Автоматический поиск ошибок синхронизации в приложениях на платформе Android

В.П. Иванников, С.П. Вартанов, М.К. Ермаков [1]
ivan@ispras.ru, svartanov@ispras.ru, mermakov@ispras.ru

Аннотация. В данной статье рассматривается задача автоматического поиска ошибок синхронизации при проведении динамического анализа приложений в рамках платформы Android. Приводится теоретический обзор существующих методов проведения подобного анализа, затем рассматриваются основные особенности приложений платформы Android с точки зрения применения данных методов. Приводится обзор существующих инструментов для анализа компонентов Android-приложений как в нативном кода на языке Си, так и реализованных на языке Java. Также в статье описан механизм обнаружения ошибок синхронизации, используемый инструментом динамического анализа байт-кода Coffee Machine и его основные аспекты: инструментирование, генерация трассы отношений предшествования и использование инструмента ThreadSanitizer Offline для обнаружения состояния гонок по сгенерированной трассе для платформы Android.

Ключевые слова: динамический анализ, поиск ошибок синхронизации, Android

1. Введение

Платформа Android, вышедшая на рынок операционных систем для мобильных устройств в 2008 году, прочно занимает одно из лидирующих мест в настоящее время. Разработкой приложений для платформы занимается большое число разработчиков, как в составе крупных компаний в сфере информационных технологий, так и выполняющих работу в небольших командах или в индивидуальном порядке. Как и в общем случае при разработке программного

1 Работа проводится при финансовой поддержке Министерства образования и науки Российской Федерации

обеспечения, контроль качества (в том числе устранение дефектов) является одной из приоритетных задач, что в свою очередь обуславливает востребованность средств и инструментов, позволяющих производить подобные действия автоматически.

Для платформы Android приоритетность проведения анализа разрабатываемого программного обеспечения в контексте производительности и стабильности является большей в сравнении с соответствующим аспектом в разработке пользовательских приложений, рассчитанных на использование на персональных компьютерах. Данное различие возникает за счёт ограниченности ресурсов на мобильных устройствах и более жёсткой системы контроля распределения доступных ресурсов, реализованной на платформе Android.

1.1. Приложения платформы Android

Приложения для платформы Android разрабатываются на языке программирования Java или на других языках, транслируемых на этапе компиляции в байткод Java. Полученный байткод затем транслируется в другой формат для виртуальной машины Dalvik, производящей непосредственное выполнение приложения в операционной системе Android. Виртуальная машина Dalvik реализована на языке Си и предоставляет возможности исполнения нативного кода в рамках Java-приложения. Таким образом, проведение статического или динамического анализа приложений с целью обнаружения дефектов или выявления сегментов, влияющих на производительность, должно покрывать как приложения, рассматриваемые в рамках виртуальной машины Dalvik, так и компоненты, реализованные в нативном коде. Несмотря на общие рекомендации по минимальному использованию нативного кода, сформулированные создателями платформы Android, большое количество приложений реализованы таким образом, что фрагменты, отвечающие за вычисления с плотной нагрузкой, вынесены за пределы исходного кода на языке Java.

1.2. Многопоточное исполнение

Операционная система Android практически полностью базируется на использовании графических пользовательских интерфейсов. Реализация интерфейсов на платформе Android выполняется в рамках виртуальной машины Dalvik и, соответственно, использует механизмы и структуры языка Java.

Реализация графического пользовательского интерфейса требует использования многопоточной схемы, в которой для активного приложения существует как минимум одна нить выполнения, осуществляющая регистрацию событий, соответствующих действия пользователя, и как минимум одна нить, осуществляющая выполнение необходимых ответных действий. Приложения, работающие в

операционной системе Android, помимо этого общего ограничения подчиняются стандартным протоколам, определяющих цикл жизни процессов и возможные состояния — необходимо обеспечивать единый подход к использованию мобильного устройства и производить корректное распределение ресурсов.

Указанные особенности и базовые положения платформы Android обуславливают востребованность средств, производящих автоматический анализ Java-составляющей приложений с целью выявления ошибок синхронизации и удовлетворения стандартным положениям реализации графического пользовательского интерфейса.

Начиная с версии Android 3.0 [1] разработчиками была добавлена совместимость с многоядерными архитектурами. Изменения затронули составляющие Android (виртуальную машину Dalvik, основную библиотеку bionic и т. д.), так и предоставили возможность пользоваться параллельными вычисления в нативном коде. В свою очередь, это ставит вопрос об эффективности применения автоматических средств для обнаружения возможных ошибок синхронизации в нативном коде.

