Прикладное применение динамического анализа программ, исполняющихся в интерпретирующих средах[1]

С. П. Вартанов <mermakov@ispras.ru>
М. К. Ермаков <mermakov@ispras.ru>
А. Ю. Герасимов <agerasimov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Сложность современного программного обеспечения постоянно растет, в связи с чем возникает потребность в автоматических инструментах выявления ошибок в разработанных программах. В рамках данной статьи мы представляем решение некоторых задач, возникающих в процессе разработки программного обеспечения. Профилирование работы программ с динамической оперативной памятью, символьное исполнение программ с графическим пользовательским интерфейсом, обнаружение ошибок в параллельных программах – небольшой, но крайне важный класс задач, решение которых востребовано индустрией разработки программного обеспечения. В связи с отсутствием в виртуальной машине Dalvik операционной системы Android стандартных средств подключения агентов, на базе которых возможно проведение динамической инструментации байт-кода, в статье рассмотрен подход к профилированию использования динамической памяти Java-программами при помощи инструмента, реализованного как модифицированная виртуальная машина Dalvik операционной системы Android. Показана обоснованность примененного подхода, приведены практические результаты анализа нескольких программ из комплекта поставки операционной системы Android. Также в статье описано решение задачи динамического символьного исполнения программ с графическим пользовательским интерфейсом с целью генерации минимальных последовательностей управляющих воздействий на пользовательский интерфейс, обеспечивающих тестовое покрытие программы на базе статической инструментации байт-кода Java-программ и модификации инструмента генерации тестовых наборов для приложений с графическим пользовательским интерфейсом GUITAR. В завершении статьи рассматривается применение особенности реализации инструмента обнаружения ошибок синхронизации параллельных программ на языке Java, исполняемых виртуальной машиной Dalvik операционной системы Android на базе статической инструментации байт-кода Java-программ и применения инструмента ThreadSanitizer.

[1] Работа проводится в рамках научно-исследовательских работ Института системного программирования РАН в 2014 – 2017 годах

Ключевые слова: динамический анализ программ; анализ программ.

DOI: 10.15514/ISPRAS-2017-29(1)-9

Для цитирования: Вартанов С.П., Ермаков М.К., Герасимов А.Ю. Прикладное применение динамического анализа программ, исполняющихся в интерпретирующих средах. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 135-148. DOI: 10.15514/ISPRAS-2017-29(1)-9

1. Введение

В настоящее время в индустрии программного обеспечения наблюдается развитие сектора разработки программного обеспечения на языках программирования с интерпретирующей средой, например, Java Virtual Machine (JVM) или Common Language Runtime (CLR) как для серверного программного обеспечения, так и для настольных и мобильных приложений (например, на базе операционных систем Android или Windows Phone). В связи с этим возникает задача анализа программ, выполняющихся в интерпретирующей среде, с учетом специфики как самой среды, так и окружающего системного программного обеспечения.

Анализ программ может производиться для проверки различных свойств программного обеспечения, например, на наличие дефектов и уязвимостей, производительности и ресурсоемкости целевого программного обеспечения, а также с целью восстановления алгоритмов и моделей работы целевой программы. Таким образом наличие методик и средств анализа программ является одной из важнейших компонент обеспечения качества продуктов на протяжении всего жизненного цикла программного обеспечения.

Исследования в рамках данного проекта направлены на разработку подходов и средств автоматического и полуавтоматического анализа приложений, выполняемых в рамках интерпретирующих сред. Подобный анализ может проводиться для поиска дефектов и уязвимостей, обнаружения частей программ, осуществляющих неэффективное использование временных и системных ресурсов и др. Актуальность подобных средств анализа в настоящее время определяется распространённостью приложений, исполняющихся в интерпретирующих средах (например для языка Java, являющегося целевым для данного проекта, или для языка C# со средой выполнения CLR), в частности на мобильных платформах, таких как Android и Windows Phone.

В статье рассмотрены особенности методов динамического анализа, нацеленных на профилирование программ, а также методов для анализа программ, обладающих графическим пользовательским интерфейсом и использующих параллельные вычисления. Также в работе помимо анализа бинарного кода для архитектуры x86 и x86-64, рассматриваются особенности анализа программ, работающих на архитектуре ARM, и программ, написанных на языке Java.

