Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Информационно-аналитические системы

Алексин Илья Николаевич

Исследование аномалий работы компилятора языка Kotlin

Бакалаврская работа

Научный руководитель:
к.т.н., доцент Брыксин Т. А.

Рецензент:
Инженер по тестированию ПО ООО "ИнтеллиДжей Лабс" Петухов В. А.

Санкт-Петербург
2019

Saint-Petersburg State University
Computer Science

Analitical Information Systems

Ilya Alexin

Kotlin language compiler anomaly research

Graduation Thesis

Scientific supervisor:
assistant professor Timofey Bryksin

Reviewer:
Quality Assurance Engineer, IntelliJ Labs Co. Ltd. Petukhov Victor

Saint-Petersburg
2019

Оглавление

Введение

На текущий момент сфера разработки программного обеспечения активно развивается. Например, с 2008 года по ноябрь 2018 года количество репозиториев на GitHub[1] возросло с 33 тыс. до 100 млн[2]. Вследствие такого стремительного роста количества разрабатываемых приложений всё чаще возникают проблемы проверки их корректности и производительности. При этом разработчики программного обеспечения имеют возможность оптимизировать только программный код и не могут полностью контролировать процесс его преобразования в машинный код. В таких условиях становится особенно важно отслеживать уязвимости в компиляторах, так как они могут отрицательно сказываться на качестве приложений, созданных на соответствующем языке программирования.

п В последние годы одним из самых быстро развивающихся языков программирования является Kotlin[3], разрабатываемый компанией JetBrains и применяющийся для разработки на платформах Java Virtual Machine (JVM) и Anrdoid. Так, по данным, предоставленным сервисом GitHub, за последний год количество разработчиков на Kotlin возросло в 2,6 раза, что является самым высоким показателем среди всех языков программирования[4]. Также активно развивается проект Kotlin/Native, предоставляющий возможность компилировать исходный код на языке программирования Kotlin в оптимизированный машинный код для различных платформ. Вместе с этим язык Kotlin достаточно молод (первый релиз состоялся в феврале 2016 года), а проект Kotlin/Native находится в статусе Beta, и в данный момент разработчики Kotlin заинтересованы в оптимизации работы языка на разных уровнях, и в частности на уровне компиляции.

Одним из подходов к нахождению проблем производительности компилятора является поиск кодовых аномалий, примеров синтаксически

[1]https://github.com/

[2]https://blog.github.com/2018-11-08-100M-repos

[3]https://kotlinlang.org/

[4]https://octoverse.github.com/project

верного исходного кода на языке программирования Kotlin, являющегося нетипичным по каким-либо признакам. Для разработчиков компилятора языка Kotlin такие фрагменты кода могут быть интересны по следующим причинам. Во-первых, найденные кодовые аномалии могут быть добавлены в тесты производительности компилятора. Во-вторых, на примере кодовых аномалий есть возможность оценить объём сгенерированных инструкций для особенных случаев применения языка. Полученные данные могут указать на выражения, генерирующие слишком много инструкций и поэтому требующие оптимизации со стороны компилятора.

Несколько методов исследования кодовых аномалий и реализация инструментов для их извлечения описаны в дипломных работах [17, 18]. Поиск аномалий в них осуществлялся с помощью анализа синтаксических деревьев и байт-кода JVM. Также были рассмотрены так называемые условные аномалии: фрагменты кода, которые являются аномальными при анализе байт-кода, но не являются аномальными на уровне синтаксических деревьев. В данной работе продолжаются исследования условных аномалий, а также рассматриваются аномалии в Kotlin/Native проектах, которые используют промежуточное представление Low Level Virtual Machine[5] (LLVM IR) для генерации оптимизированного машинного кода.

Постановка задачи

Целью данной работы является исследование условных аномалий на основе байт-кода, а также поиск кодовых аномалий в Kotlin/Native проектах. Её результатом станет отчёт разработчикам компилятора языка Kotlin, содержащий примеры найденных аномалий. Для достижения цели работы требуется выполнить следующие задачи:

1. Провести обзор методов поиска аномалий в данных высокой размерности.

[5]https://llvm.org/docs/LangRef.html

2. Применить различные алгоритмы поиска аномалий.

3. Реализовать инструмент для извлечения LLVM IR из Kotlin/Native проектов.

4. Предоставить отчёт разработчикам компилятора языка Kotlin.

