DOI: 10.15514/ISPRAS-2024-36(3)-1

Статический анализ для языка Scala

[1,2] *Афанасьев В. О.,* ORCID: 0000-0002-8036-0633 <*vafanasiev@ispras.ru*>
[1] *Бородин А. Е.,* ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1,3] *Белеванцев А. А.,* ORCID: 0000-0003-2817-0397 <*abel@ispras.ru*>

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Национальный Исследовательский Университет Высшая Школа Экономики, 101000, Россия, г. Москва, ул. Мясницкая, д. 20*
[3] *Московский государственный университет им. М.В. Ломоносова, Россия, г. Москва, Ленинские горы д. 1*

Аннотация. В работе описывается статический анализатор для поиска ошибок в программах на языке Scala. Предлагаемая схема анализа использует JVM-байткод, полученный при компиляции программ. Полученный байткод передаётся на вход межпроцедурному статическому анализатору *Svace*. В отличие от анализа других языков, поддерживаемых анализатором *Svace*, в данной статье мы рассматриваем подход, не требующий модификаций компилятора и таким образом сильно упрощающий поддержу языка. Подобный подход также может использоваться в статических анализаторах, которые нацелены на поддержку большого количества языков.

Ключевые слова: статический анализ; поиск ошибок; уязвимости; Scala; JVM; байткод; Svace.

Для цитирования: Афанасьев В.О., Бородин А.Е., Белеванцев А.А. Статический анализ для языка Scala. Труды ИСП РАН, 2024, том 36 вып. 3, с. 9–20. DOI: 10.15514/ISPRAS–2024–36(3)–1.

Static Analysis for Scala

[1,2] *Afanasyev V. O.*, ORCID: 0000-0002-8036-0633 <*vafanasiev@ispras.ru*>
[1] *Borodin A. E.*, ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1,3] *Belevantsev A. A.*, ORCID: 0000-0003-2817-0397 <*abel@ispras.ru*>

[1] *Ivannikov Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia
[2] *National Research University Higher School of Economics,*
20, Myasnitskaya Str., Moscow, 101000, Russia.
[3] *Moscow State University,*
1, Leninskie gory, Moscow, 119991, Russia.

Abstract. The paper describes a static analyzer for finding defects in Scala programs. The proposed analysis scheme uses JVM bytecode produced during compilation. The generated bytecode is used as an input for inter-procedural static analyzer *Svace*. In contrast to the analysis of other languages supported by *Svace*, in this work we describe an approach that does not require compiler modifications and therefore simplifies language support. This approach can also be used in static analyzers that aim to support a large number of programming languages.

Keywords: static analysis; search for defects; vulnerabilities; Scala; JVM; bytecode; Svace.

For citation: Afanasyev V.O., Borodin A.E., Belevantsev A.A. Static analysis for Scala. Trudy ISP RAN/Proc. ISP RAS, 2024, vol. 36, issue 3, pp. 9–20 (in Russian). DOI: 10.15514/ISPRAS–2024–36(3)–1.

1. Введение

В данной работе мы описываем наш опыт, полученный при добавлении поддержки языка Scala в статический анализатор Svace [1-2].

Scala – мультипарадигмальный язык программирования со статической типизацией, поддерживающий как объектно-ориентированную, так и функциональную парадигму. В качестве основной платформы для этого языка используется JVM [3]. Также имеется возможность компиляции Scala-кода в JavaScript-код [4] и в нативный код при помощи платформы LLVM [5].

Для Scala существует две независимые версии языка, которые развиваются параллельно – Scala2 и Scala3. Особенность данных версий в том, что между ними нет совместимости на уровне исходного кода — компилятор для одной версии языка не сможет скомпилировать произвольный код, написанный для другой версии языка. Как следствие, для Scala имеется два компилятора, развиваемых независимо.

Согласно индексу TIOBE за март 2024 года [6], Scala находится на 34-м месте по популярности. С одной стороны, этот язык менее популярен чем некоторые другие JVM-языки: Java (4-е место) и Kotlin (19-е место). С другой стороны, язык Scala достаточно популярен, чтобы войти в топ-50 – зачастую Scala используют в качестве альтернативы в Java-проектах, так как этот язык полностью совместим с Java, но использует более выразительную систему типов и имеет более полную поддержку функциональной парадигмы [7].