2. Профилирование программ

Динамический анализ эффективно применяется для проверки свойств времени выполнения программы, таких как количество потребляемой программой памяти, объём потребляемых вычислительных ресурсов, статистика использования кэша и т. п. В рамках данного проекта было проведено исследование методов сбора информации об эффективности использования памяти программами операционной системы Android [1]. Основная часть программ для этой платформы написана на языке Java и выполняется виртуальной машиной Dalvik [2]. Особенности профилирования программ тесно связаны с особенностями как самой операционной системой, так и с особенностями используемой виртуальной машины.

Виртуальная машина Dalvik реализует лишь часть стандартных интерфейсов виртуальных машин Java. Так, в ней присутствуют механизм вызова функций в бинарном коде JNI, протокол передачи данных JDWP и механизм создания слепков кучи, но отсутствуют средства поддержки динамически подключаемых агентов. Последнее ограничение означает невозможность использования стандартных средств динамической инструментации Java программ и средств динамического анализа программ, реализация которых основана на применении динамической инструментации. Протокол JDWP описывает способы обмена информацией между отладчиком и виртуальной машиной Dalvik. С его помощью можно запрашивать информацию о текущем состоянии памяти виртуальной машины, управлять потоками, механизмом сборки мусора и модифицировать параметры объектов и классов, а также описывать события, при возникновении которых данные о состоянии виртуальной машины автоматически отправляются отладчику.

Виртуальная машина Dalvik предоставляет интерфейс для извлечения в заданные моменты времени слепков кучи, которые содержат в себе информацию о состоянии объектов, хранящихся в памяти. Существует множество инструментов, нацеленных на анализ слепков, сделанных на разных стадиях процесса выполнения анализируемой программы, однако точность таких инструментов напрямую зависит от частоты извлечения слепков, создание которых требует полной остановки работы виртуальной машины. К тому же при помощи анализа разницы между слепками не представляется возможным сделать выводы о том, какие из выделенных и освобождённых объектов были использованы в ходе выполнения программы. Это не позволяет в полной мере делать выводы об эффективности использования памяти анализируемой программой.

В связи с указанными особенностями в настоящем проекте с целью извлечения всей необходимой информации для полноценного анализа использования памяти приложениями платформы Android предлагается производить постоянный сбор всех событий, связанных с выделением, освобождением и использованием памяти при помощи внесения изменений в виртуальную машину Dalvik, упаковку собранной информации в специальные пакеты и

передачу их с использованием протокола JDWP специальному программному средству, анализирующему указанную информацию.

2.1 Сбор информации

Осуществляется сбор следующих типов событий: загрузка и инициализация класса, создание экземпляра класса, использование экземпляра при помощи интерфейсов доступа к нему, а также освобождение памяти при помощи сборщика мусора.

Собранная информация упаковывается в пакеты определённых типов:

- "request allocations" для информации о созданных объектах;
- "request freed" для информации об их освобождении;
- "object usage" для информации об использовании созданных объектов;
- "class information" для информации о внутренней структуре загружаемых классов.

Создание новых объектов в виртуальной машине Dalvik происходит в рамках единой процедуры. Это позволяет использовать её для точечного сбора данных о выделенных объектах.

Освобождение созданных объектов происходит в рамках процедуры сбора мусора и осуществляется в соответствии с алгоритмом MarkSweep [3]. Сбор информации об освобождённой памяти происходит на финальном этапе работы алгоритма, в рамках которого известен полный список объектов, память которых необходимо освободить.

Анализ использования памяти может быть произведён с разной степенью точности в зависимости от задач, стоящих перед анализатором. В первом случае, если достаточно информации о том, какие из объектов были использованы перед тем, как занимаемая ими память была освобождена, а какие – нет, и при этом не имеет значения, какая именно часть объектов была использована, достаточно использовать единственный дополнительный флаг на каждый объект, а передачу информации об использовании можно объединить с передачей отладчику информации об освобождении памяти, разделив объекты на два множества: те, к которым был осуществлён доступ по крайней мере один раз, и те, которые не были использованы вплоть до их уничтожения сборщиком мусора.

