Метод легковесного статического анализа для поиска состояний гонок[1]

[1]*П.С. Андрианов <andrianov@ispras.ru>*
[1]*В.С. Мутилин <mutilin@ispras.ru>*
[1,2,3,4]*А.В. Хорошилов <khoroshilov@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1.
[3]*Московский физико-технический институт (государственный университет),*
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] *Национальный исследовательский университет «Высшая школа экономики»*
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В этой статье представлен подход легковесного статического анализа для поиска состояний гонок, названный CPALockator. Он учитывает такую специфику ядер операционных систем, как сложный параллелизм и особые примитивы синхронизации. Метод основан на алгоритме Lockset, но использует две эвристики, которые призваны уменьшить количество ложных предупреждений: модель памяти и модель параллелизма. В качестве примитивов синхронизации рассматриваются блокировки. Основным предметом нашего исследования являются ядра операционных систем, но предложенный подход может быть применен также и для других программ. Метод основан на идее адаптивного статического анализа (Configurable Program Analysis, CPA) и реализован в инструменте CPAchecker. Реализация метода состоит из двух стадий: сначала определяется множество разделяемых переменных для каждой точки программы, затем производится анализ примитивов синхронизации. На второй стадии собирается информация о всех возможных доступах к разделяемым переменным и захваченных примитивах синхронизации. После этого создается отчет, содержащий предупреждения для тех переменных, для которых было найдено хотя бы два доступа, образующие потенциальное состояние гонки. Для каждого доступа приводится один из возможных путей выполнения программы, который ведет к нему. Инструмент был апробирован на ядре операционной системы реального времени объемом приблизительно 200 000 строк кода. Он позволил найти 20 новых состояний гонки, признанных разработчиками. Кроме того, был произведен пилотный запуск инструмента на драйверах операционной системы Linux с помощью инфраструктуры инструмента LDV, который использовался для подготовки заданий для инструмента CPALockator. Дальнейшим направлением исследований является разработка более точной модели потоков, интеграция с более точными тяжеловесными техниками анализа.

Ключевые слова: статический анализ, состояние гонки, ядро операционной системы, разделяемые данные.

DOI: 10.15514/ISPRAS-2015-27(5)-6

Для цитирования: Андрианов П.С., Мутилин В.С., Хорошилов А.В. Метод легковесного статического анализа для поиска состояний гонок. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 87-116. DOI: 10.15514/ISPRAS-2015-27(5)-6.

[1] Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №13-01-00694

1. Введение

Несмотря на большой прогресс в области верификации программного обеспечения, ошибки, связанные с параллельным выполнением кода остаются одними из наиболее труднообнаруживаемых. Более того, такие ошибки довольно многочисленны, например, в среднем они составляют около 20% от всех ошибок в файловых системах ядра операционной системы Linux [1]. Наиболее частыми причинами ошибок, связанных с параллельным выполнением ядра операционной системы, являются состояния гонки, при которых происходит одновременный доступ к разделяемым данным из нескольких потоков. В частности анализ исправлений за год разработки ядра Linux показал, что ошибки, связанные с состояниями гонки, образуют наиболее многочисленный класс и составляют 17% от всех типичных ошибок [2].

Существуют два пути для поиска состояний гонки автоматически: динамический анализ и статический анализ. Техники динамического анализа позволяют получить относительно небольшой процент ложных срабатываний. Примерами инструментов, реализующих методы динамического анализа, являются Eraser [3], RaceHound [4] и DataCollider [5]. Они способны находить потенциальные состояния гонки только на тех путях выполнения программы, которые происходят в течение реального исполнения. Состояние гонки определяется двумя практически одновременными доступами к одним и тем же данным, что усложняет ее обнаружение. Инструменты, которые используют методы на основе векторных часов, могут работать с двумя разнесенными во времени доступами к памяти, но они чувствительны к порядку операций. Кроме того, известно, что значительную часть путей исполнения программы достаточно сложно воспроизвести в тестовом окружении.

Методы статического анализа имеют свои проблемы. Тяжеловесные техники достаточно точны, но требуют большого количества времени для анализа. В случае задачи поиска гонок общее число мест, где может возникнуть состояние гонки, слишком велико. Проводились некоторые эксперименты по

верификации модулей ядра, и результаты показали, что тяжеловесный подход не масштабируется[6]. Происходит комбинаторный взрыв состояний, поэтому даже для небольших модулей количество затраченного времени и памяти было гигантским.

Методы легковесного статического анализа, например, метод, реализованный в инструмента Locksmith [7], работают очень быстро, но число ложных срабатываний обычно бывает очень велико. Для инструмента Locksmith средний процент ложных сообщений об ошибках составляет 73% на некоторых POSIX приложениях и около 96% на нескольких драйверах [8]. Существующие методы не принимают во внимание некоторую специфику ядра операционной системы, описанную ниже, поэтому большинство драйверов и особенно ядро само по себе анализировать существующими инструментами, предназначенными для пользовательских приложений, очень сложно.

Параллелизм в ядрах операционных систем устроен сложным образом, так как они являются асинхронными. Многие функции ядра могут быть выполнены параллельно друг с другом, и определить, когда начинается параллельное исполнение, очень сложно. Кроме того в ядрах операционных систем используются дополнительные примитивы синхронизации, такие как отключение прерываний или планирования. Еще одна важная особенность — это активное использование адресной арифметики. Как результат, поиск состояний гонок в ядрах операционных систем является более сложной задачей, чем в пользовательских приложениях.

В этой статье мы предлагаем новый метод легковесного статического анализа для обнаружения состояний гонок, названный CPALockator. Он может легко масштабироваться на большие объемы исходного кода, оставляя процент ложных срабатываний на приемлемом уровне и принимая во внимание специфику ядра операционной системы.

Оставшаяся часть статьи организована следующим образом. В разделе 2 даются необходимые определения. В разделе 3 описывается основная идея предложенного метода. После этого рассказывается о подходе адаптивного статического анализа. Реализация метода обсуждается в разделе 5. Следующий раздел посвящен интеграции в систему LDV [12]. В разделе 7 мы рассказываем о полученных результатах, потом в разделе 8 — о близких работах. В заключении кратко описываются планы на будущее.

2. Определения

В этой статье термин **поток** используется, чтобы обозначить независимый поток выполнения инструкций в ядре операционной системы, например, прерывание от аппаратуры или выполнение системных вызовов от имени пользовательского потока. Если некоторый системный вызов может быть

прерван прерыванием от аппаратуры, мы считаем, что этот системный вызов и прерывание могут выполняться параллельно друг с другом.

Блокировка — это объект, использующийся для предотвращения одновременного доступа к памяти. Если некоторая блокировка захвачена из одного потока, то другой поток, пытающийся захватить ту же самую блокировку, не может продолжить свое выполнение до тех пор, пока блокировка не будет освобождена. Типичными примерами блокировок могут быть мьютексы и спинлоки. Мы считаем такие примитивы синхронизации ядра, как отключение прерываний или планирования, специальными блокировками. Например, функция `irq_disable()` отключает планирование и тем самым запрещает любое параллельное исполнение, поэтому мы считаем, что воображаемая глобальная блокировка `irq_disable` захвачена. Некоторые блокировки могут быть захвачены несколько раз, в этом случае имеет место рекурсивный захват блокировки.

Разделяемые данные — это область памяти, которая доступна из нескольких потоков. В языке Си разделяемые данные представлены глобальными переменными и указателями на память, доступную из нескольких потоков через корректные конструкции языка Си. Важно отметить, что разделяемость данных является характеристикой, зависящей от времени. Локальные данные могут стать разделяемыми в некоторой точке программы и вернуть статус локальности позже.

Использование данных — чтение или запись данных.

Состояние гонки — это ситуация, при которой происходят два неупорядоченных использования одних и тех же разделяемых данных и по крайней мере одно из них является записью. Состояние гонки не всегда приводит к ошибке (так называемое доброкачественное состояние гонки), но является симптомом ее.