1. Обзор

Задача обнаружения аномалий уже была поставлена в области анализа данных [4]. Одним из способов её решения является поиск выбросов в анализируемых данных, то есть нахождение объектов, отличающихся от большинства по определённым признакам. С помощью такого подхода в работе [17] задача поиска кодовых аномалий в байт-коде и синтаксических деревьях была представлена в качестве классической задачи машинного обучения. Для этого достаточно сформировать соответствующий набор данных, выбрать способ его векторного представления и затем применить методы поиска выбросов [10].

1.1. Представление данных

В данной работе используется 3 различных представления исходного кода: синтаксические деревья, байт-код и LLVM IR. Способы их векторного представления могут быть разделены на две группы. К первой группе относятся техники выделения явных признаков, основанные на подсчёте определённых метрик. Для синтаксических деревьев такими метриками могут быть высота дерева или частота встречаемости отдельных узлов [7]. Для байт-кода JVM[6] и LLVM IR[7] в качестве метрик могут выступить шаблоны инструкций для выполнения определённых действий (создание объекта, вызов функции), классы инструкций (побитовые, унарные, инструкции доступа к памяти), частота использования отдельных инструкций. Преимущество явных векторных представлений заключается в установлении очевидной связи между исходным объектом и соответствующим ему вектором, благодаря чему не составляет труда понять, почему тот или иной фрагмент кода отмечен алгоритмом поиска выбросов как аномальный. С другой стороны, при использовании явных представлений игнорируются потенциально полезные признаки, которые не были учтены при составлении метрик.

Ко второй группе векторных представлений относятся техники из-

[6]https://docs.oracle.com/javase/specs/jvms/se9/html/index.html
[7]https://llvm.org/docs/LangRef.html#instruction-reference

влечения неявных признаков путём преобразования объекта в вектор без подсчёта заранее заданных метрик. Для этого могут быть использованы техники извлечения bag of words [1] и N-грамм [2], применяемые при обработке естественных языков. N-грамм представление позволяет рассмотреть связи между соседними узлами или инструкциями, но теряет информацию о структуре кода вцелом. Также могут использованы представления с помощью автокодировщиков, применяемые для вычисления схожести языковых конструкций [8]. Для синтаксических деревьев предложены специфические способы векторизации методами хэширования [5] и глубокого обучения [3]. Для представления LLVM IR и байт-кода, имеющих линейную структуру, допускается использование любых методов, применимых к массивам данных или деревьям.

Таким образом, неявные векторные представления сохраняют большое количество информации о кодируемых объектах и позволяют рассмотреть их более сложные признаки, которые могут быть трудны для реализации с помощью метрик. Однако, результатом кодирования являются векторы высокой размерности, что влияет на скорость работы и эффективность многих алгоритмов поиска выбросов и поэтому требует дополнительного использования методов уменьшения размерности данных.

1.2. Поиск аномалий

Статистические методы поиска аномалий дают хорошие результаты при анализе временных рядов [6] и в том случае, если при построении модели исходить из предположения, что данные удовлетворяют заранее определённому распределению [4].

Также нужно учесть, что в контексте данной работы необходимы методы, способные давать положительный результат на неразмеченных наборах данных высокой размерности. Поэтому следует рассмотреть алгоритмы детектирования выбросов, основанные на методах машинного обучения без учителя. Основные алгоритмы, относящиеся к этому классу, рассмотрены далее.

1.2.1. Local Outlier Factor (LOF)

Основной идеей является расчёт меры аномальности для объекта на основе его расстояния в какой-либо метрике до k ближайших соседей в наборе данных [13]. Точнее, рассматривается локальная плотность точки: коэфициент достижимости точки из k ближайших к ней. Мерой анномальности точки A считается как отношение локальной плотности A к средней локальной плотности в этой области. Благодаря такому подсчёту значений, точка a1 на рисунке 1 будет определяться как аномалия. Однако, как и многие другие методы, обнаруживающие выбросы с помощью расстояний до ближайших соседей, LOF показывает плохие результаты на разреженных данных высокой размерности [9]. Для улучшения результатов работы алгоритма требуется нормализовать данные и уменьшить количество признаков. Также следует рассмотреть сэмплирование набора данных, то есть запусить LOF на нескольких подвыборках с последующим усреднением результатов.