Если же для анализа необходима информация о том, какие именно части объекта были использованы (например, для определения эффективности выбора внутренней структуры классов), в ходе работы программы происходит отслеживание всех методов, осуществляющих доступ к внутренним структурам объектов через внешние интерфейсы. В этом случае для передачи информации отладчику используется специальный тип пакетов, в которых сохраняется идентификатор объекта и смещение, по которому был осуществлён доступ. Информация о внутренней структуре загруженных классов используется для восстановления полей объектов по сохранённым смещениям.

2.2 Инструментальное средство

На основе описанных подходов был реализован прототип инструментального средства, который предназначен для работы на внешней рабочей станции, имеющей подключение к устройству с платформой Android, и который обеспечивает сохранение, обработку и анализ информации, поступающей от модифицированной виртуальной машины Dalvik. Инструментальное средство производит установку соединения с виртуальной машиной и отправляет команду запуска сбора информации. В ходе работы анализируемого приложения инструмент прослушивает канал связи и осуществляет приём пакетов описанных типов с информацией об использовании памяти.

Инструментальное средство также обладает графическим пользовательским интерфейсом, отображающим процесс использования памяти в анализируемой программе в режиме реального времени, и системой команд для управления отображением.

2.3 Практические результаты

С целью проверки практической применимости разработанного прототипа инструмента и предложенных методов, прототип инструмента был использован для анализа работы с памятью в ряде стандартных программ операционной системы Android: Mms (программа для обмена сообщениями), Calendar (органайзер), Browser (средство для просмотра содержимого страниц сети Интернет), Deskclock (стандартные часы) и Contacts (средство для работы со списком контактов). В качестве целевой платформы анализа использовалась версия операционной системы Android 4.2.2, работающая на устройстве PandaBoard. Сбор информации об использовании памяти производился для различных сценариев использования приложений. По результатам исследований был выявлен ряд закономерностей, свойственных всем проанализированным программам.

Для отдельных классов объектов программ Java были выявлены шаблоны частот использования полей этих классов, на основе которых можно сделать вывод об эффективности применения механизма наследования.

Было выяснено, что основная доля выделенной и используемой памяти приходится на массивы примитивных типов. Так от 50 до 80 % всей выделенной памяти составляли объекты типов `byte[]`, `char[]` и `int[]`. Это объясняется использованием их для хранения изображений, которые формируются для графического пользовательского интерфейса, работой со строками и использованием системы обработки исключительных ситуаций соответственно. По результатам проведенного исследования программ для операционной системы Android можно сделать вывод, что как в конечных классах, так и в классах стандартной библиотеки Android, от которых они были унаследованы существует ряд полей, которые не используются ни в одном из созданных объектов. Если применить методы рефакторинга стандартной

библиотеки классов операционной системы Android и классов приложений, то можно значительно сократить использование оперативной динамической памяти программами операционной системы Android.

3. Анализ программ с графическим интерфейсом

Использование графического пользовательского интерфейса с одной стороны облегчает взаимодействие пользователя с программным средством и расширяет возможности его использования, с другой стороны значительно усложняет логику программного обеспечения. Механизмы графического пользовательского интерфейса подразумевают ожидание действий пользователя, как правило представляемых в программе в виде событий, и реакцию на них, как правило в виде вызова функций-обработчиков событий графического пользовательского интерфейса, что при значительном количестве активных графических элементов порождает множество различных сценариев работы программы. В связи с большим количеством возможных комбинаций событий и, следовательно, вызовов обработчиков событий, в программах с графическим пользовательским интерфейсом, ручное тестирование подобных программ становится весьма трудоёмким процессом, что приводит к сложности как обнаружения ошибок в программе, так и воспроизведения и отладки. Поэтому, возникает необходимость создания инструментальных средств автоматической генерации тестовых наборов для программ с графическим пользовательским интерфейсом, не требующих постоянного участия человека в процессе анализа поведения программы.

Для анализа программ с графическим пользовательским интерфейсом классические методы анализа, основанные на символьном выполнении, не подходят без дополнительных модификаций, поскольку потоки данных часто проходят через функции библиотек, отвечающих за отображение графического интерфейса, чем сильно усложняют формулу в булевских ограничениях операциями, не влияющими на логику исполнения программы. В связи с этим, для проведения анализа методами символьного исполнения требуется предварительное построение модели графического пользовательского интерфейса и структуры анализируемой программы.