3. Легковесный метод для поиска состояний гонки

Метод CPALockator основан на алгоритме Lockset [3], который строит множество *C(v)* потенциальных блокировок для каждой разделяемой ячейки памяти *v*. Это множество содержит в себе те блокировки, которые защищают v для дальнейших действий. Блокировка *l* находится в *C(v)* в текущий момент времени, если для каждого потока доступ к *v* всегда происходил при захваченной блокировке *l*. *C(v)* инициализируется всеми возможными блокировками. Когда происходит доступ к данным, *C(v)* обновляется, как пересечение *C(v)* и того множества блокировок, которые захвачены на данный момент в текущем потоке. Если *C(v)* становится пустым, имеет место потенциальное состояние гонки.

Чтобы задать алгоритм поиска состояний гонок мы должны ответить на следующие вопросы:

- Когда начинается параллельное исполнение?

- Что такое данные?
- Какие данные считаются одинаковыми?
- Что такое блокировки и какие существуют правила для операций с ними?
- Какие блокировки считаются одинаковыми?

Инструмент динамической верификации Eraser, который первым реализовал Lockset, использует точки создания потоков, чтобы определить, когда начинается параллельное выполнение. Для ядра операционной системы определить, когда начинается параллельное выполнение, достаточно сложно. Мы считаем, что каждый системный вызов или обработчик прерывания может выполняться параллельно с любым другим, включая себя. В реальности взаимосвязь между ними более сложная. Модель потоков в методе CPALockator представлена функцией main, которая содержит вызовы всех системных вызовов и обработчиков прерываний.

Eraser оперирует ячейками памяти при реальном выполнении. Метод CPALockator считает все переменные и поля структур единицей данных по умолчанию. Существуют ситуации, в которых доступ к различным полям структур должен быть защищен блокировкой. Предположим, что у нас есть тип структуры, представляющий разделяемый связный список с полями next и prev. Пусть у нас есть два доступа: к полю next одной переменной этого типа и к полю prev другой переменной этого же типа. Все статические методы, оперирующие с ячейками памяти столкнутся с проблемой в этом случае, так как всегда очень сложно понять, что два различных указателя могут указывать на одну и ту же область памяти. В нашем методе мы имеем возможность считать этот случай двумя доступами к одним данным и выдать предупреждение об ошибке. Этот способ требует ручной аннотации, тем не менее достаточно прост в использовании. Подробнее он будет описан в Разделе 5.3.

Так как Eraser оперирует ячейками памяти при реальном исполнении программы, то в нем данные считаются разделяемыми, если два доступа происходят по одному адресу. Статический инструменты, такие как Locksmith, строят граф потоков данных, чтобы определить, какие указатели указывают на одну и ту же память. Однако для ядра операционной системы сложно построить граф потоков данных из-за активного использования адресной арифметики и массивного параллелизма. Поэтому такой метод работает не так хорошо, как для пользовательских программ. В методе CPALockator равенство ячеек памяти следует только из синтаксических правил. Глобальный указатель всегда считается указывающим на одну и ту же область памяти. Аналогичное предположение действует и для локального указателя в заданной функции. Считается, что два поля структуры указывают на одну область памяти, если совпадает тип структуры и имена полей. Важно отметить, что имена самих переменных в этом случае не учитываются. Так, если указатели на структуры A и B имеют одинаковый тип, доступы A->x и B->x будут рассматриваться, как доступы к одной и той же памяти. Если структуры A и B не связаны друг с другом, это предположение приведет к ложному сообщению об ошибке. На практике из-за этой эвристики происходит 18% всех ложных предупреждений.

Eraser рассматривает блокировки, как объект, который может быть захвачен. Он поддерживает только две операции с ним: захват и освобождение. Метод CPALockator позволяет описать блокировку: задать функции захвата и освобождения (возможны несколько вариантов), их аргументы, рекурсивность. Равенство блокировок следует из равенства имен объектов (переменных) в обоих методах.

4. Адаптивный статический анализ

Для реализации метода CPALockator был выбран инструмент адаптивного статического анализа CPAchecker (Configurable Program Analysis, CPA) [10]. Он позволяет комбинировать различные техники анализа, встраивать дополнительные подходы, такие как CEGAR, BMC. В этом заключается одно из важных отличий описываемого метода от существующих техник легковесного анализа. Рассмотрим кратко теорию адаптивного статического анализа.

Адаптивный статический анализ может быть составлен из нескольких алгоритмов, предлагающих различные типы анализа. Кроме того, возможна дополнительная настройка алгоритмов путем выбора оператора слияния и способа проверки необходимости завершения анализа.

Адаптивный статический анализ (`D`, `transfer`, `merge`, `stop`) состоит из абстрактного домена `D`, отношения переходов `transfer`, оператора слияния `merge` и оператора останова `stop`. Эти четыре компонента задают алгоритм анализа и влияют на его точность и потребление ресурсов.

Абстрактный домен `D` определяет множество абстрактных состояний. Каждому абстрактному состоянию соответствует его абстрактное значение, то есть множество конкретных состояний, которое оно представляет. Конкретное состояние программы — это отображение переменных программы во множество значений этих переменных.

Отношение переходов `transfer` определяет для каждого состояния **e** потенциальные следующие абстрактные состояния {**e'**}, для которых каждый переход помечен соответствующей дугой графа потока управления (ГПУ).

Оператор `merge` позволяет объединить информацию от нескольких путей анализа. Он определяет, когда два узла дерева достижимости сливаются в один, а когда они должны быть рассмотрены по-отдельности. В классических легковесных подходах объединение всегда происходит, в случае если абстрактные состояния относятся к одной вершине ГПУ. В классических тяжеловесных техниках абстрактные состояния никогда не объединяются.

Оператор `stop` проверяет, является ли текущее состояние покрытым данным множеством уже проанализированных состояний. Он определяет, когда анализ пути завершается в текущей вершине. В классических легковесных подходах останов происходит когда полученное абстрактное состояние не содержит новых конкретных, то есть достигнута неподвижная точка. В тяжеловесных техниках останов происходит, в случае если множество конкретных состояний полученного абстрактного состояния является подмножеством множества конкретных состояний, соответствующих некоторому уже проанализированному абстрактному состоянию.

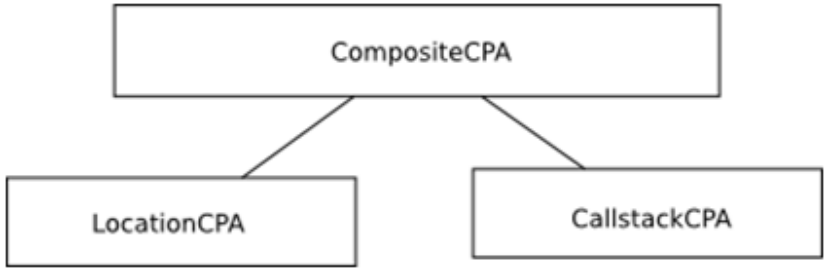

Рис. 1. Пример дерева конфигураций CPA

Рассмотрим пример дерева конфигурации CPA. В нем изображены три анализа. Основным является *CompositeCPA*. Он включает в себя *LocationCPA* и *CallstackCPA*.

Состнояние *LocationCPA* содержит в себе вершину ГПУ, то есть номер строки с исходным кодом. Таким образом, его абстрактный домен является множеством всех возможных вершин ГПУ. Оператор `transfer` меняет номер строки текущего состояния на номер строки вершины, в которую входит соответствующая дуга. Для этого анализа оператор `merge` никогда не объединяет состояния. Останов происходит только если состояние уже было проанализировано ранее.

Состояние *CallstackCPA* состоит из стека вызовов функций. В случае если вызывается новая функция, ее имя помещается на вершину стека. Когда производится возврат, имя функции удаляется из стека. Это описание работы оператора перехода. Операторы stop и merge такие же, как и для предыдущего анализа.