1.3. Структура статьи

В настоящей статье рассматривается подход к проведению динамического анализа приложений для платформы Android, позволяющий автоматически выявлять ошибки синхронизации. В разделе 2 приводятся основные алгоритмы, используемые в современных инструментах, их достоинства и недостатки. В разделе 3 рассматриваются особенности проведения анализа нативного кода для приложений Android и приводится обзор существующих инструментов. В разделе 4 приводится описание подхода к проведению анализа для Java-составляющей приложений Android и данные о реализации подхода и полученных результатах. В разделе 5 подводятся итоги проведённого исследования.

2. Динамический поиск ошибок синхронизации

На данный момент инструменты динамического анализа, позволяющие автоматически обнаруживать ошибки синхронизации, реализуют подходы, основанные на одном из двух общих методов: рассмотрении набора блокировок (*lockset*) или на основе отношений предшествования (*happens-before relations*). Существуют также гибридные подходы, сочетающие в себе преимущества обоих алгоритмов и позволяющие уменьшить влияние недостатков алгоритмов на эффективность анализа. Все три типа алгоритмов основаны на выделении событий синхронизации, производимых на основе стандартных средств (семафоров, сигналов и т. д.) и выделении событий доступа к разделяемым между несколькими нитями исполняемой программы. Непосредственная оценка событий доступа и событий синхронизации в

контексте положения о том, что выполнение инструкций отдельными нитями может идти в произвольном порядке (друг относительно друга), позволяет обнаружить состояния гонки и взаимной блокировки.

2.1. Набор блокировок

Метод набора блокировок основан на рассмотрении каждой отдельной нити выполнения как последовательности событий, где каждому событию соответствует некоторое состояние. Данное состояние включает в себя набор объектов, над которым производятся действия, набор блокировок данной нити и набор параметров производимых действий (в частности, тип доступ к объектам – по записи или по чтению). Соответственно, возможные события включают в себя работу с объектами, выставление блокировки для остальных нитей, т. е. получение эксклюзивного доступа для работы в критической секции, и освобождение блокировок (выход из критической секции).

Непосредственный анализ приложения на предмет ошибок синхронизации производится путём рассмотрения подобных последовательностей для всех нитей процесса. Предполагается, что работа с разделяемыми ресурсами является корректной только тогда, когда выполняется следующее условие — доступ к некоторому объекту или набору объектов, одновременно выполняемый двумя или более параллельными нитями, где хотя бы одна из нитей осуществляет изменение содержимого объектов, должен происходить, когда все нити имеют хотя бы одну общую блокировку. С точки зрения средств синхронизации данное условие можно выразить более просто — для каждой критической секции должно иметься хотя бы одно средство синхронизации, которое осуществляет защиту данной секции и используется каждой нитью, претендующей на доступ к данной секции.

Выявление ошибок синхронизации при помощи данного метода возможно как во время работы, так и после завершения по сохранённой трассе событий отдельных нитей. Обновление наборов блокировок происходит напрямую при получении событий входа и выхода из критических секций.

Основным недостатком рассмотренного метод поиска ошибок синхронизации на основе наборов блокировок является использование строго критерия непосредственно ошибочной ситуации. Некоторые механизмы взаимодействия (прямая передача между нитями с помощью каналов, переработка объектов) нарушают заданный критерий, однако являются корректными с точки зрения параллельной работы программы. Соответственно анализ приложений, использующих данные механизмы, приведёт к ложным срабатываниям. Тем не менее данный подход является довольно простым в разработке.

2.2. Отношения предшествования

Концепция отношений предшествования была введена Лэсли Лэмпортом для исследования взаимодействия компонентов распределённых систем [2]. Два события *i* и *j* называются связанными отношением предшествования, если:

1. *i* и *j* произошли в рамках одной нити, причем набор инструкций, вызывающий событие *i,* был выполнен до набора инструкций, вызывающего событие *j*;
2. *i* и *j* произошли в нитях **A** и **B** соответственно, однако *i* является событием отправки сообщения от нити **A** к нити **B**, а *j* является событием получения данного сообщения.