Большинство современных программ обладают сложной структурой и большим количеством элементов графического интерфейса, и их структура часто не полностью определяется статически, а дополняется или модифицируется в процессе выполнения программы. Существующие средства анализа осуществляют построение модели графического интерфейса на основе запуска программы. Одним из наиболее эффективных инструментов для автоматического анализа программ с графическим интерфейсом на платформе Android является инструмент GUITAR [4], который позволяет проводить без участия эксперта проводить построение моделей программ, графический интерфейс которых основан на использовании AWT/Swing, SWT, и программ платформы Android. Основным недостатком этого инструмента является то, что

построенная модель графического пользовательского интерфейса основывается на информации, собранной в процессе одного запуска приложения и конкретном сценарии его использования. При различных сценариях работы с программой эта модель может быть разной, а значит требуется извлекать информацию о структуре графического пользовательского интерфейса программы с учётом разных запусков.

В рамках проекта рассмотрен гибридный метод, совмещающий автоматическое построение модели графического интерфейса на основе нескольких запусков программы с анализом, основанном на символьном исполнении.

Существует множество инструментов, осуществляющих динамический анализ программ на языке Java методами символьного исполнения. К таким можно отнести инструмент Java PathFinder [5], который представляет собой символьную виртуальную машину Java. Инструмент JDart [6] позволяет строить новые пути выполнения программы при помощи символьного исполнения и основан на инструменте Java PathFinder. Инструмент Coffee Machine [7], использующий динамическое символьное исполнение, основан на методах статической инструментации байт-кода языка Java и позволяет производить анализ вне зависимости от виртуальной машины.

В связи с тем, что на платформе Android для запуска Java-программ используется собственная виртуальная машина Dalvik, в рамках проекта использовался механизм статической инструментации инструмента Coffee Machine, а также возможности инструмента GUITAR для построения модели графического пользовательского интерфейса приложения.

GUITAR строит модель графического пользовательского интерфейса приложения, основываясь на иерархической структуре элементов графического пользовательского интерфейса и при помощи построения ориентированного графа потока событий. Ребро графа потока событий между событиями A и B обозначает возможность возникновения события B после события A. Связи между элементами в этом графе означают возможные события между графическими элементами. Таким образом, пути в построенном графе соответствуют некоторым сценариям работы с анализируемым приложением. Одним из компонентов инструмента осуществляется построение таких сценариев, а другим — их воспроизведение.

Механизм символьного исполнения применительно к программам с графическим пользовательским интерфейсом предлагается использовать, основываясь на следующих принципах:

- традиционная трасса символьного выполнения строится исключительно для событий, связанных с графическим интерфейсом;
- помеченными (a, соответственно, и символьными) считаются не только данные, косвенно зависящие от входных, но и поля примитивных типов и массивы примитивного типа и поля объектов, к которым существует доступ из статических полей классов;

- в трассу добавляются специальные маркеры начала и конца события.

Общий механизм динамического символьного выполнения применяется не только для инвертирования условий, но и для перестановки частей трассы в построенных ограничениях, что позволяет исследовать различные пути выполнения программы и различные сценарии с точки зрения пользовательского интерфейса.

Практическая применимость предложенных методов анализа программ с графическим пользовательским интерфейсом была проверена на программах Calculator и ProgCalc операционной системы Android. В результате было обнаружено 17 и 2 дефекта соответственно. Для первого приложения причиной аварийного завершения явилась не обработанная должным образом исключительная ситуация типа `NumberFormatException`, для второго — `NumberFormatException` и `ArithmeticException`.

4. Анализ многопоточных программ

В настоящее время активно развивается отрасль создания вычислительных устройств, обладающих двумя и более вычислительными ядрами, а также создаются механизмы синхронизации между вычислительными процессами средствами операционных систем. В связи с этим, для более эффективного использования вычислительных ресурсов активно разрабатываются программы, использующие параллельные вычисления. Особенно это справедливо для программ, использующих графический пользовательский интерфейс, поскольку для снижения задержек при работе с графическим пользовательским интерфейсом программы используется несколько вычислительных потоков, в одном из которых реализуется цикл обработки сообщений от элементов графического пользовательского интерфейса.

