Анализ программ на языке Java в инструменте Svace

[1] *А.П. Меркулов <steelart@ispras.ru>*
[1] *С.А. Поляков <inly@ispras.ru>*
[1,2] *А.А. Белеванцев <abel@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1.

Аннотация. В статье описываются работы, выполненные для поддержки анализа программ на языке Java в статическом анализаторе Svace, разрабатываемом в ИСП РАН. Приводятся методы построения внутреннего представления для анализа Java, включая изменения в компоненте контролируемой сборки, доработки компилятора OpenJDK, трансляцию байткода Java в окончательное представление для анализа. Описываются особенности анализа Java-программ – алгоритм девиртуализации, спецификации методов стандартной библиотеки Java, некоторые специфичные детекторы. Представлены результаты выполнения анализа для исходного кода операционной системы Android 5.

Ключевые слова: статический анализ; Java; девиртуализация; байткод.

DOI: 10.15514/ISPRAS-2017-29(3)-5

Для цитирования: А.П. Меркулов, С.А. Поляков, А.А. Белеванцев. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74. DOI: 10.15514/ISPRAS-2017-29(3)-5

1. Введение

Высокая сложность программ делает практически невозможным создание программного продукта без дефектов. Причём с увеличением размера программного обеспечения возрастает не только количество дефектов, но и их плотность. Поэтому растёт необходимость в инструментах и методах поиска дефектов. Одним из таких методов является статический анализ программ, который осуществляется без их реального выполнения. При этом происходит исследование всего кода программы, в том числе редко достигаемых участков кода, что позволяет найти ошибки, которые сложно воспроизвести, и которые обычно остаются незамеченными в ходе тестирования.

В рамках проекта Svace в Институте системного программирования РАН ведутся работы по реализации статического анализа кода с целью поиска дефектов и ошибок в программах. Изначально инструмент был реализован для анализа кода на языках C и C++. В данной статье описана реализация поддержки анализа программ на языке Java и связанные с этим особенности и проблемы. Описание устройства инструмента Svace можно найти в статьях [1-2]. Кратко поясним основные этапы работы Svace для ясности дальнейшего изложения.

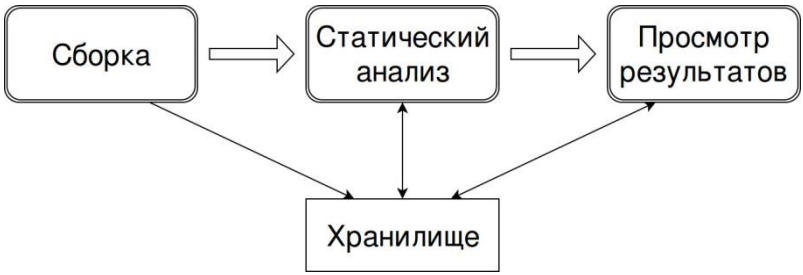


Рис. 1. Ход работы анализатора Svace
Fig. 1. Svace analyzer workflow

Процесс анализа программы Svace можно разбить на 3 этапа, работающие с одним хранилищем (см. рис. 1): контролируемая сборка проекта, статический анализ проекта, исследование результатов анализа пользователем в web-интерфейсе. Сначала Svace выполняет мониторинг оригинальной командой сборки интересуемого проекта. В качестве результата выполнения этой сборки генерируется контейнер с данными для анализа, которые включают в себя скомпилированное промежуточное представление, исходные коды, данные компоновки и так далее. На втором этапе запускается статический анализ для собранного контейнера с данными, его результатом становится контейнер с набором выданных предупреждений. Этот контейнер отправляется в хранилище результатов, которое отлеживает историю предупреждений. На третьем этапе пользователь открывает web-интерфейс и просматривает найденные предупреждения. Благодаря наличию истории становится возможным показывать, к примеру, только новые предупреждения или только пропавшие предупреждения.

Для добавления поддержки языка программирования Java было необходимо модифицировать этап сборки проекта и этап статического анализа проекта. Решение задачи добавления языка Java можно разбить на следующие подзадачи: модификации системы контролируемой сборки, трансляции байт-кода Java [3] во внутреннее представление Svace (раздел 2), модификации компилятора javac [4] (раздел 3), составления спецификаций стандартной библиотеки Java, девиртуализации вызовов (раздел 4). Девиртуализация была реализована перед этапом построения графа вызовов и является новой фазой

по сравнению с анализом кода на языках C и C++. В разделе 5 представлены результаты анализа исходного кода ОС Android 5, а в разделе 6 – заключение.

2. Перехват компиляции Java-программ

Для анализа проекта с помощью Svace необходимо выполнить контролируемую анализатором сборку проекта для компиляции исходных файлов проекта с помощью специально модифицированного компилятора Svace и с включёнными отладочными опциями. Общая схема этапа сборки приведена на рисунке 2. Процесс перехвата запускает оригинальную команду сборки, инструментируя её таким образом, чтобы перехватывать все запускаемые процессы и выделять интересующие нас команды сборки. Важно не влиять при этом на исходную сборку, чтобы ее результаты совпали с выполнением сборки без мониторинга.