Задача *CompositeCPA* — объединение анализов, описанных выше. Ее абстрактный домен является декартовым произведением доменов *LocationCPA* и *CallstackCPA*. Оператор перехода *CompositeCPA* вызывает операторы переходов вложенных анализов. Сначала он получает новое состояние от *LocationCPA*, затем — от *CallstackCPA* и объединяет их вместе. Это объединение состояние является новым состоянием *CompositeCPA*.

`Merge` и `stop` операторы также объединяют соответствующие операторы вложенных анализов. Для того, чтобы объединить два состояния *CompositeCPA*, нужно сначала объединить состояния *LocationCPA*, которые включены в данные состояния *CompositeCPA*, а потом — состояния *CallstackCPA*. Оператор stop работает похожим образом: если все вложенные CPA решают остановить анализ, *CompositeCPA* также останавливает его.

Рассмотрим, как такая композиция CPA анализирует простую программу (Рис. 2). На рисунке 3 изображен абстрактный граф достижимости для этой программы.

```
1  int g(int a) {
2    int b = 0;
3    if (a == 0) {
4      b++;
5    }
6    return b;
7  }
8  int f() {
9    return 0;
10 }
11 int main() {
12   int t;
13   t = f();
14   g(t);
15 }
```

Рисунок 2 — Пример программы

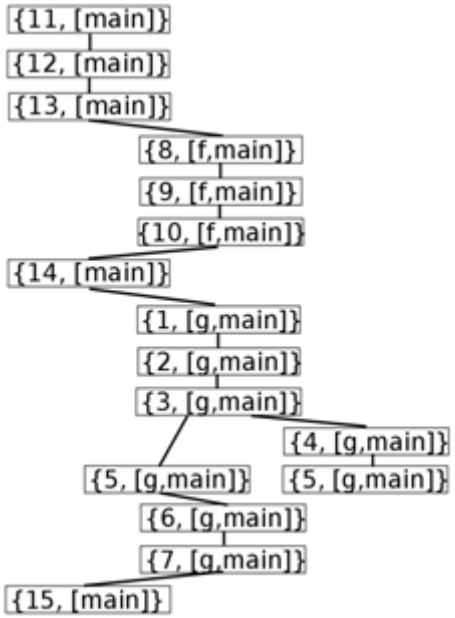

Рисунок 3 — Пример анализа. Первое число в скобках представляет собой состояние LocationCPA (номер строки) и затем идет стек вызовов функций

Инструмент начинает из функции `main`, затем он анализирует функцию `f`, после этого — переходит в функцию `g`. В этой фунции он встречает условие на строке 3. Он анализирует две ветви и получает одинаковые состояния на строке 5. Это означает, что одно состояние покрывается другим, поэтому нет необходимости анализировать оба, а анализ продолжается только для одного из них.

Классические анализы, реализующие подход CPA решают задачу достижимости, то есть, доказывают достижимость некоторой метки в программе (вызова функции и т.п.). Состояние гонки характеризуется двумя точками и для того, чтобы пойти по пути классических анализов, нужно было бы определить состояние нашего анализа, как декартово произведение состояний в двух потоках, то есть получить семантику чередования (англ. interleaving). Мы же используем модульную абстракцию для описания взаимодействия нескольких потоков. Этот подход предполагает использование упрощенной модели взаимодействия потоков. При необходимости эта модель может быть уточнена, но так как в текущем варианте анализа не учитываются условия, то взаимодействие потоков не может быть учтено ни при какой его модели.

5. Реализация

Реализация метода CPALockator состоит из двух этапов. Сначала определяются разделяемые данные, затем для каждого использования разделяемых данных сохраняется множество захваченных блокировок. На рисунке 4 представлены эти этапы.

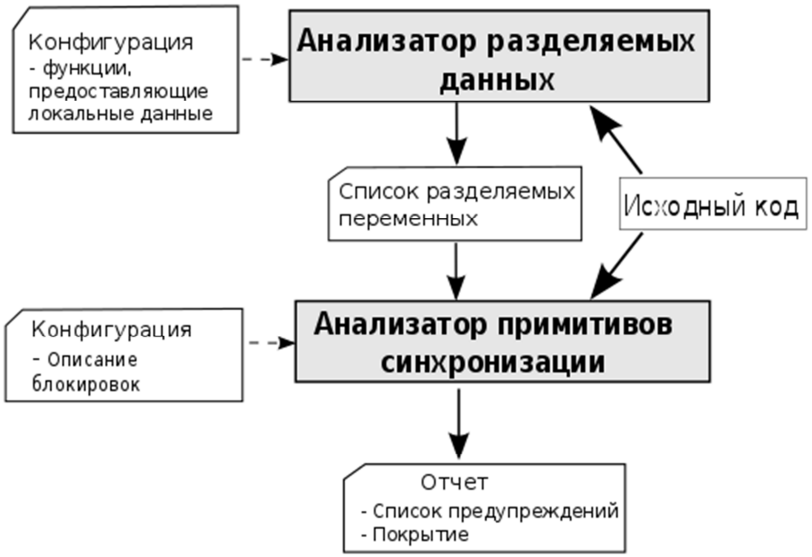

Рисунок 4 — Этапы CPALockator

Конфигурация CPA для Анализатора разделяемых данных состоит из функций, предоставляющих локальные данные, например, `calloc`, `malloc` и др. Мы предполагаем, что указатели, возвращаемые этими функциями,

указывают на локальные данные, которые не могут быть разделяемыми в соответствующей точке программы. Конфигурация для Анализатора блокировок включает в себя описание блокировок и аннотации, которые описываются в секции 5.3.

5.1 Конфигурация CPA для Анализатора разделяемых данных

Анализатор разделяемых данных используется для формирования списка разделяемых переменных для каждой точки программы (см. Рис 5).

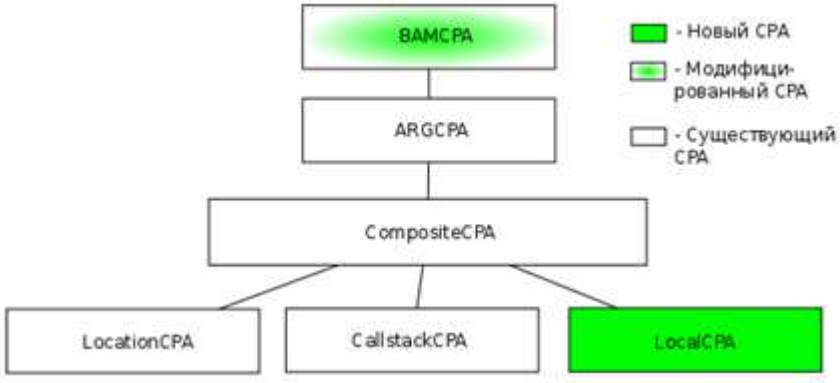

Рисунок 5 — Конфигурация Анализатора разделяемых данных

BAMCPA (Block Abstraction Memorization) [11] отвечает за модульность анализа. Если функция была уже проанализирована с некоторым состоянием до текущего вызова и было сохранено множество конечных состояний, то она не анализируется еще раз, а используются сохраненные состояния. Мы добавили в оригинальный *BAMCPA* возможность обработки рекурсии и некоторые способы для взаимодействия *BAMCPA* и наших новых CPA.

ARGCPA (Abstract Reachability Graph) отвечает за возможность восстановления пути из любого состояния до начального. Он хранит для каждого состояния множество предыдущих и последующих, и предоставляет эти данные для обхода и восстановления пути.

CompositeCPA, *LocationCPA* и *CallstackCPA* были уже описаны.