Высокая сложность логики программы, использующей параллельные вычисления, приводит к возникновению специфических ошибок, связанных с синхронизацией вычислительных потоков. Основная сложность таких ошибок заключается в том, что их проявление зависит не только от входных данных программы, но также и от алгоритмов работы планировщика потоков операционной системы, т. е. недетерминированно с точки зрения пользователя, что сильно затрудняет их поиск, отладку и воспроизведение.

К наиболее распространённым ошибкам синхронизации относятся взаимная блокировка потоков и состояние гонки.

В рамках данного исследования рассматривается анализ пользовательских приложений для платформы Android. Большинство приложений этой операционной системы используют графический пользовательский интерфейс и параллельные вычисления. Также приложения для Android разрабатываются на языке Java и других, которые могут быть транслированы в Java байт-код, и выполняются на виртуальной машине Dalvik.

4.1 Методы поиска ошибок синхронизации

Основными механизмами, которые используются для поиска ошибок синхронизации, являются построение и отслеживание множеств блокировок и построение модели на основе отношения предшествования между событиями.

Первый механизм предполагает сопоставление каждому отдельному событию в программе состояния, которое включает в себя информацию о правилах предоставления отдельным нитям эксклюзивного доступа к разделяемым данным, которые действуют в выполнении данного события. Описанные состояния обновляются при входе и выходе из критической секции. Недостатком этого метода является то, что он постулирует слишком строгое условие отсутствия ошибки синхронизации.

Второй механизм предполагает установление на всём множестве событий конкретного пути выполнения программы отношения предшествования по следующему правилу: одно событие называется предшествующим другому, если оно предшествует ему (возможно транзитивно) в рамках одного вычислительного потока, либо одно событие является отправкой, а другое — принятием одного и того же сообщения.

В рамках проекта предполагается использование гибридного метода, который сочетает в себе идеи обоих методов.

4.2 Существующие решения

На сегодняшний день существует множество средств, предоставляющих возможность поиска ошибок синхронизации в программах, использующих параллельные вычисления. Это инструменты Intel Thread Checker [8], Sun Thread Analyzer [9], а также инструменты, основанные на среде динамической инструментации Valgrind [10]: helgrind [11], DRD [12] и ThreadSanitizer [13].

Первые два приведённых инструмента реализуют метод, основанный на отношениях предшествования, и гибридный метод соответственно. Они, однако, не доступны для архитектур семейства ARM.

Инструменты, основанные на фреймворке Valgrind, поддерживают архитектуру ARM. Наиболее эффективным из них является инструмент ThreadSanitizer.

4.3 ThreadSanitizer

ThreadSanitizer представляет собой не отдельный инструмент, а целый набор инструментов, предназначенных для поиска ошибок взаимной блокировки и состояния гонки в программах и основанных на общих подходах для решения этой проблемы. Одним из его компонентов является ThreadSanitizer Offline, инструмент, способный осуществлять поиск ошибок в построенной модели выполнения программы, другим — Java ThreadSanitizer, осуществляющий построение такой модели для программ, которые были транслированы в Java байт-код.

Основой методов, реализованных в интрументах набора ThreadSanitizer, является рассмотрение участков программы, в которых отсутствуют синхронизационные событие, т. е. участков, результат выполнения которых не зависит от последовательности выполнения нитей. Под синхронизационными событиями здесь понимаются операции доступа к общей памяти, выставление или снятие блокировок, операции передачи и принятия сигналов, а также операции над потоками.

В основе алгоритма лежит построение модели выполнения программы с использованием отношения предшествования на множестве описанных участков программы. На основе построенной модели ThreadSanitizer Offline принимает решение о том, какие события, осуществляющие доступ к разделяемым данным (одно из которых их модифицирует), не связаны отношением предшествования. Это означает, что выполнение программы зависит от порядка выполнения нитей, что в большинстве случаев не предусмотрено программистом и означает наличие в программе дефекта. Формат модели достаточно универсален и подходит для описания выполнения программ, использующих параллельные вычисления, написанных на различных языках.