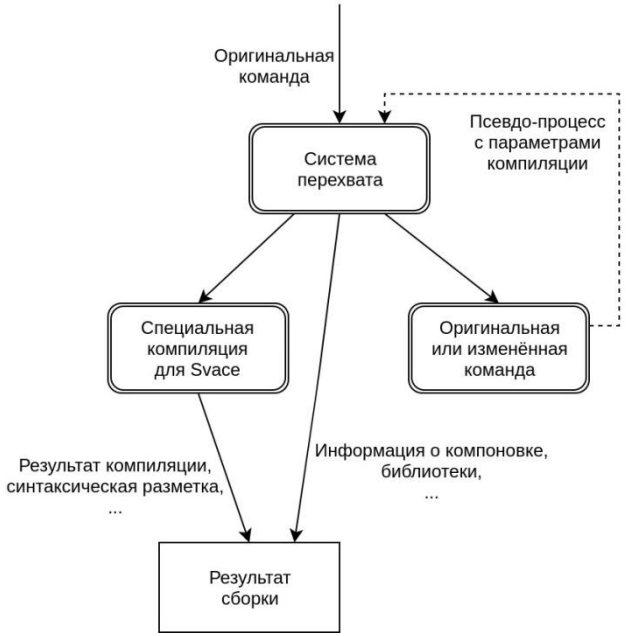


Рис. 2. Контролируемая сборка программы
Fig. 2. Monitored program build

В тот момент, когда в процессе сборки обнаруживается, что была запущена команда компиляции, дополнительно запускается модифицированный компилятор, который собирает код так, чтобы его было удобно впоследствии анализировать. Эта компиляция производится с выключенными оптимизациями и с включённой отладочной информацией. Сам компилятор

при этом модифицируется таким образом, чтобы максимально упростить генерируемый код, увеличить объём отладочной информации, а так же сделать специальные пометки на автогенерированный код. Кроме этого, модифицированный компилятор сохраняет дополнительную информацию о программе, в частности, генерирует файлы с синтаксической разметкой исходного кода, которые понадобятся при показе предупреждений пользователю.

Для компиляции кода на языке Java было решено проводить отладочную сборку с помощью модифицированного компилятора javac из пакета OpenJDK [4], так как OpenJDK является де-факто стандартным компилятором Java. Другим возможным выбором является компилятор ECJ среды Eclipse [9], однако его использование в контексте анализа в Svace не дает особых преимуществ – возможность разбора исходного кода с ошибками, важная для интегрированной среды разработки, на данный момент несущественна для Svace. В результате отладочной компиляции получаются файлы с байткодом Java (class-файлы), карта соответствия файлов промежуточного представления исходным файлам, синтаксическая разметка исходных файлов.

Дополнительной сложностью перехвата компиляций Java является возможность программно вызывать компилятор через Java Compiler API [5] без запуска нового процесса. Этой возможностью активно пользуются среды сборки Java-приложений, в частности, Ant, Maven, Gradle [6,7]. Задача перехвата в данном случае решается с использованием Java-агента [10]. При запуске любого Java-приложения можно указать стандартизированным образом библиотеку, через которую будет фильтроваться весь байткод запускаемого приложения. При этом Java-агент имеет возможность модифицировать этот байткод и, таким образом, запускаемое приложение.

Нами был создан Java-агент, который ищет в байткоде запускаемой программы вызовы компилятора (класса com.sun.tools.javac.main.Main) и инструментирует их так, чтобы дополнительно к компиляции вызывался специальный метод в java-агенте, в который передаются параметры компиляции. Метод формирует запуск специального псевдопроцесса, через параметры которого передаются параметры компиляции. Запуск этого псевдопроцесса перехватывается, как и все остальные процессы, после чего извлекаются параметры компиляции, и по ним формируется команда отладочной компиляции. При этом псевдопроцесс даже не запускается. На рисунке 2 пунктирной стрелкой показан вызов такого процесса. Таким образом, в реализованном решении перехваченные из оригинальной сборки запуски Java-программ модифицируются перед реальным запуском путём указания библиотеки собственного Java-агента.

3. Модификация компилятора javac

В качестве базового компилятора Java был выбран компилятор javac из пакета OpenJDK для Java 8. Оригинальный компилятор пришлось модифицировать,

так как необходимо было решить ряд проблем. Часть информации об исходной программе терялась или искажалась в процессе компиляции. В частности, возникали случаи дублирования байткода, основанного на одном и том же исходном коде, например, в процессе трансляции finally-блоков, а так же трансляции разделов инициализации полей класса в случае, если они инициализируются прямо в строке объявления, такие инициализации дублируются на каждый конструктор.

