Применение статической инструментации байт-кода языка Java для динамического анализа программ

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

Аннотация. В статье рассматривается задача проведения динамического анализа программ на языке Java при условии, что исходный код программы отсутствует, а запуск программы может происходить на виртуальных машинах, которые интерпретируют байт-код формата, отличного от формата Java Virtual Machine. Приводится обзор методов инструментации и особенностей инструментации байт-кода языка Java для проведения итеративного динамического анализа с целью покрытия наибольшего числа путей выполнения программы. Для такого рода анализа используется построение входных данных для покрытия ранее не пройденных базовых блоков при помощи отслеживания потока помеченных данных, построения ограничений пути и проверки их выполнимости. В качестве метода решения поставленной задачи рассматривается применение статической инструментации байт-кода. Основными достоинствами подобного подхода являются увеличение скорости анализа (за счёт того, что инструментация проводится один раз, до начала работы итеративного механизма) и возможность конвертировать инструментированный байт-код в другие форматы для запуска на нестандартных виртуальных машинах (например, DEX для виртуальной машины Dalvik). В статье также рассматривается реализация предложенных методов в инструменте Coffee Machine. Инструментация осуществляется с помощью BCEL (библиотеки для манипулирования байт-кодом) и разделяется на три этапа: определение классов и методов для инструментации, инструментация на уровне классов и методов, инструментация на уровне отдельных инструкций. На основе Coffee Machine показано, как статическая инструментация может быть применена для печати информации о выполнившихся инструкциях, отслеживания помеченных данных, построения ограничений пути выполнения, а также для построения трассы синхронизационных событий. В качестве одного из ограничений предложенного подхода рассматривается невозможность доступа к динамическим данным в ходе выполнения программы и некоторым методам системных классов. Эти ограничения могут быть сняты за счёт увеличения накладных расходов на повторную инструментацию анализируемой программы и написание методов, симулирующих работу требуемых системных методов. Для использования методов-симуляторов используется специальный механизм, который сопоставляет их имена и имена реальных методов в процессе работы программы и производит дублирование вершины стека для передачи фактических параметров методу-симулятору.

Ключевые слова: статическая инструментация, динамический анализ, анализ программ.
DOI: 10.15514/ISPRAS-2015-27(1)-2
Для цитирования: Вартанов С.П., Ермаков М.К.. Применение статической инструментации байт-кода языка Java для динамического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 25-38. DOI: 10.15514/ISPRAS-2015-27(1)-2.

1. Введение

Поиск ошибок в программном обеспечении — неотъемлемая часть процесса разработки программ, ввода их в эксплуатацию и дальнейшей поддержки. Причём стоимость обнаружения и исправления ошибок постоянно увеличивается с течением времени в жизненном цикле программного обеспечения.

Методы ручного или полуавтоматического тестирования программ требуют от программиста, во-первых, больших временных затрат, во-вторых, достаточно высокого уровня понимания принципов и деталей работы конкретной программы.

Для полностью автоматического анализа программ с целью поиска ошибок используются методы статического и динамического анализа. Динамический анализ известен как более тяжеловесный подход и требует значительных временных и вычислительных ресурсов. Основное достоинство динамического анализа по сравнению со статическим — отсутствие ложных срабатываний при определённых условиях, накладываемых на исходную программу (таких как детерминированность). Проведение динамического анализа требует внесения определённых изменений в ход выполнения программы. Это может быть осуществлено за счёт использования инструментации кода программы или преобразования среды выполнения программы в зависимости от требований, предъявляемых к анализу.

В статье рассматривается применение статической инструментации программ для проведения динамического анализа с целью поиска ошибок и уязвимостей в программах на языке Java, либо в программах, транслируемых в Java байт-код. Реализация описанного метода будет рассмотрена на примере инструмента Coffee Machine, разрабатываемого в ИСП РАН.[1]

2. Динамический анализ программ на языке Java с целью поиска ошибок и уязвимостей

Под динамическим анализом программ понимается процесс, требующий запуска анализируемой программы на исполнение. В противоположность ему, статические методы анализа основываются на различных моделях,

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

построенных на базе кода анализируемой программы. Динамический анализ программ имеет ряд достоинств и недостатков по сравнению со статическими методами.