Важной особенностью отношения предшествования является транзитивность — если событие *i* предшествует событию *j*, а событие *j* предшествует событию *k*, то событие *i* также предшествует событию *k*. Именно эта особенность позволяет рассматривать полное объединение событий, произошедших в нитях программы при выполнении и вынести суждение о наличии или отсутствии ошибочных ситуаций.

В рамках данного алгоритма критерием ошибки синхронизации для двух событий доступа к разделяемым объектам (хотя бы одно из которых производить изменение содержимого объектов) является невозможность упорядочить эти события по отношению предшествования. В самом деле, если нельзя утверждать ни то, что одно событие предшествует второму, ни то, что второе событие предшествует первому, то потенциально возможны оба варианта отношения времени возникновения этих событий, то есть имеется ситуация гонки.

Эффективная реализация данного метода является более технически сложной задачей, чем реализация метода наборов блокировок, однако рассмотрение отношений предшествования для событий доступа к разделяемым объектам потенциально устраняет возможность получения ложных срабатываний. Реальная степень устранения ложных срабатываний ограничена полнотой обработки возможных средств прямой коммуникации между параллельными нитями. Как и метод на основе наборов блокировок, подход, рассматривающий отношения предшествования, может применяться во время выполнения программы или по окончании выполнения на основе собранной трассы событий.

Важной особенностью данного метода является более узкое покрытие набора ошибочных ситуаций по сравнению с методом наборов блокировок. Ошибка синхронизации, обнаруженная при рассмотрении отношений предшествования, будет однозначно обнаружена при отслеживании состояний нитей и наборов блокировок, однако обратное утверждение не является верным [3]. При рассмотрении поиска дефектов с точки зрения ложных срабатываний соотношение двух методов меняется на противоположное — рассмотрение отношений предшествования можно производить при фильтрации сообщений об

обнаруженных ошибках синхронизации, найденных на основе исследования наборов блокировок.

2.3. Гибридный метод

Рассмотренные положения однозначно оправдывают эффективность использования гибридного метода, который предоставляет меньшее число ложных срабатываний и покрывает больший объем реальных ошибок синхронизации. Большая часть работ последних лет (например, [3, 7]) посвящена выработке наиболее эффективного применения двухсторонней проверке с помощью гибридного метода.

3. Анализ нативного кода

3.1. Особенности анализа нативного кода

Анализ нативного кода приложений Android затрудняется несколькими особенностями:

- Обращения к нативным методам производятся напрямую из виртуальной машины Dalvik, что делает необходимым анализ самой виртуальной машины для получения полной картины происходящего.
- Процесс запуска приложения (инициализация виртуальной машины Dalvik, загрузка необходимых компонентов) проводится специальным образом, что затрудняет проведение анализа выполнения.

Первая особенность значительно снижает эффективность применения статического анализа кода, в то время как вторая особенность создаёт значительные затруднения как раз для методов динамического анализа.

Общая схема запуска приложения выглядит следующим образом (см. рис. 1):

1. При запуске операционной системы создаётся процесс Zygote, инициализирующий виртуальную машину Dalvik, загружая необходимые системные классы и т. д.
2. Каждый последующий экземпляр Dalvik запускается с помощью стандартного механизма *fork-exec* из процесса Zygote. Непосредственно запросы на запуск копий виртуальной машины процесс Zygote получает от соответствующих сервисов в системе прослушивая специльный канал передачи данных.
3. Для нового экземпляра загружаются непосредственно классы запускаемого приложения, однако часть данных наследуется от процесса Zygote, породившего данный экземпляр.

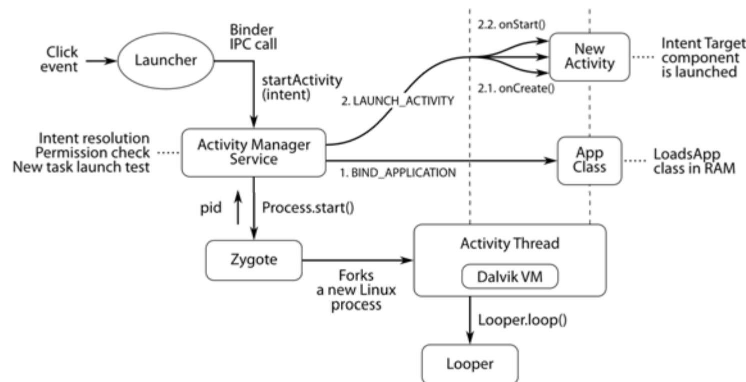

Рис. 1: Запуск Android-приложения

Непосредственная сложность для динамического анализа заключается в том, что должен или осуществляться перехват запуска процесса, или же проводится полный анализ выполнения и процесса Zygote.