В случае трансляции finally-блоков необходимость в дублировании байткода возникает из-за того, что необходимо выполнить finally-блок сразу на нескольких путях: на успешном пути выполнения try-блока, на пути пойманного исключения в catch-блоке и на пути непойманного исключения, которое продолжит свой путь выше по стеку вызовов. Во всех этих случаях требуется выполнить один и тот же байткод finally-блока. Логичным решением данной задачи является генерация функции, которая выполняла бы finally-блок. Проблема тут заключается в том, что код finally-блока, как правило, работает с локальными переменными функции, а поскольку в Java байткоде невозможно взять адрес локальных переменных, генерация отдельной функции для обработки finally-блока не представляется возможным.

В ранних версиях Java в байткоде существовали инструкции `jsr/ret`, которые фактически выполняли локальный вызов. Инструкция jsr при этом запоминает адрес возврата в стек и переходит на указанную метку, а инструкция ret, переходит по адресу, указанному в стеке. Таким образом, в компилятор уже получается встроенным механизм генерации finally-блоков без дублирования кода. К сожалению, данный механизм не используется в поздних версиях Java.

Для анализа дублирование байткода неудобно тем, что байткод перестаёт взаимно-однозначно отображаться на исходный код, и необходимо учитывать отладочную информацию для установления, какой же код был продублирован. В противном случае возможны ситуации, при которых будут выданы ложные сообщения об ошибках. Дело в том, что одним из критериев выдачи сообщений в Svace является установление факта наличия ошибки на всех путях, проходящих через определённую точку. То есть, анализатор пытается найти точки в программе, обладающие следующим свойством: если программа попадает в эту точку, то неминуемо произойдёт или уже произошла ошибка – в этом случае данное место в программе либо является мертвым кодом, либо в программе содержится ошибка.

В случае дублирования кода данный критерий теряет свою актуальность, поскольку одна и та же точка в исходной программе будет соответствовать нескольким точкам в байткоде. Тем самым анализатор не может рассчитывать на то, что все точки программы должны хоть когда-то исполняться. В табл. 1 приводится код метода и получающийся стандартным компилятором соответствующий байткод. Как видно из данного примера, код,

соответствующий finally-блоку, дублируется в трёх экземплярах (дублируемые инструкции байт-кода выделены серым, остальной код – мелким шрифтом). И, если не исправить и не учитывать это дублирование, то на выполнении кода на ветке без исключений возникает недостижимый код: переменная `err` будет равна нулю, и далее следует бессмысленное сравнение с единицей.

Табл. 1. Дублирование кода компилятором javac
Table 1. Code duplication performed by javac compiler

```
int err = 0;
try {
    return something();
} catch(RuntimeException e) {
    err = 1;
} finally {
    // Возможное ложное
    // срабатывание
    if (err == 1) {
        report();
    }
    finish();
}
return err;
```

```
Exception table:
from    to  target  type
  2     7    22 RuntimeException
  2     7    41      any
 22    25    41      any
 41    43    41      any

 0: iconst_0
 1: istore_2
 2: aload_0
 3: invokevirtual something:()I
 6: istore_3
 7: iload_2

 8: iconst_1
 9: if_icmpne     16
12: aload_0
13: invokevirtual report:()V
16: aload_0
17: invokevirtual finish:()V

20: iload_3
21: ireturn
22: astore_3
23: iconst_1
24: istore_2
25: iload_2

26: iconst_1
27: if_icmpne     34
30: aload_0
31: invokevirtual report:()V
34: aload_0
35: invokevirtual finish:()V

38: goto   59
41: astore 4
43: iload_2

44: iconst_1
45: if_icmpne     52
48: aload_0
49: invokevirtual report:()V
52: aload_0
53: invokevirtual finish:()V
```

```
56: aload  4
58: athrow
59: iload_2
60: ireturn
```

Другой проблемой является автогенерированный компилятором код. К примеру, оператор отрицания нередко раскрывается в if-ветвление. В результате выдаются ложные срабатывания про недостижимый код. В примере из табл. 2 инструкция №9 является недостижимым кодом, однако более правильно в данном случае сообщать о константном результате вычисления выражения `!x`. Чтобы решить проблему автогенерации, был изменен компилятор javac таким образом, чтобы он генерировал вспомогательный блок информации к функции с разметкой сгенерированных ветвлений (серым цветом в примере помечены инструкции такого ветвления).

Табл. 2. Генерация условного оператора
Table 2. If operator generation

```
if (x)
 {
    // Сообщение о недостижимом
    // коде будет некорректно
    something(!x);
 }
```

```
 0: iload_1
 1: ifeq 17
 4: aload_0
 5: iload_1
 6: ifne 13
 9: iconst_1
10: goto 14
13: iconst_0
14:           invokevirtual
something:(Z)V
17: return
```

4. Статический анализ Java-программ

Этап статического анализа в инструменте Svace можно разбить на несколько последовательных основных фаз: построение графа вызовов, разрыв циклов в графе вызовов и составление топологического порядка обхода графа вызовов в порядке от листовых вершин к корневым, обход графа вызовов с разорванными циклами в топологическом порядке и анализ каждой функции. При этом по результатам анализа функции формируется резюме, которое используется вызывающими функциями, чем достигается межпроцедурность анализа.

