен	33	33	74
Время CPU (с)	23 700	24 600	68 100

MergeJoin показывает результаты лучше, чем конфигурация MergeSep. Это происходит, в основном, из-за большого количества переходов в окружении, которые MergeSep рассматривает по одному, что приводит к большому количеству проверок совместности, вызовов cpa-операторов $stop$ и $transfer$. MergeJoin объединяет все эффекты в один и применяет за один раз. Это позволяет сохранить огромное количество времени. Однако, это приводит к небольшому падению точности даже на таких искусственных примерах.

Два лишних некорректных результата были получены из-за того, что в данной конфигурации отсутствует падение инструмента при уточнении, которое присутствует в первых двух конфигурациях. Небольшое отличие времен работы MergeJoin и MergeEq объясняется некоторыми флуктуациями работы внешнего компонента – решателя. Например, в некоторых примерах при одинаковом количестве уточнений построение абстракции для MergeEq занимает меньше времени, чем для MergeJoin.

Результаты экспериментов на множестве задач *drivers/net/* представлены в таблице 3.

Данные результаты подтверждают основные выводы, полученные в предыдущем пункте: MergeJoin и MergeEq требуют меньше времени для получения вердикта, а использование конфигурации MergeSep не приводит к более корректным результатам. Значительно меньшая разница в затраченном времени между конфигурациями объясняется большим количеством таймаутов, которые возникают для всех трех конфигураций. Если же вычислить затраченное время для полу-

Таблица 3 — Сравнение на наборе задач *drivers/net/*

	CPALockator-TM		
Подход	MergeJoin	MergeEq	MergeSep
Вердикт «ошибка»	8	7	5
из них корректных	8	7	5
из них некорректных	0	0	0
Вердикт «нет ошибки»	261	260	259
из них корректных	261	260	259
из них некорректных	0	0	0
Анализ не завершен	156	158	161
Время CPU (с)	139 000	139 000	143 000

ченных вердиктов (за исключением задач, для которых анализ не был завершен), в этом случае картина будет более полярной. Кроме того, следует помнить, что незавершенный анализ не всегда означает именно исчерпание отведенного времени, иногда это могут быть падения инструмента, которые могут случиться на первых секундах.

Основным выводом данного пункта является тот факт, что конфигурация MergeJoin является наиболее оптимальной для большинства задач: как искусственных, так и приближенных к реальным программам. Это объясняется тем, что при проверке многопоточной программы к ошибке обычно приводит ситуация, при которой имеет место одновременный доступ к переменной, при этом уже становится неважным, какие еще области памяти были изменены вместе с ней. Таким образом, перебирать все возможные изменения памяти по одному, как это делает MergeSep, становится бессмысленно.

4.3.2. Сравнение влияния оптимизаций

Для проведения экспериментов были использованы следующие конфигурации:

- Base. Базовая реализация предикатного анализа с отключенными оптимизациями, в том числе использование SBE (англ. Single Block Encoding).
- Undef. Оптимизация с присваиванием неопределенного значения.
- Relevance. Оптимизация с применением только релевантных эффектов.
- ABE. Оптимизация ABE.

– Havoc. Игнорирование разделяемых данных в формуле пути.
– Imprecise. Использование локального уточнения.

Результаты сравнения на множестве задач SV-COMP представлены в таблице 4.

Таблица 4 — Сравнение оптимизаций на наборе SV-COMP

	CPALockator-Reach					
Подход	Base	Undef	Relevance	ABE	Havoc	Imprecise
Вердикт «ошибка»	1027	1033	1025	1026	1052	1052
из них корректных	802	805	802	803	808	808
из них некорректных	225	228	223	225	244	244
Вердикт «нет ошибки»	21	21	23	21	5	2
из них корректных	21	21	23	21	5	2
из них некорректных	0	0	0	0	0	0
Анализ не завершен	34	28	34	33	25	28
Время CPU, с	29 900	21 300	29 200	25 700	11 600	13 700

Оптимизация Undef, как и следовало ожидать, позволяет достаточно сильно сократить затрачиваемое на анализ время, но это приводит не только к новым корректно найденным ошибкам, но и к ложным предупреждениям. Ложные предупреждения возникают как раз из-за того, что мы абстрагируемся от конкретных значений, которые могут быть записаны в соответсвующие разделяемые переменные.

Оптимизация Relevance практически ничего не дает, а небольшое изменение времени находится в рамках погрешности. Это связано с тем, что данная оптимизация показывает себя в тех случаях, когда используется достаточно большой набор предикатов для построения абстракции. В этом случае, вычисление тех предикатов, которые являются релевантными при изменении окружения будет иметь смысл. Для искусственных примеров SV-COMP обычно требуется лишь несколько предикатов, чтобы отсечь или подтвердить ошибку.

Оптимизация ABE заметно повышает скорость анализа, хотя и не так сильно как оптимизация Undef. Однако, в отличие от нее, ABE не приводит к некорректным результатам. Такой результат также является закономерным.

Оптимизации Havoc и Imprecise, по сути, являются двумя реализациями одной и той же идеи: исключение из абстракции разделяемых переменных. Havoc исключает разделяемые переменные еще на этапе построения формулы в PredicateCPA, а Imprecise – на этапе уточнения. Собственно, результаты подтверждают, что исключение данных на более ранних этапах является эффективным.

Однако, что и следовало ожидать, предположение, что все разделяемые данные могут принимать любые значения, не позволяет доказать корректность нескольких примеров, что увеличивает количество ложных предупреждений об ошибках. Положительным эффектом является обнаружение корректных ошибок в тех примерах, которые раньше приводили к исчерпанию времени в более точных конфигурациях.

Результаты сравнения на множестве драйверов *drivers/net/* представлены в таблицах 5.

Таблица 5 — Сравнение на наборе задач *drivers/net/*

	CPALockator-TM					
Подход	Base	Undef	Relevance	ABE	Havoc	Imprecise
Вердикт «ошибка»	5	5	5	7	17	16
из них корректных	5	5	5	7	13	12
из них некорректных	0	0	0	0	4	4
Вердикт «нет ошибки»	258	258	258	260	255	254
из них корректных	258	258	258	260	255	254
из них некорректных	0	0	0	0	0	0
Анализ не завершен	162	162	162	158	153	155
Время CPU, с	121 000	116 000	117 000	97 400	96 800	107 000

Интересное отличие от предыдущего пункта заключается в том, что оптимизация Undef не влияет на вердикты. Это объясняется тем, что в драйверах ОС Linux редко встречается присваивание явных значений во внутренние структуры. В основном, это может быть инициализация, которая обычно происходит в начале работы в однопоточном режиме. А все штатные сценарии работы драйвера манипулируют внутренними данными, то есть, вычислить явное их значение при статическом анализе не удается. Это означает, что почти все эффекты окружения и так записывают в разделяемую память неопределенное значение, то есть оптимизация Undef теряет смысл.

Оптимизация Relevance в этом наборе более заметна, хотя и не позволяет получить новые вердикты. Это связано с тем, что данный набор задач является более сложным, и в некоторых случаях требуется большое количество уточнений, чтобы получить вердикт. В том случае, если было накоплено большое множество предикатов, данная оптимизация способна ускорить работу.

Для достаточно длинных ошибочных трасс оптимизация ABE вносит более весомый вклад, так как блоки получаются более длинные за счет большого коли-

чества работы с локальными данными, чего не было в искусственных примерах SV-COMP. Это позволяет даже получить несколько дополнительных вердиктов.

Оптимизации Havoc и Imprecise позволяют найти несколько дополнительных ошибок за счет своей скорости, однако приводит к появлению некоторых ложных предупреждений. Тем не менее, стоит заметить, что ее точности хватает, чтобы доказать отсутствие ошибок у большого множества задач. Таким образом, в некоторых случаях имеет смысл жертвовать точностью, отказываясь от точного рассмотрения разделяемых данных для того, чтобы получить новые вердикты.

Сравнение оптимизации ABE на конфигурации $CPALockator - Inv$ не проводилось, так как в этом случае бы, по сути, использовалась бы исходная реализация ABE, то есть без каких-либо модификаций, а значит, никакой научной новизны такие результаты не представляют.

4.4. Сравнение различных вариантов реализации ThreadCPA

4.4.1. Сравнение различных подходов

Для проведения экспериментов были использованы следующие конфигурации:

– Simple. Простой вариант инвариантный к эффектам окружения.
– Env. Вариант с эффектами окружения.
– Base. Расширенный вариант инвариантный к эффектам окружения.

Результаты сравнения представлены в таблице 6.

Все конфигурации демонстрируют примерно одинаковые результаты. Как и ожидалось, конфигурация Simple является самой слабой, что не позволяет доказать корректность нескольких задач. Конфигурация Base немного быстрее из-за того, что она не рассматривает переходы в окружении, как Env, и является более точной, чем Simple, что также сокращает количество состояний.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 4.4.1.

Реализация Simple не может точно определить, что некоторые эффекты являются несовместным, и из-за этого выдает лишние предупреждения. При этом, такая простота реализации никак не помогает при поиске дополнительных оши-

Таблица 6 — Сравнение на наборе SV-COMP

	CPALockator-Reach		
Подход	Simple	Env	Base
Вердикт «ошибка»	982	973	972
из них корректных	793	793	793
из них некорректных	189	180	179
Вердикт «нет ошибки»	9	21	19
из них корректных	9	21	19
из них некорректных	0	0	0
Анализ не завершен	91	88	91
Время CPU, с	26 800	25 100	21 500

Таблица 7 — Сравнение на наборе *drivers/net/*

	CPALockator-TM			CPALockator-Inv	
Подход	Simple	Env	Base	Simple	Base
Вердикт «ошибка»	120	9	10	147	38
из них корректных	10	9	10	28	29
из них некорректных	110	0	0	119	9
Вердикт «нет ошибки»	150	262	262	145	256
из них корректных	150	262	262	145	256
из них некорректных	0	0	0	0	0
Анализ не завершен	155	154	152	133	131
Время CPU, с	136 000	136 000	133 000	111 000	109 000

бок, и время анализа, наоборот, увеличивается, так как много времени тратится на применение большого количества эффектов окружения.

Реализация Env с использованием эффектов окружения потока демонстрирует примерно такие же результаты, как и Base, так как обычно эффектов создания/удаления потока небольшое количество, и их присутствие не усложняет анализ.

Результаты сравнения реализаций ThreadCPA на ОС РВ представлены в таблице 8.

Таблица 8 — Сравнение на ядрах ОС РВ

	CPALockator-Inv			
	ОС РВ 1		ОС РВ 2	
Подход	Simple	Base	Simple	Base
Число предупреждений	461	461	981	981
Общее время CPU, с	534	582	887	886
время ThreadCPA, с	0,11	0,17	0,148	0,213

В этом случае играет определенную роль свойство исходного кода. Дело в том, что для ядра ОС РВ искусственно подготавливатся код, который создает

все требуемые потоки, которые при реальном выполнении ОС могут быть запущены после каких-нибудь событий, например, после возникновения прерывания. Все выделенные потоки создаются последовательно в одной функции, и в этом случае даже простой анализ потоков способен справиться, так как доступов к разделяемым данным до создания потока нет.

Основной вывод заключается в том, что в данном случае нет определенно бесполезных конфигураций. Env и Base обеспечивают корректные результаты на любых примерах, а конфигурация Simple может использоваться тогда, когда известны дополнительные свойства исходного кода.

4.4.2. Сравнение различных вариантов обработки повторно создаваемого потока

Для проведения экспериментов были использованы следующие конфигурации:

– Fail. Падение при попытке повторно создать поток.
– Skip. Игнорирование повторно создаваемого потока.
– Self. Абстракция от повторно создаваемого потока (создание самопараллельного потока).

Результаты сравнения на наборе SV-COMP представлены в таблице 9.

Таблица 9 — Сравнение на наборе SV-COMP

	CPALockator-Reach		
Подход	Fail	Skip	Self
Вердикт «ошибка»	972	989	1028
из них корректных	793	802	805
из них некорректных	179	187	223
Вердикт «нет ошибки»	19	63	21
из них корректных	19	60	21
из них некорректных	0	3	0
Анализ не завершен	91	30	33
Время CPU, с	21 500	22 500	23 600

Результаты реализации Fail демонстрируют, что в наборе SV-COMP присутствуют задачи, в которых используется создание потока с присваиванием в уже использованную структуру потока. Если игнорировать такие случаи и не созда-

вать такой поток заново, это позволяет успешно доказать отсутствие ошибки для большого количества задач, однако такой подход позволяет пропустить ошибку, если ее появление зависит как раз от взаимодействия таких потоков. Это демонстрируют три примера, в которых была пропущена ошибка. Наиболее правильным вариантом в таком случае будет абстракция от количества созданных потоков, то есть конфигурация Self. Она не позволяет пропустить ошибку, но может получить большее количество вердиктов, чем конфигурация Fail.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 10.

Таблица 10 — Сравнение на наборе *drivers/net/*

	CPALockator-TM			CPALockator-Inv		
Подход	Fail	Skip	Self	Fail	Skip	Self
Вердикт «ошибка»	10	10	7	37	36	35
из них корректных	10	10	7	29	28	27
из них некорректных	0	0	0	8	8	8
Вердикт «нет ошибки»	262	262	261	256	255	255
из них корректных	262	262	261	256	255	255
из них некорректных	0	0	0	0	0	0
Анализ не завершен	153	153	157	132	134	135
Время CPU, с	132 000	133 000	139 000	110 000	110 000	116 000

Полученные результаты демонстрируют, что для задач из набора *drivers/net/* конфигурация Self является избыточной, хотя и по-прежнему является наиболее точной. Это объясняется особенностями модели окружения, которая отвечает за создание потоков. Дело в том, что для такой конфигурации становится невозможным ситуация, при которой несколько созданных потоков привязываются к одной структуре. Зная этот факт, мы можем использовать более эффективные конфигурации Fail или Skip, которые становятся равнозначными, так как такие ситуации, приводящие к падению, произойти не могут.

Результаты сравнения на ОС РВ представлены в таблице 11.

Таблица 11 — Сравнение на ядрах ОС РВ

	CPALockator-Inv ОС РВ 1			CPALockator-Inv ОС РВ 2		
Подход	Fail	Skip	Self	Fail	Skip	Self
Число предупреждений	461	461	461	981	981	981
Общее время CPU, с	582	565	607	886	836	937
Время работы ThreadCPA	0,15	0,17	0,12	0,213	0,22	0,16
Число созданных потоков	497	497	497	37	37	37

Эти результаты демонстрируют тот же эффект, что и результаты сравнени на наборе *drivers/net/*: используя особенности исходного кода, можно применять более эффективные в данном случае решения. Однако, следует помнить, что такие конфигурации являются некорректными в общем случае, что подтверждают результаты на наборе SV-COMP.

4.5. Сравнение различных вариантов реализации LockCPA

4.5.1. Сравнение вариантов реализации cpa-оператора $merge$

Для проведения экспериментов были использованы следующие конфигурации:

– Sep. Базовая реализация LockCPA с cpa-оператором $merge_{Sep}$, при котором состояния никогда не объединяются.
– Join. Реализация LockCPA с cpa-оператором $merge_{Join}$, при котором объединение состояний задается пересечением их множеств захваченных блокировок.

Результаты экспериментов на наборе SV-COMP представлены в таблице 12.

Таблица 12 — Сравнение на наборе SV-COMP

	CPALockator-Reach	
Подход	Sep	Join
Вердикт «ошибка»	1028	1028
из них корректных	805	805
из них некорректных	223	223
Вердикт «нет ошибки»	21	21
из них корректных	21	21
из них некорректных	0	0
Анализ не завершен	33	33
Время CPU, с	25 600	27 300

Результаты не демонстрируют никакой разницы между подходами. Это объясняется тем, что задачи в этом наборе достаточно простые, и в них практически невозможны ситуации, при которых в одной и той же точке программы могут быть захвачены различные блокировки. Более того, почти во всех примерах вообще ис-

пользуется единственная блокировка, в то время как для объединения необходима хотя бы пара отличающихся состояний.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 13.

Таблица 13 — Сравнение на наборе *drivers/net/*

	CPALockator-TM		CPALockator-Inv	
Подход	Sep	Join	Sep	Join
Вердикт «ошибка»	31	31	37	36
из них корректных	23	23	29	28
из них некорректных	8	8	8	8
Вердикт «нет ошибки»	253	253	256	256
из них корректных	253	253	256	256
из них некорректных	0	0	0	0
Анализ не завершен	141	141	132	133
Время CPU, с	121 000	120 000	110 000	110 000

В данном случае имеют место минимальные отличия случайного характера, так как в данных задачах также используется небольшое количество блокировок. И даже если возможны некоторые ситуации, в которых возникнет отличие работы одного варианта LockCPA от другого, они не будут влиять на общий вердикт задачи.