К достоинствам динамического анализа можно отнести отсутствие ложных срабатываний при определённых ограничениях на анализируемую часть программы: отсутствие элементов, вносящих недетерминированность в ход выполнения, отсутствие элементов взаимодействия с пользователем или детерминированное описание этого взаимодействия в виде последовательности событий. Также достоинством динамического анализа можно назвать возможность доступа к фактическим данным, значения которых известны лишь на этапе выполнения программы.

К недостаткам динамического анализа относятся значительные временные затраты на его проведение по сравнению со статическим анализом. В случае, если целью анализа является обход всех возможных путей выполнения программы или достижение максимального покрытия кода программы по одному из критериев, возникает проблема экспоненциального роста числа путей анализируемой программы.

В рамках задачи поиска ошибок в приложениях на языке Java было принято решение применить методы полностью автоматического динамического анализ, поскольку эти методы наиболее эффективны для поиска редко проявляющихся критических ошибок, которые приводят к аварийному завершению программы.

2.1 Требования к решению

В связи с особенностями анализируемых программ к решению предъявлялся ряд требований:

- возможность проведения анализа на виртуальных машинах, интерпретирующих байт-код формата, отличного от де-факто принятого за стандарт формата Java Virtual Machine;
- также для анализа может быть недоступен исходный код программы — анализируемая программа может представлять собой набор класс-файлов или файлов типа JAR.

Решение основано на принципах, заложенных в инструменте динамического анализа Avalanche [1]. В цели инструмента входит обеспечение как можно более полного покрытия базовых блоков исходной программы и поиск ошибок на обнаруженных путях выполнения. Для этих целей используется инвертирование условных переходов. После каждого инвертирования условия в условном переходе при помощи решателя булевых формул STP осуществляется проверка выполнимости ограничений пути. В случае выполнимости строится новый набор входных данных, который в силу детерминированности анализируемой программы определяет ранее не пройденный путь выполнения.

Сбор информации о ходе выполнения программы, который необходим для построения ограничений пути, осуществляется при помощи механизма динамической инструментации, предоставляемого фреймворком Valgrind.

Применение инструмента Avalanche нецелесообразно при анализе Java программ в связи с тем, что анализ проводится на уровне машинного кода процессора компьютера, на котором выполняется виртуальная машина Java, и основные вычислительные мощности в основном будут затрачены на анализ работы самой виртуальной машины. К тому же на уровне инструментации, применяемом в инструменте Valgrind, нет непосредственного доступа к высокоуровневым структурам языка Java, что также неоправданно усложняет анализ Java программ, например обнаружение необработанных исключительных ситуаций в Java программе.

В связи с этим было принято решение о создании нового инструмента предназначенного для непосредственного анализа байт-кода Java. Принципы, на которых построен инструмент, подробно описаны в статьях [2, 3]. Данная статья посвящена описанию механизма статической инструментации байт-кода Java программы и описанию ограничений данного подхода, обнаруженных в процессе разработки инструмента.

3. Инструментация программ

Под инструментацией понимается процесс внедрения дополнительных действий в ход выполнения программы таким образом, чтобы исходная функциональность не была затронута или влияние на неё было бы минимальным.

Инструментация может применяться для извлечения различной информации о ходе выполнения программы в рамках задачи профилирования — определение времени событий (начала и завершения работы подпрограмм), длительности работы отдельных инструкций или фрагментов, время, затраченное на обмен информацией между различными компонентами, извлечение информации о состоянии памяти и т. д. Также среди извлекаемой информации могут быть данные, необходимые для оптимизации работы систем, управляющих выполнением программ, — точность предсказаний условных переходов, частота возникновения ситуаций, когда запрашиваемые данные отсутствуют в кэше, эффективность систем автоматической сборки мусора, а также для обнаружения дефектов.