Для анализа функции сначала читается промежуточное представление, сгенерированного специальным компилятором, и транслируется в

промежуточное представление Svace. Для каждой анализируемой функции строится граф потока управления и топологически обходится в глубину, начиная с входной точки в функцию. Ядро статического анализа Svace осуществляет символьное исполнение инструкций промежуточного представления. Поиском дефектов занимаются детекторы, которые подписываются на интересующие их события (например, на события разыменования и сравнения для детектора поиска разыменования переменной после её сравнения с нулём). Детекторы декларируют набор атрибутов, которые распространяются по графу потока управления в процессе символьного исполнения, а также корректируются в обработчиках детектора при наступлении интересующих его событий. Проверка на возможный дефект также происходит в обработчиках детектора. Найденные дефекты записываются в предварительный буфер, который впоследствии фильтруется с целью устранить дублирующиеся и похожие сообщения.

Анализ Java-программ происходит по аналогичной схеме. На первом этапе происходит быстрый просмотр всех файлов промежуточного представления. При этом ведётся сбор информации об иерархии наследования, содержании классов, строится граф вызовов. На втором этапе происходит девиртуализация, которая существенно упрощает граф вызовов. По готовому графу вызовов строится порядок обхода функций. На последнем этапе происходит обход этого графа и основной описанный выше анализ.

Таким образом, для добавления поддержки языка Java было необходимо реализовать трансляцию Java байт-кода в промежуточное представление Svace, девиртуализацию, реализацию спецификаций стандартной библиотеки Java, а также ряд специфичных для Java детекторов.

4.1 Трансляция Java байткода в промежуточное представление Svace

Внутреннее представление Svace пригодно для анализа различных языков, но по своему уровню близко к биткоду LLVM, т.к. изначально использовалось для программ на языках C и C++, а собственным компилятором Svace для этих языков является модифицированный Clang. Представление Svace является низкоуровневым трехадресным типизированным ассемблером в SSA-форме. При добавлении поддержки языка Java в Svace было решено использовать байткод Java как входное для статического анализатора представление, а перед анализом транслировать его в имеющееся внутреннее представление.

Это решение было обосновано несколькими факторами. Во-первых, байткод Java является стандартным выходным форматом для компилятора javac, и этот формат поддерживают многие системные утилиты Java, среди которых популярная библиотека ASM. Во-вторых, кроме непосредственно компилируемой программы, в проекте обычно присутствуют сторонние библиотеки в виде JAR-файлов, также содержащие байткод. Таким образом,

становится возможным проанализировать ещё и использующиеся при компиляции библиотеки.

Для чтения байткода используется библиотека ASM. Несмотря на то, что Java байткод представляет из себя стек-машину, эта стек-машина получается из трансляции абстрактного синтаксического дерева метода и потому имеет ряд ограничений. В частности, для каждого базового блока фиксирована входная глубина и максимально возможная глубина стека. Таким образом, возможна трансляция этой стек-машины в обычное трёхадресное представление.

Табл. 3. Трансляция выражения z=x+y
Table 3. Translating z=x+y expression

Байт-код	Тривиальная трансляция	Трансляция с оптимизацией
`iload_1`	`tmp1 = deref addr_x` `pmove tmp1 to stack0`	`tmp1 = deref addr_x`
`iload_2`	`tmp2 = deref addr_y` `pmove tmp2 to stack1`	`tmp2 = deref addr_y`

`iadd`	`tmp3 = deref stack1`
	`tmp4 = deref stack0`
	`tmp5 = tmp3 + tmp4`
	`pmove tmp5 to stack0`
		`tmp3 = tmp1 + tmp1`

`istore_3`	`tmp6 = deref stack0` `pmove tmp6 to addr_z`	`pmove tmp3 to addr_z`

Для трансляции был разработан следующий алгоритм. Достаточно поддерживать для каждой инструкции текущую глубину стека. Для каждой глубины стека заводится соответствующая ячейка памяти. Загрузка значения из стека транслируется, как загрузка значения из ячейки памяти, соответствующей текущей глубине стека. Пример исходного кода из табл. 3 показывает трансляцию выражения `z=x+y` для целочисленных `x`, `y` и `z`.

Здесь была произведена следующая оптимизация. Вместо того, чтобы транслировать каждое обращение в стек и из стека в инструкции обращения к ячейкам памяти, алгоритм трансляции может запоминать промежуточные значения и класть их в ячейки памяти, соответствующие глубине стека, при условии, что транслируемый базовый блок заканчивает выполняться с непустым стеком. Соответственно, брать значения из ячеек памяти можно только тогда, когда неизвестны SSA-значения, которые там должны были лежать. Такая ситуация возможна, только если транслируется базовый блок с изначально непустым стеком, что бывает, например, при трансляции тернарных операторов. В результате в байткоде получается разветвление, каждая ветка которого по-своему заполняет единственный элемент в стеке.