Результаты сравнения на ядрах ОС РВ представлены в таблице 14.

Таблица 14 — Сравнение на ядрах ОС РВ

	CPALockator-Reach			
	ОС РВ 1		ОС РВ 2	
Подход	Sep	Join	Sep	Join
Число предупреждений	461	461	981	981
Общее время CPU (с)	582	592	886	889
Время работы LockCPA, с	1,6	1,3	3,0	3,1
Операций с блокировками	38526	41604	108367	108367

Несмотря на некоторые отличия в количестве операций, изменение сра-оператора $merge$ снова никак не влияет на результаты. Таким образом, можно сделать вывод о том, что состояния LockCPA являются слишком малочисленными для каждой конкретной точки программы. Даже если там возникают несколько состояний, одно из них будет старше по решетке, чем другие, например, не будет содержать блокировок вообще. А значит, операции объединения становятся бессмысленными.

4.5.2. Сравнение оптимизаций BAM

Как было описано в разделе 3.6.5, возможно применение различных стратегий для операций $reduce/expand$ для LockCPA: удаление неиспользуемых примитивов синхронизации и сокращение счетчика рекурсивных захватов. Для сравнения были выбраны следующие варианты:

– None. Базовая реализация LockCPA с отключенными оптимизациями, то есть операция $reduce$ является тождественной.
– Block. Использование абстракции от счетчика рекурсивных захватов блокировки только для тех абстрактных блоков, где не производится никаких действий с данной блокировкой.
– All. Использование абстракции от счетчика рекурсивных захватов блокировки для всех абстрактных блоков.
– Locks. Использование абстракции от неиспользуемых примитивов синхронизации.
– Block-Locks. Одновременное использование и оптимизации Block, и оптимизации Locks.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 15

Таблица 15 — Сравнение на наборе *drivers/net/*

	CPALockator-Inv				
Подход	None	Block	All	Locks	Block-Locks
Вердикт «ошибка»	35	34	35	32	32
из них корректных	27	26	27	25	25
из них некорректных	8	8	8	7	7
Вердикт «нет ошибки»	255	255	255	255	255
из них корректных	255	255	255	255	255
из них некорректных	0	0	0	0	0
Анализ не завершен	135	136	135	138	134
Время CPU (с)	89 600	91 000	90 000	90 400	91 700

В данном случае отличия являются минимальными и в рамках погрешности. Это объясняется тем, что в драйверах обычно используется небольшое множество примитивов синхронизации, которые редко предполагают возможность рекурсивного захвата.

Результаты сравнения на ядрах ОС РВ представлены в таблицах 16, 17.

Таблица 16 — Сравнение на ядре ОС РВ 1

	CPALockator-Inv				
Подход	None	Block	All	Locks	Block-Locks
Число предупреждений	461	461	461	461	461
Общее время CPU, с	963	759	717	616	606
Время reduce/expand при анализе, с	0,3	0,4	0,4	0,5	0,6
Количество попаданий в BAM кэш	34305 (47%)	33893 (50%)	31656 (48%)	32893 (63%)	32893 (63%)

Таблица 17 — Сравнение на ядре ОС РВ 2

	CPALockator-Inv				
Подход	None	Block	All	Locks	Block-Locks
Число предупреждений	981	981	892	981	981
Общее время CPU, с	1211	1090	777	909	858
Время reduce/expand при анализе, с	1,5	3,7	1,3	1,5	1,7
Количество попаданий в BAM кэш	284253 (85%)	285655 (89%)	157505 (87%)	239473 (91%)	239473 (91%)

Для случая с ОС РВ можно сделать вывод, что использование эффективного варианта реализации $reduce/expand$ является принципиально важным, и может значительно ускорить анализ. Это объясняется тем, что, в отличие от драйверов, в ядре операционной системе достаточно много сильно связанного кода, который может вызываться под различными блокировками.

Снижение количества найденных предупреждений для варианта All объясняется наличием сложных случаев захвата блокировки под условиями. Например, в начале функции может быть проверка, если данная блокировка еще не захвачена, то она захватывается. Тогда, в случае All состояние на входе в функцию не будет содержать информацию о захваченной ранее блокировке, и она захватится второй раз. Но на самом деле она будет захвачена дважды, что может сыграть определенную роль при ее освобождении.

4.5.3. Использование уточнения

– Base. Базовая реализация LockCPA, которая рассматривает все возможные примитивы синхронизации с самого начала.
– Refinement. Реализация LockCPA с использованием уточнения.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 18.

Таблица 18 — Сравнение на наборе *drivers/net/*

	CPALockator-Inv	
Подход	Base	Refinement
Вердикт «ошибка»	33	31
из них корректных	25	23
из них некорректных	8	8
Вердикт «нет ошибки»	255	255
из них корректных	255	255
из них некорректных	0	0
Анализ не завершен	137	139
Время CPU (с)	95 200	94 100

В целом, результаты двух конфигураций на наборе *drivers/net/* отличаются не сильно. Это объясняется тем, что обычно используется небольшой набор примитивов синхронизации, который может быть легко вычислен за одну дополнительную итерацию уточнения.

Результаты сравнения на ядрах ОС РВ представлены в таблице 19.

Таблица 19 — Сравнение на ядрах ОС РВ

	CPALockator-Inv			
	ОС РВ 1		ОС РВ 2	
Подход	Base	Refinement	Base	Refinement
Выданные предупреждения	461	488	981	1128
Время CPU, с	582	4952	886	5170
Время на LockCPA, с	1,6	15	1,7	4,8
Операций с блокировками	38526	310422	108367	645978
Количество уточнений	0	10	0	10

В случае ОС РВ отличия становятся более масштабными, так как количество примитивов синхронизации резко возрастает и требуется значительное количество итераций, чтобы исключить все ложные предупреждения об ошибках, которые появились именно из-за неточностей LockCPA. При этом возможный выигрыш из-за построения менее точной абстракции существенно меньше.

Таким образом, можно сделать вывод, что реализация LockCPA с использованием уточнения является неэффективной и для небольших программ, и для достаточно объемных.

4.6. Сравнение вклада в точность анализа дополнительных CPA

4.6.1. Оценка эффекта оптимизации BAM

– Base. Вариант анализа без использования BAM.
– BAM. Использование оптимизации BAM.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 20.

Таблица 20 — Сравнение на наборе *drivers/net/*

	CPALockator-Inv	
Подход	Base	BAM
Вердикт «ошибка»	5	37
из них корректных	3	29
из них некорректных	2	8
Вердикт «нет ошибки»	248	256
из них корректных	248	256
из них некорректных	0	0
Анализ не завершен	172	132
Время CPU (с)	51 500	110 200

Результаты наглядно демонстрируют, что оптимизация BAM позволет значительно ускорить анализ, при этом новые ложные вердикты связаны не с самой оптимизацией BAM, а с другими свойствами анализа. Оптимизация BAM лишь позволяет проявить эти свойства, избежав исчерпания лимита времени или памяти. Отдельно отметим, что общее время с использованием оптимизации BAM было затрачено больше из-за большого количества падений конфигурации Base из-за нехватки памяти. Такие падения происходили достаточно быстро, что сокращало общее время работы.

Результаты сравнения на ядрах ОС РВ представлены в таблице 21.

Таблица 21 — Сравнение на ядрах ОС РВ

	CPALockator-Inv			
	ОС РВ 1		ОС РВ 2	
Подход	Base	BAM	Base	BAM
Выданные предупреждения	-	461	-	981
Время CPU, с	10024	582	7843	886
Проанализированных потоков	92	497	1	37

Для анализа такого сложного исходного кода, как ядро ОС РВ, оптимизация BAM является принципиально важной. По количеству проанализированных потоков можно оценить насколько эта оптимизация ускоряет анализ. Дело в том, что в отличие от драйверов в ядре ОС РВ присутствует большое количество сильно связанного кода, то есть кода, доступного из многих мест. Это приводит к большому количеству повторно анализируемых функций, что является очень затратным, если не переиспользовать предыдущие результаты.

4.6.2. Оценка эффекта использования анализа разделяемых данных

– Base. Конфигурация инструмента без использования анализа разделяемых данных.
– Shared. Конфигурация инструмента с использованием анализа разделяемых данных.

Результаты сравнения на наборе *drivers/net/* представлены в таблице 22.

Таблица 22 — Сравнение на наборе *drivers/net/*

	CPALockator-Inv	
Подход	Base	Shared
Вердикт «ошибка»	31	33
из них корректных	23	24
из них некорректных	8	9
Вердикт «нет ошибки»	255	254
из них корректных	255	254
из них некорректных	0	0
Анализ не завершен	139	138
Время CPU (с)	93 300	95 100

В данном случае использование анализа разделяемых данных не приводит к значительному улучшению. Это объясняется тем, что выделение новой памяти и ее инициализация производится обычно на начальном этапе загрузки модуля в однопоточном режиме, и только после этого обработчики драйвера регистрируются в системе и становятся доступными для вызова. Таким образом, анализ потоков успешно справляется с задачей исключения таких ложных ситуаций без анализа разделяемых данных.

Некоторое незначительное ухудшение работы связано с тем, что, тем не менее, некоторые пути к доступам к данным, ранее считавшихся разделяемыми, были отброшены в конфигурации Shared. И вместо отброшенных путей стали рассматриваться другие, которые содержали более сложные условия для анализа предикатов, что привело к более длительной процедуре уточнения и построению более детальной абстракции.

Результаты сравнения на ядрах ОС РВ представлены в таблице 23.

Таблица 23 — Сравнение на ядрах ОС РВ

	CPALockator-Inv			
	ОС РВ 1		ОС РВ 2	
Подход	Base	Shared	Base	Shared
Выданные предупреждения	461	218	981	447
Время CPU, с	582	1034	886	568
Преданализ, с	–	21	–	19

В данном случае анализ разделяемых данных значительно сокращает количество ложных предупреждений об ошибках, так как отсутствуют секции инициализации данных. Таким образом, только анализ разделяемых данных может определить, что предупреждения является ложным.

Некоторое увеличения времени анализа не связано напрямую с анализом разделяемых данных, а происходит из-за дополнительных расходов на вычисление статистики. Это время является незначительным для второго случая из-за отличий кода.

4.6.3. Оценка эффекта использования предикатного анализа

– Predicate. Вариант анализа с использованием предикатной абстракции.
– Lightweight. Использование анализа без использования предикатной абстракции

Результаты экспериментов на наборе SV-COMP представлены в таблице 24.

Результаты подтверждают тот факт, что без анализа предикатов невозможно доказать недостижимость ошибочной метки, то есть корректность задачи, так как во всех задачах так или иначе присутствует условие. Другими словами, все ошибочные метки являются синтаксически достижимыми.

Таблица 24 — Сравнение на наборе SV-COMP

	CPALockator-Inv	
Подход	Predicate	Lightweight
Вердикт «ошибка»	1028	1058
из них корректных	805	808
из них некорректных	223	250
Вердикт «нет ошибки»	21	0
из них корректных	21	0
из них некорректных	0	0
Анализ не завершен	33	24
Время CPU, с	25 600	8 020

Результаты сравнения на наборе *drivers/net/* представлены в таблице 25.

Таблица 25 — Сравнение на наборе *drivers/net/*

	CPALockator-TM		CPALockator-Inv	
Подход	Predicate	Lightweight	Predicate	Lightweight
Вердикт «ошибка»	31	144	37	168
из них корректных	23	86	29	86
из них некорректных	8	58	8	82
Вердикт «нет ошибки»	253	226	256	226
из них корректных	253	226	256	226
из них некорректных	0	0	0	0
Анализ не завершен	128	55	132	31
Время CPU (с)	121 000	37 700	110 000	14 200

Результаты показывают, что использование анализа предикатов существенно повышает точность анализа, хотя для некоторых примеров оказывается возможным доказать отсутствие состояний гонки и без него.

Результаты сравнения на ядрах ОС РВ представлены в таблице 26.

Таблица 26 — Сравнение различных вариантов реализации cpa-оператора $merge$

	CPALockator-Inv			
	ОС РВ 1		ОС РВ 2	
Подход	Predicate	Lightweight	Predicate	Lightweight
Выданные предупреждения	461	521	981	1037
Время CPU, с	582	107	886	195

Заметим, что даже без использования уточнения предикатный анализ способен сократить количество ложных срабатываний за счет оптимизации ABE, которая способна отсекать недостижимые пути в рамках одного блока, в котором строится полная формула пути. При этом для дальнейшего отсеивания ложных

предупреждений необходмо проведение процедуры уточнения для добавления предикатов.

4.7. Анализ причин пропуска ошибок в модулях ядра ОС Linux

Анализ причин пропуска ошибок производился на множестве задач, полученных на основе существующих исправлений в стабильных версиях ядра Linux [4] за 2014 год. Из всего множества в 4047 коммитов по ключевым словам было выделено 795 коммитов, которые были проанализированы вручную. Ключевыми словами были слова, связанные с исправлением ошибки: fix, error, race, bug, failure, crash и др. Подробно методика анализа коммитов описана в работе [100]. Из полученного списка были выделены коммиты, исправляющие ошибки, связанные с состояниями гонки по данным. То есть, в это множество не включались высокоуровневые гонки. Из полученных 43 коммитов были исключены те, в которых встречались примитивы синхронизации, не основанные на блокировках, так как инструмент в данный момент не поддерживает другие примитивы синхронизации. Далее, из множества в 28 коммитов были исключены те, которые не относились к модулям ядра, так как это является ограничением системы подготовки задач Klever. В полученном списке осталось 13 коммитов, для каждого из которых была запущена система Klever с целью подготовки верификационных задач. Для 5 модулей не были получены задачи из-за внутренних проблем системы Klever (падения компонентов). Оставшиеся 8 задач были получены, и на них производился запуск иструмента CPALockator с целью проверить, обнаруживается ли то состояние гонки, которое исправляется в данном коммите.

Результаты анализа полученных 8 задач представлены в таблице 27. Ошибки обозначаются идентификаторами коммитов, исправляющими ошибку. Соответственно, верификационная задача была получена для состояния репозитория для предыдущего коммита, то есть, до исправления. Подробное описание исправлений приведено в приложении Б.

Таким образом, из 8 коммитов

– В двух модулях однаруживаются именно те ошибки, которые исправляются в коммитах. Для этих задач были подготовлены парные задачи для

[4] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

Таблица 27 — Результаты запуска инструмента на известных ошибках в стабильных версиях ядра ОС Linux

Коммит	Модуль	Рез-т	Комментарий
0e2400e	drivers/char/virtio_console.ko	+	
7357404	fs/hfsplus/hfsplus.ko	±	Спец. конфигурация
f1a8a3f	drivers/net/bonding/bonding.ko	∓	Спец. конфигурация; нецелевая ошибка
1a81087	net/ipv4/tcp_illinois.ko	±	Найдена нецелевая ошибка
f0c626f	drivers/target/iscsi/iscsi_target_mod.ko	∓	Спец. конфигурация; нецелевая ошибка
aea9dd5	fs/btrfs/btrfs.ko	−	
10ef175	sound/soc/snd-soc-core.ko	−	Таймаут
4036523	drivers/gpu/drm/i915/i915.ko	−	

состояния репозитория после исправления. Для одного модуля соответствующее предупреждение пропадает полностью, а для другого выдается новое предупреждение об ошибке, но уже ложное. Это подтверждает корректность обнаружения ошибки.

– Еще в трех модулях однаруживаются другие, нецелевые состояния гонки на ту же переменную, что и исправляется в коммите. Из-за того, что CPALockator выдает только одно предупреждение для каждой переменной, целевой исправление не было обнаружено.
– Три модуля являются слишком объемными, что приводит к исчерпанию лимита времени (15 минут) при их анализе.

Ошибки, которые исправляются в коммитах 7357404, f1a8a3f, f0c626f, находятся только при ограниченном количестве итераций CEGAR. В противном случае за отведенные лимиты времени CPALockator не успевает исключить все ложные пути. При ограничении количества итераций CEGAR уточнение будет прервано, и все оставшиеся предупреждения будут выданы.