Подход к анализу программ, применяемый в анализаторе Svace [8, 9], подразумевает перехват оригинальной сборки программы и модификацию компилятора с целью генерации промежуточного представления, лучше подходящего для статического анализа. Из-за низкой популярности языка, а также наличия двух компиляторов, мы решили отойти от этой схемы и поэкспериментировать с более простой реализацией, которая не включает модификацию компиляторов.

2. Схема анализа Svace

Статический анализатор Svace осуществляет поиск ошибок в программах, написанных на языках C, C++, Java, Kotlin, Python, Go. Для двух языков – Java и Kotlin – в качестве промежуточного представления используется байткод для Java Virtual Machine (JVM).

Схема анализа JVM-языков в Svace состоит из следующих фаз:

1) Контролируемая сборка проекта с сохранением байткода и прочей информации, необходимой для анализа;
2) Чтение байткода, построение из него унифицированного представления и анализ получившегося представления.

Во время фазы контролируемой сборки Svace проводит мониторинг процессов, запускаемых во время сборки проекта. Если этот процесс является командой компиляции, то Svace дополнительно запускает процесс компиляции при помощи собственного модифицированного компилятора для соответствующего языка, передавая ему опции от оригинальной команды, изменяя или дополняя их при необходимости. Под модифицированным компилятором подразумевается некоторая доработанная версия исходного компилятора, которая производит более удобный для дальнейшего анализа байткод – в частности, с выключенными оптимизациями и с отладочной информацией. Общая схема данного процесса сборки приведена на рис. 1.

Таким образом, для анализа языков Java и Kotlin анализатор Svace сначала строит промежуточное представление, которое представляет собой модифицированный байткод. При этом совместимость с оригинальным байткодом сохранена, отличие заключается только в том, что мы добавляем туда дополнительную информацию об оригинальной программе. Затем на основе сгенерированного байткода строится внутреннее представление Svace и запускается анализ. Исходный код программы уже не участвует при анализе.

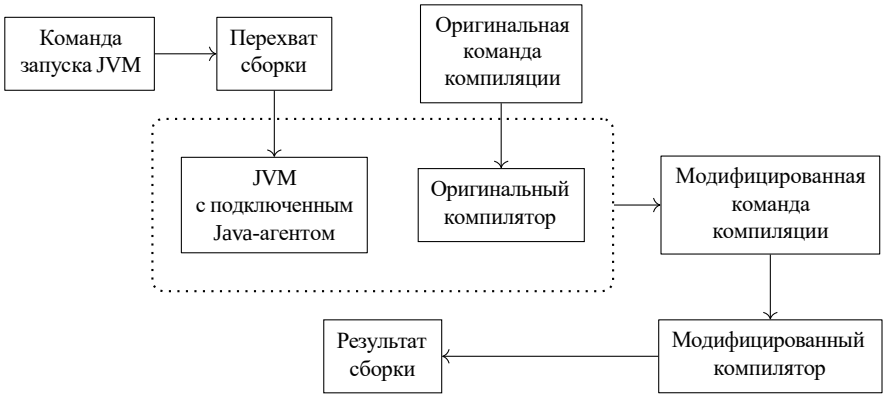

Рис. 1. Процесс сборки при наличии модифицированного компилятора
Fig. 1. Build process in presence of modified compiler

Наша идея анализа языка Scala заключается в том, чтобы подать на вход анализатору байткод, сгенерированный оригинальными компиляторами. Ни сами компиляторы, ни байткод, выдаваемый ими, никак при этом не модифицируются.

С одной стороны, отсутствие модифицированного компилятора может влиять на качество анализа в худшую сторону, так как байткод может содержать конструкции, которые не были написаны непосредственно программистом, а появились в результате трансляции. Например, компиляторы могут генерировать дублирующийся или недостижимый код; код, содержащий излишние проверки; вспомогательные методы и классы. В таких случаях Svace не может

отличить код, написанный программистом, от кода, фиктивно добавленного компилятором, и поэтому может выдавать ложные предупреждения [1].