Возможно окончательно избавиться от ячеек памяти, эмулирующих стек. Для этого после построения исходного представления необходимо переместить на псевдорегистры все переменные из памяти, про которые известны их адреса и которые не остаются живыми после окончания работы функции. Подобного рода устранение ячеек памяти реализовано в трансформации mem2reg в рамках инфраструктуры LLVM. Однако при этом необходимо аккуратно сохранять соответствие с исходным кодом программы, чтобы не потерять информацию о том, какие псевдорегистры находились в памяти (например, для корректного поиска утечек памяти). В настоящий момент данное преобразование в инструменте Svace не реализовано.

Следует также отметить ещё несколько тонкостей, связанных с трансляцией. Первой из них является тот факт, что для значений типа `long` и `double` выделяется 2 ячейки памяти. При этом под ссылки на объекты выделяется одна ячейка памяти. Такое деление является устаревшим, так как в 64х-битных системах возможны ситуации, при котором объектов в программе будет больше, чем 2^{32}. Нужно отслеживать тип операндов у операций со стеком для корректной генерации кода.

Второй тонкостью является выделение двух ячеек памяти под локальные переменные типа `long` и `double`. Из-за этого нумерация локальных переменных является не сплошной. Кроме того, локальные переменные для Java-машины по сути реализуют адресуемую память. Соответствие реальным локальным переменным можно построить только по отладочной информации. При этом одной и той же ячейке этой адресуемой памяти могут соответствовать несколько реальных локальных переменных в анализируемой программе, если области видимости этих переменных не перекрываются.

4.2 Девиртуализация

Девиртуализация была реализована перед этапом построения графа вызовов и является новой фазой по сравнению с анализом кода на языках C и C++. На данный момент реализованы простые эвристики девиртуализации, основанные на иерархии классов. Считается, что при анализе доступна вся информации о наследовании. Конечно, это не всегда так, но на реальных проектах это предположение себя оправдывает.

Исходя из иерархии наследования, очень часто можно сказать, из какого именно класса будет вызвана та или иная функция. На рис. 3 изображён пример с иерархией наследования из четырёх классов с базовым классом A и рассмотрена ситуация, как будет девиртуализирован вызов виртуального метода f(). Для объектов, тип которых указан как B, C и D, можно однозначно определить, какая именно функция будет вызвана. В классе C метод f() объявлен как абстрактный, однако поскольку единственной реализацией в примере является реализация из класса D, то можно точно сказать, что эта реализация и будет вызвана. В то же время, базовый класс A имеет реализацию метода f(), однако если переменная имеет тип A, то её реальный

тип может быть A, B или D, что не позволяет девиртуализовать вызовы метода f() у переменных с типом A.

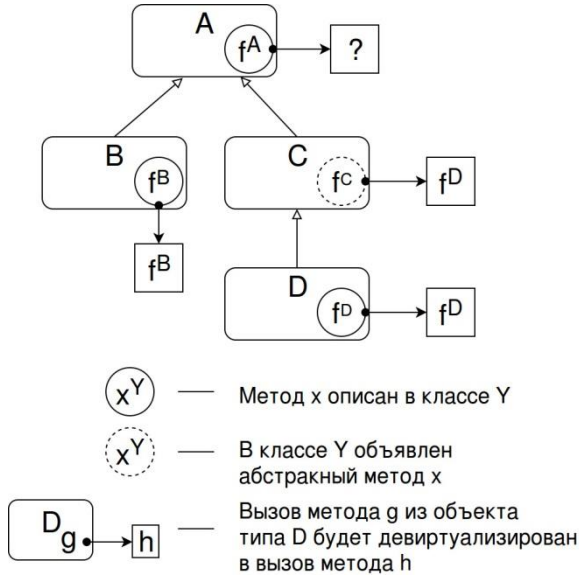


Рис. 3. Пример девиртуализации
Fig. 3. Devirtualization example

Такой метод наиболее эффективен для девиртуализации публичных (`public`) методов, не имеющих переопределений. Поскольку практика указания `final` в описании методов не является распространённой среди Java-программистов, то подобного рода девиртуализация на основе иерархии наследования позволяет устранить избыточную формальную виртуальность в вызовах.

4.3 Спецификации

Статическому анализатору важно знать, какой эффект на программу производят функции стандартной библиотеки, особенно функции выделения ресурсов. Можно пытаться понять смысл функций стандартной библиотеки автоматически по их исходному коду, но таковой чаще всего недоступен для анализа, нет гарантий, что заранее проанализированный код совпадает с используемой библиотекой, и информация, полученная от такого анализа, избыточна. Поэтому в Svace используются спецификации для описания эффектов функций стандартной библиотеки. Спецификация функции стандартной библиотеки представляет из себя краткое резюме эффекта

исполнения этой функции с точки зрения видимых извне действий функции, интересных анализатору.