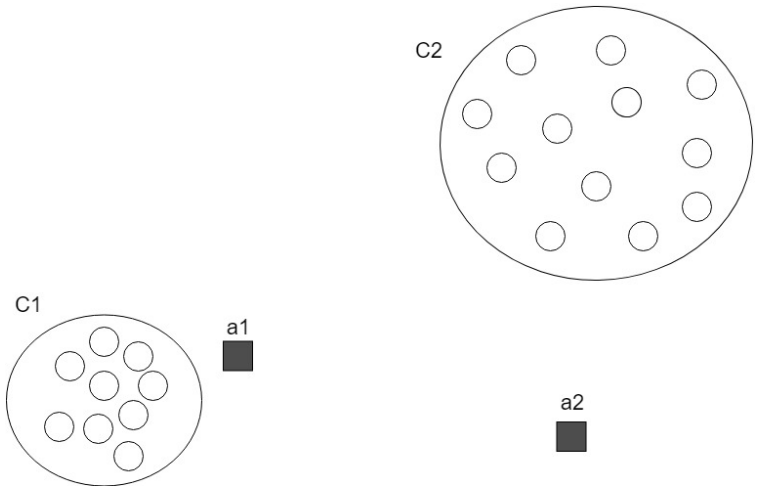

Рис. 1: a1, a2 - аномалии

1.2.2. Isolation Forest (IF)

В отличие от LOF и подобных ему алгоритмов, IF не использует для вычисления аномалий расстояние между объектами. IF работает по принципу случайного леса, отделяя объект от выборки [16]. Каждое изолирующее дерево строится следующим образом: на каждом ветвле-

нии случайно выбирается признак и его значение, по которому делится выборка. Процесс продолжается, пока не будут разделены все элементы. Мерой нормальности объекта является средняя по всем деревьям длина пути от корня. Соответсвенно, чем раньше объект будет отделяться от выборки, тем больше будет его мера аномальности.

1.2.3. Автокодировщик

Также к задаче поиска выбросов применимы автокодировщики, нейронные сети прямого распространения [15, 11]. Простейшая архитектура автокодировщика представляет собой входной и выходной слой с одинаковым количеством нейронов и скрытый слой, обязательно содержащий меньше нейронов, чем на входе и выходе. Задача автокодировщика - научиться отображать на выходе значение, максимально близкое к значению на входе. Таким образом, в результате обучения скрытый слой будет содержать код (латентное пространство) входных данных размерности, равной числу нейронов на скрытом уровне. Далее латентное пространство может быть использовано для определения аномальности объектов.

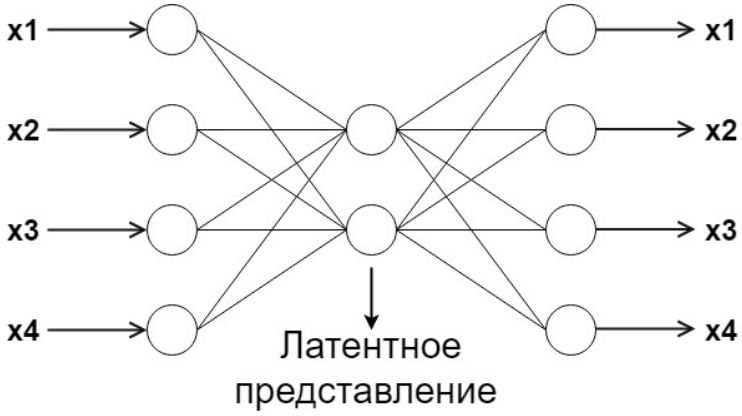

Рис. 2: Автокодировщик

Рассмотренные алгоритмы используют различные подходы к обнаружению выбросов и показывают хорошие результаты при анализа данных малой размерности. Они также могут быть эффективны при анализе данных высокой размерности, если произвести предобработку данных и настроить параметры запуска алгоритмов [12].

2. Анализ байт-кода

В ходе предыдущих работ в области поиска кодовых аномалий языка Kotlin [18, 17] были разработаны инструменты для получения исходного кода на языке Kotlin со всех открытых репозиториев и .class-файлов (байт-кода) релизов проектов, выложенных на GitHub. Также в работе [17] рассмотрен поиск аномалий в байт-коде путем преобразования инструкций в список N-грамм и последующим запуском автокодировщика, реализованного с использованием фреймворка Keras[8]. В результате были получены кодовые аномалии, состоящие из файла с исходным кодом и соответсвующего ему байт-кода. Далее вручную производилась классификация аномалий и анализ причин их возникновения. Процесс обработки байт-кода представлен на рисунке 3. Также были исследованы условные аномалии: фрагменты коды, которые являются аномальными на байт-коде, но не являются аномальнымми на синтаксических деревьях. В данной работе это исследование продолжается с доработкой механизмов извлечения признаков и применением новых методов поиска аномалий.