Таким образом, 5 из 8 (62,5%) выбранных исправлений так или иначе находятся. Для оставшихся трех задач был выдан вердикт *unknown*. Это демонстрирует, что инструмент CPALockator способен не пропускать ошибки при заранее известных ограничениях.

4.8. Анализ причин ложных срабатываний на компонентах ядра ОС

4.8.1. Ложные предупреждения на драйверах ОС Linux

Анализ ложных предупреждений проводился на задачах, основанных на драйверах подсистемы *drivers/net/* ОС Linux, которые были использованы для сравнения конфигураций. Для 473 задач из подсистемы *drivers/net/* было подготовлено 425 задач. Для 261 было доказано отсутствие ошибок. Потенциальные состояния гонки были обнаружены в 22 задачах. При этом только для 8 задач был выдан финальный вердикт *false*. На остальных 161 задаче анализ завершился с вердиктом *unknown* из-за исчерпания лимита времени в 15 минут. При этом для 14 задач инструмент выдал некоторое количество предупреждений, несмотря на завершение по таймауту. Так, для 22 задач было получено 85 предупреждений. Эти предупреждения были проанализированы вручную, и в таблице 28 представлены результаты их анализа.

Таблица 28 — Анализ предупреждений в модулях ядра ОС Linux

Количество	Процент	Описание
41	48%	Истинных предупреждений
25	29%	Неточность подготовки задачи
19	22%	Неточность метода
8	9%	Неточность модели памяти
7	8%	Специфика ядра ОС (прерывания)
4	5%	Неточность отдельных CPA

Потенциальным ошибкам соответствовало 41 предупреждений (48%). При этом 10 из них были отмечены, как безобидные. Это, например, состояния гонки на счетчиках статистики, при печати вспомогательной информации на экран и т. п. Большая часть найденных ошибок (18 предупреждений) соответствует ситуации, при которой новый поток создается в тот момент, когда еще не завершена инициализация основных данных драйвера. Например, регистрация структуры девайса или регистрация обработчиков прерывания может происходить до того, как выделяется память под те данные, которые могут быть использованы в обработчиках этого девайса. При этом для каждого устройства возможно некоторая защита на уровне железа. Например, обработчик прерывания регистрируется в системе, однако соответствующие прерывания само устройство может генериро-

вать только значительно позже. Такие свойства очень сложно обнаружить только по исходному коду драйвера. Важно отметить, что для одного состояния гонки может быть выдано несколько предупреждений на различные переменные, работа с которыми производится похожим образом. Поэтому число предупреждений значительно превышает число уникальных состояний гонки. Так, полученные 41 предупреждение соответствует лишь 15 модулям.

Значительная часть ложных сообщений об ошибках связана с несовершенством модели окружения. Из-за этой причины было получено 25 ложных предупреждений (29%). Заметим, что весь этот класс ошибок не связан с инструментом CPALockator. В основном, неточность модели окружения приводит к тому, что параллельно выполняются участки кода, которые не могут выполняться одновременно при реальном исполнении программы. Еще одним подклассом данного типа ложных предупреждений является неточная модель для отсутствующих функций. Например, некоторые функции могут активировать некоторые потоки, вызывать функции по указателю, но их код не включается в верификационную задачу. В этом случае должна быть подготовлена модель, которая содержит в себе необходимые действия (активацию потока, вызов функции и т.д.). Другим подклассом данного типа ошибок является неточное выделение памяти под структуры драйвера. Это приводит к последующим неточностям в процессе анализа, в первую очередь, «потери значений» в анализе предикатов. То есть, при чтении значения из невыделенной памяти считается, что там может быть любое значение.

Оставшиеся ложные предупреждения связаны непосредственно с инструментом CPALockator. Наиболее значительный класс ложных предупреждений связан с неточностью модели памяти – 8 ложных предупреждений. Из-за неточной модели памяти анализ не может определить, что данные на самом деле являются разными. В первую очередь это касается полей структур, доступ к которым встречается наиболее часто. Из-за использования указателей на структуры мы не можем определить, что в двух потоках используется различная память одного типа, и, чтобы не пропустить ошибку, вынуждены выдавать предупреждения.

Еще одной важной причиной ложных предупреждений является низкоуровневые особенности ядра, в первую очередь прерываний. В данный момент считается, что обработчик прерывания выполняется в обычном потоке, однако в реальной системе это не так. Выполнение прерывания может быть прервано только в некоторых случаях и только другим прерванием, что означает, что при выполнении его с другим обычным потоком оно должно выполниться атомарно. Кро-

ме того, регистрация обработчика прерывания не всегда означает, что они могут быть активны. В некоторых случаях необходимо активировать их на устройстве отдельно.

Далее идут незначительные классы ложных предупреждений, связанных с неточностями анализа разделяемых данных (4 предупреждений), особенностями работы со сложными структурами данных (1 предупреждение) и модельными переменным (1 предупреждения)

4.8.2. Ложные предупреждения на ОС РВ

Из анализа 52 предупреждений было найдено:

- 11 реальных предупреждений, из них 4 безобидных;
- 43 ложных предупреждений, из которых
    - 11 связанны с неточностью модели окружения;
    - 16 связанных с неточностью анализа разделяемых данных;
    - 9 связанных с неточностью анализа примитивов синхронизации;
    - 4 связанных с неточностью модели памяти;

Модель окружения для ОС РВ представляет собой набор потоков, которые выполняют код заранее заданных системных вызовов, обработчиков прерываний и т.д. Таким образом, для всех потоков известно, что они могут выполняться в любой момент времени, а значит, снижается число ложных предупреждений об ошибках из-за неточности модели окружения.

Как и следовало ожидать, при анализе ОС РВ повышается роль анализа разделяемых данных, и несколько снижаются требования к модели окружения. Слишком малое количество ложных предупреждений из-за неточности модели памяти, скорее всего, связано с тем, что такие проблемы проявляются в последнюю очередь. То есть, после решения проблем, связанных с неточностью модели окружения и анализа разделяемых данных, могут возникнуть проблемы из-за неточной модели памяти.

4.9. Выводы по результатам экспериментов

4.9.1. Выводы по использованным конфигурациям

Как невозможно создать универсальный инструмент, способный находить любые ошибки в любых программах, так и невозможно указать универсальную конфигурацию, которая будет лучше для всех задач. Однако, имеет смысл обозначить те или иные случаи, в которых будет лучше та или иная конфигурация.

Среди различных видов реализации cpa-оператора $merge$ для PredicateCPA в большинстве случаев оказывается достаточным реализация Join. Sep является слишком медленной, а ее точность не нужна почти во всех случаях. Реализация Eq близка к Join как по временным показателям, так и по количеству найденных вердиктов.

Оптимизации PredicateCPA обладают различными характеристиками. ABE позволяет ускорить анализ во всех случаях, но для больших задач полученное ускорение становится наиболее заметным. Оптимизация с применением релевантных эффектов становится значимой только при большом количестве уточнений, то есть для сложных задач. Тем не менее, эти оптимизации не снижают точность анализа, а значит, могут применяться в любых случаях. Оптимизация с присваиванием неопределенных значений ускоряет анализ, но для искусственных тестов это сопровождается снижением точности. Для исходного кода более сложных задач такое падение точности не наблюдается, а значит, такая оптимизация может применяться в таких случаях. Оптимизации с игнорированием разделяемых данных значительно снижают точность, но позволяют ускорить анализ. Но более важное их свойство состоит в том, что они делают анализ предикатов инвариантным к эффектам окружения, что позволяет применять более сложные оптимизации, в том числе, BAM. Таким образом, они могут быть использованы при анализе большого объема исходного кода, который не позволяет провести полноценный точный анализ.

Реализации ThreadCPA с эффектами окружения и без них практически равнозначны, однако, для использования других оптимизаций выгоднее иметь такой вариант анализа потоков, который будет инвариантным к эффектам окружения. Простая реализация ThreadCPA является слишком простой для решения задач,

в которых предполагается порождение потоков внутри других, однако является достаточным при анализе искуственно подготовленного кода для некоторых ОС РВ. Повторно создаваемые потоки могут встречаться в различных искусственных тестах, однако это является нетипичным случаем для модели окружения драйверов и ОС РВ. Таким образом, в общем случае вариант с абстракцией от повторно создаваемого потока поможет обеспечить отсутствие пропуска ошибки, а при решении прикладных задач более эффективным будет вариант с игнорированием таких ситуаций.

Реализации LockCPA с использованием cpa-оператора $merge_{Join}$ и с использованием уточнения являются бессмысленными на всех наборах, на которых проводилось исследование. Таким образом, эффективнее становится использовать простой вариант анализа примитивов синхронизации, в котором отсутствует объединение состояний. Реализации $reduce/expand$ становятся значимыми только при анализе большого объема исходного кода. Тем не менее, существуют вариант реализации этих cpa-операторов, которые могут привести к пропуску ошибок в определенных ситуациях, хотя при этом позволяют получить существенное ускорение анализа.

Использование анализа разделяемых данных не приводит к сколько-нибудь заметному снижению скорости анализа, но при этом возможно снижение числа выданных ложных предупреждений об ошибках. Это число сильно зависит от возможностей анализа потоков, который также может отсеивать часть ложных сообщений, связанных с инициализацией, поэтому наиболее заметный эффект достигается при анализе кода, который не содержит таких зависимостей (ядро ОС РВ).

Использование оптимизации BAM позволяет существенно сократить требования по расходуемой памяти и времени для проведения анализа. Существенное ограничение данной оптимизации состоит в том, что она требует инвариантности к эффектам окружения, хотя это и не является принципиальным ограничением. Другими словами, оптимизация позволяет расширение на случай подхода с раздельным рассмотрением потоков, однако это не является темой данной работы и может быть дальнейшим развитием.

Использование анализа предикатов позволяет существенно повысить точность анализа, однако имеется возможность полностью отказаться от него, что, по сути, является реализацией варианта простого легковесного анализа потоков

данных. Такой вариант также может иметь место для поиска ошибок в большом объеме исходного кода или в жестких временных рамках.

4.9.2. Выводы по результатам анализа предупреждений об ошибках

Одной из важных причин и ложных срабатываний, и пропуска ошибок при анализе реального кода становится подготовка неточной верификационной задачи. С точки зрения инструмента статической верификации это является сторонним процессом, и ложное предупреждение из-за неточности модели окружения является истинным с точки зрения инструмента. Для реального пользователя интересен результат, и поэтому задача соответствия верификационной задачи реальному процессу выполнения программы является достаточно важной. Однако, эта тема выходит за рамки данной работы.

При этом подготовка качественной модели окружения требуется для того исходного кода, который подразумевает сложную схему работы. Так, при анализе ОС РВ, при котором требовалось проверить корректную работу системных вызовов при любом варианте выполнения, было достаточно самой простой модели. В то время как для драйверов ОС Linux была нужна более точная модель, так как различные обработчики драйвера могут активироваться в различное время.

Наиболее частой причиной ложных предупреждений для системного программного обеспечения является неточная модель памяти, при которой отождествляются различные объекты. Данная проблема возникает почти у всех инструментов статического анализа. Попытки применения более точных моделей совместно с анализом алиасов приводят к другим проблемам, связанным с эффективностью, а также проблемам применения к недоопределенному коду. Таким образом, данное направление является одним из приоритетных для дальнейшего развития инструмента.

Другие причины ложных срабатываний, связанных с неточностями различных компонентов анализа проявляются в различной степени на различном целевом исходном коде. Например, ложные предупреждения из-за неточности анализа разделяемых данных почти не проявляются на драйверах ОС Linux из-за особенностей их устройства и анализа потоков, который способен повысить точность.

Однако, для ОС РВ процент ложных предупреждений из-за анализа разделяемых данных повышается, так как анализ потоков играет меньшую роль.

Тоже самое верно и для длугих CPA, в том числе LockCPA. Для того исходного кода, который использует обычные блокировки для синхронизации, данный анализ работает достаточно точно, однако, он становится бесполезным при анализе кода с другими примитивами синхронизации, например, с помощью посылки сообщений. И для такого кода будет требоваться разработка нового CPA.

Таким образом, можно заключить, что разработанный инструмент поиска состояний гонки способен обеспечивать высокую точность анализа

Заключение

Основные результаты работы заключаются в следующем.

1. Был разработан метод поиска состояний гонки на основе раздельного анализа потоков, использующий средства абстракции состояний и переходов для управления точностью и ресурсоемкостью верификации.
2. Был разработан алгоритм построения окружения потока, который позволяет гибко настраивать уровень абстракции над взаимодействием потоков, и доказана его корректность.
3. Был разработан новый алгоритм, который является обобщением существующего алгоритма статической верификации программ при помощи метода CPA, расширяющий типовой набор CPA-анализаторов средствами верификации многопоточных программ с раздельным анализом потоков, и доказана его корректность.

Эксперименты показали преимущества подхода на больших верификационных задачах перед существующими техниками статической верификации. Небольшие задачи со сложным взаимодействием потоков лучше решаются другими инструментами, так как предложенный подход абстрагируется от такого взаимодействия, что приводит к потере точности, существенной для небольших искусственных задач. Однако разработанный подход также не пропускает ошибок (при некоторых предположениях) и может быть развит в будущем.

Таким образом, можно заключить, что основные требования к новому инструменту для анализа корректности синхронизации были выполнены, так как он успешно применяется к различным программным системам, в том числе, к драйверам ОС Linux и ядрам ОС реального времени. Также как и в классических методах статической верификации, может быть получена гарантия отсутствия ошибок при выполнении указанных предположений: требований на операторы CPA и условий корректного разбиения на непересекающиеся регионы BnB модели. Предложенная общая теория позволяет описывать достаточно сложные варианты анализа, в том числе и те, которые не включаются в понятие анализа с раздельным рассмотрением потоков. Тем не менее, эта теория не исчерпывает всех возможных подходов, и для эффективного описания масштабируемого подхода с частичным вычислением чередований потребуется ее расширение. Однако, эта тема выходит за рамки данной работы. Одним из возможных направлений развития подхода яв-

ляется добавление взаимодействия потоков, возможно, не в полном объеме, чтобы сохранить масштабируемость. Это может быть отдельный анализ CPA, который будет динамически настраивать свою точность с помощью алгоритма CEGAR, обеспечивая некоторый промежуточный вариант между подходом с раздельным анализом потоков и перебором чередований. Другим возможным улучшением инструмента может стать интеграция его с другим подходом. Например, комбинация быстрого подхода с раздельным анализом потоков в качестве первого этапа, а затем классический тяжеловесный анализ – на втором этапе. Это может быть реализовано в соответствии с идеей кооперативной верификации. Еще одним возможным направлением развития подхода является построение реального пути с чередованием потоков на основе пути с примененными эффектами окружения. Этот вариант был бы полезен для исследования и уточнения абстракции.

В заключение автор выражает благодарность и большую признательность научному руководителю Хорошилову А. В. и Мутилину В. С. за поддержку, помощь, обсуждение результатов и научное руководство. Также автор благодарит Новикова Е. М., Захарова И. С. и Щепеткова И. В. за помощь в работе и обсуждение результатов. Автор также благодарит всех, кто сделал настоящую работу возможной.

Список литературы

1. *П.С. Андрианов*. Анализ корректности синхронизации компонентов ядра операционных систем // *Труды Института системного программирования РАН*. — 2019. — Т. 5, № 31. — С. 203–232.

2. *Andrianov Pavel*. Analysis of Correct Synchronization of Operating System Components // *Programming and Computer Software*. — 2020. — Vol. 46, no. 8. — P. 712–730.

3. *Andrianov Pavel, Mutilin Vadim*. Scalable Thread-Modular Approach for Data Race Detection // *Frontiers in Software Engineering Education*. — 2020. — Pp. 371–385.

4. *П.С. Андрианов, В.С. Мутилин, А.В. Хорошилов*. Метод легковесного статического анализа для поиска состояний гонок // *Труды Института системного программирования РАН*. — 2015. — Т. 5, № 27. — С. 87–116.

5. *П.С. Андрианов, В.С. Мутилин, А.В. Хорошилов*. Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных абстракций // *Труды Института системного программирования РАН*. — 2016. — Т. 6, № 28. — С. 65–86.

6. *Andrianov Pavel, Mutilin Vadim, Khoroshilov Alexey*. Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel // Tools and Methods of Program Analysis: 4th International Conference, TMPA 2017, Moscow, Russia, March 3-4, 2017, Revised Selected Papers / Ed. by Vladimir Itsykson, Andre Scedrov, Victor Zakharov. — Cham: Springer International Publishing, 2018. — Pp. 11–23. — URL: https://doi.org/10.1007/978-3-319-71734-0_2.