В версии Android 3 разработчиками был добавлен специальный механизм обёртки вызовов *exec*, производящихся из процесса Zygote [4], что в значительной степени упрощает проведение динамического анализа, предоставляя пользователю возможность автоматически запускать необходимое средство анализа вместо нового экземпляра виртуальной машины Dalvik (соответственно новый экземпляр создаётся под контролем средства анализа).

Таким образом, проведение динамического анализа представляется более эффективным подходом для автоматического выявления ошибок сихронизации.

3.1. Средства динамического анализа

Среди современных средств динамического анализа с целью обнаружения ошибок синхронизации можно выделить несколько распространённых инструментов:

- Intel Thread Checker [5],
- Sun Thread Analyzer [6],
- helgrind, DRD, ThreadSanitizer [7] (в составе Valgrind).

Intel Thread Checker реализует метод на основе отношений предшествования и доступен (на пробной и коммерческой основе) на платформах Windows и Linux, однако не предоставляется для архитектуры ARM (на которой работает большинство мобильных

устройств с операционной системой Android). Sun Thread Checker производит анализ на основе гибридного метода, однако также не является доступным для архитектур семейства ARM.

Инструменты helgrind, DRD и ThreadSanitizer реализованы в рамках среды разработки средств динамического анализа Valgrind [8], распространяемого свободно. Поддержка архитектуры ARM и платформы Android была добавлена разработчиками в версиях 3.6.0 и 3.7.0 соответственно [9]. Инструменты helgrind и DRD основаны на методе отношений предшествования, но имеют ряд различий на более техническом уровне. Инструмент ThreadSanitizer был разработан независимо от основного комплекта Valgrind и основан на гибридном методе [7], позволяющем демонстрировать более высокую эффективность анализа по сравнению с helgrind и DRD. Реализация инструмента также работает быстрее остальных инструментов и обладает рядом дополнительных особенностей:

1. Помимо общего механизма подавления ошибок (*suppression*), используемом в Valgrind для фильтрации сообщений о ложных срабатываниях, в ThreadSanitizer имеется система фильтрации компонентов, загруженных в оперативную память и используемых при выполнение программы (по библиотекам и по отдельным функциям). Поддержка платформы Android в составе комплекса Valgrind не является полностью завершённой и, в частности, для инструментов helgrind и DRD не накоплено достачное число сигнатур ошибок, обеспечивающих подавление сообщений о ложных срабатываниях.
2. ThreadSanitizer предоставляет поддержку аннотаций исходного нативного кода, которые указывают на отсутствие реального дефекта в «подозрительных» фрагментах. Использование данных аннотаций позволяет снизить число ложных срабатываний в более сложных ситуациях.

Использование ThreadSanitizer является возможным при помощи указанного средства обёртки вызовов *exec*. Проведение анализа в данном случае охватывает большой объем нативного кода (фактически, рассматриваются ситуации, происходящие внутри виртуальной машины Dalvik), однако является единственным способом обработки вызовов нативного кода и выявления потенциальных ошибок синхронизации.

На основе проведенного обзора проведение динамического анализа применение инструмента ThreadSanitizer является наиболее эффективным подходом к автоматическому обнаружению возможных проблем зацикливания.

4. Анализ Java-приложений

4.1. ThreadSanitizer

Как было сказано ранее, ThreadSanitizer — проект, посвящённый динамическому поиску ошибок синхронизации.

С точки зрения алгоритма инструмента, важны следующие типы операций:

чтение и запись, т. е. операции доступа к памяти,

операции блокировок,

сигналы и ожидания сигналов,

операции работы с потоками.

Будем называть такие операции «синхронизационными».

Для алгоритма важно понятие сегментов потока. В терминах ThreadSanitizer, сегмент — непрерывная последовательность инструкций потока, в которой не встречаются указанные выше операции.

В основе алгоритма лежит механизм определения отношения предшествования между различными сегментами и синхронизационными операциями различных потоков. Отношение предшествования означает, что набор операций, стоящий в левой его части не может встречаться в трассе выполнения программы после набора операций, стоящего в правой части.