С другой стороны, отсутствие модифицированного компилятора сильно упрощает поддержку языка в анализаторе, так как не приходится дополнительно поддерживать кодовую базу для компиляторов этого языка. Для Scala это наиболее критично, так как для этого языка существуют сразу два невзаимозаменяемых компилятора: scalac для Scala2 и dotty для Scala3.

Также отметим, что даже базовый анализ языка Scala на основе байткода может повлиять на качество анализа в лучшую сторону. Проекты на платформе JVM допускают, что их можно разрабатывать сразу на нескольких языках, так как байткод является полностью совместимым между разными языками. Если код на одном языке вызывает код на другом языке, то для успешного анализа Svace должен получить промежуточное представление для каждого языка. В ином случае, если для некоторого языка отсутствует промежуточное представление, то вызовы функций из этого языка будут трактоваться как неизвестные – для них не будет построено внутреннее представление и не будет выполнен анализ.

3. Перехват сборки

Компилятор языка Scala представляет собой JAR-библиотеку. Для запуска компилятора напрямую пользователи могут использовать скрипт (bash-скрипт на ОС семейства Linux и bat-скрипт на ОС семейства Windows), вызывающий конкретную точку входа в Java-приложение. Также для Scala существуют системы автоматической сборки, например SBT [10], которые запускают компилятор не напрямую, а при помощи его API [11].

Для перехвата оригинальной сборки в Svace для Scala используется Java-агент, подключаемый к компилятору. Java-агент – это JAR-библиотека, которая позволяет манипулировать с байткодом загружаемых классов. В случае Svace, данный Java- агент инструментирует код компилятора Scala таким образом, чтобы перехватить два события: запись файла с байткодом (.class) на диск и завершение компиляции[2]. Во время записи файла с байткодом Svace сохраняет информацию о том, из какого файла с исходным кодом (.scala) этот байткод был сгенерирован и по какому пути он сохранён. При завершении компиляции все файлы с исходным кодом и байткодом, а также информация о том, какому исходному файлу соответствуют файлы с байткодом, сохраняются для дальнейшего анализа.

Описанная инструментация позволяет запустить оригинальный компилятор, при этом собрав всю необходимую для анализа информацию. В отличие от Java и Kotlin, для сборки которых в Svace используются модифицированные компиляторы, для языка Scala используется байткод, полученный оригинальным компилятором. Общая схема процесса сборки приведена на рис. 2.

Стоит отметить одну особенность автоматических систем сборки. Некоторые из таких систем позволяют оптимизировать процесс сборки при помощи запуска вспомогательных процессов: например, демон-процесс для Gradle [12] и клиент-серверный режим для SBT [13]. В данных режимах существуют процессы, которые можно назвать клиентским и серверным: клиентский процесс отправляет серверному процессу команды, а сервер их исполняет. В качество команд могут выступать: разрешение сторонних зависимостей, сборка проекта, выполнение автоформатирования и т.д. Таким образом, процесс компиляции инициируется клиентским процессом, но выполняется серверным. При этом клиентский и серверный процесс могут быть неродственными – более того, данные процессы могут выполняться на разных вычислительных машинах. Анализатор Svace во время сборки перехватывает только те процессы компиляции, которые будут запущены исходной командой и её дочерними

[1] Проблема заключается в том, что по сгенерированному байткоду требуется судить об оригинальном исходном коде.

[2] В случае вызова компилятора через API завершением компиляции считается не завершение самого компилятора, а выход из метода, являющегося точкой входа в компилятор через API.

процессами, поэтому в нашей реализации отсутствует поддержка для подобных режимов сборки. Следовательно, пользователи нашего анализатора должны избегать использования подобных режимов[3].

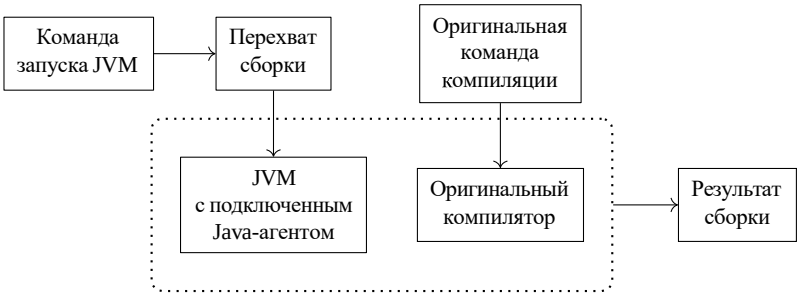

Рис. 2. Процесс сборки при отсутствии модифицированного компилятора
Fig. 2. Build process in absence of modified compiler

4. Анализ

Анализ для Scala основывается на компоненте, который выполняет анализ для языков Java [14] и Kotlin [9] на основе байткода JVM. При этом возможен совместный анализ кода на Java и Kotlin, так как в реальных проектах код этих языков может быть смешан.