Анализ извлекаемой информации может происходить как непосредственно в ходе выполнения (on-the-fly analysis), так и после завершения работы (postmortem analysis). Во втором случае извлекаемая информация записывается в так называемую трассу событий. Построенная трасса в дальнейшем может использоваться без изменений или преобразовываться для построения более абстрактных моделей работы программы и их

последующего анализа (например, поиск ошибочных ситуаций, допускаемых моделью, либо проверка модели на соответствие спецификации).

В качестве способа построения новых путей выполнения программы выбран метод инвертирования условных переходов, который включает в себя отслеживание потока помеченных данных для определения возможности инвертирования отдельного перехода. Помеченными называются входные данные, а также данные, которые от них зависят. Для инвертирования условного перехода необходимо, чтобы в его условии присутствовали помеченные данные. В ином случае не существует возможности в рамках принятых ограничений повлиять на выполнение программы таким образом, чтобы оно пошло по альтернативному пути. Отслеживание помеченных данных требует хранения информации о том, является ли отдельный блок памяти, используемый в программе, помеченным, а также сбора информации о том, как помеченные блоки связаны между собой.

Как и методы анализа программ, методы инструментации подразделяют на статические и динамические в зависимости от того, на каком этапе относительно непосредственного процесса выполнения программы происходит процесс инструментации.

3.1 Динамическая инструментация

Динамическая инструментация происходит в процессе выполнения программы.

В качестве примера среды динамической инструментации бинарного кода можно привести фреймворк Valgrind [4]. Принцип работы инструментации Valgrind заключается в разбиении исходного бинарного кода на блоки нефиксированной длины по заранее определённым правилам (не содержащие условных переходов, содержащие ограниченное число безусловных переходов и т. д.). Полученные блоки переводятся во внутреннее представление, инструментируются, конвертируются обратно в машинный код и запускаются на исполнение.

Для увеличения эффективности динамической инструментации часто используется кэширование инструментированных блоков таким образом, что при запросе на выполнение блока кода, вместо его инструментации будет использован ранее сохранённый блок, если он был проинструментирован ранее.

3.2 Статическая инструментация

В отличие от динамической, статическая инструментация полностью завершается до момента начала выполнения программы (однократного или многократного).

Основное достоинство статической инструментации состоит в том, что повторно исполняемые блоки программы не требуют кэширования или

переинструментации. Также в случае статической инструментации, дополнительные действия, помимо самого инструментационного кода, минимальны или отсутствуют вовсе.

Выбор в пользу одного из представленных способов инструментации осуществляется в зависимости от того, с какой целью производится преобразование программы. Статическая инструментация эффективней, если основное требование к инструментации — минимальное снижение производительности инструментируемой программы. Если производится многократное выполнение программы на путях, имеющих общие фрагменты кода, статическая инструментация также может быть эффективней за счёт отсутствия затрат на кэширование и возможных промахов кэша. Динамическая инструментация эффективней, если запуск программы производится один раз и заранее неизвестно, какая часть программы будет выполнена.

3.3 Инструментация байт-кода языка Java

Программы, написанные на Java, обычно транслируются в специальный байт-код, исполняемый виртуальной машиной. Для инструментации программ в языке Java, начиная с версии 1.5.0, существует специальный пакет `java.lang.instrument`.

Для инструментации байт-кода существует множество различных инструментов и библиотек. ASM [5] — одна из популярных библиотек для динамической инструментации байт-кода, использующих интерфейс `java.lang.instrument`. К сожалению, `java.lang.instrument` поддерживается не всеми виртуальными машинами. В частности, опция `-javaagent` отсутствует у виртуальной машины Dalvik платформы Android, что делает невозможным использование средств инструментации, подобных ASM.

Для статической инструментации байт-кода существует другая популярная библиотека — BCEL (bytecode engineering library) [6]. Она позволяет осуществлять генерацию и модификацию байт-кода на уровне отдельных инструкций и отдельных записей таблиц class-файлов.

4. Реализация инструментации в инструменте Coffee Machine

Инструментация байт-кода в инструменте Coffee Machine осуществляется на трёх основных уровнях:

- Предварительная подготовка классов и методов.
- Инструментация классов и методов.
- Инструментация инструкций.