Инструмент Java ThreadSanitizer строит модель в описанном формате для программы, представленной в виде байт-кода языка Java, с использованием динамической инструментации, которая предоставляется библиотекой ASM [14]. Эта библиотека использует стандартный механизм виртуальной машины Java для проведения динамической инструментации, который отсутствует в виртуальной машине Dalvik операционной системы Android, что не позволяет использовать инструмент Java ThreadSanitizer на данной платформе.

4.4 Coffee Machine

Инструмент Coffee Machine, разработанный в ИСП РАН, производит статическую инструментацию байт-кода языка Java на основе библиотеки BCEL [15]. Это означает, что преобразование программы происходит до её выполнения, а не во время. Такой подход позволяет нужным образом модифицировать программу вне зависимости от того, каким образом она будет выполняться и какая виртуальная машина для этого будет применена.

В рамках проекта на базе инструмента Coffee Machine был реализован гибридный метод, совмещающий в себе отслеживание множеств блокировок и вычисление отношений предшествования между синхронизационными событиями, что в сочетании с использованием инструмента ThreadSanitizer Offline для проверки построенной модели позволило использовать его для поиска синхронизационных ошибок в программах платформы Android.

4.5 Результаты

Практические подтверждение применимости предложенного подхода проводились на ряде приложений операционной системы Android. Для автоматического построения сценариев работы использовалась модифицированная версия инструмента GUITAR. Результатом стало обнаружение некоторых реальных дефектов, связанных с ошибками в реализации синхронизации потоков. В качестве примера можно привести дефект, обнаруженный в приложении Email, которое входит в набор предустановленных программ операционной системы. Обнаруженная ситуация гонки при определённом порядке выполнения потоков приводит к возникновению исключению `NullPointerException` и аварийному завершению программы.

5. Заключение

В данной статье рассмотрены прикладные задачи, решение которых возможно при помощи динамического анализа программ. Описанные методы реализованы в виде экспериментальных прототипов инструментов для анализа приложений на языке Java и проверены на программах из поставки операционной системы для мобильных устройств Android.

Список литературы

[1]. Официальный анонс выпуска Android 3.0 [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html). Обращение от 01.12.2016

[2]. D. Bornstein. Dalvik VM internals (http://sites.google.com/site/io/dalvik-vm-internals) Обращение от 01.12.2016

[3]. David Detlefs, Christine Flood, Sete Heller, Tony Printezis. Garbage-first garbage collection. *ISMM'4 Proceedings of the 4th international symposium on Memoty management,* pp. 37-48. Vancouver, BC, Canada. October 24-25, 2004

[4]. Bao N. Nauyen, Bryan Robbins, Ishan Banerjee, Atif Memon. GUITAR: an innovative tool for automated testinf of GUI-driven software. *Automated Software Engineering, Volutme 21, Issue 1, March 2014*. pp. 65-105.

[5]. Willem Visser, Corina S. Pāsāreanu, Sarfranz Khurshid. Test input generation with java PathFinder. *ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. pp. 97-107. Boston, Massachusetts, USA. July 11-14, 2004.

[6]. Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, Vashwanath Raman. JDart: a dynamic symbolic analysis framework. *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Volume 9636.* pp. 442-459. Springer-Verlang New Yor, Inc, New York, NY, USA. April 02-08, 2016.

[7]. С.П. Вартанов, М.К. Ермаков. Применение статической инструментации байт-кода языка Java для динамического анализа программ. Труды ИСП РАН, том 27, выпуск 1, 2015 г., стр. 25-38. DOI: 10.15514/ISPRAS-2015-27(1)-2.

[8]. 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

[9]. Руководство по инструменту Sun Thread Analyzer [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html) Обращение от 01.12.2016

[10]. 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

[11]. Ali Jannesari, Walter F. Tichy, Victor Pankratius, Kaibin Bao. Helgrind+: an efficient dynamic race detector. *Parallel and Distributed Processing Symposium, International(2009)*. pp. Rome, Italy, May 23-29, 2009

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

[13]. Ali Jannesari, Markus Westpahl-Futuya, Walter F Tichy. Dynamic data race detection for correlated variables. *ICA3PP'11 Proceedings of the 11th international conference on Algorithms and architectures for parallel processing, Colume Part I.* pp. 14-26. Melbourne, Austratia. October 24-26, 2011

[14]. E. Bruneton, R. Lenglet, T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible systems*, November 2002. Grenoble, France.