7. *P.S. Andrianov, V.S. Mutilin, A.V. Khoroshilov*. An approach to lightweight static data race detection // Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering. — 2014. — Pp. 27–33.

8. *P.S. Andrianov, V.S. Mutilin, A.V. Khoroshilov*. Lightweight Static Analysis for Data Race Detection in Operating System Kernel // Proceedings of the internation-

al conference ”Tools and Methods of Programs Analysis”. — 2014. — Pp. 128–135.

9. *Кулямин В. В.* Методы верификации программного обеспечения. — 2008. — 111 pp.

10. *Raza Aoun*. A Review of Race Detection Mechanisms // Computer Science – Theory and Applications / Ed. by Dima Grigoriev, John Harrison, Edward A. Hirsch. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. — Pp. 534–543.

11. Evaluation and Comparison of Ten Data Race Detection Techniques // *Int. J. High Perform. Comput. Netw.* — 2017. — jan. — Vol. 10, no. 4–5. — P. 279–288.

12. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs / Stefan Savage, Michael Burrows, Greg Nelson et al. // *SIGOPS Oper. Syst. Rev.* — 1997. — oct. — Vol. 31, no. 5. — Pp. 27–37. — URL: http://doi.acm.org/10.1145/269005.266641.

13. *Lamport Leslie*. Time, Clocks, and the Ordering of Events in a Distributed System // *Commun. ACM*. — 1978. — jul. — Vol. 21, no. 7. — P. 558–565. — URL: https://doi.org/10.1145/359545.359563.

14. *Flanagan Cormac, Freund Stephen N.* FastTrack: Efficient and Precise Dynamic Race Detection // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI ’09. — New York, NY, USA: ACM, 2009. — Pp. 121–133. — URL: http://doi.acm.org/10.1145/1542476.1542490.

15. *Flanagan Cormac, Freund Stephen N.* FastTrack: Efficient and Precise Dynamic Race Detection // *SIGPLAN Not.* — 2009. — jun. — Vol. 44, no. 6. — Pp. 121–133. — URL: http://doi.acm.org/10.1145/1543135.1542490.

16. *Marino Daniel, Musuvathi Madanlal, Narayanasamy Satish*. LiteRace: Effective Sampling for Lightweight Data-race Detection // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI ’09. — New York, NY, USA: ACM, 2009. — Pp. 134–143. — URL: http://doi.acm.org/10.1145/1542476.1542491.

17. *Marino Daniel, Musuvathi Madanlal, Narayanasamy Satish*. LiteRace: Effective Sampling for Lightweight Data-race Detection // *SIGPLAN Not.* — 2009. — jun. — Vol. 44, no. 6. — Pp. 134–143. — URL: http://doi.acm.org/10.1145/1543135.1542491.

18. *Bond Michael D., Coons Katherine E., McKinley Kathryn S.* PACER: Proportional Detection of Data Races // Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '10. — New York, NY, USA: ACM, 2010. — Pp. 255–268. — URL: http://doi.acm.org/10.1145/1806596.1806626.

19. *Bond Michael D., Coons Katherine E., McKinley Kathryn S.* PACER: Proportional Detection of Data Races // *SIGPLAN Not.* — 2010. — jun. — Vol. 45, no. 6. — Pp. 255–268. — URL: http://doi.acm.org/10.1145/1809028.1806626.

20. Dynamic Race Detection with LLVM Compiler / Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, Dmitriy Vyukov // Proceedings of the Second International Conference on Runtime Verification. — RV'11. — Berlin, Heidelberg: Springer-Verlag, 2012. — Pp. 110–114. — URL: http://dx.doi.org/10.1007/978-3-642-29860-8_9.

21. *Serebryany Konstantin, Iskhodzhanov Timur*. ThreadSanitizer – data race detection in practice. // Proceedings of the Workshop on Binary Instrumentation and Applications. — NYC, NY, U.S.A.: 2009. — Pp. 62–71. — URL: http://doi.acm.org/10.1145/1791194.1791203.

22. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-world Applications / Yao Qi, Raja Das, Zhi Da Luo, Martin Trotter // Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. — PADTAD '09. — New York, NY, USA: ACM, 2009. — Pp. 5:1–5:11. — URL: http://doi.acm.org/10.1145/1639622.1639627.

23. *Yoga Adarsh, Nagarakatte Santosh, Gupta Aarti*. Parallel Data Race Detection for Task Parallel Programs with Locks // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA: ACM, 2016. — Pp. 833–845. — URL: http://doi.acm.org/10.1145/2950290.2950329.

24. Effective Data-Race Detection for the Kernel / John Erickson, Madan Musuvathi, Sebastian Burckhardt, Kirk Olynyk // Operating System Design and Implementation (OSDI'10). — USENIX, 2010. — October. — URL: https://www.microsoft.com/en-us/research/publication/effective-data-race-detection-for-the-kernel/.

25. Effective Data-Race Detection for the Kernel / John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk // Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. — OSDI'10. — USA: USENIX Association, 2010. — P. 151–162.

26. A study of the internal and external effects of concurrency bugs / P. Fonseca, Cheng Li, V. Singhal, R. Rodrigues // 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN). — 2010. — June. — Pp. 221–230.

27. *Fonseca Pedro, Li Cheng, Rodrigues Rodrigo*. Finding Complex Concurrency Bugs in Large Multi-threaded Applications // Proceedings of the Sixth Conference on Computer Systems. — EuroSys '11. — New York, NY, USA: ACM, 2011. — Pp. 215–228. — URL: http://doi.acm.org/10.1145/1966445.1966465.

28. *Park Soyeon, Lu Shan, Zhou Yuanyuan*. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places // Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. — ASPLOS XIV. — New York, NY, USA: ACM, 2009. — Pp. 25–36. — URL: http://doi.acm.org/10.1145/1508244.1508249.

29. *Park Soyeon, Lu Shan, Zhou Yuanyuan*. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places // *SIGARCH Comput. Archit. News*. — 2009. — mar. — Vol. 37, no. 1. — Pp. 25–36. — URL: http://doi.acm.org/10.1145/2528521.1508249.

30. *Park Soyeon, Lu Shan, Zhou Yuanyuan*. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places // *SIGPLAN Not.* — 2009. — mar. — Vol. 44, no. 3. — Pp. 25–36. — URL: http://doi.acm.org/10.1145/1508284.1508249.

31. *Kusano Markus, Wang Chao*. Flow-sensitive Composition of Thread-modular Abstract Interpretation // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA: ACM, 2016. — Pp. 799–809. — URL: http://doi.acm.org/10.1145/2950290.2950291.

32. *Sung Chungha, Kusano Markus, Wang Chao*. Modular Verification of Interrupt-driven Software // Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. — ASE 2017. — Piscataway, NJ, USA: IEEE Press, 2017. — Pp. 206–216. — URL: http://dl.acm.org/citation.cfm?id=3155562.3155592.

33. *Pratikakis Polyvios, Foster Jeffrey S., Hicks Michael*. LOCKSMITH: Practical Static Race Detection for C // *ACM Trans. Program. Lang. Syst.* — 2011. — jan. — Vol. 33, no. 1. — Pp. 3:1–3:55. — URL: http://doi.acm.org/10.1145/1889997.1890000.

34. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs / George C. Necula, Scott McPeak, Shree P. Rahul, Westley Weimer // Compiler Construction / Ed. by R. Nigel Horspool. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. — Pp. 213–228.

35. *Voung Jan Wen, Jhala Ranjit, Lerner Sorin*. RELAY: Static Race Detection on Millions of Lines of Code // Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. — ESEC-FSE '07. — New York, NY, USA: ACM, 2007. — Pp. 205–214. — URL: http://doi.acm.org/10.1145/1287624.1287654.

36. *Steensgaard Bjarne*. Points-to Analysis in Almost Linear Time // Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '96. — New York, NY, USA: ACM, 1996. — Pp. 32–41. — URL: http://doi.acm.org/10.1145/237721.237727.

37. *Radoi Cosmin, Dig Danny*. Effective Techniques for Static Race Detection in Java Parallel Loops // *ACM Trans. Softw. Eng. Methodol.* — 2015. — sep. — Vol. 24, no. 4. — Pp. 24:1–24:30. — URL: http://doi.acm.org/10.1145/2729975.

38. *Di Peng, Sui Yulei*. Accelerating Dynamic Data Race Detection Using Static Thread Interference Analysis // Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores. — PMAM'16. — New York, NY, USA: ACM, 2016. — Pp. 30–39. — URL: http://doi.acm.org/10.1145/2883404.2883405.

39. *Sui Yulei, Di Peng, Xue Jingling*. Sparse Flow-sensitive Pointer Analysis for Multithreaded Programs // Proceedings of the 2016 International Symposium on Code Generation and Optimization. — CGO '16. — New York, NY, USA: ACM, 2016. — Pp. 160–170. — URL: http://doi.acm.org/10.1145/2854038.2854043.

40. May-happen-in-parallel Analysis with Static Vector Clocks / Qing Zhou, Lian Li, Lei Wang et al. // Proceedings of the 2018 International Symposium on Code Generation and Optimization. — CGO 2018. — New York, NY, USA: ACM, 2018. — Pp. 228–240. — URL: http://doi.acm.org/10.1145/3168813.

41. Parallel bug-finding in concurrent programs via reduced interleaving instances / T. L. Nguyen, P. Schrammel, B. Fischer et al. // 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2017. — Oct. — Pp. 753–764.

42. Static Data Race Detection for Concurrent Programs with Asynchronous Calls / Vineet Kahlon, Nishant Sinha, Erik Kruus, Yun Zhang // Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. — ESEC/FSE '09. — New York, NY, USA: ACM, 2009. — Pp. 13–22. — URL: http://doi.acm.org/10.1145/1595696.1595701.

43. *Seidl Helmut, Vojdani Vesal*. Region Analysis for Race Detection // Proceedings of the 16th International Symposium on Static Analysis. — SAS '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 171–187. — URL: http://dx.doi.org/10.1007/978-3-642-03237-0_13.

44. Ad Hoc Synchronization Considered Harmful / Weiwei Xiong, Soyeon Park, Jiaqi Zhang et al. // Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. — OSDI'10. — Berkeley, CA, USA: USENIX Association, 2010. — Pp. 163–176. — URL: http://dl.acm.org/citation.cfm?id=1924943.1924955.

45. *Smith Adam R., Kulkarni Prasad A.* Localizing Globals and Statics to Make C Programs Thread-safe // Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems. — CASES '11. — New York, NY, USA: ACM, 2011. — Pp. 205–214. — URL: http://doi.acm.org/10.1145/2038698.2038730.

46. *Mukherjee Suvam, Kumar Arun, D'Souza Deepak*. Detecting All High-Level Dataraces in an RTOS Kernel // Verification, Model Checking, and Abstract Interpretation / Ed. by Ahmed Bouajjani, David Monniaux. — Cham: Springer International Publishing, 2017. — Pp. 405–423.

47. *Chopra Nikita, Pai Rekha, D'Souza Deepak*. Data Races and Static Analysis for Interrupt-Driven Kernels // Programming Languages and Systems / Ed. by Luís Caires. — Cham: Springer International Publishing, 2019. — Pp. 697–723.

48. *Cordeiro Lucas, Fischer Bernd*. Verifying Multi-threaded Software Using Smt-based Context-bounded Model Checking // Proceedings of the 33rd International Conference on Software Engineering. — ICSE '11. — New York, NY, USA: ACM, 2011. — Pp. 331–340. — URL: http://doi.acm.org/10.1145/1985793.1985839.

49. *Musuvathi Madanlal, Qadeer Shaz*. Iterative Context Bounding for Systematic Testing of Multithreaded Programs // *SIGPLAN Not.* — 2007. — jun. — Vol. 42, no. 6. — Pp. 446–455. — URL: http://doi.acm.org/10.1145/1273442.1250785.

50. SATABS: SAT-Based Predicate Abstraction for ANSI-C / Edmund Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by Nicolas Halbwachs, Lenore D. Zuck. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. — Pp. 570–574.

51. *Lahiri Shuvendu K., Qadeer Shaz, Rakamarić Zvonimir*. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers // Proceedings of the 21st International Conference on Computer Aided Verification. — CAV '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 509–524. — URL: http://dx.doi.org/10.1007/978-3-642-02658-4_38.

52. *DeLine Rob, Leino Rustan*. BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs // Technical Report MSR-TR-2005-70. — Microsoft Research, 2005. — March. — P. 13. — URL: https://www.microsoft.com/en-us/research/publication/boogiepl-a-typed-procedural-language-for-checking-object-oriented-programs/.

53. A Reachability Predicate for Analyzing Low-level Software / Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, Zvonimir Rakamaric // Proceedings of the 13th

International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — TACAS'07. — Berlin, Heidelberg: Springer-Verlag, 2007. — Pp. 19–33. — URL: http://dl.acm.org/citation.cfm?id=1763507.1763513.

54. *Ghafari Naghmeh, Hu Alan J., Rakamarić Zvonimir*. Context-bounded Translations for Concurrent Software: An Empirical Evaluation // Proceedings of the 17th International SPIN Conference on Model Checking Software. — SPIN'10. — Berlin, Heidelberg: Springer-Verlag, 2010. — Pp. 227–244. — URL: http://dl.acm.org/citation.cfm?id=1928137.1928160.

55. Symbolic Counter Abstraction for Concurrent Software / Gérard Basler, Michele Mazzucchi, Thomas Wahl, Daniel Kroening // Proceedings of the 21st International Conference on Computer Aided Verification. — CAV '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 64–78. — URL: http://dx.doi.org/10.1007/978-3-642-02658-4_9.

56. *Kahlon Vineet, Sankaranarayanan Sriram, Gupta Aarti*. Semantic Reduction of Thread Interleavings in Concurrent Programs // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by Stefan Kowalewski, Anna Philippou. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. — Pp. 124–138.

57. *Kahlon Vineet, Wang Chao, Gupta Aarti*. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique // Proceedings of the 21st International Conference on Computer Aided Verification. — CAV '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 398–413. — URL: http://dx.doi.org/10.1007/978-3-642-02658-4_31.

58. Optimal Dynamic Partial Order Reduction / Parosh Abdulla, Stavros Aronis, Bengt Jonsson, Konstantinos Sagonas // *SIGPLAN Not.* — 2014. — jan. — Vol. 49, no. 1. — Pp. 373–384. — URL: http://doi.acm.org/10.1145/2578855.2535845.

59. Effective verification of low-level software with nested interrupts / Daniel Kroening, Lihao Liang, Tom Melham et al. // Design, Automation & Test in Europe Conference & Exhibition (DATE 2015). — ACM, 2015. — mar. — Pp. 229–234. — URL: http://dl.acm.org/citation.cfm?id=2755803.

60. *Torre Salvatore, Madhusudan P., Parlato Gennaro*. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability // Proceedings of the 21st International Conference on Computer Aided Verification. — CAV '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 477–492. — URL: http://dx.doi.org/10.1007/978-3-642-02658-4_36.

61. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization / Omar Inverso, Ermenegildo Tomasco, Bernd Fischer et al. // Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. — New York, NY, USA: Springer-Verlag New York, Inc., 2014. — Pp. 585–602. — URL: http://dx.doi.org/10.1007/978-3-319-08867-9_39.

62. Verifying Concurrent Programs by Memory Unwinding / Ermenegildo Tomasco, Omar Inverso, Bernd Fischer et al. // Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035. — New York, NY, USA: Springer-Verlag New York, Inc., 2015. — Pp. 551–565. — URL: https://doi.org/10.1007/978-3-662-46681-0_52.

63. Model-based Kernel Testing for Concurrency Bugs through Counter Example Replay / Moonzoo Kim, Shin Hong, Changki Hong, Taeho Kim // *Electronic Notes in Theoretical Computer Science*. — 2009. — Vol. 253, no. 2. — Pp. 21 – 36. — Proceedings of Fifth Workshop on Model Based Testing (MBT 2009). URL: http://www.sciencedirect.com/science/article/pii/S1571066109004034.

64. *J.Holzmann Gerard*. The Model Checker SPIN. — Vol. 23. — IEEE Trans. on Software Engineering, 1997. — may. — Pp. 279–295.

65. *Atig Mohamed Faouzi, Bouajjani Ahmed, Qadeer Shaz*. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads // Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,. — TACAS '09. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 107–123. — URL: http://dx.doi.org/10.1007/978-3-642-00768-2_11.