Стандартная библиотека Java обладает существенно большим размером, нежели стандартная библиотека языка Си. Кроме нее, нами была поддержана библиотека платформы Android [8], так как эта платформа является важной для Java. Основной проблемой при реализации спецификаций стала иерархия классов в стандартной библиотеке. К примеру, необходимо учитывать тот факт, что при вызове метода `close` у объекта через интерфейс базового класса будет очищаться выделенный ресурс. Таким образом, описанной выше девиртуализации становится недостаточно, чтобы отслеживать утечки ресурсов, так как вызов остался бы виртуальным. В текущей реализации задача решается через спецификации интерфейсов и абстрактных функций. Таким образом, виртуальный вызов функции из стандартной библиотеки превращается в подстановку спецификации, что позволяет справиться с виртуальностью в стандартной библиотеке Java.

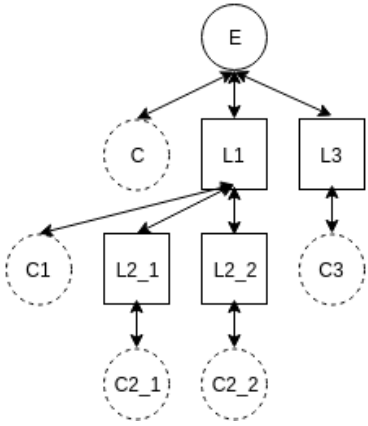
Отличительной особенностью стандартной библиотеки Java является то, что она содержит достаточно много классов-обёрток, таких как `BufferedInputStream`. В результате, надо учитывать тот факт, что вызов вызов `close` из базового интерфейса в случае стандартной библиотеки будет, как правило, закрывать все дочерние ресурсы. Данное предположение реализовано в спецификациях стандартной библиотеки Java, то есть считается, что метод `close` рекурсивно вызывает `close` у всех дочерних объектов. Использование такой эвристики приводит к незначительной потере истинных срабатываний, однако при этом позволяет избежать значительного количество ложных срабатываний или усложнения алгоритмов анализа и девиртуализации.

4.4 Специфичные для Java детекторы

Большая часть детекторов для Java входит в набор детекторов для Си/Си++ в том или ином виде, и их реализация является общей. Однако имеется ряд специфичных для Java детекторов. Прежде всего, можно выделить детекторы на синхронизацию. В Java существует оператор синхронизации, который синтаксически гарантированно всегда будет сбалансирован. Это позволяет писать для Java более простые и эффективные детекторы, чем это пришлось бы делать для Си и Си++. Кратко опишем устройство детекторов ошибок синхронизации в объеме, возможном для данной статьи.

В статическом анализаторе Svace реализованы следующие детекторы для обнаружения ошибок синхронизации: детектор для обнаружения взаимных блокировок DEADLOCK и статистический детектор NO_LOCK.STAT для обнаружения состояний гонки. Работа данных детекторов основана на анализе *дополненного графа вызовов* — модели параллельной программы, описывающей выполнение программы в несколько потоков.

Локальным дополненным графом вызовов будем называть двунаправленный граф, имеющий вершины трех типов: начало метода, захват ресурса, вызов метода. Вершина любого типа содержит в себе ссылку на место в исходном коде программы, где произошло то или иное событие. Вершины начала или вызова метода, кроме того, содержат уникальный идентификатор метода. Вершина захвата ресурса содержит абстрактный ресурс, идентифицирующий ресурс, который необходимо захватить потоку для входа в критическую секцию. Локальный граф строится, соединяя вершины захвата ресурсов согласно путям в графе потока управления, проходящим через них и начало метода, и аналогично соединяя вершины с вызовами методов. Целью является представить информацию о том, какие ресурсы необходимо захватить, чтобы добраться до вызова метода (см. рис. 4 – квадратами показаны вершины второго типа, прерывистыми кругами – третьего).




```
public void E() {
    C();
    synchronized(L1) {
        C1();
        synchronized(L2_1) {
            C2_1();
        }
        synchronized(L2_2) {
            C2_2();
        }
    }
    synchronized(L3) {
        C3();
    }
}
```

Рис. 4. Локальный дополненный граф вызовов
Fig. 4. Local enriched graph

Дополненным графом вызовов будем называть граф, полученный из локальных, путем соединения соответствующих вершин первого и третьего типов. Так как граф вызовов обходится анализом от вызываемых функций к вызывающим, то дополненный граф можно строить инкрементально по ходу анализа – локальные графы для вызываемых функций в точке вызова уже будут известны.

Ошибкой взаимной блокировки потоков в параллельной программе называется ситуация, когда несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками. Детектор DEADLOCK нацелен на обнаружение взаимной блокировки, определенной следующим образом. Пусть t и t′ – два абстрактных потока, а l_1, l_2, l_1', l_2' – инструкции блокировки в исходном коде программы. *Взаимная блокировка* потоков t и t′

будет иметь место, если t и t′ будут ждать освобождения ресурсов r_1 и r_2 при выполнении l_2 и l_2', при этом владея ресурсами r_2 и r_1 после выполнения l_1 и l_1' соответственно. Определенный таким образом дефект легко обнаруживается во время обхода в глубину расширенного графа вызовов.