Svace строит граф вызовов и выполняет анализ на основе резюме [15]. При таком подходе после анализа каждой функции строится её резюме, которое затем используется для анализа инструкций вызова функции. Анализ на основе резюме имеет высокую масштабируемость и скорость анализа.

Для языка Scala мы включили детекторы для следующих видов ошибок:

- Двойное закрытие ресурсов
- Использование ресурсов после закрытия
- Утечки ресурсов
- Небезопасное использование данных из непроверенных источников
- Утечка конфиденциальной информации
- Непроверенные возвращаемые значения функций
- Неиспользуемые возвращаемые значения функций
- Деление на ноль

Всего в процесс анализа было включено 55 детекторов.

Вместе с тем, из-за использования байткода, сгенерированного оригинальным компилятором, часть детекторов не была включена. Во-первых, это все детекторы на недостижимый код и на бесполезные сравнения в исходном коде. Для реализации таких детекторов требуется добавление дополнительной информации об инструкциях, генерация которых была вызвана непосредственно исходным кодом.

Также в Svace используется подход поиска неконсистентности в исходном коде [16]. При таком подходе для переменных языка выводятся предположения на основе их использования. Например, если переменную сравнивают с нулём, то предполагается, что она может иметь нулевое значение. Если в функции есть инструкция разыменования этой переменной без проверки на ноль, то будет выдано предупреждение о неконсистентной работе со ссылкой. Все такие детекторы также были выключены для языка Scala.

[3] Данное ограничение не относится к Scala непосредственно, так как подобные режимы присутствуют и в системах сборки для других языков.

4.1 Анализ сущностей, вычисление их метрик и отношений между ними

Для языков, анализ которых поддерживается в Svace, реализован отдельный модуль, который вычисляет для анализируемой программы содержащиеся в ней сущности: классы, методы, поля и глобальные переменные, а также файлы и каталоги; далее для всех сущностей определяется ряд числовых характеристик, метрик, и вычисляются отношения — связи между двумя сущностями (по вызову, чтению, записи, наследованию и проч.). При этом в значительной степени используются те же входные данные, что и при поиске ошибок: информация от контролируемой сборки о выполненных компиляциях и компоновках, внутреннее представление исходного кода, отладочная информация. Модуль может выступать и как независимый инструмент анализа. Более подробно вопросы поддержки языков C, C++ освещены в статье [17], поддержки языка Kotlin — в [9].

Языки Kotlin и Java поддерживаются в упомянутом модуле по традиционной схеме: метрики, связанные с организацией исходного кода по методам и файлам (число строк в функции или классе, число комментариев, пробельных строк и т.п.), вычисляются в компиляторе, генерирующем внутреннее представление; там же вычисляются метрики потока управления методов (число циклов, условных операторов, операторов выбора, цикломатическая сложность и пр.); наконец, на этапе анализа в модуле считывается байткод для всех тел методов и построенные ранее метрики, после чего вычисляются отношения и агрегируются финальные метрики, требующие знания устройства всей программы (например, число потомков класса, глубина определенного класса в дереве наследования и подобные). Применение байткода позволяет находить отношения (например, вызовы), возникающие между сущностями разных языков: в программах на языке Kotlin нередки вызовы методов из Java-кода и наоборот.