66. Effective Verification for Low-Level Software with Competing Interrupts / Lihao Liang, Tom Melham, Daniel Kroening et al. // *ACM Trans. Embed. Com-*

put. Syst. — 2017. — dec. — Vol. 17, no. 2. — Pp. 36:1–36:26. — URL: http://doi.acm.org/10.1145/3147432.

67. *Deligiannis Pantazis, Donaldson Alastair F., Rakamaric Zvonimir*. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T) // Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). — ASE '15. — Washington, DC, USA: IEEE Computer Society, 2015. — Pp. 166–177. — URL: http://dx.doi.org/10.1109/ASE.2015.30.

68. *Flanagan Cormac, Qadeer Shaz*. Thread-modular Model Checking // Proceedings of the 10th International Conference on Model Checking Software. — SPIN'03. — Berlin, Heidelberg: Springer-Verlag, 2003. — Pp. 213–224. — URL: http://dl.acm.org/citation.cfm?id=1767111.1767125.

69. *Henzinger Thomas A., Jhala Ranjit, Majumdar Rupak*. Race Checking by Context Inference // Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. — PLDI '04. — New York, NY, USA: ACM, 2004. — Pp. 1–13. — URL: http://doi.acm.org/10.1145/996841.996844.

70. *Malkis Alexander, Podelski Andreas, Rybalchenko Andrey*. Thread-Modular Verification Is Cartesian Abstract Interpretation // Theoretical Aspects of Computing - ICTAC 2006 / Ed. by Kamel Barkaoui, Ana Cavalcanti, Antonio Cerone. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. — Pp. 183–197.

71. Thread-Modular Abstraction Refinement / Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Shaz Qadeer // Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings / Ed. by Warren A. Hunt, Fabio Somenzi. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. — Pp. 262–274. — URL: http://dx.doi.org/10.1007/978-3-540-45069-6_27.

72. Thread-Modular Shape Analysis / Alexey Gotsman, Josh Berdine, Byron Cook, Mooly Sagiv // *SIGPLAN Not.* — 2007. — jun. — Vol. 42, no. 6. — P. 266–277. — URL: https://doi.org/10.1145/1273442.1250765.

73. *Gupta Ashutosh, Popeea Corneliu, Rybalchenko Andrey*. Threader: A Constraint-Based Verifier for Multi-threaded Programs booktitle="Computer Aided Verifi-

cation / Ed. by Ganesh Gopalakrishnan, Shaz Qadeer. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. — Pp. 412–417.

74. *Gupta Ashutosh, Popeea Corneliu, Rybalchenko Andrey*. Predicate Abstraction and Refinement for Verifying Multi-threaded Programs // Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '11. — New York, NY, USA: ACM, 2011. — Pp. 331–344. — URL: http://doi.acm.org/10.1145/1926385.1926424.

75. *Gupta Ashutosh, Popeea Corneliu, Rybalchenko Andrey*. Predicate Abstraction and Refinement for Verifying Multi-threaded Programs // *SIGPLAN Not.* — 2011. — jan. — Vol. 46, no. 1. — Pp. 331–344. — URL: http://doi.acm.org/10.1145/1925844.1926424.

76. *Popeea Corneliu, Rybalchenko Andrey*. Threader: A Verifier for Multi-threaded Programs // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by Nir Piterman, Scott A. Smolka. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. — Pp. 633–636.

77. *Cohen Ariel, Namjoshi Kedar S.* Local Proofs for Global Safety Properties // *Form. Methods Syst. Des.* — 2009. — apr. — Vol. 34, no. 2. — Pp. 104–125. — URL: http://dx.doi.org/10.1007/s10703-008-0063-8.

78. VCC: Contract-based modular verification of concurrent C / M. Dahlweid, M. Moskal, T. Santen et al. // 2009 31st International Conference on Software Engineering - Companion Volume. — 2009. — May. — Pp. 429–430.

79. *Burnim Jacob, Sen Koushik*. Asserting and Checking Determinism for Multi-threaded Programs // Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. — ESEC/FSE '09. — New York, NY, USA: ACM, 2009. — Pp. 3–12. — URL: http://doi.acm.org/10.1145/1595696.1595700.

80. *Leino K. Rustan, Müller Peter*. A Basis for Verifying Multi-threaded Programs // Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. — ESOP '09. — Berlin, Heidelberg: Springer-Verlag,

2009. — Pp. 378–393. — URL: http://dx.doi.org/10.1007/978-3-642-00590-9_27.

81. *Beyer Dirk, Henzinger Thomas A., Théoduloz Grégory*. Configurable software verification: concretizing the convergence of model checking and program analysis // Proceedings of CAV. — Berlin, Heidelberg: Springer-Verlag, 2007. — Pp. 504–518.

82. *Beyer D., Henzinger T.A., Theoduloz G.* Program Analysis with Dynamic Precision Adjustment // Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on. — 2008. — sept. — Pp. 29–38.

83. *Beyer D., Keremoglu M.E., Wendler P.* Predicate Abstraction with Adjustable-Block Encoding // Formal Methods in Computer-Aided Design, 2010. FMCAD 2010. — 2010.

84. Bounded Model Checking Using Satisfiability Solving / Edmund Clarke, Armin Biere, Richard Raimi, Yunshan Zhu // Formal Methods in System Design. — 2001. — Pp. 7–34. — URL: https://doi.org/10.1023/A:1011276507260.

85. *Hoder Kryštof, Bjørner Nikolaj*. Generalized Property Directed Reachability // Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing. — SAT'12. — Berlin, Heidelberg: Springer-Verlag, 2012. — P. 157–171. — URL: https://doi.org/10.1007/978-3-642-31612-8_13.

86. Counterexample-guided abstraction refinement / E.M. Clarke, O. Grumberg, S. Jha et al. // *Proc. CAV, LNCS*. — 2000. — Vol. 1855. — Pp. 154–169.

87. *Мандрыкин М.У., Мутилин В.С., Хорошилов А.В.* Введение в метод CEGAR – уточнение абстракции по контрпримерам // *Труды ИСП РАН*. — 2013. — Т. 24. — С. 219–292.

88. *Craig William*. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory // *Journal of Symbolic Logic*. — 1957. — Sep. — Vol. 22, no. 3. — Pp. 269–285. — URL: https://www.cambridge.org/core/article/three-uses-of-the-herbrand-gentzen-theorem-in-relating-model-theory-and-proof-the7674DE501824D8FC294FB396CD5617DB.

89. Lazy Abstraction / Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Grégoire Sutre // *SIGPLAN Not.* — 2002. — jan. — Vol. 37, no. 1. — Pp. 58–70. — URL: http://doi.acm.org/10.1145/565816.503279.

90. *Lipton Richard J.* Reduction: A Method of Proving Properties of Parallel Programs // *Commun. ACM*. — 1975. — dec. — Vol. 18, no. 12. — Pp. 717–721. — URL: http://doi.acm.org/10.1145/361227.361234.

91. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph / Ron Cytron, Jeanne Ferrante, Barry K. Rosen et al. // *ACM Trans. Program. Lang. Syst.* — 1991. — oct. — Vol. 13, no. 4. — Pp. 451–490. — URL: http://doi.acm.org/10.1145/115372.115320.

92. *Beyer Dirk*. Software Verification and Verifiable Witnesses // Tools and Algorithms for the Construction and Analysis of Systems. — Vol. 9035 of *Lecture Notes in Computer Science*. — Springer Berlin Heidelberg, 2015. — Pp. 401–416. — URL: http://dx.doi.org/10.1007/978-3-662-46681-0_31.

93. Witness Validation and Stepwise Testification Across Software Verifiers / Dirk Beyer, Matthias Dangl, Daniel Dietsch et al. // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. — ESEC/FSE 2015. — New York, NY, USA: ACM, 2015. — Pp. 721–733. — URL: http://doi.acm.org/10.1145/2786805.2786867.

94. Exchanging Verification Witnesses between Verifiers / Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann // Software Engineering 2017 / Ed. by Jan Jürjens, Kurt Schneider. — Bonn: Gesellschaft für Informatik e.V., 2017. — P. 93.

95. Correctness Witnesses: Exchanging Verification Results between Verifiers / Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA: Association for Computing Machinery, 2016. — P. 326–337. — URL: https://doi.org/10.1145/2950290.2950351.

96. *Novikov Evgeny, Zakharov Ilja*. Towards Automated Static Verification of GNU C Programs // Perspectives of System Informatics / Ed. by Alexander K. Petrenko,

Andrei Voronkov. — Cham: Springer International Publishing, 2018. — Pp. 402–416.

97. *Novikov Evgeny, Zakharov Ilja*. Verification of Operating System Monolithic Kernels Without Extensions // Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice / Ed. by Tiziana Margaria, Bernhard Steffen. — Cham: Springer International Publishing, 2018. — Pp. 230–248.

98. *Beyer Dirk, Löwe Stefan, Wendler Philipp*. Reliable benchmarking: requirements and solutions // *International Journal on Software Tools for Technology Transfer*. — 2019. — Feb. — Vol. 21, no. 1. — Pp. 1–29. — URL: ”https://doi.org/10.1007/s10009-017-0469-y.

99. *Beyer Dirk, Friedberger Karlheinz*. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker // Proceedings of the 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2016, Telč, Czechia, October 21-23) / Ed. by J. Bouda, L. Holík, J. Kofroň et al. — EPTCS 233. — ArXiV, 2016. — Pp. 61–71.

100. *Мутилин В.С., Новиков Е.М., Хорошилов А.В.* Анализ типовых ошибок в драйверах операционной системы Linux // *Труды ИСП РАН*. — 2012. — Vol. 22. — Pp. 349–374.

Приложение А

Доказательство теорем

А.1. Доказательство теоремы 1

Нам нужно доказать следующее утверждение:

$$[\![CPA_{PS}(\mathbb{D}, e_0, \pi_0)]\!] \supseteq Reach_{\longrightarrow}([\![\{e_0\}]\!])$$

Доказательство состоит из двух частей. Сначала доказывается, что алгоритм 3 без использов ания структуры waitlist аппроксимирует сверху множество достижимых конкретных состояний. Вторым шагом показывается, что алгоритм 1 эквивалентен алгоритму 3.

Data: адаптивный статический анализ с частичными состояниями $\mathbb{D}$,
начальное абстрактное состояние e_0 с точностью $\pi_0 \in \Pi$,
множество $reached$ из элементов $E \times \Pi$
Result: множество достижимых абстрактных состояний $reached$
$reached := \{e_0, \pi_0\}$;
for *each* $e \in reached$ **do**
  for *each* e': $e \overset{reached}{\rightsquigarrow} (e', \pi')$ **do**
    $(\widehat{e}, \widehat{\pi}) = prec(e', \pi', reached)$;
    for *each* $(e'', \pi'') \in reached$ **do**
      $e_{new} = merge(\widehat{e}, e'', \widehat{\pi})$;
      if $e_{new} \neq e''$ **then**
        $reached := reached \setminus \{(e'', \pi'')\} \cup \{(e_{new}, \widehat{\pi})\}$;
      end
    end
    if $!stop(\widehat{e}, reached, \widehat{\pi})$ **then**
      $reached := reached \cup \{(\widehat{e}, \widehat{\pi})\}$;
    end
  end
end

Algorithm 3: $\overline{CPA}(\mathbb{D}, e_0, \pi_0)$

Доказательство. Нужно доказать, что $[\![\overline{CPA}(\mathbb{D}, e_0, \pi_0)]\!] \supseteq Reach_{\longrightarrow}([\![\{e_0\}]\!])$. Определим рекурсивную функцию $\overline{CPA}_n(\mathbb{D}, e_0, \pi_0)$, которая будет вычислять множество достижимых состояний алгоритма на n-той итерации. Обозначим ее для краткости $reached_n \equiv \overline{CPA}_n(\mathbb{D}, e_0, \pi_0)$.

$$
\begin{aligned}
&reached_0 = \{e_0\} \\
&s_n = \{e' \mid e \in reached_n \wedge e \overset{reached_n}{\rightsquigarrow} e'\} \\
&s'_n = \{\widehat{e} \mid \widehat{e} = prec(e', \pi', reached_n) \wedge e' \in s_n\} \\
&\widehat{s}_n = \{\overline{e} \mid \exists e^1 \in reached_n, \widehat{e} \in s'_n . \overline{e} = merge(\widehat{e}, e^1, \pi))\} \\
&\widehat{s}^1_n = reached_n \setminus \widehat{s}_n \\
&\widehat{s}^2_n = \widehat{s}_n \setminus reached_n \\
&\tilde{s}_n = \{e \mid e \in s' \wedge !(stop(e, reached_n, \pi))\} \\
&reached_{n+1} = reached_n \setminus \widehat{s}^1_n \cup \widehat{s}^2_n \cup \tilde{s}_n = \widehat{s}_n \cup \tilde{s}_n
\end{aligned}
\tag{A.1}
$$

Покажем, что

$$
[\![reached_{n+1} \cup \hat{R}]\!] \supseteq [\![Reach(reached_n) \cup \hat{R}]\!] \tag{A.2}
$$

$$
\begin{aligned}
&[\![reach_{n+1} \cup \hat{R}]\!] = (def.) = [\![reached_n \setminus \widehat{s}^1_n \cup \widehat{s}^2_n \cup \tilde{s}_n \cup \hat{R}]\!] \supseteq \\
&(eq.\ 2.2{,}2.6) \supseteq [\![reached_n \cup \tilde{s}_n \cup \hat{R}]\!] \supseteq (eq.\ 2.2{,}2.7) \\
&\supseteq [\![reached_n \cup s'_n \cup \hat{R}]\!] \supseteq (eq.\ 2.2{,}2.8) \supseteq [\![reached_n \cup s_n \cup \hat{R}]\!] = \\
&= (def.\ 2.4) = [\![Reach(reached_n) \cup \hat{R}]\!]
\end{aligned}
$$

Теперь докажем по индукции, что

$$
\forall \hat{R} \in E, k \in \mathbb{N}, [\![reached_{n+k} \cup \hat{R}]\!] \supseteq [\![Reach^k(reached_n) \cup \hat{R}]\!] \tag{A.3}
$$

При $k = 0$ отношение тривиально: $[\![reached_n \cup \hat{R}]\!] \supseteq [\![reached_n \cup \hat{R}]\!]$. Пусть теперь инвариант A.3 выполнен при некотором k, рассмотрим его при $k+1$:

$$
\begin{aligned}
&[\![reach_{n+k+1} \cup \hat{R}]\!] \supseteq (eq.\ .2) \supseteq [\![Reach(reached_{n+k}) \cup \hat{R}]\!] = \\
&= (def.) = [\![reached_{n+k} \cup \{e' \mid e \overset{reached_{n+k}}{\rightsquigarrow} e' \wedge e \in reached_{n+k}]\!] \cup \hat{R}\} \\
&\supseteq (eq.\ .3) \supseteq [\![Reach^k(reached_n) \cup \\
&\{e' \mid e \overset{reached_{n+k}}{\rightsquigarrow} e' \wedge e \in reached_{n+k}\} \cup \hat{R}]\!] \supseteq (eq.\ 2.3) \\
&\supseteq [\![Reach^k(reached_n) \cup \{e' \mid e \overset{Reach^k(reached_n)}{\rightsquigarrow} e' \wedge e \in Reach^k(reached_n)\} \\
&\cup \hat{R}]\!] = (def.\ 2.4) = [\![Reach^{k+1}(reached_n) \cup \hat{R}]\!]
\end{aligned}
$$

Таким образом, мы имеем,

$$
[\![reach_{n+k}]\!] \supseteq [\![Reach^k(reached_n)]\!] \supseteq (eq.\ 2.5) \supseteq \bigcup_{\tau \in [\![reached_n]\!]} \{\tau' \mid \tau \longrightarrow \tau'\} \tag{A.4}
$$

Когда алгоритм заканчивает свое выполнение (доходит до неподвижной точки) $reached_n = reached_{n+1}$. Обозначим финальное множество, как $Reached = lim_{n\to\infty}(reached_n)$. Нам остается доказать, что $\bigcup_{\tau \in [\![Reached]\!]} \{\tau' \mid \tau \longrightarrow \tau'\} \supseteq Reach_{\longrightarrow}([\![\{e_0\}]\!])$.