LocalCPA отвечает за определение локальности всех переменных, доступных в данной точке программы. Отдельну следует отметить, что под переменными мы также понимаем указатели, то есть, анализ также учитывает информацию о том, куда они указывают. Оператор перехода `transfer` распространяет информацию о разделяемости данных через операторы присваивания и вызовы функций. Например, если указатель `b` указывает на разделяемую область памяти и существует присваивание `a = b`, тогда разделяемость памяти

*b переностится на область памяти *a. После присваивания считается, что a также указывает на разделяемую память. На точках объединения оператор merge объединяет результаты. В случае неопределенности всегда выбирается наихудший результат, то есть, статус shared. Рассмотрим следующий пример:

```
if (condition) {
    a = b;
} else {
    a = c;
}
```

Если b указывает на локальные данные, а c — на разделяемые, то после анализа этого блока кода считается, что a указывает на разделяемые данные.

Результатом этого этапа анализа является список разделяемых переменных для каждой точки программы.

5.2 Конфигурация CPA для Анализатора примитивов синхронизации

Анализатор блокировок используется для определения множества захваченных блокировок для каждого использования разделяемых данных, которые были представлены на предыдущем этапе анализа.

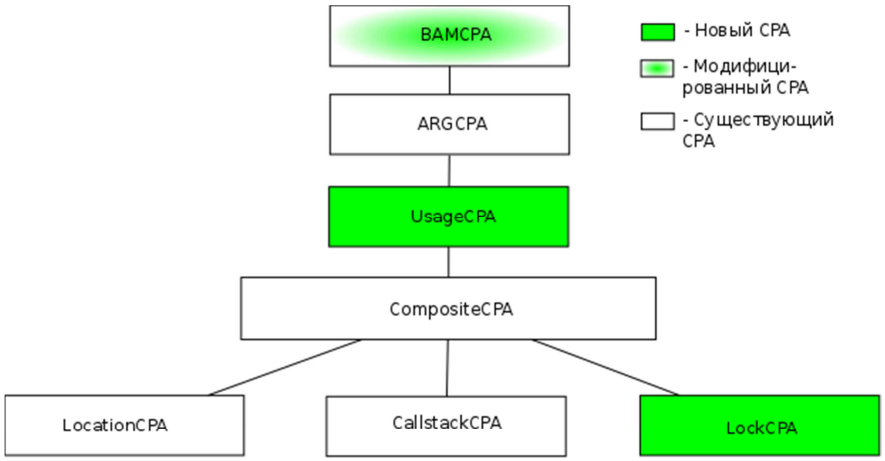

Рисунок 6 — Конфигурация Анализатора примитивов синхронизации

BAMCPA, *ARGCPA*, *LocationCPA* и *CallstackCPA* являются теми же самыми.

UsageCPA собирает статистику использования данных. Оператор transfer определяет переменные, использованные в выражении в некотором доступе к

данным, сохраняет стек вызовов и множество захваченных блокировок для каждого использования.

В конце анализа мы получаем информацию обо всех использованиях для каждой разделяемой переменной. Каждое использование состоит из:

- множества захваченных блокировок;
- стека вызовов функций;
- номера строки;
- типа дуги ГПУ (вызов функции, присваивание и т.д.);
- тип доступа (чтение или запись).

LockCPA следит за множеством захваченных блокировок. Его состояние содержит множество блокировок, которые были захвачены на текущем пути анализа. Для каждой блокировки содержится информация о:

- имя блокировки;
- счетчик рекурсивного захвата;
- стек вызовов функций.

Оператор перехода меняет состояние, которое содержит множество захваченных блокировок. Когда вызывается функция захвата блокировки, соответствующая блокировка добавляется во множество или увеличивается счетчик, если она уже присутствует. При вызове функции освобождения функции счетчик уменьшается, а если он становится равным нулю, блокировка удаляется из множества.

Состояния всех CPA в этой конфигурации никогда не объединяются. Анализ останавливается, если это состояние уже было проанализировано.

5.3 Аннотации

Аннотации используются для более точной настройки метода на специфический код. Всего есть три класса аннотаций:

- Аннотации влияния функции на примитивы синхронизации.
- Аннотации влияния функции на разделяемость данных.
- Аннотации целевых данных.

Рассмотрим следующий пример для объяснения первого типа аннотации.

```
int f() {
  if (isGlobalPointer) {
    lock();
  }
  (*pointer)++;
  if (isGlobalPointer) {
    unlock();
```

```
    }
}
```

В этом примере увеличение разделяемого счетчика всегда происходит под блокировкой. В случае же локального указателя захват блокировки не нужен. Наш анализ рассматривает четыре пути, так как для каждого из двух условных операторов анализируются ветви then и else. Два из этих путей оканчиваются с захваченной блокировкой, а оставшиеся два оканчиваются в состоянии с пустым множеством блокировок. Пара состояний с захваченной блокировкой является недостижимой, так как условия в условных операторах одинаковы.

Такие ситуации не так часто происходят, но каждая из них порождает значительное число ложных сообщений об ошибке, так как финальное состояние в функции с захваченной блокировкой сильно влияет на дальнейший пути анализа. Аннотации функций используются для того, чтобы разобраться с такими случаями. Это лишь способ подсказать анализу, что функция всегда освобождает или захватывает блокировку.

В приведенным выше примере достаточно добавить аннотацию, что функция f всегда освобождает блокировку.

Аннотации описывают функции в терминах состояний *LockCPA*. После того, как функция была проанализирована, состояние уточняется в соответствии с аннотацией.

В данный момент используется 4 типа аннотаций используются:

- Захват блокировки — функция всегда захватывает блокировку.
- Освобождение блокировки — функция всегда освобождает блокировку.
- Сброс блокировки — если блокировка может быть захвачена несколько раз рекурсивно, функция полностью освобождает ее.
- Восстановление блокировки — функция может модифицировать множество блокировок, но все изменения будут забыты при выходе из функции.

К этому типу аннотаций можно добавить конфигурацию блокировок. Поддерживается возможность определить функции захвата, освобождения и сброса, а также глубину рекурсивного захвата. Эти аннотации обрабатываются в *LockCPA*.

Следующий тип аннотаций описывает влияние на разделяемые данные. Функция может вернуть разделяемые данные или инициализировать указатель, передаваемый, как аргумент, локальными данными. Также данные могут стать разделяемыми после вызова функции. Все эти случаи могут быть специфицированы аннотациями, чтобы повысить точность анализа. В данный момент поддерживаются только описание функций, возвращающих локальные данные. Эти аннотации обрабатываются в *LocalCPA*.

Третий тип аннотаций используется для установки равенства между переменными так, чтобы они рассматривались, как одинаковые данные. Это требуется, например, для списков, в которых элементы обычно имеют одинаковые имена *next*. Если не удается различить элементы различных списков, будет выдано большое количество ложных предупреждений, так как доступ к различным спискам может защищаться различными множествами блокировок. Поэтому мы связываем переменные, представляющие элементы списка, с самим списком. Для этой цели конфигурация содержит функции, которые используются для работы со списком. Например, выражение `e = getElement(list)` связывает переменную *e* с переменной *list*, передаваемой как параметр. Эти аннотации обрабатываются в *UsageCPA*.

Для операционной системы реального времени, на которой метод был апробирован, было аннотировано около 2% функций, что в абсолютных величинах составило 90 функций.

6. Интеграция инструмента

Метод CPALockator был интегрирован в систему LDV (Linux Driver Verification), разработанную в рамках проекта по верификации драйверов операционной системы Linux (см. Рис. 7) [12].