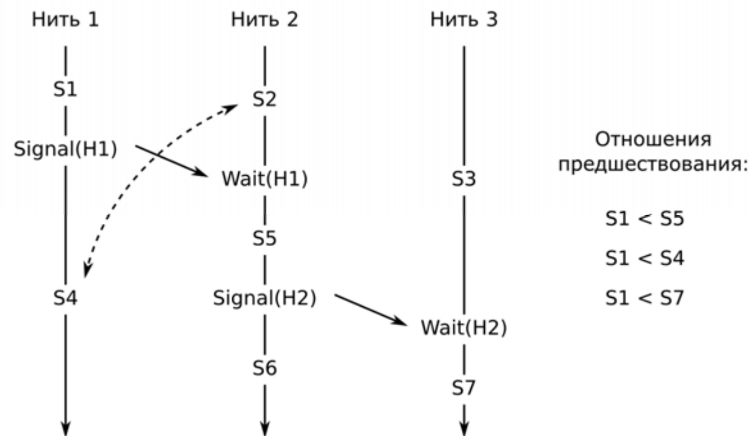

Рис. 2: Отношения предшествования

В терминах ThreadSanitizer, состояние гонки — доступ к общим данным из двух различных потоков в точках, между которыми нет отношения предшествования, если один из доступов есть доступ типа запись.

Если инструмент обнаружил дефект, это означает, что в соответствии с построенными отношениями предшествования, состояние гонки может возникнуть (но не возникает обязательно) на одном из запусков приложения. Обнаруженный дефект может быть ложным в случае, если построены не все отношения предшествования. Например, это может происходить, если некоторые части программы не доступны для анализа — не была произведена инструментация библиотеки или отдельного класса и соответствующие события не были зафиксированы в трассе.

Даже если обнаруженный дефект действительно имеет место, возникает проблема воспроизведения данной конкретной ошибки.

Основная часть проекта ThreadSanitizer — инструмент для обнаружения состояния гонки для программ, написанных на языке C++, однако здесь речь пойдёт о других, экспериментальных инструментах, созданных в рамках этого проекта — ThreadSanitizer Offline и Java ThreadSanitizer. Оба этих инструмента также основаны на методе построения отношений предшествования.

4.2. ThreadSanitizer Offline

ThreadSanitizer Offline — экспериментальный инструмент, реализующий вторую часть алгоритма обнаружения. Этот инструмент принимает на вход информацию о выполнившихся синхронизационных инструкциях в виде трассы событий и строит наборы отношений предшествования для поиска состояний гонки.

Таким образом, ThreadSanitizer Offline не зависит ни от языка программирования, на котором написано анализируемое приложения, ни от механизма генерации трассы событий. Необходимо лишь, чтобы трасса событий соответствовала синтаксису инструмента.

4.3. Java ThreadSanitizer

Java ThreadSanitizer — инструмент, производящий инструментацию байт-кода. Он реализует первую часть алгоритма обнаружения ошибок.

Инструмент производит динамическую инструментацию в ходе выполнения анализируемого приложения. Такая инструментация позволяет извлекать информацию о выполнении и строить трассу событий. Построенная трасса может быть использована инструментом ThreadSanitizer Offline для обнаружения состояний гонки.

Таким образом, совместно Java ThreadSanitizer и ThreadSanitizer Offline позволяют произвести поиск ошибок синхронизации в приложении на языке Java.

Для инструментации байт-кода Java ThreadSanitizer использует библиотеку ASM [10]. Эта библиотека использует шаблон проектирования посетитель и java agent для динамической инструментации байт-кода.

Для работы инструмента на платформе Android это представляет проблему. В связи с тем, что в операционной системе Android вместо виртуальной машины Java используется виртуальная машина Dalvik, использование java agent не представляется возможным. Другая проблема связана с отсутствием доступа к системным библиотекам и классам виртуальной машины Dalvik и таким образом часть информации о процессе выполнения может быть потеряна.

4.4. Coffee Machine

Coffee Machine — разработанный в рамках данного исследования инструмент для динамического анализа приложений, код которых может быть транслирован в байт-код виртуальной машины Java.

В соответствии с несколькими задачами, которые решает инструмент, его структура состоит из нескольких компонент:

- **Instrumentator**. Обработка анализируемго приложения, поиск зависимостей, построение списка классов для инструментации и статическая инструментация.
- **Interpreter**. Анализ трассы исполнения и построение булевых ограничений для решателя STP [11].
- **Main module**. Обеспечение взаимодействия компонент и запуск анализируемого приложения на исполнение.