Предварительная подготовка заключается в создании списка классов и методов, которые необходимо проинструментировать. Наиболее простое

решение этой проблемы заключается в инструментации всех методов всех классов, которые доступны в ходе выполнения программы. Для того, чтобы класс мог быть использован Java-машиной, он должен находится в файле с расширением `.jar` или `.class`, который располагается в одном из перечисленных мест:

- в системной директории Java-машины (где располагаются такие системные библиотеки как `rt.jar`);
- в специальной папке, предназначенной для расширений;
- в директории из списка `classpath`, который может быть передан Java-машине при помощи опции или может быть прочитан из переменной окружения.

В общем случае инструментация всех методов во всех возможных классах избыточна. Статически построенный список методов может, с одной стороны, содержать лишние методы (которые не будут вызваны), с другой стороны, может быть неполным. Первое происходит из-за того, что в большинстве случаев путь, по которому будет проходить выполнение, заранее не известен, а задача определения недостижимости кода является алгоритмически неразрешимой в общем случае. Второе происходит по причине того, что метод класса может быть вызван динамически.

Если число всех доступных методов сильно превышает число методов, действительно используемых в ходе выполнения программы, инструментация всех доступных методов приводит к чрезмерным накладным расходам. Данная проблема может быть решена использованием статического определения списка методов с рядом дополнений, реализованных в инструменте.

Если пользователю анализатора заранее известен список методов, которые с большой долей вероятности будут вызваны динамически, инструмент предоставляет возможность указать их для инструментации. На следующем шаге в этот список добавляются методы, вызовы которых явно присутствуют в байт-коде программы. Далее этот список может быть расширен целиком классами, в которых присутствуют эти методы, исходя из предположения, что если из класса динамически вызывается метод, с большой долей вероятности может быть вызван и другой метод того же класса.

Если в ходе выполнения программы обнаруживается метод, который не был проинструментирован ранее, выполнение останавливается и происходит инструментация пропущенного метода, а также других на основе информации о классе, которому он принадлежит. После этого программа перезапускается на тех же входных данных для продолжения анализа.

4.1 Инструментация классов и методов

После определения списка необходимых методов, управление передаётся компоненту, который разделяет методы на отдельные инструкции. Для этих целей используется BCEL.

4.2 Инструментация инструкций

Компонент, производящий инструментацию отдельных инструкций, имеет модульную структуру. Основной модуль представляет собой класс с набором абстрактных методов, соответствующих каждому типу инструкции байт-кода. Каждый конкретный модуль инструментации является наследником основного с необходимым образом реализованными методами инструментации. Для каждого типа инструкции переопределяются два метода: первый создаёт список инструкций, которые необходимо вставить до инструментируемой инструкции, второй — список инструкций, которые нужно вставить после.

В инструменте реализованы три модуля, производящих инструментацию отдельной инструкции:

- печать трассы инструкций,
- печать трассы условий на помеченные данные,
- печать трассы синхронизационных событий.

4.2.1 Печать трассы инструкций

Первый из модулей инструментации тривиален и демонстрирует возможность использования инструментации для снятия трасс работы анализируемой программы. Его работа заключается в печати информации о выполнившейся инструкции.

Для большинства инструкций это достигается путём вставки следующего набора инструкций после её выполнения:

- `getstatic` – получение потока вывода,
- `push` – запись выводимой строки в стек,
- `invokevirtual` – вызов функции `java.lang.String.println`.

Таким образом, вместо инструкции `new` для класса `java.io.FileReader` в байт-коде появятся четыре следующие команды (список команд приведён в формате библиотеки BCEL, в скобках указаны номера записей в константном пуле):

```
0: getstatic      java.lang.System.out Ljava/io/PrintStream; (51)
3: ldc            "Main.main 0 187 java/io/FileReader" (67)
5: invokevirtual  java.io.PrintStream.println (Ljava/lang/String;)V
(56)
8: new            java.io.FileReader (2)
```

Листинг 1. Инструкции байт-кода, в котрые будет преобразована инструкция `new` для класса `java.io.FileReader`

Исключением являются инструкции перехода, такие как `ifeq` (условный переход в случае равенства значений аргументов) или `return` (возврат из

метода), поскольку инструкции, помещаемые после них, не выполняются или могут не выполняться в случае условных переходов.