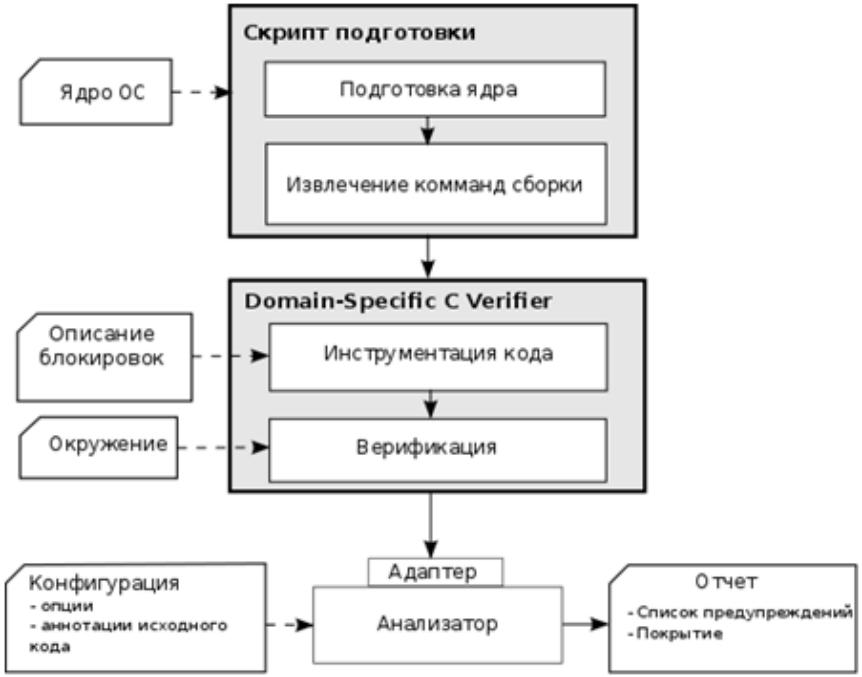

Рисунок 7 — Интеграция с архитектурой LDV

Сначала подготавливается ядро операционной системы. В течение этой стадии вызовы компилятора заменяются на вызовы нашего инструмента для извлечения команд сборки. Затем поток команд сборки извлекается с помощью специальных скриптов. Этот поток передается в компонент DSCV (Domain-Specific C Verifier). Он инструментирует исходный код, используя описания блокировок. Например, он заменяет макросы, используемые для захвата и освобождения блокировок на вызовы модельных функций, аннотированных в конфигурации. Это делается из-за того, что макросы могут быть развернуты в очень сложную последовательность команд, тогда как модельные функции анализировать значительно проще.

После этого включается модель окружения. Она представлена функцией main, содержащей системные вызовы и обработчики прерываний. Считается, что все они могут выполняться параллельно.

После всей подготовки исходный код анализируется нашим инструментом поиска гонок. Он генерирует отчет, содержащий список предупреждений с детальной информацией о каждом из них.

Для визуализации трасс ошибок используется другой компонент LDV. Когда инструмент генерирует предупреждение о состоянии гонки, оно должно быть показано пользователю. Более того, пользователь должен иметь возможность проверить, является ли это предупреждение истинной ошибкой или ложным предупреждением. Таким образом, необходимо представлять трассу ошибки и ее связь с исходным кодом. Состояние гонки характеризуется по крайней мере двумя использованиями с непересекающимися наборами блокировок. Визуализированная трасса ошибки содержит стек вызовов функций для двух использований с точками захватов блокировок. Визуализатор преобразует данные, полученные от верификатора и связывает их с исходным кодом. Для представления результатов генерируется HTML отчет. Главная страница отчета содержит общую статистику (Таблица 1).

Табл. 1 — пример отчета для драйвера Linux floppy.ko

Статистика	Общая	Предупреждения
Глобальные:	195	29
Переменные	122	23
Указатели	73	6
Локальные:	3	0
Переменные	0	0

Указатели	3	0
Поля структур:	118	24
Переменные	105	24
Указатели	13	0
Всего:	316	53

На ней отображены общее число переменных для каждой из трех категорий: глобальные, локальные и поля структур. Во втором столбце указано число переменных, для которых были получены предупреждения. Обозначение «Указатели» означает доступ по указателю, а «Переменные» — доступ к самой переменной. После этого идет список всех найденных блокировок. После этого расположен список всех предупреждений, для каждого из которых содержится пара использований с непересекающимся множеством блокировок.

Пример выдаваемого предупреждения об ошибке представлен на рисунках 8-9.

Source code

Race_example.c

```
1  #line 1 "../cil-files/Race_example.c"
2  int global;
3
4  int print() {
5    if (global % 2 == 0) {
6            printf("global is even: %d", global);
7      } else {
8            printf("global is odd: %d", global);
9    }
10 }
11
12 int increase() {
13   lock();
14   global++;
15   unlock();
16 }
17
18 int main() {
19   switch(undef_int()) {
20           case 0:
21                   print();
22                   break;
23
24           case 1:
25                   increase();
26                   break;
27     }
28 }
```

Рисунок 8 — Пример исходного кода. Параллельное выполнение print() и increase() может привести к состоянию гонки

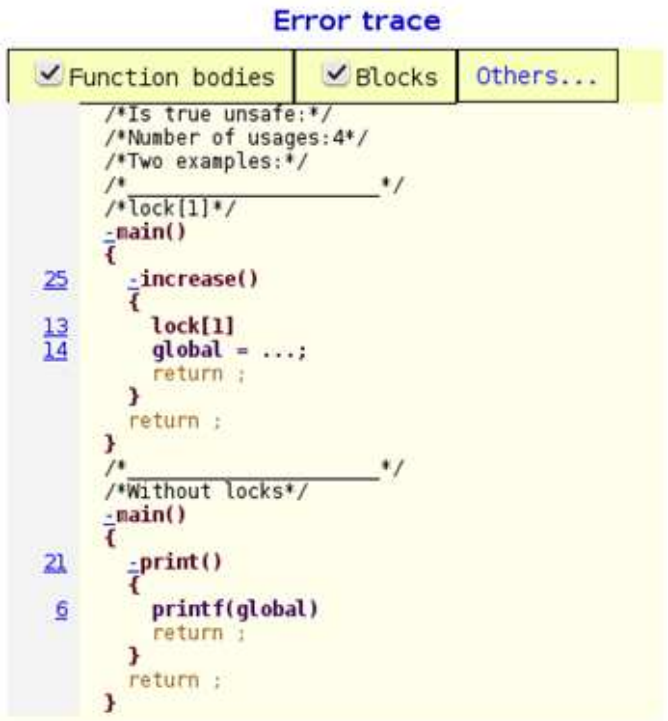

Рисунок 9 — Пример сообщения об ошибке. Трасса ошибки с точками захвата блокировок и точками вызова функций, каждая из которых ссылается на соответствующую строку исходного кода (см. рис. 8)

Еще раз подчеркнем, что мы используем модель параллелизма, и две функции, вызываемые из функции *main*, считаются выполняющимися параллельно.

Функция print печатает информацию о переменной `global`, а `increase` увеличивает ее значение под блокировкой. В таком примере возможно состояние гонки, потому что функция `increase` может записывать переменную одновременно с ее проверкой в функции `print`. Наш инструмент генерирует предупреждение для переменной `global` с трассой ошибки, показанной на рисунке 9.

В первой строке указано общее число найденных использований. Только два из них визуализируются далее. Первое использование означает, что доступ к `global` был в вызове функции. Второе происходит в присваивании при захваченной блокировке `lock`.

Кроме того имеется опция для генерации покрытия исходного кода. Оно показывает код, который был проанализирован верификатором, и его связь с общим кодом ядра.

7. Результаты

Инструмент был применен для анализа операционной системы реального времени, которая уже была протестирована и находилась в использовании уже несколько лет. Общий объем кода был около 200 000 строк кода, но только 50 000 были проанализированы. Основной причиной является то, что не все возможные функции были включены в функцию main. Мы обнаружили 20 новых состояний гонки, которые были подтверждены разработчиками. Общее число предупреждений было 139. Без Анализатора разделяемых данных число предупреждений было 378. В данный момент большая часть ложных сообщений об ошибках происходит из-за неточности в анализе выражений. Например, сейчас анализ не полностью поддерживает условные операторы.