[15]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel) Обращение от 01.12.2016

Applying dynamic analysis to programs running in interpreted environments

S.P. Vartanov <svartanov@ispras.ru>
M.K. Ermakov <mermakov@ispras.ru>
A.Y. Gerasimov <agerasimov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The present-day trends in software engineering include the steady increase of code and design complexity which reinforces the high demand in automated software testing and analysis tools. In this paper, we showcase several dynamic program analysis applications and present our solutions. These applications include memory profiling, automated test generation using dynamic symbolic execution and automated detection of concurrency bugs in multithreaded programs. Our memory profiling tool is designed for Java applications for Android and it is implemented through Android Dalvik VM modification. This approach allowed us to overcome existing Dalvik VM limitations that make existing profiling tools based on dynamic bytecode instrumentation inaccessible. We have successfully applied our tool to several core Android applications - the results provided in the paper outline the effectiveness of the approach. The second solution we discuss in the paper - dynamic symbolic execution for test generation automation - allows us to efficiently generate test scenarios for Java program graphical user interface. The core technologies of the approach include the use of static bytecode instrumentation and automatic GUI model extraction. We implement the approach on top of a user interface test automation framework GUITAR. Finally, we present our

approach to automatically identify concurrency bugs in multithreaded Java applications. The approach is based on static bytecode instrumentation for trace generation and employs ThreadSanitizer defect detection tool for identifying bugs.

Keywords: dynamic analysis; program analysis.

DOI: 10.15514/ISPRAS-2017-29(1)-9

For citation: Vartanov S.P., Ermakov M.K., Gerasimov A.Y. Applications of dynamic analysis of programs executed inside of interpreting environments. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 1, 2017. pp. 135-148 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-9

References

[1]. The official announce of the Android 3.0 release [HTML] (http://developer.android.com/about/versions/android-3.0-highlights.html). Accessed at 01.12.2016

[2]. D. Bornstein. Dalvik VM internals (http://sites.google.com/site/io/dalvik-vm-internals) Accessed at 01.12.2016

[3]. David Detlefs, Christine Flood, Sete Heller, Tony Printezis. Garbage-first garbage collection. *ISMM'4 Proceedings of the 4th international symposium on Memoty management.* pp. 37-48. Vancouver, BC, Canada. October 24-25, 2004

[4]. Bao N. Nauyen, Bryan Robbins, Ishan Banerjee, Atif Memon. GUITAR: an innovative tool for automated testinf of GUI-driven software. *Automated Software Engineering, Volutme 21, Issue 1, March 2014*. pp. 65-105.

[5]. Willem Visser, Corina S. Pāsāreanu, Sarfranz Khurshid. Test input generation with java PathFinder. *ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. pp. 97-107. Boston, Massachusetts, USA. July 11-14, 2004.

[6]. Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, Vashwanath Raman. JDart: a dynamic symbolic analysis framework. *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Volume 9636.* pp. 442-459. Springer-Verlang New Yor, Inc, New York, NY, USA. April 02-08, 2016.

[7]. S. P. Vartanov, M. K. Ermakov. Applying Java bytecode static instrumentation for software dynamic analysis. Trudy ISP RAN / Proc. ISP RAS, volume 27, issue 1, 2015, pp. 25-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-2.

[8]. 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

[9]. The manual for Sun Thread Analyzer [HTML] (http://docs.oracle.com/cd/E19205-01/820-4155/tha.html) Accessed at 01.12.2016

[10]. 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

[11]. Ali Jannesari, Walter F. Tichy, Victor Pankratius, Kaibin Bao. Helgrind+: an efficient dynamic race detector. *Parallel and Distributed Processing Symposium, International (2009)*. pp. 1-13. Rome, Italy, May 23-29, 2009

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

[13]. Ali Jannesari, Markus Westpahl-Futuya, Walter F Tichy. Dynamic data race detection for correlated variables. *ICA3PP'11 Proceedings of the 11th international conference on Algorithms and architectures for parallel processing, Colume Part I.* pp. 14-26. Melbourne, Austratia. October 24-26, 2011

[14]. E. Bruneton, R. Lenglet, T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible systems*, November 2002. Grenoble, France.

[15]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel) Accessed at 01.12.2016