Перед тем как выдать предупреждение о дефекте, анализатор проводит дополнительные проверки для подавления ложных предупреждений. Основной причиной ложных предупреждений является gate lock — общий для потоков t и t' ресурс, завладеть которым необходимо до выполнения инструкций l_1 и l_1'. Для вершин дополненного графа вызовов, соответствующим l_1 и l_1', формируется множество доминирующих вершин, имеющих тип захват ресурса. Если пересечение полученных множеств не пусто, необходимо отменить выдачу предупреждения.

Использование предложенной модели параллельных программ не ограничивается поиском состояний взаимной блокировки. Ее также можно использовать для обнаружения состояний гонки. Данная ошибка может возникнуть в случае, когда несколько потоков имеют доступ к одному и тому же ресурсу. Детектор NO_LOCK.STAT накапливает статистику обращения к полям классов во время обхода графа потока управления. Если обращение к полю произошло внутри критической секции, записывается позитивный результат в статистику. Также сохраняется информация о том, какой ресурс необходимо захватить потоку, чтобы попасть в данную критическую секцию. Если обращение к полю произошло вне критической секции, записывается негативный результат в статистику. При значительном превышении позитивной статистики над негативной для случаев негативной статистики выдается предупреждение о потенциальном состоянии гонки.

Поскольку алгоритм не учитывает возможные контексты вызова метода, внутри которого произошло небезопасное обращение к полю, возможна выдача ложных предупреждений. Такая ситуация встречается, например, когда контракт метода предусматривает вызов данного метода только при захвате потоком необходимого ресурса. Для уменьшения числа ложных срабатываний используется дополненный граф вызовов. Исследуются все пути в расширенном графе вызовов, ведущие в вершину начала метода, внутри которого произошло небезопасное обращение к полю. Если на каждом пути встречается вершина захвата ресурса, необходимого для обращения к данному полю, то предупреждение о состоянии гонки не будет выдано.

Рассмотрим еще один характерный для Java детектор – NO_BASE_CALL. Он нацелен на поиск методов, которые не вызывают свою реализацию из базового класса, хотя должны это делать. Этот детектор делится на два поддетектора. Детектор NO_BASE_CALL.LIB обладает базой знаний о библиотечных методах, при реализации которых надо вызывать базовый метод. К примеру, при реализации метода `clone` необходимо использовать `clone` родительского класса и так до `Object.clone()`, который создаёт объект нужного типа, возвращает его, а производные классы его заполняют.

Типичной ошибкой при реализации является создание объекта текущего класса. В этом случае производный класс не сможет нормально себя склонировать.

Вторым поддетектором является NO_BASE_CALL.STAT. Этот детектор работает с произвольными методами и статистически пытается понять, когда программист забыл вызвать метод из базового класса, то есть детектор ищет ситуации, когда производный класс вызывает почти везде соответствующий метод базового класса, но есть случаи, когда такого вызова не происходит. В этом случае есть подозрение на забытый вызов, и генерируется предупреждение об ошибке.

5. Экспериментальные результаты

Описанные алгоритмы трансляции программ на языке Java и доработки анализатора были реализованы в инструменте Svace. Основные структуры данных и алгоритмы анализа являлись общими между Java, Си и Си++. По результатам тестирования было установлено, что качество анализа кода на языке Java находится на уровне успешных коммерческих аналогов. Анализ исходного кода ОС Android 5 занимает примерно полтора часа на сервере с 32 логическими ядрами. Результаты истинных срабатываний для некоторых детекторов приведены в табл. 4.

Табл. 4. Результаты тестирования для ОС Android 5
Table 4. Experimental results for OS Android 5

Название детектора	Срабатываний всего	Исследовано срабатываний	TP%
DEREF_AFTER_NULL	143	108	96.3%
DEREF_AFTER_NULL.EX	250	100	98.0%
DEREF_OF_NULL	10	7	100.0%
DEREF_OF_NULL.ASSIGN	245	166	100.0%
DEREF_OF_NULL.CONST	1001	775	100.0%
DEREF_OF_NULL.EX	160	57	96.5%
DEREF_OF_NULL.RET.LIB	574	188	91.2%
DEREF_OF_NULL.RET.LIB.PROC	124	12	100.0%
DEREF_OF_NULL.RET.USER	1407	397	98.0%
DEREF_OF_NULL.RET.USER.PROC	144	43	100.0%
HANDLE_LEAK	1165	268	96.8%
HANDLE_LEAK.EXCEPTION	984	240	96.9%

HANDLE_LEAK.FRUGAL	381	132	98.4%
HANDLE_LEAK.FRUGAL.EXCEPTION	158	46	95.7%
NO_BASE_CALL.LIB	179	73	93.2%
NO_BASE_CALL.STAT	235	82	100.0%
NO_CHECK_IN_LOCK	66	30	86.2%
NO_LOCK.GUARD	47	32	93.8%
NO_LOCK.STAT.EX	1332	101	94.1%
NULL_AFTER_DEREF	219	95	78.6%
UNREACHABLE_CODE	211	80	89.9%
WRONG_LOCK_OBJECT	52	37	100.0%

6. Заключение

В статье была описана реализация поддержки языка программирования Java в рамках инфраструктуры инструмента статического анализа Svace, изначально разработанного для анализа кода на языках C и C++. В целом можно отметить, что базовые алгоритмы анализа подходят и для языка Java, трудности заключаются в организации перехвата компиляций через Java VM, девиртуализации, работе со стандартной библиотекой. Результаты работы на промышленном коде являются приемлемыми, и анализатор используется в компании Samsung для анализа собственных мобильных приложений для ОС Android и собственных ее расширений.