Один запуск инструмента на машине с 6 Гб памяти и восьмиядерным процессором 2.80 ГГц занял 3 минуты на ядре описанной выше операционной системы. Кроме этого, был пробный запуск инструмента на ядре операционной системы Linux 3.8 на директории `drivers`. Общее число проанализированных модулей было около 3500. Инструмент сообщил о 900 предупрежденях. Некоторые из них были проанализированы, и одна реальная ошибка была найдена, однако в текущей версии эта ошибка была уже исправлена.

8. Похожие работы

Мы не будем рассматривать методы динамического анализа, которые имеют свои преимущества. Также мы рассмотрим только методы анализа программ на языке Си.

8.1 Тяжеловесные техники анализа

Традиционно идеи тяжеловесных подходов применяются к небольшим программам. Зачастую в таких подходах предполагается, что взаимодействие между потоками производится только через глобальные переменные. Так как в реальных программных системах число предупреждений, которое приходится на поля структур, сопоставимо с предупреждениями, полученными для глобальных переменных, а иногда даже выше, то для нас было важно учитывать в анализе поля структур.

Еще одной важной особенностью тяжеловесных инструментов, нацеленных на анализ многопоточного программного обеспечения, в том, что они проверяют другое свойство, а именно свойство достижимости ошибочной метки в условиях параллельного окружения. Для того чтобы проверить наличие состояния гонки, необходимо дополнительно изощряться, чтобы записать это

требование, как условие достижимости. Например, после каждого присваивания в разделяемую переменную добавить проверку, что сейчас в данной переменной находится именно то, что мы только что туда записали. Это значительно усложняет анализ. Более того, операции захвата и освобождения блокировки тоже могут быть представлены, как операции с переменными. Например, в инструменте ESBMC [13] моделирование примитивов синхронизации происходит с помощью конструкции assume(P), которая рассматривает пути выполнения программы, для которых в данный момент выполнен предикат P. Операция захвата блокировки m представляет собой атомарную последовательность assume(m==0); m=1. Моделирование захвата блокировки таким образом усложняет анализ, так как теперь приходится учитывать значение еще одной переменной.

Методы тяжеловесного анализа обыкновенно рассматривают все варианты параллельной работы программы, то есть анализируются возможные варианты переключений потоков (англ. interleavings). Одним из подходов является анализ параллельной композиции потоков, при котором так или иначе строится дерево достижимых состояний (ART), которые представляют собой произведение состояний каждого потока. Далее неизбежно возникает проблема комбинаторного взрыва, которую различные инструменты решают по-своему. Например, наблюдение, что состояние гонки проявляется уже при небольшом числе переключений контекста, позволило ограничить число переключений некоторым фиксированным K — подход, получивший название проверка моделей с ограниченным переключением контекста (англ. context-bounded model checking). Примерами инструментов, реализующих методы тяжеловесного анализа, могут быть ESBMC [13] и SATABS [14], которые используют различные подходы к анализу программ (BMC[15] и CEGAR [16, 17] соответственно)

В предыдущих инструментах число потоков было ограничено несколькими экземплярами и известно заранее. Однако большие программные системы, такие как ядро операционной системы, работают со значительно большим количеством потоков. Для анализа таких систем был предложен метод абстракции по счетчику состояний, который реализован, например, в инструменте Boom[18]. Идея метода заключается в том, чтобы рассматривать не состояния каждого потока, а количество потоков в том или ином состоянии. Кроме того, были предложены такие оптимизации как символическое представление, то есть состояние локального потока описывается не конкретным значением, а некоторым символическим. Однако такой подход применим только в случае, если один и тот же код выполняется в несколько потоков. В сложных программных системах большая часть кода, выполняющегося параллельно, является уникальным.

Еще одним вариантом борьбы с большим пространством состояний является использование редукции частичных порядков.

Вторым большим классом тяжеловесных подходов является построение модульной абстракции, реализованное в инструменте Threader [19, 20, 21] или BLAST[22]. Такие подходы не рассматривают все возможные чередования, а строят абстракцию взаимодействия между потоками. Сначала используется неточная модель, которая предполагает, что все потоки могут изменять любые разделяемые данные произвольным образом, затем эта модель уточняется в процессе анализа. Возможно уточнение как множества точек переключения контекста, так и влияния потоков друг на друга. Потенциально такой подход должен дать лучшие результаты нежели те, которые были получены для инструментов, типа ESBMC, однако разработчики приводят результаты только на тестах порядка нескольких сотен строк исходного кода, поэтому остаются вопросы о масштабируемости такого подхода.

Задача верификации многопоточных программ может решаться с помощью трансляции параллельной программы в последовательную с последующей ее верификацией с помощью существующих инструментов или какую-нибудь другую промежуточную форму, например, формулу, которая затем может быть проверена решателем. В основном, методы трансляции опираются на контекстно-ограниченную проверку моделей (англ. Context-Bounded Model Checking [23, 24, 25, 26]) - подход к проверке многопоточных программ, при котором число активных потоков ограничивается некоторым числом в процессе верификации. Для преобразования параллельной программы фиксируется число возможных потоков n и число возможных переключений контекста K. Требуется построить такую последовательную программу, которая покрывала бы все возможные варианты выполнения n потоков, каждый из которых прерывался бы не более K раз.

Некоторые тяжеловесные инструменты предпочитают работать не с программой на языке Си, а на некотором более простом языке. В таком случае сначала применяется трансляция исходной программы в другой язык. Например, для инструмента Storm [27] — это язык Boogie [28], в котором нет таких конструкций, как динамическое выделение памяти, арифметика указателей, преобразование типов. Параллельная программа преобразуется в последовательную, и для нее уже строятся формулы, которые проверяются с помощью SMT-решателя. Для моделирования переключения контекста используется понятие карты памяти, которое означает локальную копию всей памяти, занимаемой глобальными переменными. В данном подходе для каждого потока создается копия карты памяти. Переключение контекста моделируется переключением работы с одной картой памяти на другую. Все потоки выполняются один за другим. 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 верификации циклы разворачиваются на конечную глубину, что может

привести к потере точности, а вызовы функций заменяются на их тела. Кроме того, принимается предположение о том, что в системных программах редко встречается рекурсия, чтобы можно было заменить вызов функции на ее тело.

В статье [24] авторы предлагают три варианта трансляции параллельной программы в логическую формулу.

Первый вариант - это метод Explicit Program Counter (EMC) [29], использование явного счетчика команд. В этом случае состояние программы включает в себя все локальные переменные и счетчик команд для каждого потока. В возможных точках переключения контекста вставляется специальный оператор недетерминированного выбора, который выбирает поток для переключения и корректную строку кода, на которой закончилось его выполнение в прошлый раз. Это самый простой способ, но он требует много времени и памяти.

Второй вариант - Lal-reps(LR) [23]. В этом случае каждый поток символически выполняется независимо от других, с учетом того, что значения глобальных переменных могут измениться случайным образом. Задача упрощается тем, что заранее задается, когда происходит переключения контекста, а значит, задаются и точки, когда глобальным переменным присваиваются новые значения.

Третий вариант - LMP [30]. Его основное отличие от предыдущего подхода в том, что он присваивает глобальным переменным не случайные значения, а только те, которые возможны во время выполнения программы. После переключения контекста текущее локальное состояние потока сбрасывается, и чтобы его восстановить приходится заново исполнять поток из начального состояния с учетом того, что становятся известны значения глобальных переменных до переключения.

Попытки применения подхода с контекстно-ограничиваемыми проверками показали, что он плохо масштабируется и на реальных программах пока не может быть использован [27]. Авторы сравнивают свой способ верификации, основанный на генерации логических формул со способом логической проверки моделей, при котором все переменные программы преобразуются к бинарным, для которых уже генерируются формулы для проверки решателем [31]. Такой подход, например, использован в статье [18]. Эксперименты показали, что эти две парагдимы сильно отличаются, и для них нужны различные способы трансляции. LR - лучший способ для проверки логических формул. Что интересно, самый простой способ EMC оказался лучше, чем LMP почти во всех тестах.