Покажем, что $\forall \{\tau_i\} \in \mathcal{T} : \tau_0 \in [\![\{e_0\}]\!] \wedge \forall 1 \leqslant k \leqslant N : \tau_k \longrightarrow \tau_{k+1} \implies \forall 1 \leqslant k \leqslant N : \tau_k \in [\![Reached]\!]$ по индукции. Алгоритм 3 не удаляет состояния, поэтому $e_0 \in Reached \vee e_0 \sqsubseteq e' \in Reached$. Таким образом, $\tau_0 \in [\![Reached]\!]$, то есть, базис индукции выполнен. Предположим, что утверждение выполнено для некоторого $n \in \mathbb{N}$:

$$
\begin{aligned}
&\forall 1 \leqslant k \leqslant n : \tau_k \in [\![Reached]\!] \\
&\forall \tau \in [\![Reached]\!] : \{\tau' \mid \tau \longrightarrow \tau'\} \subseteq (eq.\ 2.5) \subseteq [\![Reach^k(Reached)]\!] = \\
&= [\![Reached]\!] \implies \tau_{k+1} \in \{\tau' \mid \tau \longrightarrow \tau'\} \subseteq [\![Reached]\!]
\end{aligned} \tag{A.5}
$$

Так, мы показали, что $[\![\overline{CPA}(\mathbb{D}, e_0, \pi_0)]\!] \supseteq Reach_{\longrightarrow}([\![\{e_0\}]\!])$.

Теперь нам нужно показать, что алгоритм использующий очередь состояний ($waitlist$) эквивалентен алгоритму без нее, то есть, что $\forall e \in \overline{CPA}(\mathbb{D}, e_0, \pi_0) \exists e' \in CPA(\mathbb{D}, e_0, \pi_0) : e \sqsubseteq e'$.

Заметим, что итерации алгоритмов абсолютно одинаковые, поэтому изменения полученных состояний являются эквивалентными. Единственным отличием является то, что $CPA(\mathbb{D}, e_0, \pi_0)$ итерируется по очереди состояний, а

$\overline{CPA}(\mathbb{D}, e_0, \pi_0)$ – по всем возможным подмножествам R. Два алгоритма могут быть синхронизированы следующим образом: они начинают с одного и того же начального множества $reached$. Каждая итерация алгоритма $CPA(\mathbb{D}, e_0, \pi_0)$ повторяется алгоритмом $\overline{CPA}(\mathbb{D}, e_0, \pi_0)$. Так как они выполняют одинаковые действия и модифицируют множество $reached$ одинаково, получаемое множество $reached$ будет совпадать у обоих алгоритмов. Когда $CPA(\mathbb{D}, e_0, \pi_0)$ заканчивает выполнение, $\overline{CPA}(\mathbb{D}, e_0, \pi_0)$ продолжает выполнять все оставшиеся итерации.

Теперь нужно показать, что рассмотренных итераций алгоритма $CPA(\mathbb{D}, e_0, \pi_0)$ будет достаточно для построения финального множества $reached$.

По аналогии с предыдущим пунктом нам потребуется рекурсивное определение функций $reached_n$ и $waitlist_n$:

$$
\begin{aligned}
&reached_0 = \{e_0\} \\
&waitlist_0 = \{e_0\} \\
&e_n = get(waitlist_n) \text{ выдает первое состояние в очереди} \\
&s_n = \{e'_n \mid e_n \overset{reached_n}{\rightsquigarrow} e'_n\} \\
&s'_n = \{\widehat{e}_n \mid \widehat{e}_n = prec(e'_n, \pi', reached_n) \wedge e'_n \in s_n\} \\
&\widehat{s}_n = \{\overline{e_n} \mid \exists e^1_n \in reached_n, \widehat{e}_n \in s'_n . \overline{e_n} = merge(\widehat{e}_n, e^1_n, \pi))\} \\
&\widehat{s}^1_n = reached_n \setminus \widehat{s}_n \\
&\widehat{s}^2_n = \widehat{s}_n \setminus reached_n \\
&\tilde{s}_n = \{e_n \mid e_n \in s'_n \wedge !(stop(e_n, reached_n, \pi))\} \\
&reached_{n+1} = reached_n \setminus \widehat{s}^1_n \cup \widehat{s}^2_n \cup \tilde{s}_n = \widehat{s}_n \cup \tilde{s}_n \\
&waitlist_{n+1} = waitlist_n \setminus \{e_n\} \setminus \widehat{s}^1_n \cup \widehat{s}^2_n \cup \tilde{s}_n
\end{aligned}
\tag{A.6}
$$

Докажем следующий инвариант для всех итераций алгоритмов.

$$
\begin{aligned}
&\forall e \in reached_n : \\
&e \in waitlist_n \vee (\forall \widehat{R} \subseteq E : [\![\{e' \mid e \overset{reached_n}{\rightsquigarrow} e'\} \cup reached_n \cup \widehat{R}]\!] \subseteq [\![reached_n \cup \widehat{R}]\!])
\end{aligned}
\tag{A.7}
$$

Этот инвариант означает, что для любого перехода из $reached_n$ либо он находится в $waitlist_n$, либо все следующие переходы тоже находятся в $reached_n$, то есть в процессе анализа переходы не теряются.

Докажем инвариант A.7 по индукции. Для начального шага инвариант выполнен: $reached_0 = \{e_0\} \wedge waitlist_0 = \{e_0\} \implies e_0 \in waitlist_0$. Пусть теперь ин-

вариант выполнен для некоторой итерации k. Рассмотрим следующую итерацию $k+1$ и возьмем случайный переход $e \in reached_{k+1}$. Возможны четыре варианта:

1. $e \in (reached_{k+1} \setminus reached_n) \setminus \{e_n\}$. Для этого состояния ничего не меняется, для него был выполнен инвариант на предыдущей итерации, и будет выполнен на этой.
2. $e = e_k$. На прошлой итерации инвариант был выполнен, так как $e_k \in waitlist_k$, но $e_k \notin waitlist_{k+1}$. По определению $reached_{k+1} = reached_k \setminus \widehat{s}_k^1 \cup \widehat{s}_k^2 \cup \tilde{s}_k$ и $[\![reached_{k+1} \cup \widehat{R}]\!] = [\![reached_k \setminus \widehat{s}_k^1 \cup \widehat{s}_k^2 \cup \tilde{s}_k \cup \widehat{R}]\!] \supseteq (eq.\ 2.6) \supseteq [\![reached_k \cup \tilde{s}_k \cup \widehat{R}]\!] \supseteq (eq.\ 2.7) \supseteq [\![reached_k \cup s'_k \cup \widehat{R}]\!] \supseteq (eq.\ 2.8) \supseteq [\![reached_k \cup \{e' \mid e \overset{reached_k}{\rightsquigarrow} e'\} \cup \widehat{R}]\!]$. Откуда следует $[\![\{e' \mid e \overset{reached_{k+1}}{\rightsquigarrow} e'\} \cup reached_{k+1} \cup \widehat{R}]\!] \subseteq [\![reached_{k+1} \cup \widehat{R}]\!])$.
3. $e \in \widehat{s}_{k+1}^2$. По определению $waitlist_{k+1}$ это означает, что $\widehat{s}_{k+1}^2 \subseteq waitlist_{k+1} \Longrightarrow e \in waitlist_{k+1}$. А значит, инвариант выполнен.
4. $e \in \tilde{s}_k$. Опять же, по определению $waitlist_{k+1}$ это означает, что $\tilde{s}_k \subseteq waitlist_{k+1} \Longrightarrow e \in waitlist_{k+1}$. А значит, инвариант выполнен.

Инвариант A.7 выполнен на всех итерациях алгоритма $CPA(\mathbb{D}, e_0, \pi_0)$, включая последнюю, в которой $waitlist_k = \varnothing$, $reached_k = Reached$, поэтому $[\![\{e' \mid e \overset{Reached}{\rightsquigarrow} e'\} \cup Reached]\!] \subseteq [\![Reached]\!]$. Отсюда следует, что все остальные операции алгоритма $\overline{CPA}(\mathbb{D}, e_0, \pi_0)$ являются бесполезными. Действительно, пусть $\overline{CPA}(\mathbb{D}, e_0, \pi_0)$ нашел переход $e \in Reached : e \overset{Reached}{\rightsquigarrow} e' \wedge e' \notin Reached$. Используя инвариант мы получаем, что $[\![\{e'\} \cup Reached]\!] \subseteq [\![Reached]\!]$. То есть, этот абстрактный переход не дает новых конкретных переходов. Отсюда следует, что

$[\![\overline{CPA}(\mathbb{D}, e_0, \pi_0)]\!] \subseteq (req.\ 2.2) \subseteq [\![CPA(\mathbb{D}, e_0, \pi_0)]\!]$

Мы доказали следующее неравенство:

$$Reach_{\rightarrow}([\![\{e_0\}]\!]) \subseteq [\![\overline{CPA}(\mathbb{D}, e_0, \pi_0)]\!] \subseteq [\![CPA(\mathbb{D}, e_0, \pi_0)]\!]$$

□

A.2. Доказательство условия 2.13 для анализа без абстракции

Нам нужно доказать условие 2.13, связывающее операторы $transfer$, $\bigoplus$ и $apply$.

Доказательство. Рассмотрим случайный переход $\tau \longrightarrow \tau'$, где $\tau = (c,t,g), \tau' = (c',t',g')$ и предположим, что $\exists t_0, t_1, \ldots, t_n \in T, t_i \neq t_j$ $\tau \in \bigoplus_I ((e_0,t_0), \{(e_1,t_1), \ldots, (e_n,t_n)\})$. По определению 2.23 это означает, что $s_i = (t_i, c(t_i), c_l(t_i), c_g, c_s)$. По определению $\rightsquigarrow_Q$ существует состояние $s'_0 = (t_0, c'(t_0), c'_l(t_0), c'_g, c'_s)$. Так как $\{e_i\}$ были совместны (сра-оператор $check_C(\{e_0, \ldots, e_n\})$, это означает, что все переходы в окружении соответствует переходу: $\forall 1 \leqslant i \leqslant n : q_i = (gl, s, gl', s')$. Таким образом, следующие состояния имеют вид $e_i \rightsquigarrow e'_i$, $e'_i = (s'_i, t_i, q'_i)$, где $s'_i = (t_i, c(t_i), c_l(t_i), c'_g, c'_s)$. То есть, глобальные части других частичных состояний меняются в соответствии с переходом в основном потоке. При этом по определению $\rightsquigarrow_Q$, который выдает все возможные переходы, $\exists 0 \leqslant k \leqslant n : g' = q'_k$, то есть новый переход в потоке должен содержатся среди полученных после переходов в окружении. Зафиксируем это значение k и рассмотрим проекцию этого нового перехода в потоке: $e'_k|_p = e'_p = (s'_k, t_k, (gl', s', gl'', cs''))$. Глобальное часть состояний s'_i была получена с помощью одинакового перехода, поэтому она удовлетворяет условию совместности $compatible_Q$. Поэтому можно применить сра-оператор $apply$ $\forall 0 \leqslant i \leqslant n, i \neq k : \tilde{e_i} = apply(e'_i, e'_p) = (s'_i, t_i, (gl', s', gl'', cs''))$. И тогда, состояния $e'_k, \tilde{e'_0}, \ldots, \tilde{e'_{k-1}}, \tilde{e'_{k+1}}, \ldots, \tilde{e'_n}$ удовлетворяют условию $check_C$. А значит, могут быть объединены в глобальный переход сра-оператором $\bigoplus$, при этом τ' будет в этом множестве по его построению.

В доказательстве опущен второй случай с операцией создания потока thread_create. Рассуждения полностью повторяют описанные выше с той лишь поправкой, что они проводятся для $n+1$ элемента. □

А.3. Доказательство условия 2.26 сра-оператора $strengthen$

Покажем, что требование 2.26 выполнено для простого сра-оператора $\downarrow$ (определение 2.25).

Доказательство. Дакажем условие от противного. Предположим, что

$$\exists \tau \in \mathcal{T}, e_0, \dots, e_n \in E, t_0, \dots, t_n \in T :$$
$$\tau \in \bigoplus\nolimits_{\mathscr{C}} ((e_0, t_0), \{(apply_{\mathscr{C}}(e_1, e_0), t_1), \dots, (apply_{\mathscr{C}}(e_n, e_0), t_n)\})$$
$$\tau \notin \bigoplus\nolimits_{\mathscr{C}} ((\downarrow e_0, t_0), \{(apply_{\mathscr{C}}(\downarrow e_1, \downarrow e_0), t_1), \dots, (apply_{\mathscr{C}}(\downarrow e_n, \downarrow e_0), t_n)\})$$

По определению 2.24 это означает, что существует некоторый внутренний CPA, для которого

$$\exists \tau \in \mathcal{T}, e_0^i, \dots, e_n^i \in E_I, t_0, \dots, t_n \in T :$$
$$\tau \in \bigoplus\nolimits_{I} ((e_0^i, t_0), \{(e_1^i, t_1), \dots, (e_n^i, t_n)\})$$
$$\tau \notin \bigoplus\nolimits_{I} ((\tilde{e}_0^i, t_0), \{(apply_I(\tilde{e}_1^i, \tilde{e}_0^i), t_1), \dots, ((apply_I(\tilde{e}_n^i, \tilde{e}_0^i), t_n)\})$$

Где $\downarrow e_j = (\tilde{e}_j^1, \dots, \tilde{e}_j^m)$.

Учитывая явный вид перехода $e = (s,q)$ и вид cpa-оператора epp_I(определение 2.16), а также то, что cpa-оператор усиления не меняет абстрактного состояния (определение 2.25), получаем, что можно исключить из рассмотрения абстрактные состояния:

$$\exists g \in G, e_0^i, \dots, e_n^i \in E_I, t_0, \dots, t_n \in T :$$
$$g \in \|q_0^i\| \cap \|q_1^i\| \cap \dots \cap \|q_n^i\|$$
$$g \notin \|\tilde{q}_0^i\| \cap \|\tilde{q}_1^i\| \cap \dots \cap \|\tilde{q}_n^i\|$$

Это означает, что $\exists 0 \leqslant j \leqslant n : q_j^i \neq \tilde{q}_j^i$. При этом не обязательно единственный. По определению 2.25 отсюда следует, что $\exists g' \in G, g \neq g' : \exists 0 \leqslant j \leqslant n : \tilde{q}_j^i = g'$, так как только в этом случае дуга может измениться. Однако, по тому же определению 2.25 такое изменение возможно только если дуга g' является единственно возможной, а значит, $\tau = (c, g', t)$, что противоречит условию $g' \neq g$. А значит,предположение было неверно, и условие 2.26 выполнено. □

A.4. Доказательство условия 2.13 для CompositeCPA

Нам нужно показать, что требование 2.13 выполнено для $\mathscr{C}$, если требование 2.27 выполнены для всех его внутренних элементов.

Доказательство. Рассмотрим случайный конкретный переход:

$$\tau \in \bigoplus\nolimits_{\mathscr{C}} \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right)$$

где $e_i = (e_i^1, \ldots, e_i^n)$.

По определению 2.24, это означает что

$$\forall j : 1 \leqslant j \leqslant m : \tau \in \bigoplus\nolimits_{\Delta_j} \left(\begin{pmatrix} e_0^j \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1^j \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m^j \\ t_m \end{pmatrix} \right\} \right)$$

Возьмем случайный переход $\tau \longrightarrow \tau', \tau = (c,g,t), g \in G, t \in T$, и покажем, что $\exists e'_0, \ldots, e'_m, \tilde{e}'_0, \ldots, \tilde{e}'_{k-1}, \tilde{e}'_{k+1}, \ldots, \tilde{e}'_m \in E$ и $\tau' \in \bigoplus_{\mathscr{C}} \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} \tilde{e}'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} \tilde{e}'_m \\ t_m \end{pmatrix} \right\} \right)$ и при этом $\tilde{e}'_0, \ldots, \tilde{e}'_m$ будут получены из $e'_0, \ldots, e'_m$, а они – из $e_0, \ldots, e_m$ указанными в 2.13 способами.