Решение данной задачи заключается в том, чтобы помещать печать дополнительных инструкций до инструкций перехода и изменять адреса переходов на эти инструкции на адрес первой дополнительной инструкции.

4.2.2 Печать трассы условий на помеченные данные

Одна из функций инструмента Coffee Machine — поиск ошибок в программах на языке Java с помощью итеративной генерации наборов входных данных, каждый из которых увеличивает покрытие кода программы, и запуска программы на этих входных данных.

Первый запуск анализируемой программы производится на введённых пользователем или случайным образом сгенерированных входных данных. Построение новых путей выполнения программы происходит при помощи инвертирования условий в условных переходах в уже известных путях выполнения. Для того, чтобы проверить в каждом конкретном случае, является ли инвертирование корректным, т. е. приводит ли оно к действительно существующему новому пути выполнения, в ходе выполнения программы производится отслеживание помеченных данных.

Для этих целей служит специальный инструментационный класс, методы которого вызываются в ходе выполнения программы. Этот класс также содержит структуры данных, соответствующие стеку, регистрам и памяти Java-машины, на которой происходит выполнение программы. Ячейки каждой структуры данных соответствуют реальным переменным Java-машины и хранят в себе ссылки на метапеременные. В терминах метапеременных описываются связи между реальными переменными в ходе выполнения программы. Эти связи записываются в виде булевых ограничений на языке запросов инструмента STP [7].

Для печати ограничений переопределены методы модуля инструментации для большинства типов инструкций байт-кода. В большинстве случаев метод содержит в себе вставку в байт-код инструкций, дублирующих необходимые параметры стека, помещающие в стек определённые параметры, и вставку вызова соответствующего метода инструментационного класса. Параметры инструментационных методов должны в точности соответствовать дублированным и добавленным значениям, чтобы не повлиять на результат работы инструкций, которые будут выполнены после.

4.2.3 Печать трассы синхронизационных событий

Другая функция инструмента Coffee Machine – поиск состояний гонки при помощи инструмента ThreadSanitizer Offline [8].

Инструментация в этом случае производится для ограниченного набора инструкций — работы с памятью, полями, исключениями, вызовов методов и специальных инструкций `monitorenter` и `monitorexit`. Каждая инструкция указанного типа инструментируется с целью печати информации в трассу синхронизационных событий в формате инструмента ThreadSanitizer Offline.

Пример формата трассы приведён ниже. В него входят информация о начале и завершении потоков и методов, установка и снятие блокировок, чтение и запись объектов.

```
...
RTN_EXIT 2 22c 0 0
SBLOCK_ENTER 2 937 0 0
WRITER_LOCK 2 938 2a1fc2b466 0
READ 2 939 1abbb49a1967433 1
WRITE 2 939 1abbb49a1967433 1
UNLOCK 2 93a 2a1fc2b466 0
RTN_EXIT 2 93b 0 0
SBLOCK_ENTER 2 475 0 0
RTN_EXIT 2 476 0 0
THR_END 2 383 0 0
THR_JOIN_AFTER 0 383 2 0
...
```

Листинг 2. Пример трассы синхронизационных событий

Рассмотрим теперь подробнее инструментацию вызовов методов.

4.2.4 Инструментация методов

Особые действия выполняются при инструментации инструкции вызова метода. Всего в байт-коде языка Java для вызова метода используются пять инструкций:

- `invokevirtual` (код $B6_{16}$),
- `invokespecial` (код $B7_{16}$),
- `invokestatic` (код $B8_{16}$),
- `invokeinterface` (код $B9_{16}$),
- `invokedynamic` (код BA_{16}).

В параметры перечисленных инструкций входят ссылка на вызываемый метод, а также (за исключением инструкций `invokestatic` и `invokedynamic`) ссылка на объект, метод которого вызывается. При возможности, на этапе инструментации извлекается имя и сигнатура вызываемого метода. Если реализация метода отсутствует в теле класса, соответствующий класс ищется среди классов-предков.