Еще два варианта трансляции рассматриваются в статье [32]. В ней представлен подход ленивой трансляции параллельной программы в последовательную с ограниченным числом переключений контекста. Важной особенностью описываемого метода является то, что множество достижимых состояний полученной программы является тем же, что и для исходной программы. Известные методы сохраняют локальное состояние потока перед

переключением контекста. Для того чтобы избежать этого, предлагается использовать копии глобальных переменных для каждого из переключений контекста. Таким образом, после каждого переключения ведется работа со своей картой памяти, а условия на равенство их между собой будут выписаны позднее. При ленивой трансляции выполнение происходит постепенно, с реальным переключением. Инициализируется только первая карта памяти. Далее, при переключении контекста она копируется во вторую и происходит анализ второго потока. При переключении контекста обратно в первый поток происходит его выполнение заново, но уже до следующей точки переключения контекста.

Кроме того, была реализована вторая стратегия, противоположная, - нетерпеливая трансляция, которая предполагает, что поток выполняется сразу от начала и до конца. В начале карты памяти инициализируются произвольными значениями. В точках переключения контекста происходит смена карты памяти. Для сравнения этих стратегий использовались несколько тестов, которые показали, что нетерпеливая трансляция тратит больше времени и памяти на проверку.

8.2 Методы легковесного статического анализа кода

Легковесный статический анализ традиционно применяется на больших программных системах, на которых нельзя провести формальную проверку моделей. Он отличается большей скоростью, но не может претендовать на формальное доказательство отсутствия ошибок.

Статья [6] представляет инструмент Locksmith для статического поиска гонок. Этот метод является наиболее близким к нашему. Он также основан на алгоритме Lockset, но имеет другие подходы для анализа блокировок и разделяемых данных.

Сначала происходит построение графа потока данных — такого графа, в узлах которого стоят переменные, а дуги ведут в ту область памяти, на которую они указывают. Этот анализ чувствителен к полям структур (field-sensitive), это значит, что каждое поле моделируется независимо от других. Такой способ точного анализа потоков данных неизбежно приводит к проблемам с адресной арифметикой. Например, возникают различные проблемы с моделированием указателей типа void, и в статье было рассмотрено несколько вариантов их решения. Важно отметить, что блокировки также считаются данными, для которых строится граф потоков данных. Такой подход имеет как плюсы, так и минусы. Имея граф потоков данных, инструмент может понять, что две различные переменные указывают на одну блокировку и тем самым повысить точность анализа. Однако, если блокировке соответствует очень сложный граф, например, она берется из поля несколько раз вложенной структуры, которая берется из списка, то анализ этого графа сам по себе становится нетривиальной задачей.

Анализ разделяемых данных определяет те области памяти, которые доступны из нескольких потоков. Важной особенностью такого алгоритма определения разделяемых данных является то, что разделяемые переменные фиксируются для всей программы (location-insensitive), и становится невозможным учитывать случаи, в которых некоторая локальная переменная становится разделяемой после некоторых действий.

Еще одним инструментом, который основан на алгоритме Lockset, является Relay[33]. Он описывает изменения во множествах блокировок и доступах к областям памяти относительно точки входа в функцию. Модель потоков очень похожа на вариант, используемый в методе CPALockator — некоторое количество функций, отобранных вручную, считается выполняемыми параллельно. После анализа может быть применен ряд эвристик, которые удалят некоторые предупреждения, возникшие из-за неточной модели окружения.

Одним из важных отличий этого подхода от подхода CPALockator является определение разделяемых данных с учетом алиасов, то есть, данные являются разделяемыми, если равны адресные выражения или одно адресное выражение входит в множество возможных (may)-алиасов другого адресного выражения. Блокировки являются объектами, которые могут быть захвачены и освобождены специальными функциями. Для каждой функции вычисляется множество захваченных блокировок относительно точки входа в функцию. Оно представляет собой пару из точно захватываемых блокировок и возможно освобождаемых. Важно отметить, что блокировки также являются данными, вычисляемыми относительно точки входа в программу. Поэтому, если блокировка является аргументом вызова функции, то для дальнейшего анализа она будет обновлена в терминах переменных вызываемой функции. Наконец, для каждой функции сохраняется множество доступов с захваченными блокировками и эффект функции, который включает в себя относительное множество блокировок.

Метод, предложенный в [34], фокусируется на быстром вычислении must-алиасов, то есть множества таких переменных, которые являются алиасами при любых путях выполнения программы. В этом подходе используются три модели потоков. Первый напоминает модель в подходе CPALockator — каждая функция выполняется параллельно с другими. Вторая модель основана на функциях работы с потоками, такими, как `fork` и `join`. Параллельное выполнение начинается после создания нового потока операцией `fork`, а заканчивается на точке слияния (операция `join`). Третья модель отличается от предыдущих отсутствием точек слияния и ограничением на количество задач, способных выполняться на одном потоке. Данными являются области памяти, а равные области памяти определяются множеством must-алиасов. Блокировки также определяются множеством must-алиасов. Таким образом, основным отличием от нашего метода является акцент на анализе алиасов.

Статья [35] является развитием предыдущей и посвящена проблеме построения графа потока управления для многопоточной программы, если в ней имеются вызовы по функциональным указателям, а также проблеме анализа алиасов в этом случае. В случае рекурсии граф потока управления раскручивается до тех пор, пока не будет найдена неподвижная точка в терминах разбиения Стинсгаарда [36]. Полученный граф состояний обходится и анализируется на предмет наличия состояний гонок с помощью стандартного алгоритма Lockset. Разделяемые переменные определяются, как алиасы к глобальным переменным.

9. Заключение

В данной статье мы предложили новый легковесный метод для поиска состояний гонок, который реализован на основе инструмента CPAchecker. Метод CPALockator учитывает такую специфику ядра операционной системы, как сложный параллелизм, примитивы синхронизации и активное использование адресной арифметики. Еще одной особенностью является возможность масштабируемости на большие объемы исходного кода. Основными отличиями описанного метода является модель памяти, модель параллелизма и способ определения разделяемых данных.

Основной проблемой данного метода является большое количество ложных предупреждений об ошибках. Чтобы уменьшить их количество, планируется интегрировать идеи метода CEGAR (Counterexample Guided Abstraction Refinement), который позволяет учесть условия с помощью предикатной абстракции. В случае задачи поиска состояний гонки метод CEGAR должен быть адаптирован для анализа двух потоков.

Еще одним направлением дальнейшего развития является аппробация предложенного метода на ядре операционной системы Linux.

Список литературы

[1]. Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A Study of Linux File System Evolution, 11th USENIX Conference on File and Storage Technologies (FAST ’13)

[2]. Мутилин В.С., Новиков Е.М., Хорошилов А.В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, С. 349-374, 2012.

[3]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[4]. Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К., Хорошилов А.В., Цыварев А.В. Тестирование операционных систем. Труды ИСП РАН, том 26, С. 73-107, 2014.

[5]. John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk. Effective Data-Race Detection for the Kernel Operating System Design and Implementation (OSDI’10), 2010, USENIX.

[6]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher. Model checking concurrent linux device drivers. ASE'07, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 501-504, ACM, New York, NY, USA, 2007.

[7]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Practical Static Race Detection for C, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.

[8]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006

[9]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, Model Checking Concurrent Linux Device Drivers, ASE'07, November 4–9, 2007.

[10]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[11]. Daniel Wonisch, Block Abstract Memorization for CPAchecker, TACAS 2012, LNCS 7214, pp. 531-533.

[12]. Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е. Архитектура Linux Driver Verification. Труды ИСП РАН, том 20, 2011. С. 163-187.

[13]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)

[14]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.

[15]. CBMC A Tool for Checking ANSI-C Programs, Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pages 193–207, 1999.