Дальнейшим направлением развития является разработка и внедрение более сложных алгоритмов девиртуализации. В частности, можно попробовать объединять резюме от потенциальных наследников и объединять их в одно резюме. В этом случае граф вызовов становится намного более связным, и потребуется существенно больше удалений рёбер из граф вызовов, чтобы сделать его ациклическим. Чтобы компенсировать удалённые рёбра, может потребоваться делать несколько обходов по ациклическому графу вызовов. Более точную девиртуализацию также можно проводить в случаях, когда на этапе потоково-чувствительного анализа удаётся установить реальный тип объекта или хотя бы сузить диапазон возможных типов.

Список литературы

[1]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, 2014 г., стр 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2]. А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[3]. Спецификация виртуальной машины Java. http://docs.oracle.com/javase/specs/jvms/se7/html/, дата обращения 20.06.2017
[4]. Компилятор Javac. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, дата обращения 20.06.2017
[5]. Программный интерфейс компиляции в Java. http://openjdk.java.net/groups/compiler/guide/compilerAPI.html, дата обращения 20.06.2017
[6]. Система сборки Ant. http://ant.apache.org/, дата обращения 20.06.2017
[7]. Система сборки Maven. https://maven.apache.org/, дата обращения 20.06.2017
[8]. ОС Android. https://source.android.com/, дата обращения 20.06.2017
[9]. Компилятор Eclipse ECJ. https://mvnrepository.com/artifact/org.eclipse.jdt.core.compiler/ecj, дата обращения 20.06.2017
[10]. Инструментация байткода Java через java-агенты. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, дата обращения 20.06.2017

Supporting Java programming in the Svace static analyzer

[1] *A.P. Merkulov <steelart@ispras.ru>*
[1] *S.A. Polyakov <inly@ispras.ru>*
[1,2] *A.A. Belevantsev <abel@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. The paper is devoted to the works performed within the Svace static analysis tool to support Java language. First, the approach to intercept compilation process for transparently building the analyzer internal representation should be extended to cover usage of the Java compiler API that is popular in Ant and Maven tools. We achieve this goal with implementing our custom Java agent that instruments all calls to the compiler API and notifies the analyzer with the actual compilation parameters. Second, the modified Javac compiler builds the analyzer IR. The changes we made to the compiler include avoiding unnecessary bytecode duplication for easier mapping of bytecode instructions to source code and properly marking the code added by the compiler itself. Third, we discuss the process of bytecode translation to the Svace IR proper (which is a low-level 3-address IR akin to the LLVM IR). It is a straightforward code generation algorithm with further code cleanups that treats stack locations as local variables made possible by the fact that we know the maximum stack size consumed by the method. Finally, we discuss the devirtualization heuristics that assume we know the full class hierarchy and specific Java checkers including synchronization issue checkers. Experimental results obtained on Android 5 source code show that the checkers have high quality (more than 80% true positives). It can be seen that the general infrastructure for analysis and checkers implemented in Svace works well for the Java programming language with the adaptations described in the paper.

Keywords: static analysis; Java; bytecode; devirtualization

DOI: 10.15514/ISPRAS-2017-29(3)-5

For citation: Merkulov A.P., Polyakov S.A., Belevantsev A.A. Supporting Java programming in the Svace static analyzer. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017. pp. 57-74 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-5

References

[1]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/ Procf ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[2]. A. Borodin, A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAS/ Proc. ISP RAS, vol. 27, issue 6, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8

[3]. Java virtual machine specification. http://docs.oracle.com/javase/specs/jvms/se7/html/, accessed 20.06.2017

[4]. The Javac compiler. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, accessed 20.06.2017

[5]. Java compiler API. http://openjdk.java.net/groups/compiler/guide/compilerAPI.html, accessed 20.06.2017

[6]. Ant build system. http://ant.apache.org/, accessed 20.06.2017

[7]. Maven build system. https://maven.apache.org/, accessed 20.06.2017

[8]. Android operating system. https://source.android.com/, accessed 20.06.2017

[9]. The Eclipse ECJ compiler. https://mvnrepository.com/artifact/org.eclipse.jdt.core.compiler/ecj, accessed 20.06.2017

[10]. Instrumenting Java bytecode with Java agents. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, accessed 20.06.2017