Для языка Scala подход, описанный нами в данной статье, не позволяет найти метрики, которые необходимо вычислять в компиляторе, но дает возможность использовать повторно имеющийся модуль анализа сущностей, обрабатывающий байткод, и посчитать сущности, отношения и агрегирующие метрики. Как и для Kotlin, сразу будут определяться связи между сущностями на разных языках, что актуально для Scala. Чтобы поддержать остальные метрики, мы планируем применить подход унифицированного абстрактного синтаксического дерева [18], который реализован в Svace для легковесного поиска ошибок в программах на ряде поддерживаемых языков. Суть подхода заключается в том, что в компиляторе языка генерируется, помимо внутреннего представления для глубокого анализа, унифицированное АСД, а далее детекторы ошибок кодирования и стиля пишутся для этого дерева единожды и работают для всех языков.

Реализация этого подхода и для Scala позволит, во-первых, находить ошибки уже реализованным легковесным анализом, во-вторых, вычислить информацию, необходимую для навигации по коду. После того, как завершатся ведущиеся работы по адаптации модуля, считающего метрики и отношения между сущностями по байт- коду, для его работы с представлением унифицированного дерева, можно будет вычислять метрики потока управления методов и для Scala. Тем не менее, чтобы вычислять метрики структуры кода (число строк), в любом случае понадобится модифицировать компилятор Scala.

5. Результаты

Мы проанализировали 14 проектов с открытым исходным кодом при помощи Svace. Часть из этих проектов используют одновременно и Scala, и Java, а какие-то написаны только на Scala.

Ниже в табл. 1 и 2 представлены результаты по времени сборки и анализа. Для проектов, которые используют в своём коде язык Java, мы также сделали дополнительные замеры времени при отключённом перехвате сборки для Java, но включённом для Scala.

Замедление сборки со Svace только Scala-кода относительно оригинальной сборки в среднем составило около 3%. Замедление сборки со Svace Scala- и Java-кода относительно сборки со

Svace только Scala-кода в среднем составило около 38%. Как можно заметить, сборка Java-кода замедляется при перехвате Svace сильнее, чем сборка Scala-кода. Это связано с тем, что при компиляции Java Svace запускает дополнительный процесс компиляции, использующий модифицированный компилятор. Для Scala же дополнительных запусков компилятора не происходит, так как используется байткод, созданный оригинальным компилятором. Таким образом, более простая реализация дала ожидаемое ускорение в контролируемой сборке из-за отсутствия запуска модифицированного компилятора.

Табл. 1. Время сборки и анализа проектов, использующих Scala-код и Java-код
Table 1. Build and analysis time for Scala and Java projects

Проект	Размер, KLOC (Scala + Java)	Время оригинальной сборки, сек.[4]	Время сборки со *Svace* (Scala), сек.[4]	Время сборки со *Svace* (Scala + Java), сек.[4]	Время анализа, сек.[4]
apache/spark	879 + 88	385	398	491	650
lampepfl/dotty	492 + 6	67	67	72	294
akka	306 + 209	110	116	193	244
playframework	77 + 34	32	32	54	60
apache/incubator-streampark	24 + 24	147	154	191	35

Табл. 2. Время сборки и анализа проектов, использующих только Scala-код
Table 2. Build and analysis time for Scala only projects

Проект	Размер, KLOC (Scala)	Время оригинальной сборки, сек.[4]	Время сборки со *Svace* (Scala), сек.[4]	Время анализа, сек.[4]
zio	125	85	87	110
broadinstitute/rawls	102	39	41	59
DataBiosphere/leonardo	69	52	53	75
sangria-graphql/sangria	60	35	36	79
chipsalliance/chisel	49	25	25	55
ghostdogpr/caliban	35	50	51	59
slick	28	35	36	81
gvolpe/trading	6	26	27	22
ucb-bar/berkeley-hardfloat	3	8	9	15

В табл. 3 для данных проектов приведены результаты разметки предупреждений. Также в табл. 4 приведены результаты для каждого из видов ошибок. Были размечены все выданные предупреждения. Истинные предупреждения составили 69% от общего количества. Мы считаем это неплохим результатом. Для сравнения, текущая версия Svace на Java коде проекта Android 14 имеет средний процент истинных срабатываний 73,6%, а на исходном коде компилятора Kotlin 1.5.30 – 75,75%.

Оценивать качество Scala проектов было сложнее из-за отсутствия разметки кода – информации о взаимосвязи между токенами проекта, которую использует веб-интерфейс показа предупреждений. Для Java и Kotlin эту информацию сохраняют модифицированные

[4] Все замеры времени проводились на вычислительной машине с характеристиками: CPU: 12 логических ядер, 4.6GHz; RAM: 32Gb.