Coffee Machine производит статическую инструментацию байт-кода, посредством добавления инструкций вызова методов классов `Daemon` и `Concurrency`, включёнными в состав инструмента.

Методы класса `Daemon` производят трассировку помеченных данных (информации, которая зависит от входных данных) и генерируют набор булевых ограничений для решателя STP. В ходе выполнения приложения `Daemon` для каждого условного перехода строит булево выражение, определяющее переход по альтернативной ветке. STP проверяет набор ограничений на выполнимость и в случае успеха возвращает список присваиваний для входных данных. В соответствии с построенными входными данными один или несколько условных переходов инвертируются и процесс выполнения происходит по ранее не покрытому пути.

Методы класса `Concurrency` вызываются для встретившихся в ходе выполнения операций синхронизации. В их задачи входит сбор информации о выполняющихся потоках, отслеживание блокировок, сигналов и объединение полученных данных в трассу событий. Трасса

событий соответствует синтаксису, который понятен инструменту ThreadSanitizer Offline.

Процесс инструментирования байт-кода в Coffee Machine основывается на статическом подходе. Для этих целей используется библиотека BCEL (Byte Code Engineering Library [12]). Статический подход означает, что инструментирование кода происходит один раз, до процесса выполнения приложения. У используемого подхода есть ряд недостатков. Инструментирования с использованием BCEL происходит медленнее, например, фреймворка для динамической инструментации ASM и требует предварительного определения списка классов для инструментирования, которые могут быть потенциально использованы при выполнении приложения (при динамическом подходе исключается возможность инструментации не использованных классов). Основное преимущество данного подхода в возможности конвертирования инструментированного байт-кода в формат для требуемой виртуальной машины.

В рамках поставленой задачи это означает, что после этапа инструментирования код может быть преобразован в байт-код формата DEX, интерпретируемый виртуальной машиной Dalvik. Таким образом он может исполняться на платформе Android.

Построенная после выполнения приложения трасса событий подаётся на вход инструменту ThreadSanitizer Offline, который осуществляет поиск ошибок синхронизации.

Таким образом, Coffee Machine, в отличие от Java ThreadSanitizer, имеет возможность извлекать трассу событий для Java-приложений на платформе Android.

Одной из существенных проблем является невозможность инструментирования системных библиотек и стандартных синхронизационных методов Java. В связи с этим, класс `Concurrency` отдельно отслеживает вызовы таких методов и производит их частичную симуляцию. Например, при вызове в приложении метода инициализации класса

```
java.util.concurrent.locks.ReentrantReadWriteLock
```

также будет вызван метод

```
a_java_util_concurrent_locks_ReentrantReadWriteLock_
init(Object o)
```

класса `Concurrency` с соответствующим образом переданными ему параметрами. Этот метод сохранит необходимую информацию.

Для выполнения подобных действий могут быть необходимы параметры, передаваемые методу и его возвращаемое значение. Для этого вершина стека виртуальной машины Java, включающая в себя передаваемые параметры и ссылку на объект (в случае, если метод не статический) дублируются дважды, затем происходит вызов метода,

производящего предварительные действия. У этого метода нет возвращаемого значения, а параметры соответствуют входным параметрам целевого метода. Далее происходит вызов целевого метода. Метод, производящий завершающие действия, принимает на вход входные параметры целевого метода и его возвращаемое значение, которое возвращает без изменений. В случае, если для выполнения действий не требуются входные параметры или возвращаемое значение, отдельные шаги процесса могут быть опущены. Схема процесса приведена на рис. 3.