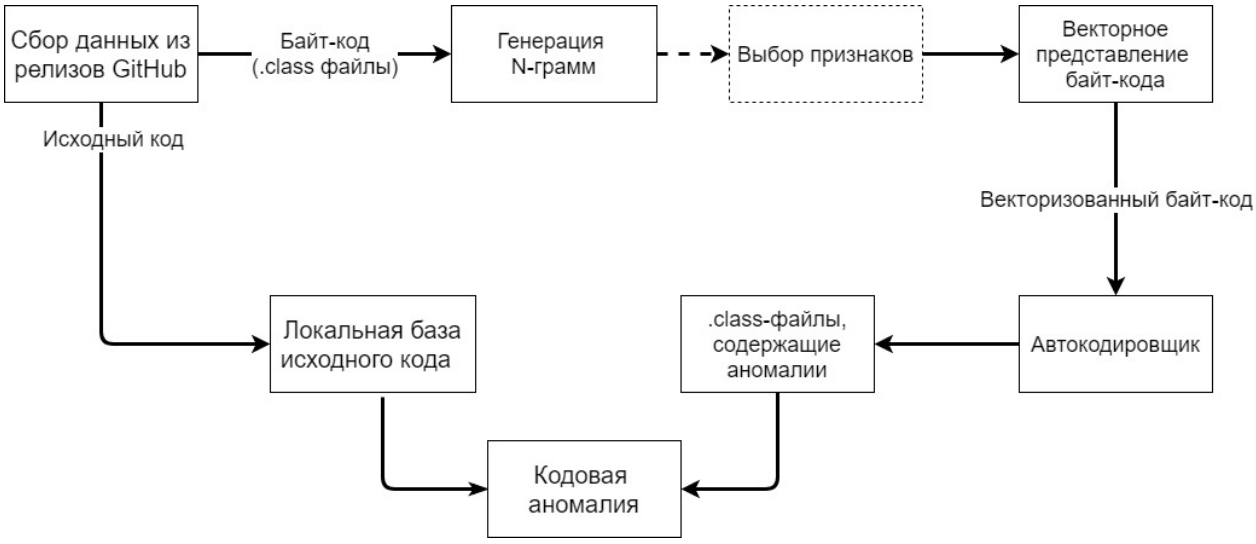

Рис. 3: Процесс поиска аномалий

[8]https://keras.io/

2.1. Структурные единицы кода

Поиск кодовых аномалий может проиизводиться на разных уровнях, начиная от строк и заканчивая файлом. В диссертации [17] единицей исходного кода для извлечения признаков из байт-кода были выбраны классы. Такой подход был обусловлен тем, что классы содержат достаточно много информации для анализа и позволяют найти аномалии как в функциях, так и в конструкторах. С другой стороны, нередко классы содержат сотни строк кода, что затрудняет классификацию и оценку аномалий. Поэтому в данной работе было принято решение произвести поиск аномалий на функциях, фактически отвечающих за единственную операцию в рамках класса, а также в среднем содержащих значительно меньше строк кода.

2.2. Генерация N-грамм

Алгоритмы векторного представления с помощью N-грамм приведены в работе [17]. Для кодирования синтаксического дерева производится обход дерева и на основе связи ”родитель-потомок” между узлами генерируются 1-граммы, 2-граммы, ..., N-граммы. Для N=3 данный процесс изображён на рис. 4.

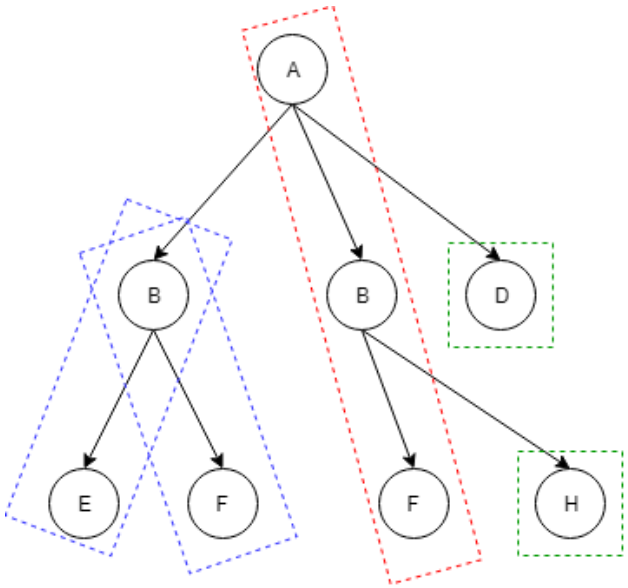

Рис. 4: Представление синтаксических деревьев

Алгоритм приведения последовательности инструкций байт-кода к вектору приведён на рисунке 5: для N=3 берётся окно из первых 3 инструкций и на их основе генерируются три 1-граммы, две 2-граммы и одна 3-грамма, затем окно смещается на одну инструкцию вправо, после чего строятся одна 1-грамма, одна 2-грамма, одна 3-грамма.