При использовании статической инструментации нет возможности проинструментировать ряд не нативных системных методов. Это происходит

в связи с возникающими зависимостями, которые не удаётся разрешить на этапе инициализации виртуальной машины. Такие методы, как и нативные, требуют симуляции, если принимают на вход помеченные данные.

Всего различаются три типа методов.

- Не нативные методы, к байт-коду которых у инструмента имеется доступ. Эти методы добавляются в список инструментации, а значит либо были проинструментированы ранее, либо будут проинструментированы. Инструментационный код до или после инструкции вызова в этом случае не добавляется.

Методы, которые либо не имеют байт-кода (нативные методы), либо это системные методы, которые невозможно проинструментировать статически. Среди них можно выделить два типа методов:

- Методы, которые принимают в качестве параметров или генерируют помеченные данные. В этом случае метод либо будет проигнорирован и помеченные данные будут потеряны, либо в качестве инструментационного кода будет использован вызов метода, симулирующего его функциональность.
- Методы, среди параметров которых нет помеченных данных, и о которых известно, что они не генерируют помеченные данные. В этом случае вызов метода игнорируется.

Вызовы методов первого и третьего типа игнорируются. Для инструкции вызова метода второго типа используется следующий способ инструментации, аналогичный методу использования методов-симуляторов инструмента Java ThreadSanitizer [8].

Каждый метод, реализующий симуляцию функциональности исходного метода имеет имя вида

```
[a|b]_<имя класса>_<имя исходного метода>
```

и располагается в классе с именем, соответствующим имени пакета, в котором располагается исходный класс. Первая буква в имени метода-симулятора означает позицию, в которую будет добавлен инструментационный код относительно инструкции вызова метода: `a` (after) — после инструкции вызова, `b` (before) – до инструкции вызова. Сигнатура метода-симулятора должна повторять сигнатуру исходного метода с дополнительным первым параметром, в котором содержится ссылка на объект, метод которого вызывается (если метод не статический), а также с дополнительным последним параметром, в котором методу передаётся его возвращаемое значение (если происходит вставка вызова после инструкции и если возвращаемое значение имеет тип, отличный от `void`).

Более детально описание процесса инструментации для случаев добавления вызова метода-симулятора до, после и одновременно до и после инструкции вызова исходного метода, приведено на рис. 1.

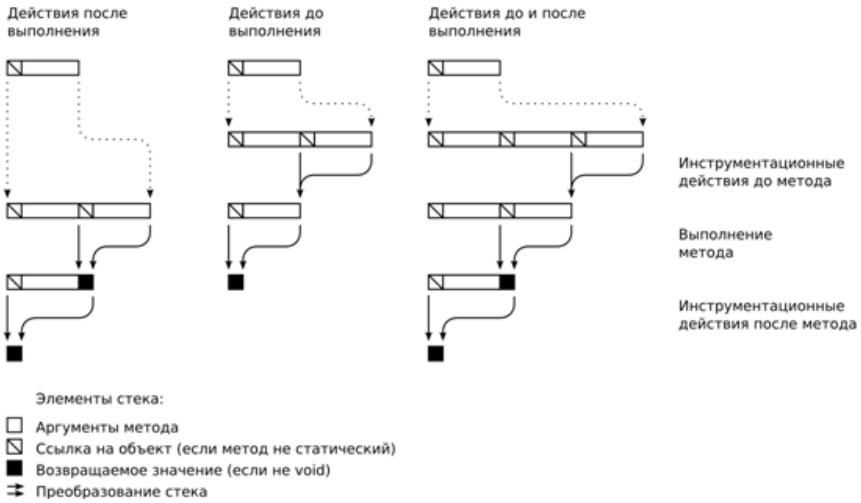

Рис. 1. Инструментация инструкции вызова метода

Например, метод-симулятор для метода `char java.lang.String.charAt(int)`, вызов которого должен быть вставлен после вызова исходного метода, будет иметь следующие имя и сигнатуру: `char a_String_charAt(String, int, char)`. Последней инструкцией этого метода должен быть возврат последнего параметра.