компиляторы. Для Scala в текущей реализации эта информация никак не генерируется и при сохранении текущего подхода её придётся генерировать отдельно.

Табл. 3. Результаты анализа проектов
Table 3. Analysis results

Проект	Число истинных	Число ложных
apache/spark	56	17
lampepfl/dotty	14	9
akka	51	3
zio	0	14
broadinstitute/rawls	0	0
playframework	0	2
DataBiosphere/leonardo	0	0
sangria-graphql/sangria	0	4
chipsalliance/chisel	19	1
ghostdogpr/caliban	1	0
apache/incubator-streampark	15	1
slick	1	19
gvolpe/trading	0	0
ucb-bar/berkeley-hardfloat	0	0
Суммарно	157	70

Табл. 4. Результаты анализа проектов по видам ошибок
Table 4. Analysis results by error types

Тип ошибки	Число истинных	Число ложных
Деление на ноль	0	2
Утечка ресурсов	111	21
Небезопасное использование данных из непроверенных источников	9	1
Непроверенные возвращаемые значения функций	35	35
Неиспользуемые возвращаемые значения функций	2	11

На листинге 1 приведён пример истинного срабатывания на проекте apache/incubator-streampark. В данной функции открываются потоки ввода из файла, но на одном из путей при завершении функции они не закрываются.

6. Похожие работы

Инструмент статического анализа SpotBugs [19-20] осуществляет анализ Java кода, получая на вход байткод. Инструмент имеет целью выдавать более высокий процент истинных срабатываний, что, в частности, достигается за счёт пропуска части срабатываний и отсутствия сложных детекторов. Анализатор не имеет модифицированный компилятор, также как и компонента контролируемой сборки. В данном случае, из-за наличия большого количества простых детекторов, такой выбор выглядит оправданным. Мы проверили результат анализатора на исходном коде самого Svace и получили процент истинных срабатываний 91,2%. Отметим, что значительная доля среди истинных была категории «формально истинных», то есть предупреждений, которые анализатор ищет, но разработчики анализируемого проекта по каким-либо причинам не планируют исправлять эти дефекты.

Возможно, имеет смысл реализовать анализ языка Scala на основе SpotBugs, или используя тот же подход. Тем не менее, нам удалось найти только упоминание того, что в инструменте SonarQube присутствует плагин, использующий SpotBugs для анализа Scala-кода [21]. Мы выполнили анализ при помощи этого плагина на приведённых выше проектах и получили процент истинных срабатываний около 10%-20% при этом суммарно было выдано более 10 тысячи предупреждений (из них размечено было около 1000). Такой низкий процент истинных срабатываний на Scala-коде связан с тем, что SpotBugs выполняет анализ на уровне байткода и не отличает языки Java, Kotlin и Scala друг от друга. Таким образом, большое количество предупреждений выдаётся для байткода, который был неявно добавлен компилятором и никак не говорит о наличии ошибки в самом исходном коде.

```
def equals(file1: File, file2: File): Boolean = {
  (file1, file2) match {
    case (a, b) if a == null || b == null => false
    case (a, b) if !a.exists() || !b.exists() => false
    case (a, b) if a.getAbsolutePath == b.getAbsolutePath => true
    case (a, b) =>
      // Открываются потоки ввода first и second
      val first = new BufferedInputStream(new FileInputStream(a))
      val second = new BufferedInputStream(new FileInputStream(b))

      if (first.available() != second.available())
          // Происходит выход из функции, но потоки не закрываются
          false;
      else {
        while (true) {
          val firRead = first.read()
          val secRead = second.read()
          if (firRead != secRead) {
            Utils.close(first, second)
            return false
          }
          if (firRead == -1) {
            Utils.close(first, second)
            return true;
          }
        }
        true
      }
    }
}
```

Листинг 1: Пример истинного срабатывания
Listing 1: Example of true positive warning

Одним из альтернативных подходов к анализу большого количества языков при помощи небольших затрат по их поддержке является анализ на основе унифицированного абстрактного синтаксического дерева [18]. Статический анализатор SonarQube позволяет анализировать проекты на большом количестве языков программирования, в том числе на языке Scala [22]. Данный инструмент использует фреймворк, именуемый SLang (SonarSource Language) [23], предоставляющий языконезависимое АСД, для которого можно писать простые синтаксические правила. Подобное решение позволяет упростить поддержку каждого отдельного языка, но анализ на основе АСД при этом имеет свои известные ограничения.