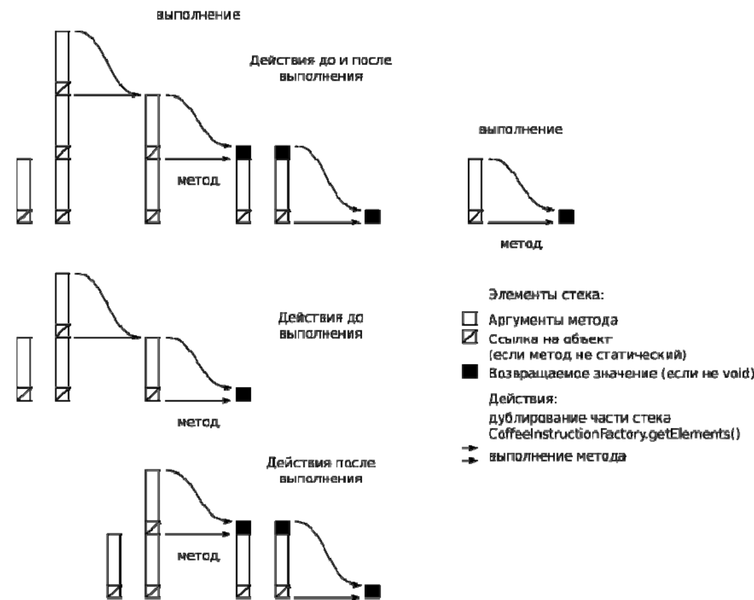

Рис. 3: Схема выполнения подготовки стека для проведения дополнительных действий до и после выполнения метода

4.5. Экспериментальные результаты

Проведение анализа с помощью инструмента Coffee Machine проводилось на наборе тестов, разработанном в рамках Java ThreadSanitizer, а также на основных приложениях операционной системы Android (Browser, Contacts, Mms и некоторые другие). Тесты в составе Java ThreadSanitizer покрывают большую часть API стандартных классов языка Java для синхронизации нескольких нитей (семафоры, барьеры и т. п.), а также дефекты параллельного выполнения, обнаруженные в реальных проектах. В рамках разработки инструмента

Coffee Machine набор тестов был расширен для покрытия специализированных ситуаций использования инструментов синхронизации Java. Для автоматический работы с приложениями операционной системы Android (генерация сценариев взаимодействия) был использована модифицированная версия комплекса инструментов для автоматического тестирования графических приложений GUItar [13, 14]. Такие факторы, как невозможность проведения инструментации непосредственно системных классов, привели к снижению эффективности анализа и увеличению числа ложных срабатываний. Были разработаны дополнительные правила для инструментации кода, использующего конструкции Android Java, позволяющие снизить число ложных срабатываний, связанных с механизмом передачи сообщений, активно используемым при работе с системным интерфейсом. Рассмотренные результаты анализа выявили несколько реальных дефектов синхронизации, которые потенциально могут происходить при разрешении специализированных ситуаций взаимодействия нескольких сервисов. Примером одной из таких ситуаций может завершение работы сервиса, отвечающего за сохранение прикреплённого к электронному письму файла, в приложении Email. Искусственное внедрение простых временных задержек в одну из нитей, работающих параллельно, позволило воспроизвести некорректную ситуацию гонки, которая привела к исключению `NullPointerException` и к аварийному завершению приложения Email. Появление исключительной ситуации требует выполнения инструкций двух нитей в строго определённой последовательности, что практически устраняет возможность выявления ошибки при обычном тестировании.

При проведении анализа набора приложений Android был произведена оценка степени замедления скорости работы, связанного с использованием инструментированного кода, на основе нескольких факторов (степень покрытия интерфейса работы с нитями, механизм генерации и последующей передачи трассы компоненту ThreadSanitizer Offline). Итоговая степень замедления при проведении инструментации исходного кода непосредственно самого приложения не превышает пятикратной отметки, причём основное увеличение приходится на этапы изначальной загрузки компонентов графического интерфейса приложения, в то время как выполнение действий над уже загруженными компонентами затрагивается в меньшем объёме (взаимодействие с приложением укладывается в рамки необходимых временных задержек в комплексе GUITAR, использующихся в модуле эмуляции наборов событий).

5. Заключение

В данной статье рассмотрены общие принципы методов автоматического обнаружения ошибок синхронизации в программном обеспечении, а также особенности данной задачи в контексте

приложений, разработанных для платформы Android. Был проведён краткий обзор методов динамического анализа программ с целью выявления ошибок подобного рода и современных средств, предлагающих реализацию данных методов, для работы с нативным кодом на языке программирования C и работы с кодом на языке Java (при разработке приложений Android используются компоненты как на языке C, так и на языке Java). Для инструментов, производящих анализ нативного кода, приведены основные сложности, связанные с особенностями работы приложений в рамках платформы Android (специфическая системы запуска виртуальной машины Dalvik), и рассмотрен подход к использованию механизма обёртки, необходимый для проведения анализа. В качестве наиболее эффективного среди существующих инструментов выделено средство ThreadSanitizer.