Сначала рассмотрим случай $g \neq thread\_create$

Используя требование 2.27 для внутренних элементов, получаем, что

$$\begin{aligned} &\forall 0 \leqslant j \leqslant m : \exists e_0^{j\prime}, e_1^{j\prime}, \ldots, e_n^{j\prime} \in E : \\ &\forall 1 \leqslant i \leqslant n : e_i^j \overset{R}{\rightsquigarrow} e_i^{j\prime} : \exists 1 \leqslant k \leqslant n : t_k = t' \wedge \\ &\tau' \in \bigoplus\nolimits_I ((e_k^{j\prime}, t_k), \{(\tilde{e_i^j}, t_i) \mid \tilde{e_i^j} = apply(e_i^{j\prime}, e_k^{j\prime}) \wedge i \neq k)\}) \end{aligned}$$

Заметим, что k является одним и тем же для всех внутренних элементов, так как есть только одно $t_k = t'$, а множество $t_0, t_1, \ldots, t_n$ – общее для всех CPA. Поэтому можно объединить все внутренние переходы следующим способом: $\forall 1 \leqslant i \leqslant n : e'_i = (e_i^{1\prime}, \ldots, e_i^{m\prime})$ Таким образом, $\tau' \in \bigoplus_{\mathscr{C}} ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\})$. По условию 2.26 $\tau' \in \bigoplus_{\mathscr{C}} ((\downarrow e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(\downarrow e'_i, \downarrow e'_k) \wedge i \neq k)\})$.

По определению $transfer$ в композитном анализе, получаем, что $\forall 1 \leqslant i \leqslant n : e_i \rightsquigarrow\downarrow e'_i$, а по определению $apply$: $\forall 1 \leqslant i \leqslant n : \tilde{e}_i = apply(e'_i, e'_k) = (\tilde{e_i^0}, \ldots, \tilde{e_i^m})$. А это означает, что выполнено условие 2.13. Это как раз то, что нам нужно было доказать.

Рассмотрим случай $g = thread\_create$.

Используя требование 2.27 для внутренних элементов, получаем, что

$$\exists e'_0, e'_1, \ldots, e'_{n+1} \in E, \forall 0 \leqslant i \leqslant n :$$
$$\begin{cases} (s_0, (l, tc_{parent}, l')) \overset{R}{\rightsquigarrow} e'_0 \\ (s_0, (l, tc_{child}, l')) \overset{R}{\rightsquigarrow} e'_{n+1} \\ e_i \rightsquigarrow e'_i, \text{ если } i \neq 0 \end{cases}$$
$$\exists 0 \leqslant k \leqslant n+1 : t_k = t' \wedge$$
$$\tau' \in \bigoplus\nolimits_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\})$$

Заметим, что k является одним и тем же для всех внутренних элементов, так как есть только одно $t_k = t'$, а множество $t_0, t_1, \ldots, t_n$ – общее для всех CPA. Поэтому можно объединить все внутренние переходы следующим способом: $\forall 1 \leqslant i \leqslant n+1 : e'_i = (e_i^{1\prime}, \ldots, e_i^{m\prime})$. Таким образом, $\tau' \in \bigoplus_{\mathscr{C}} ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\})$. По условию 2.26 $\tau' \in \bigoplus_{\mathscr{C}} ((\downarrow e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(\downarrow e'_i, \downarrow e'_k) \wedge i \neq k)\})$.

По определению $transfer$ в композитном анализе, получаем, что $\forall 1 \leqslant i \leqslant n : e_i \rightsquigarrow\downarrow e'_i \wedge e_0 \rightsquigarrow\downarrow e'_{n+1}$, а по определению $apply$: $\forall 1 \leqslant i \leqslant n : \tilde{e}_i = apply(e'_i, e'_k) = (\tilde{e_i^0}, \ldots, \tilde{e_i^m})$. А это означает, что выполнено условие 2.13. Это как раз то, что нам нужно было доказать. □

A.5. Доказательство условия 2.27 для ThreadCPA

Нам нужно доказать, что определенные в разделе 2.8 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайный конкретный переход:

$$\tau \in \bigoplus\nolimits_T \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right)$$

.

Если $op \neq thread\_create$, по определению $transfer$ $e'_i = (s_i, \top_T^T)$, то есть состояние не изменяется, а переходы рассматриваются все возможные. А значит,

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e'_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix} \right\} \right)$$

Если $op = tc_{Child} \vee op = tc_{Parent}$ состояние меняется только у дочернего потока: $e_{m+1} = (l_\nu, \top_T^T)$. То есть,

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix}, \begin{pmatrix} e'_{m+1} \\ l_\nu \end{pmatrix} \right\} \right)$$

□

A.6. Доказательство условия 2.27 для LocationCPA

Нам нужно доказать, что определенные в разделе 2.11 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайный конкретный переход $\tau \longrightarrow \tau'$ и такие абстрактные переходы $e_0, \ldots, e_n$, что :

$$\tau \in \bigoplus\nolimits_L \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right)$$

.

По определениям 2.16, 2.31 это означает, что $\tau = (c, g, t_0)$,

$$c_{pc} = \left\{ \begin{array}{c} t_0 \to l_0, \\ \ldots \\ t_m \to l_m \end{array} \right\}.$$

где $e_i = (s_i, q_i)$, $l_i \in loc(s_i)$, $q_0 \in G$, $q_0 = (l_0, op, l'_0)$.

Рассмотрим случай $g \neq thread\_create$. После перехода изменяется счетчик комманд только нулевого потока: $l'_0 \in loc(s'_0)$.

$$c'_{pc} = \left\{ \begin{array}{c} t_0 \to l'_0, \\ \ldots, \\ t_m \to l_m \end{array} \right\}.$$

Это означает, что

$$c' \in \bigoplus\nolimits_L^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix} \ , \ldots \ , \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right)$$

Теперь рассмотрим поток k в котором совершается переход $\tau' = (c', g', t_k)$. По определению сра-оператора $transfer\ e_k \rightsquigarrow e'_k$, $e'_k = (s'_k, g_k)$, где g_k – включает в себя все дуги, по которым возможны переходы из данной точки программы, то есть и g'. По определению $compatible$ полученные состояния будут совместны, а значит, можно будет применить сра-оператор $apply$, то есть $\tau' \in \bigoplus_I ((e'_k, t_k), \{(\tilde{e}_i, t_i) \mid \tilde{e}_i = apply(e'_i, e'_k) \wedge i \neq k)\})$.

Для случая $g = tc_{Child}(l_\nu)$ или $g = tc_{Parent}(l_\nu)$ доказательство принципиально ничем не отличается от предыдущего случая, так как никакой новой семантике в этих дугах нет. □

A.7. Доказательство условия 2.27 для PredicateCPA

Нам нужно доказать, что определенные в разделе 2.12 сра-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайное множество абстрактных переходов $e_0, e_1, \ldots, e_n$ и конкретный переход $\tau \in \bigoplus_P \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)$

Возьмем некоторый следующий переход $\tau \longrightarrow \tau'$, где $\tau = (c, g, t_0)$ Нам нужно показать, что $\exists e'_0, e'_1, \ldots, e'_n \in E_P$ и $\tau' \in \bigoplus_P \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)$.

Не будем подробно рассматривать случаи $g = thread\_create$, так как этот переход не меняет состояния анализа предикатов и не меняет состояние памяти (c_l и c_g). А значит, $s_i = s'_i$. Поэтому в этом случае $c' \in \bigoplus^S_P \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) = \bigoplus^S_P \left(\begin{pmatrix} s_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right)$.

Рассмотрим теперь случай $op = assign(x, exp)$. Зафиксируем значение k такое, что $t_k = t'$. Будем рассматривать переход в потоке из состояния s_k.

$e_k \rightsquigarrow e'_k$, $e'_k|_p = (\theta_{X^{local},env}(s'_k), \theta_{X^{local},env}(SP_{q'}(s'_k))$. Начальные состояния s_i были совместными, так как к ним был применим сра-оператор $\bigoplus$. Отсюда следует, что существовала общая модель $\hat{v_g} \models s_i$. По определению сра-оператора $assign$, который меняет только значение переменной x на новое, $c'_g = \hat{v_g}'$, где $\hat{v_g}'$ совпадает с $\hat{v_g}$ для всех переменных, кроме x. По определению сра-оператора перехода $s'_k = SP_{op}(s_k)$, а $s'_i = s'_i[x \to \hat{x}] \wedge SP_{op}(true) = SP_{op}(s_i)$. Таким обра-

зом, частичные состояния всех потоков меняются согласованно, и по определению $SP_{op}(\varphi)$ новое $\hat{v_g}'$ должно являться моделью для новых абстрактных состояний. Теперь необходимо проверить, что для новых переходов будет выполнен cpa-оператор $apply$, то есть, переходы будут совместны. Мы показали, что состояния останутся совместными, и осталось показать, что совместными останутся дуги, то есть, согласно 2.16, что $\forall g \in G : g \in \|g\| \cap \|\theta_{X^{local},env}(SP_g(true))\|$. Очевидно, что $g \in \|\theta_{X^{local},env}(SP_g(true))\|$, и условие выполнено. Таким образом мы показали, что условие 2.13 выполнено для анализа предикатов.

□

A.8. Доказательство условия 2.27 для LockCPA

Нам нужно доказать, что определенные в разделе 2.13 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайное множество абстрактных переходов $e_0, e_1, \ldots, e_n$ и конкретный переход $\tau \in \bigoplus_S \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)$

Возьмем некоторый следующий переход $\tau \longrightarrow \tau'$, где $\tau = (c, g, t_0)$ Так как предикатный анализ выдает все возможные следующие дуги, достаточно показать, что $\exists s'_0, s'_1, \ldots, s'_n \in E_P^S$ и $' \in \bigoplus_S^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right)$.

Не будем подробно рассматривать случаи $op = thread\_create$, $op = assign$, $op = assume$, так как эти переходы не меняет состояния анализа примитивов синхронизации и не меняют состояние c_s. А значит, $s_i = s'_i$. Поэтому в этом случае $c' \in \bigoplus_S^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) = \bigoplus_P^S \left(\begin{pmatrix} s_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right)$.

Рассмотрим теперь случай $op = acquire(\hat{s})$. Зафиксируем значение k такое, что $t_k = t'$. Будем рассматривать переход в потоке из состояния s_k.

$c \xrightarrow{t, acquire(\hat{s})} c', \; s_k \overset{acquire(\hat{s})}{\rightsquigarrow} s'_k$, нужно показать, что $c' \in \bigoplus_P^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right)$

По определению отношения переходов $c'_s(\hat{s}) = t_k$, $s'_k = s_k \cup \{\hat{s}\}$. Заметим, что если переход есть на конкретных состояниях, это означает, что блокировка $\hat{s}$

не захвачена ни одним из потоков $t_1, \ldots, t_n$. А это, в свою очередь, означает, что $s \notin s_i, 1 \leqslant i \leqslant n, i \neq k$. Таким образом, новое состояние s'_k остается совместным с остальными частичными состояниями $s'_i = s_i, i \neq k$. По определению $\bigoplus_S$

$$\bigoplus_P^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) = \{c \in C \mid \hat{s} \in s_i \implies c_s(\hat{s}) = t_i\} = C_0$$

Очевидно, $c' \in C_0$. Аналогично можно проверить, что операция $g = release(\hat{s})$ удовлетворяет условию 2.27.

□

A.9. Доказательство условия 2.27 для ValueCPA

Нам нужно доказать, что определенные в разделе 2.14 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайное множество абстрактных переходов $e_0, e_1, \ldots, e_n$ и конкретный переход $\tau \in \bigoplus_V \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)$

Возьмем некоторый следующий переход $\tau \longrightarrow \tau'$, где $\tau = (c, g, t_0)$ Так как анализ явных значений выдает все возможные следующие дуги, достаточно показать, что $\exists s'_0, s'_1, \ldots, s'_n \in E_V^S$ и $' \in \bigoplus_V^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right)$.

Не будем подробно рассматривать случаи $op = thread\_create, op = acquire, op = release$, так как эти переходы не меняет состояния анализа явных значений и не меняют состояние памяти c_g и c_l. А значит, $s_i = s'_i$. Поэтому в этом случае $c' \in \bigoplus_V^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right) = \bigoplus_V^S \left(\begin{pmatrix} s_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s_n \\ t_n \end{pmatrix} \right\} \right)$.

Рассмотрим теперь случай $op = assign(x, e)$. Зафиксируем значение k такое, что $t_k = t'$. Будем рассматривать переход в потоке из состояния s_k.

$c \xrightarrow{t, assign(x,e)} c', \; s_k \overset{assign(x,e)}{\rightsquigarrow} s'_k$, нужно показать, что $c' \in \bigoplus_V^S \left(\begin{pmatrix} s'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} s'_n \\ t_n \end{pmatrix} \right\} \right)$

Начальные состояния s_i были совместными, так как к ним был применим cpa-оператор $\bigoplus$. Отсюда следует, что существовала общая модель $\hat{v_g} \models s_i$. Заметим, что если переход есть на конкретных состояниях, это означает, что $c_g(x) =$

$e_{/c}$, если $x \in X^{global}$ или $c_l(t_k)(x) = e_{/c}$, если $x \in X^{local}$. Рассмотрим присваивание, меняющее значение глобальной переменной, как наиболее сложный. По определению отношения переходов $s'_k(x) = e_{/s}$, а значения остальных переменных не изменяется. По определению перехода в окружении $s'_i(x) = q(x) = e$, а значения остальных переменных не изменяются. Таким образом, частичные состояния всех потоков меняются согласованно, и новое отображение $\hat{v_g}'$ должно являться моделью для новых абстрактных состояний, так как и состояния, и модель изменили только одно значение переменной x. Теперь необходимо проверить, что для новых переходов будет выполнен cpa-оператор $apply$, то есть, переходы будут совместны. Мы показали, что состояния останутся совместными, и осталось показать, что совместными останутся дуги, то есть, согласно 2.16, что $\forall g \in G : g \in \|g\| \cap \|x \to e\|$. Очевидно, что $(\cdot, assign(x, e), \cdot) \in \|x \to e\|$, и условие выполнено. Таким образом мы показали, что условие 2.13 выполнено для анализа предикатов.

□

A.10. Доказательство условия 2.27 для ThreadCPA с эффектами окружения

Нам нужно доказать, что определенные в разделе 2.9 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайный конкретный переход $\tau = (c, g, t_0)$, $g = (\cdot, op, \cdot)$:

$$\tau \in \bigoplus\nolimits_T \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right)$$

.

Если $op \neq thread\_create$, по определению $transfer$ $e'_i = (s_i, \top^T_T)$, то есть состояние не изменяется, а переходы рассматриваются все возможные. Таким образом,

$$c' \in \bigoplus\nolimits_T^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_m \\ t_m \end{pmatrix} \right\} \right) = \bigoplus\nolimits_T^S \left(\begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s'_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s'_m \\ t_m \end{pmatrix} \right\} \right)$$

А значит, для $\tau' = (c', g', t_k)$:

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix} \right\} \right)$$

Рассмотрим теперь случай $op = thread\_create(l_\nu)$. При обычном переходе меняются состояния нулевого потока: $C'_0 = C_0 \cup \{l_\nu\}$, и появляется новый переход, соответствующий созданному потоку: $e_{m+1} = ((l_\nu, C_0 \cup \{l_\nu\}), \top_T^T)$.

По определению $\bigoplus_T$ все остальные переходы в окружении должны были содержать в себе абстрактную дугу, которая соответсвует переходу в потоке, то есть $q_i = (create, l_\nu)$. Таким образом, $\forall 1 \leqslant i \leqslant m, C_i = C'_0$, то есть состояния остаются совместными, а значит, cpa-оператор $apply$ будет применим. Поэтому,

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix}, \begin{pmatrix} e'_{m+1} \\ l_\nu \end{pmatrix} \right\} \right)$$

□

A.11. Доказательство условия 2.27 для расширенного ThreadCPA, инвариантного к эффектам окружения

Нам нужно доказать, что определенные в разделе 2.10 cpa-операторы $transfer$ и $apply$ удовлетворяют условию 2.27.

Доказательство. Рассмотрим случайный конкретный переход $\tau = (c, g, t_0)$, $g = (\cdot, op, \cdot)$:

$$\tau \in \bigoplus\nolimits_T \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right)$$

.

Если $op \neq thread\_create$, по определению $transfer$ $e'_i = (s_i, \top_T^T)$, то есть состояние не изменяется, а переходы рассматриваются все возможные. Таким образом,

$$c' \in \bigoplus\nolimits_T^S \left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_m \\ t_m \end{pmatrix} \right\} \right) = \bigoplus\nolimits_T^S \left(\begin{pmatrix} s'_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} s'_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s'_m \\ t_m \end{pmatrix} \right\} \right)$$

А значит, $\tau' = (c', g', t_k)$:

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix} \right\} \right)$$

Рассмотрим теперь случай $op = thread\_create(l_\nu)$. При обычном переходе меняются состояния нулевого потока: $s'_0 = s_0 \cup \{(l_\nu, Parent)\}$, и появляется новый переход, соответствующий созданному потоку: $e_{m+1} = (s \cup \{(l_\nu, Child)\}, \top_T^T)$.