5. Заключение

Различные методы инструментации применяются в зависимости от особенностей задач, которые решаются при помощи модификации исходной программы. Статическая инструментация оправдывает затраты на повторную инструментацию кода, который не будет вызван в ходе выполнения программы в том случае, если проинструментированная один раз программа используется для запуска достаточное число раз. В случае же, когда целью анализа программы является проход по всевозможным путям выполнения с целью увеличения покрытия кода, понятие кода, который не вызывается в ходе всех произведённых вызовов стремится к понятию недостижимого кода.

Статическая инструментация имеет ряд ограничений, связанных с невозможностью доступа к динамическим данным в ходе выполнения программы. Эти ограничения влияют на возможности определения списка методов, которые будут вызваны. Проблема решается при помощи

итеративной повторной инструментации исходной программы. К тому же статически невозможно проинструментировать некоторые системные классы, которые используются в ходе инициализации Java-машины. Эта проблема, как и проблема инструментации нативных методов, решается при помощи написания методов-симуляторов. Также эту проблему можно решить при помощи методов изменения среды выполнения, чему будет посвящено дальнейшая работа в рамках исследования.

Список литературы

[1]. И. К. Исаев, Д. В. Сидоров. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, 2010, № 4, с. 1–16

[2]. С. П. Вартанов, А. Ю. Герасимов. Применение динамического анализа для поиска дефектов в программах на языке Java. Стр. 9–28. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[3]. С. П. Вартанов, А. Ю. Герасимов. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. Стр. 375–394. Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[4]. N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

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

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

[7]. V. Ganesh, D. L. Dill, A Decision Procedure for Bit-Vectors and Arrays // In Proceeding of Computer Aided Verification. 2007. P. 524–536.

[8]. K. Serebryany, T. Iskhodzhanov. ThreadSanitizer-data race detection in practice

Applying Java bytecode static instrumentation for software dynamic analysis

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

Abstract. This paper focuses on dynamic analysis of Java programs. We consider the following limitations: analysis tool may not have access to target program source code, and the program may be interpreted by a non-standard virtual machine with bytecode format different from Java Virtual Machine specifications. The paper describes an approach to bytecode instrumentation which is used to perform iterative dynamic analysis for new

execution path discovery. Path discovery is performed through automatic input data generation by tracing tainted data, collecting path conditions, and satisfiability checking. The proposed approach is based on static bytecode instrumentation. The main advantages of this approach are analysis speedup (because of one-time instrumentation) and explicit access to statically generated instrumented bytecode which makes it possible to run instrumented program on different virtual machines with different bytecode formats. Proposed approaches were implemented in the Coffee Machine tool. Paper sections dedicated to this tool provide a detailed description of taint data tracing and automatic branch traversing techniques as well as a set of instrumentation utilities based on Coffee Machine allowing executed instructions printing, taint trace dumping, and synchronization events trace generation. Coffee Machine uses BCEL (bytecode instrumentation library) for instrumentation. The paper concludes with an overview of practical restrictions existing for discussed methods and possible future work directions. Main disadvantage of proposed approach is the inability to access dynamic data at run-time and instrument a set of system class methods. It may be resolved by method simulation and execution environment modifications.

Keywords: static instrumentation, dynamic analysis, program analysis.

DOI: 10.15514/ISPRAS-2015-27(1)-2

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

References

[1]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1–16. (in Russian)

[2]. S. Vartanov, A. Gerasimov. Primenenie dinamicheskogo analiza dlyz poiska defektov v programmakh na yazyke Java. [Applying dynamic analysis for defect detection in Java-applications]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 25, 2013. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), P. 9–28. (in Russian)

[3]. S. Vartanov, A. Gerasimov. Dinamicheskiy analiz programm c tselyu poiska oshibok i uyazvimostey pri pomoshchi tselenapravlennoy generatsii vkhodnykh dannykh [Dynamic program analysis for error detection using goal-seeking input data generation]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 26, issue 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), P. 375–394. (in Russian)

[4]. N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

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

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

[7]. V. Ganesh, D. L. Dill, A Decision Procedure for Bit-Vectors and Arrays // In Proceeding of Computer Aided Verification. 2007. P. 524–536.

[8]. K. Serebryany, T. Iskhodzhanov. ThreadSanitizer-data race detection in practice