Статические анализаторы Scalafix [24] и Scapegoat [25] являются плагинами к компилятору (поддерживаются как компилятор Scala2, так и Scala3), которые используют подход к анализу на основе абстрактного синтаксического дерева и системы типов языка. Данные инструменты в основном ищут разного рода опечатки и нарушения стиля кодирования: неиспользуемые переменные/функции, излишние модификаторы, пустые блоки, вызовы методов с

подозрительными аргументами и т.п. Такого рода инструменты позволяют добиться высокого процента истинных срабатываний при сравнительно недолгом времени анализа, проводимом непосредственно во время компиляции. В то же время при таком подходе чаще пропускаются, или вовсе не ищутся, более серьёзные ошибки и уязвимости.

Проприетарный инструмент Checkmarx SAST позволяет анализировать программный код на более чем 50 языках, в том числе написанный на Scala [26]. Для языка Scala поддерживается более 80 правил для обнаружения, например, SQL-инъекций, жёстко закодированных паролей и т.д. Также для данного инструмента предоставляется язык запросов CxQL [27], позволяющий пользователям в декларативной форме описывать специфичные для их кодовой базы критерии, которые должен обнаруживать анализатор в исходном коде. Результаты тестирования [28] Checkmarx SAST на наборе тестов OWASP [29] показывают средний процент истинных срабатываний в 44,9%. Про результаты конкретно для языка Scala нам неизвестно.

Учитывая разнообразие как инструментов статического анализа, так и видов ошибок, которые они ищут, наш подход к анализу хорошо дополняет вышеперечисленные анализаторы, так как ищет ошибки, которые не находятся ими.

7. Заключение

В работе мы описали реализацию статического анализа для языка Scala по упрощённой схеме, которая не включает в себя модификацию оригинального компилятора. Несмотря на недостатки, такой подход позволяет относительно просто добавить анализ программ для конкретного языка, и является компромиссом, когда нет возможностей тратить много ресурсов на модификацию и поддержку отдельной кодовой базы компилятора. На наш взгляд такой подход можно использовать и для других JVM-языков: Java, Kotlin, Groovy. Возможно, он также применим и для не JVM-платформ, когда компилятор/интерпретатор на основе исходного кода программы генерирует низкоуровневое промежуточное представление. Например, для анализа языка Python.

Другим преимуществом подобного подхода была чуть лучшая производительность перехвата сборки из-за отсутствия вызовов модифицированного компилятора.

В целом, по нашему мнению, схему упрощённого анализа можно использовать для менее популярных языков программирования. В то же время для анализа языков типа Java и Kotlin предпочтителен более трудозатратный подход, который используется в анализаторе Svace.

Список литературы / References

[1]. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН. 2014;26(1):231-250. https://doi.org/10.15514/ISPRAS-2014-26(1)-7.

[2]. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, D. Zhurikhin. Design and development of Svace static analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM):3—9, 2018.

[3]. Scala docs. Jdk compatibility. Английский. URL: https://docs.scala-lang.org/overviews/jdk-compatibility/overview.html (дата обращения 07.03.2024).

[4]. Scala.js. Harness the scala and javascript ecosystems together. develop robust apps for browsers, node.js, and serverless. Английский. URL: https://www.scala-js.org/ (дата обращения 07.03.2024).

[5]. Scala native. Английский. URL: https://scala-native.org/en/stable/ (дата обращения 07.03.2024).

[6]. Tiobe. Английский. URL: https://www.tiobe.com/tiobe-index/ (дата обращения 07.03.2024).

[7]. Scala users. Why Scala instead of Java? Английский. URL: https://users.scala-lang.org/t/why-scala-instead-of- java/9457/14 (дата обращения 07.03.2024).

[8]. A. Belevancev, A. Izbyshev, D. Zhurikhin. Monitoring program builds for svace static analyzer. System administrator, (7-8):135—139, 2017.