По определению $\bigoplus_T$ все остальные переходы в окружении должны были содержать в себе абстрактную дугу, которая соответсвует переходу в потоке, то есть $q_i = (create, l_\nu)$. Основной вопрос заключается в том, будут ли новые состояния совместными (в смысле сра-оператора $check_C$) друг с другом. Заметим, что все исходные были совместными, то есть, для каждой пары существовал общий элемент. Так как $\forall 1 \leqslant i \leqslant m s'_i = s_i$, то $\forall 0 \leqslant j \leqslant m : check((s_j, t_j), \{(s_0, t_0), \ldots, (s_m, t_m)\}) = true$. Теперь нужно проверить, что $\forall 0 \leqslant j \leqslant m : compatible(e'_j, e'_{m+1}) = true$. Для $j = 0$ совместность обеспечивается идентификатором потока l_ν, которая встречается в двух переходах с противоположными флагами. Для $j \neq 0$ совместность обеспечивается идентификатором того же потока, который обеспечивал совместность переходов e_0 и e_j.

Таким образом, $\forall 0 \leqslant i \leqslant m, check((s_j, t_j), \{(s_0, t_0), \ldots, (s_m, t_m)\}) = true$, а значит, сра-оператор $\bigoplus$ и сра-оператор $apply$ применимы. Поэтому,

$$\tau' \in \bigoplus\nolimits_T \left(\begin{pmatrix} e'_k \\ t_k \end{pmatrix}, \left\{ \begin{pmatrix} e'_0 \\ t_0 \end{pmatrix}, \ldots, \begin{pmatrix} e'_m \\ t_m \end{pmatrix}, \begin{pmatrix} e'_{m+1} \\ l_\nu \end{pmatrix} \right\} \right)$$

□

Приложение Б

Описание коммитов, содержащих исправления ошибок связанных с состоянием гонки

Коммит: f1a8a3f

Данный патч исправляет состояние гонки между функциями *bond_store_updelay/* и *bond_store_miimon*, которое может привести к делению на ноль. Важной особенностью данного патча является то, что он не полностью исправляет ошибку: для двух функций *bond_store_updelay*/*downdelay* добавляется захват блокировки *rtnl*, однако, второй поток *bond_store_miimon*, в котором производится установка нулевого значения, по-прежнему вызывается без блокировок. Таким образом, состояние гонки остается возможным. Данный коммит был включен в версию Linux v2.6.32.61, при том что в версию Linux v3.7-rc8 был ранее включен коммит fbb0c41, который добавлял соответствующую блокировку в *bond_store_miimon*. Однако, он включал в себя еще несколько изменений и, видимо, поэтому не был в это же время применен к ветке 2.6.

CPALockator обнаруживает возможное состояние гонки между *bond_store_miimo* и *bond_change_active_slave*, который печатает значение под блокировкой *rtnl*. Так как CPALockator выдает только одно предупреждение для каждой переменной, другой путь к *bond_store_updelay/downdelay* не печатается. Таким образом, нельзя заключить, что была найдена исходная ошибка, однако, найденное им предупреждение является истинным, которое не было исправлено в версии Linux v2.6.32.61, но исправляется в fbb0c41.

Для обнаружения этой ошибки потребовалось добавление модели окружения для трех структур *device_attributes*.

Коммит: 883f30e

Данный коммит исправлет потенциальное состояние гонки при доступе к данным пользователя из функций *snd_ctl_elem_user_get*, *snd_ctl_elem_user_put* и *snd_ctl_elem_user_tlv*. Данные функции вызываются через функциональные указатели, например, *snd_ctl_elem_user_put* вызывается в функции *master_put*, которая выставляется как обработчик *put* в некоторой структуре. В подготовленной верификационной задаче присутствует только один вызов по этому указателю (из

snd_ctl_elem_write), который производится под семафором *card->controls_rwsem*. Этим же семафором защищаются парные вызовы. Из описания коммита остается неясным, какой именно стек вызова приводит к ошибке. Скорее всего, эти функции могут вызываться из другого модуля, но определить из какого не удалось. При отключении поддержки семафоров данное состояние гонки находится. Изменений модели окружения не потребовалось.

Коммит: c99bd4f

Коммит исправляет ошибку связанную с использованием памяти после ее освобождения (англ. use-after-free). Теоретически, некоторый подкласс таких ошибок может быть найден с помощью инструмента CPALockator, если освобождение и доступ к памяти может осуществляться параллельно. По сути, при этом мы находим состояние гонки. Если же доступ к памяти не может происходить параллельно, но возможны ситуации, при которых доступ производится строго после, то такие ошибки мы не можем находить в текущей конфигурации, однако возможно расширение инструмента для поиска таких ошибок, то есть, концептуально это не является ограничением. В данном случае ошибка не находится из-за неточной модели памяти, при которой мы не можем сопоставить освобождение целой структуры и доступ к одному из ее полей. Данная проблема также не является принципиальной и может быть решена с использованием более точной модели. Тем не менее, в данный момент ошибка не может быть найдена.

Коммит: 1a81087

В данном коммите исправляется потенциальное состояние гонки между *tcp_illinois_info* и *rtt_reset*, которое приводит к делению на ноль. CPALockator выдает предупреждение о том, что функция *rtt_reset* может записывать данные без блокировок, однако парной функцией оказывается та же *rtt_reset*, так как обработчики из *tcp_congestion_ops* могут вызываться параллельно друг с другом. Строго говоря, данная ситуация также удовлетворяет определению состояния гонки, однако не может привести ни к чему плохому, так как записываются одинаковые данные, то есть порядок записи не может повлиять на результат выполнения. Таким образом, такое ложное срабатывание является следствием определения состояния гонки, которое используется при поиске.

Для анализа данного модуля потребовалось дополнение модели окружения регистрацией обработчиков структуры *tcp_congestion_ops*.

Коммит: 1adc906

Данный коммит добавляет спин блокировку при доступе к некоторому списку *lg->lg_prealloc_list* в функции *ext4_mb_add_n_trim*. Анализ разделяемых данных доказывает, что данные *lg* являются локальными, так как *lg = ac->ac_lg*, а *ac* выделяется с помощью *kmem_cache_alloc* в предыдущей функции *ext4_mb_new_blocks*. При ручном анализе кода понять, как эти данные могут стать разделяемыми не удалось. Таким образом, нет полной уверенности в том, что данный коммит исправляет реальную ошибку.

Коммит: 6b1246d

Данный коммит исправляет высокоуровневую гонку. При вызове функции *iscsit_get_np* сначала производится поиск подходящего сетевого портала np под спин блокировкой. Если он не был найден, то под него выделяется память и инициализируется. Затем, под той же спин-блокировкой новый портал вставляется в разделяемый список. Ошибка проявлялась в том случае, если одновременно приходило два запроса одинакового портала. Тогда в двух потоках создавался одинаковый портал, который затем добавлялся в глобальный список в двух экземплярах. Решением стало использование неразрывной блокировки от проверки существования портала до вставки его в список.

Так как доступ к разделяемым данным (в данном случае разделяемому списку порталов) производился под блокировками, CPALockator не может определить состояние гонки. Поиск высокоуровневых гонок является нетривиальной задачей в первую очередь потому, что невозможно в общем случае определить высокоуровневую гонку.

Коммит: 3627c07

Данный коммит исправляет высокоуровневую гонку, которая происходит между *acm_suspend* и *acm_start_wb*. При начале передачи данных в функции *acm_start_wb* выставляется флаг *transmitting*, что означает, что устройство не может быть остановлено до завершения передачи. В то же время передача данных не начнется, если будет выставлен флаг *susp_count*, который означает, что устрой-

ство готовится к остановке. В функции *acm_suspend* под блокировкой проверяется статус флага *transmitting*, но затем блокировка снимается и захватывается перед установкой флага *susp_count*. Таким образом, была возможна ситуация, при которой флаг *transmitting* будет установлен после проверки, и решением стало использование неразрывной блокировки между проверкой и установкой флага также, как это было сделано в *acm_suspend*.

Как и в случае с предыдущим коммитом данный тип ошибок невозможно найти с помощью CPALockator в общем случае.

Коммит: 3b91850

Состояние гонки, исправляемое данным коммитом, заключается в том, что без должной синхронизации один поток мог завершиться раньше другого, но при этом разделяемые данные, которыми оперируют эти потоки, представляют собой память на стеке одного из них. В этом случае, после завершения основного потока, эта память становится невалидной. В данном коммите встречаются сразу несколько сложностей для анализа. Первая сложность заключается в том, что явных доступов в данном случае нет, так как первый поток не изменяет свою память на стеке без блокировок. Более того, нет даже явного вызова функции освобождения этой памяти, типа *kfree*. Таким образом, это даже не классическая ошибка типа use-after-free. Вторая сложность заключается в том, что анализ разделяемых данных может не определить, что локальная переменная на стеке становится разделяемой, что приведет к пропуску ошибки. Наконец, третья особенность заключается в использовании синхронизации другого типа (*wait_for_completion*). Такая синхронизация может быть поддержана с помощью специального CPA, однако в текущей конфигурации такой CPA не реализован.

Коммит: 0e2400e

В коммите исправляется состояние гонки при большом количестве *open/close* операций, что приводит к параллельному выполнению функции *__send_control_msg*, которая добавляла сообщение в разделяемую виртуальную очередь без блокировок, что могло привести к неконсистентному ее состоянию в случае параллельной модификации. Решением было использование спин-блокировки.

CPALockator находит такое состояние гонки между *__send_control_msg*. Из коммита не совсем ясно, какой именно обработчик привел к состоянию гонки. CPALockator приводит трассу, начинающуюся с *virtcons_restore*. После исправления данная трасса не находится, что подтверждает корректность найденной ошибки. Однако, полностью доказать отсутствие ошибок не удается из-за наличия ложных предупреждений об ошибках.

Для данной верификационной задачи потребовалось добавление модельной регистрации *register_virtio_driver*.

Коммит: 2d4cf3d

В коммите исправляется ошибка, связанная с некорректным доступом к очереди, которая определена на стеке. При некорректной синхронизации один поток (*usbnet_terminate_urbs*) может завершиться раньше, чем произойдет доступ к очереди из другого потока (*usbnet_bh*). Данный модуль экспортирует объявленные обработчики, включая *usbnet_probe*, для вызова из других модулей. Таким образом, требуется более сложная схема подготовки верификационной задачи, которая позволит получать такие задачи на основе нескольких модулей.

Коммит: 0f90c9c

В коммите добавляется дополнительная проверка под блокировкой для того, чтобы избежать деления на ноль. Нулевое значение было возможно при модификации значения другим потоков. Таким образом, данная ошибка не являлась состоянием гонки в чистом виде, так как одновременного доступа к разделяемой памяти не было. Кроме того, из коммита не до конца понятно, какой именно поток мог модифицировать значение переменной. Скорее всего, это *ttm_dma_pool_release*, который выставляется, как обработчик некоторого события. Поэтму необходима более сложная модель окружения.

Коммит: aea9dd5

В коммите добавляется проверка значения некоторой переменной, которая могла быть изменена другим потоком. При этом состояние гонки не исправляется, а лишь обнаруживается, то есть, если значение было изменено, значит, произошла гонка, и возвращается код ошибки. Анализ этого модуля приводит к исчерпанию лимита по времени, так как модуль *fs/btrfs/btrfs.ko* достаточно объемный и содер-

жит около 300 000 строк кода, а также большое количество функциональных указателей, которые всегда вызывают трудности у инструментов статического анализа. При этом даже построение первой, самой неточной абстракции не успевает завершиться.

Коммит: 2e4ce49

Патч исправляет ошибку, связанную с тем, что выполнение scsi комманды (функция *complete_scsi_command*) завершается (*scsi_done*) до освобождения ресурсов (*cmd_free*). Таким образом, может оказаться, что выполнение другой команды начнется до того, как будут освобождены ресурсы. По сути операция *scsi_done* используется, как неявная синхронизация. При этом операции выделения памяти в *cmd_alloc* и ее освобождения в *cmd_free*, которые образуют гонку, сами по себе выполняются под блокировками, которые захватываются внутри соответствующих функций. То есть, данная ошибка является высокоуровневой гонкой. Кроме этого, из-за большого количества уточнений анализ данного модуля не может завершиться за отведенный лимит времени.

Коммит: f0c626f

В коммите исправляется состояние гонки, которое возможно между *iscsit_del_np* и *__iscsi_target_login_thread*. Последняя выставляет некоторый указатель на поток в ноль, а первая пытается остановить поток по этому указателю, что приводит к разыменованию нулевого указателя. Исправление переносит обнуление указателя из одной функции в другую. CPALockator выдает предупреждение для этого указателя для доступов в функциях *__iscsi_target_login_thread* и *iscsit_reset_np_thread*. При этом во второй функции перед чтением указателя присутствует проверка его значения под блокировкой, то есть данный код может быть выполнен параллельно. Таким образом, можно заключить, что найдено другое потенциальное предупреждение о состоянии гонки.

Для данного модуля потребовалось добавление регистрации структуры *target_fabric_np_base_attribute* в модель окружения.

Коммит: 34596a8

В коммите исправляется высокоуровневая гонка между функциями операциями *open* и *delete*. Удаление девайса из списка *idr* списка и его добавление про-

исходит под двумя блокировками: rtnl и мьютекс, но его поиск в этой очереди в функции *idr_find* производится только при захваченном мьютексе. Таким образом, возможна такая ситуация: при добавлении очереди к девайсу он находится в списке *idr*, затем блокировка снимается, выполняются различные действия, которые прерываются операцией *idr_remove*. Она удаляет соответствующий девайс из списка, а потом переключается обратно, и в итоге оказывается, что очередь была добавлена к удаленному девайсу.

Как и другие высокоуровневые гонки, данная ошибка не может быть найдена напрямую с помощью CPALockator.

Коммит: 7357404

В коммите исправляется два состояние гонки, связанное с параллельным доступом к счетчику *alloc_blocks*, один из ошибочных доступов в функции *hfsplus_file_extend*, а другой – в функции *hfsplus_file_truncate*. Отметим, что это именно два ошибочных доступа, которые могут образовывать состояние гонки с другими доступами. В каждой из этих функций уже находится соответствующий мьютекс, и исправление заключается в расширении области его действия. Так как CPALockator выдает одно предупреждение на каждую переменную, он находит только одно из двух мест, в данном случае он выдает предупреждение о доступах в функциях *hfsplus_file_truncate* и *hfsplus_inode_read_fork*, которые образуют потенциальное состояние гонки. Таким образом, найдено одно из двух исправлений. После исправления находится ложное сообщение об ошибке в другом месте из-за неточности анализа.

Для данного модуля потребовалось добавление функций регистрации структур *hfsplus_btree_ops* и *hfsplus_ops*, *hfsplus_file_operations* в модель окружения.

Коммит: 10ef175

Коммит исправляет ошибку связанную с состоянием гонки следующим образом Анализ данного модуля приводит к исчерпанию лимита по времени, так как модуль *sound/soc/snd-soc-core.ko* достаточно объемный и содержит около 50 000 строк кода, около 26 потоков, а также большое количество функциональных указателей, которые всегда вызывают трудности у инструментов статического анализа.

Коммит: 4036523

Коммит исправлет состояние гонки, связанное с параллельной модификацией списка, добавлением спин блокировки. Анализ данного модуля приводит к исчерпанию лимита по времени, так как модуль *drivers/gpu/drm/i915/i915.ko* достаточно объемный и содержит около 260 000 строк кода, около 24 потока, а также большое количество функциональных указателей, которые всегда вызывают трудности у инструментов статического анализа.

Коммит: 8a2629a

Коммит исправлет высокоуровневое состояние гонки. При установлении нового соединения функция *isert_connect_request* создавала структуру данных, соответствующую новому соединению, инициализировала ее, потом под мьютексом добавляла в разделяемый список и в конце будила спящий поток *isert_accept_np*, который начинал работать с новым соединением. Однако, в случае нескольких созданных запросов после создания одного соединения проснутся сразу несколько потоков *isert_accept_np*. Каждый из них проверит список соединений под блокировками, что означает, что только один из них получит соединение, а остальные снова отправяться ждать. После пяти итераций эти потоки начнут завершаться с ошибкой, что приведет к потерям запросов.