В заключительной части статьи рассматриваются инструменты для анализа компонентов, реализованных на языке Java, и, в частности, инструмент Java ThreadSanitizer. Приведён анализ особенностей платформы Android (виртуальной машины Dalvik), требующих модификации инструмента. Наконец, предложена непосредственное описание реализации инструмента Coffee Machine, использующего подход Java ThreadSanitizer, на основе статической инструментации. Приводится оценка результатов использования полученного инструмента на расширенном наборе тестов для Java ThreadSanitizer и на наборе стандартных приложений платформы Android.

Список литературы

[1]. Официальный анонс выпуска Android 3.0 [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html)

[2]. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978.

[3]. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 167–178, New York, NY, USA, 2003.

[4]. Julian Seward. Анализ текущего состояния инструмента Valgrind в контексте использования на платформе Android [HTML] (http://blog.mozilla.org/jseward/2011/09/27/valgrind-on-android-current-status)

[5]. U. Banerjee, B. Bliss, Zh. Ma, P. Petersen. Unraveling Data Race Detection in the Intel® Thread Checker. In Proceedings of STMCS '06. Manhattan, NY, USA, 2006

[6]. Руководство по инструменту Sun Thread Analyzer [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html)

[7]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. WBIA '09, New York City, NY, USA, 2009

[8]. N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007.

[9]. Примечания к выпускам версий Valgrind [HTML] (http://valgrind.org/docs/manual/dist.news.html)

[10]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[11]. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In CAV 2007, LNCS 4590, pages 519–531

[12]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[13]. Atif Memon. An event-flow model of GUI-based applications for testing. Software Testing, Verification & Reliability. Volume 17 Issue 3, pages 137-157. John Wiley and Sons Ltd. Chichester, UK, 2007

[14]. Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of The 10th Working Conference on Reverse Engineering, Nov. 2003

Automatic concurrency defect detection for Android applications

V. P. Ivannikov, S. P. Vartanov, M. K. Ermakov
ivan@ispras.ru, svartanov@ispras.ru, mermakov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper describes issues related to automatic detection of concurrency defects using dynamic analysis methods with primary target subject as the Android platform. Due to the increased popularity and spread of Android platform and, more importantly, high importance of concurrent work flow for all user applications on this platform, automatic tools for defect detection are likely to be beneficial for many developers. The paper is organised as follows: section 1 provides a brief introduction to the discussed subjects (concurrency basics, Android platform basics). Section 2 discusses theoretical base for a set of dynamic concurrency detection methods. Section 3 contains an overview of several key points in Android work flow related to native code execution in system utilities and in user applications through the JNI mechanisms. This overview is followed by a description of existing tools implementing concurrency defect detection methods for native code and the practical considerations of applying these tools for Android analysis. Section 4 provides the information on Java-oriented tools for concurrency defect detection (Java ThreadSanitizer and Coffee Machine) with the major focus on Coffee Machine, developed within the scope of this project. An overview of Coffee Machine tool functionality and base methods is given, followed by an evaluation of practical application of the tool to Android platform. Section 5 concludes the paper with an evaluation of key issues for both native and Java dynamic analysis for automatic concurrency defect detection.

Keywords: dynamic analysis, concurrency defect detection, Android

References

[1]. Android 3.0 official release statement. [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html)
[2]. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563
[3]. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming 2003. pp. 167–178. doi: 10.1145/781498.781528
[4]. Julian Seward. Valgrind on Android — Current Status. [HTML] (http://blog.mozilla.org/jseward/2011/09/27/valgrind-on-android-current-status)
[5]. U. Banerjee, B. Bliss, Zh. Ma, P. Petersen. Unraveling Data Race Detection in the Intel® Thread Checker. Presented at The First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2006.
[6]. Sun Thread Analyzer documentation [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html)

[7]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203
[8]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of ACM SIGPLAN conference on Programming language design and implementation, 2007. pp. 89-100. doi: 10.1145/1250734.1250746
[9]. Valgrind changelog [HTML] (http://valgrind.org/docs/manual/dist.news.html)
[10]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)
[11]. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. Proceedings of the 19th international conference on Computer aided verification, 2007. pp. 519-531
[12]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)
[13]. Atif Memon. An event-flow model of GUI-based applications for testing. Software Testing, Verification & Reliability. Volume 17 Issue 3, 2007. pp. 137-157. doi: 10.1002/stvr.v17:3
[14]. Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of The 10th Working Conference on Reverse Engineering, 2003. pp. 260 – 269. doi: 10.1109/WCRE.2003.1287256