[9]. Афанасьев В.О., Поляков С.А., Бородин А.Е., Белеванцев А.А. Kotlin с точки зрения разработчика статического анализатора. Труды Института системного программирования РАН. 2021;33(6):67-82. https://doi.org/10.15514/ISPRAS-2021-33(6)-5.
[10]. Sbt. A simple build tool. Английский. URL: https://www.scala-sbt.org/ (дата обращения 07.03.2024).
[11]. Sbt. Compiler interface. Английский. URL: https://www.scala- sbt.org/1.x/docs/Compiler-Interface.html (дата обращения 07.03.2024).
[12]. Gradle. Gradle daemon. Английский. URL: https://docs.gradle.org/current/userguide/gradle_daemon.html (дата обращения 07.03.2024).
[13]. Sbt. Sbt server. Английский. URL: https://www.scala-sbt.org/1.x/docs/sbt-server.html (дата обращения 07.03.2024).
[14]. Меркулов А.П., Поляков С.А., Белеванцев А.А. Анализ программ на языке Java в инструменте Svace. Труды Института системного программирования РАН. 2017;29(3):57-74. https://doi.org/10.15514/ISPRAS-2017-29(3)-5.
[15]. E. Bodden. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). В Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, страницы 85—93, 2018.
[16]. D. Engler, D.Y. Chen, S. Hallem, A. Chou, B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. ACM SIGOPS Operating Systems Review, 35(5):57—72, 2001.
[17]. Белеванцев А.А., Велесевич Е.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды Института системного программирования РАН. 2015;27(2):53-64. https://doi.org/10.15514/ISPRAS-2015-27(2)-4.
[18]. Афанасьев В.О., Бородин А.Е., Вихлянцев К.И., Белеванцев А.А. Статический анализ на основе обобщённого абстрактного синтаксического дерева. Труды Института системного программирования РАН, 2023, 35(6):103—120, https://doi.org/10.15514/ISPRAS-2023-35(6)-6.
[19]. Spotbugs. Find bugs in java programs. Английский. URL: https://https://spotbugs.github.io/ (дата обращения 18.03.2024).
[20]. N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix. Using static analysis to find bugs. IEEE software, 25(5):22—29, 2008.
[21]. Spotbugs/sonar-findbugs. Does findbugs support scala code? Английский. URL: https://github.com/spotbugs/sonar-findbugs/issues/292 (дата обращения 20.03.2024).
[22]. Sonar rules. Scala static code analysis. Английский. URL: https://rules.sonarsource.com/scala/ (дата обращения 20.03.2024).
[23]. Sonarsource. Slang. Английский. URL: https: //github.com/SonarSource/slang/blob/master/README.md (дата обращения 20.03.2024).
[24]. Scalafix. Refactoring and linting tool for scala. Английский. URL: https://scalacenter.github.io/scalafix/ (дата обращения 20.03.2024).
[25]. Scapegoat. Scala compiler plugin for static code analysis. Английский. URL: https://github.com/scapegoat- scala/scapegoat/tree/master (дата обращения 20.03.2024).
[26]. Checkmarx. Sast scanner - supported languages and frameworks. Английский. URL: https://checkmarx.com/resource/documents/en/34965- 149060-sast-scanner---supported-languages-and-frameworks.html (дата обращения 20.03.2024).
[27]. Checkmarx. Query coding example. Английский. URL: https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406757/Query+Coding+Example (дата обращения 20.03.2024).
[28]. J. R. B. Higuera, J. B. Higuera, J. A. S. Montalvo, J. C. Villalba, J. J. N. Pe´rez. Benchmarking approach to compare web applications static analysis tools detecting owasp top ten security vulnerabilities. Computers, Materials & Continua, 64(3), 2020.
[29]. Owasp. Owasp benchmark. Английский. URL: https://owasp.org/www-project-benchmark/ (дата обращения 18.03.2024).

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ – студент магистратуры факультета компьютерных наук НИУ ВШЭ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM-языки.

Vitaly Olegovich AFANASYEV – ISP RAS researcher, graduate student at the Faculty of Computer Science, NRU HSE. Research interests: compiler technologies, static analysis, JVM languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci (Phys.-Math.), Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.