[16]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), pp. 154-169, 2000.

[17]. Хорошилов А.В., Мандрыкин М.У., Мутилин В.С. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды ИСП РАН, том 24, С. 219-292, 2013.

[18]. Gerard Basler, Michele Mazzucchi1, Thomas Wahl, Daniel Kroening. Symbolic Counter Abstraction for Concurrent Software CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp. 64-78, Springer-Verlag, Berlin, Heidelberg, 2009.

[19]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[20]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.

[21]. C. Popeea, A. Rybalchenko. Threader: a verifier for multi-threaded programs In Proceedings of the 19th International Conference on Tools and Algorithms for the

Construction and Analysis of Systems (TACAS 2013), LNCS, vol. 7795, pp. 633-636, 2013.

[22]. A. Malkis, A. Podelski, A. Rybalchenko. Thread-modular counterexample-guided abstraction refinement, SAS'10 Proceedings of the 17th international conference on Static analysis, pp. 356-372, Springer-Verlag, Berlin, Heidelberg, 2010.

[23]. S. Qadeer, D. Wu. KISS: Keep it simple and sequential. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04), pp. 14 - 24, 2004.

[24]. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In Conf. on Computer-Aided Verification (CAV), pages 82–97, 2005.

[25]. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In PLDI, pages 446–455, 2007.

[26]. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In Intl. SPIN Workshop on Model Checking Software, pages 114–133, 2008.

[27]. Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp 509-524, Springer-Verlag Berlin, Heidelberg, 2009

[28]. R. DeLine, K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[29]. Naghmeh Ghafari1, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: an empirical evaluation. SPIN'10 Proceedings of the 17th international SPIN conference on Model checking software, pp. 227-244, Springer-Verlag Berlin, Heidelberg, 2010

[30]. S. La Torre, P. Madhusudan, G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 211-222, 2009.

[31]. Gerard Charly Basler, Model Checking Boolean Programs, abhandlung zur Erlangung des titels Doktor der Wissenschaften der ETH Zurich, 28 October 1978.

[32]. S. La Torre, P. Madhusudan, G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09), LNCS 5643, pp. 477-492, 2009.

[33]. Jan Wen Voung, Ranjit Jhala, Sorin Lerner, RELAY: Static Race Detection on Millions of Lines of Code. ESEC/FSE'07, 2007

[34]. Kahlon V., Yang Y., Sankaranarayanan S., Gupta A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226-239. Springer (2007)

[35]. Vineet Kahlon, NishantSinha, Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. ESEC/FSE '09 Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 13-22 ACM New York, NY, USA, 2009

[36]. B. Steensgaard. Points-to analysis in almost linear time. Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96), pp. 32-41, 1996

Lightweight Static Analysis for Data Race Detection in Operating System Kernels

[1]P.S. Andrianov <andrianov@ispras.ru>
[1]V.S. Mutilin <mutilin@ispras.ru>
[1,2,3,4]A.V. Khoroshilov <khoroshilov@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents an approach to lightweight static data race detection called CPALockator. It takes into account the specifics of operating system kernels such as complex parallelism and kernel specifics synchronization mechanisms. The method is based on the Lockset one but it implements two heuristics that are aimed to reduce amount of false alarms: a memory model and a model of parallelism. We consider locks as synchronization primitives. The main target of our research and evaluation is operating system kernels but the approach may be applied to the other programs as well. The method is based on idea of Configurable Program Analysis (CPA) and was implemented in CPAchecker tool. The implementation of the method was done in two stages. Firstly, the set of shared data is determined for every program location, then the analysis of synchronization primitives is performed. On the second stage, information about all potential accesses to shared variables and acquired synchronization primitives is also collected. After analysis the report with results is created. The tool was applied to the kernel of Real-Time operating system. It has about 200 000 lines of source code. CPALocator found 20 new race conditions, which are confirmed by developers. Also, the tool was launched on the Linux device drivers using the LDV framework for preparing tasks for CPALockator. The future investigations will be related to development more precise thread model and integration with more precise heavyweight methods of static analysis.

Keywords: static analysis, data race, operating system kernel, shared data

DOI: 10.15514/ISPRAS-2015-27(5)-6

For citation: Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Lightweight Static Analysis for Data Race Detection in Operating System Kernels. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 5, 2015, pp. 87-116 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-6.

References

[1]. Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A Study of Linux File System Evolution, 11th USENIX Conference on File and Storage Technologies (FAST '13)

[2]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analiz tipovyh oshibok v drajverah operacionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349–374, 2012 (in Russian).

[3]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[4]. Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. Testirovanie operacionnyh sistem [Testing of Operating Systems]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, pp. 73–107, 2014 (in Russian).

[5]. John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk Effective Data-Race Detection for the Kernel Operating System Design and Implementation (OSDI'10), 2010, USENIX.

[6]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher. Model checking concurrent linux device drivers. ASE'07, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 501-504, ACM, New York, NY, USA, 2007.

[7]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Practical Static Race Detection for C, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.

[8]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006

[9]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, Model Checking Concurrent Linux Device Drivers, ASE'07, November 4–9, 2007.

[10]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[11]. Daniel Wonisch, Block Abstract Memorization for CPAchecker, TACAS 2012, LNCS 7214, pp. 531-533.

[12]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).

[13]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)

[14]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.

[15]. CBMC A Tool for Checking ANSI-C Programs, Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pages 193–207, 1999.

[16]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), pp. 154-169, 2000.

[17]. Khoroshilov A.V., Mandrykin M.U., Mutilin V.S. Vvedenie v metod CEGAR — utochnenie abstrakcii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian)

[18]. Gerard Basler, Michele Mazzucchi1, Thomas Wahl, Daniel Kroening. Symbolic Counter Abstraction for Concurrent Software CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp. 64-78, Springer-Verlag, Berlin, Heidelberg, 2009.

[19]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[20]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.

[21]. C. Popeea, A. Rybalchenko. Threader: a verifier for multi-threaded programs In Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013), LNCS, vol. 7795, pp. 633-636, 2013.

[22]. A. Malkis, A. Podelski, A. Rybalchenko. Thread-modular counterexample-guided abstraction refinement, SAS'10 Proceedings of the 17th international conference on Static analysis, pp. 356-372, Springer-Verlag, Berlin, Heidelberg, 2010.

[23]. S. Qadeer, D. Wu. KISS: Keep it simple and sequential. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04), pp. 14 - 24, 2004.

[24]. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In Conf. on Computer-Aided Verification (CAV), pages 82–97, 2005.

[25]. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In PLDI, pages 446–455, 2007.

[26]. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In Intl. SPIN Workshop on Model Checking Software, pages 114–133, 2008.

[27]. Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp 509-524, Springer-Verlag Berlin, Heidelberg, 2009

[28]. R. DeLine, K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[29]. Naghmeh Ghafari1, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: an empirical evaluation. SPIN'10 Proceedings of the 17th international SPIN conference on Model checking software, pp. 227-244, Springer-Verlag Berlin, Heidelberg, 2010

[30]. S. La Torre, P. Madhusudan, G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 211-222, 2009.

[31]. Gerard Charly Basler, Model Checking Boolean Programs, abhandlung zur Erlangung des titels Doktor der Wissenschaften der ETH Zurich, 28 October 1978.

[32]. S. La Torre, P. Madhusudan, G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09), LNCS 5643, pp. 477-492, 2009.

[33]. Jan Wen Voung, Ranjit Jhala, Sorin Lerner, RELAY: Static Race Detection on Millions of Lines of Code. ESEC/FSE'07, 2007

[34]. Kahlon V., Yang Y., Sankaranarayanan S., Gupta A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226-239. Springer (2007)

[35]. Vineet Kahlon, NishantSinha, Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. ESEC/FSE '09 Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 13-22 ACM New York, NY, USA, 2009

[36]. B. Steensgaard. Points-to analysis in almost linear time. Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96), pp. 32-41, 1996