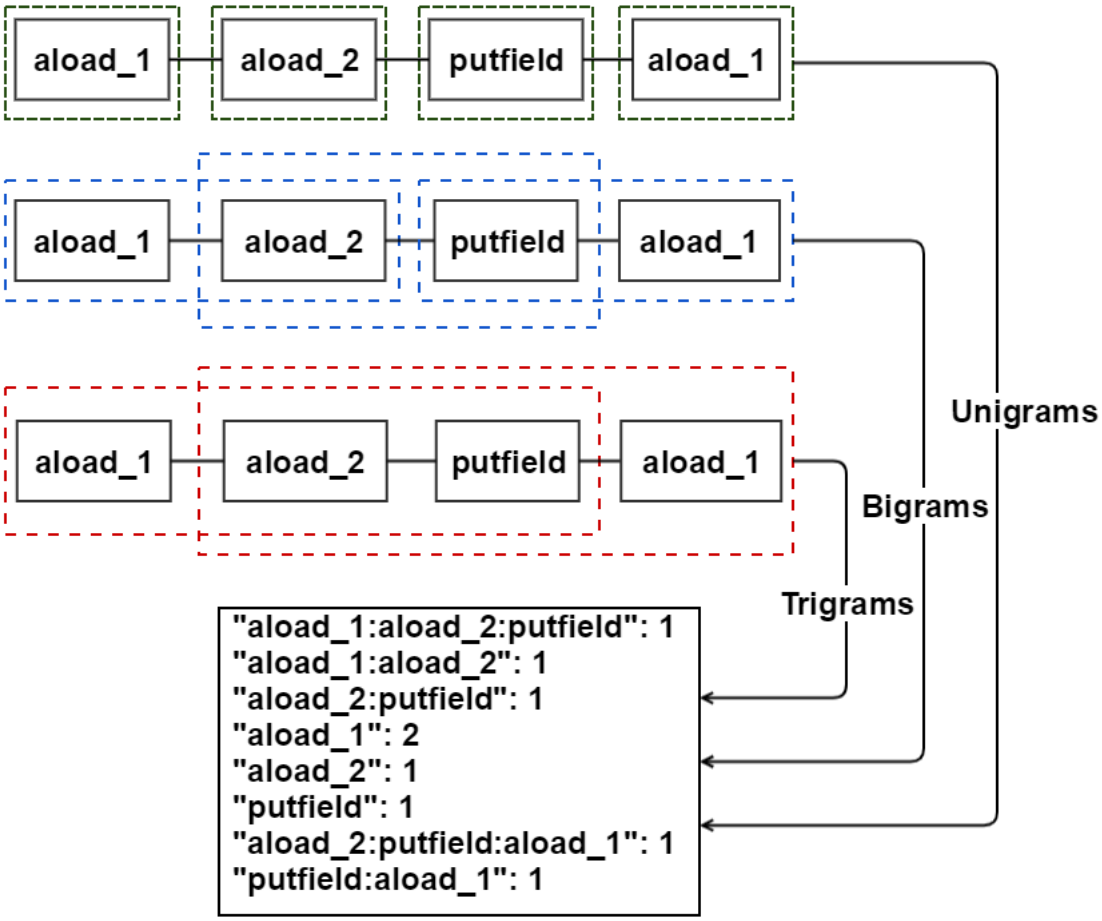

Рис. 5: Представление байт-кода

2.3. Извлечение признаков

При векторном представлении с помощью N-грамм количество сгенерированных признаков при достаточно большом наборе данных может доходить до десятков тысяч, что значительно замедляет работу алгоритмов поиска выбросов в данных, а также ухудшает качество их работы вследствие зашумленности пространства признаков. В диссер-

тации [17] для уменьшения количества признаков удалялись наиболее часто и редко встречающиеся N-граммы. В данной работе для упрощения поиска наиболее релевантных признаков предлагается рассмотреть объединение N-грамм в группы, применив несколько эвристик:

1. Заменить инструкции вида iload1, iload2, iload3, iload4, которые выполняют одно и то же действие для различных локальных переменных, на инструкцию iload. Аналогично поступить с инструкциями store, return, const.

2. Заменить инструкции iload, dload, lload, fload, выполняющие одинаковое действие для различных типов данных, на load.

3. Заменить инструкции для сравнения значений if_icmpeq, if_icmpge, if_icmpgt, if_icmple на if.

Преобразование N-грамм, содержащих эти инструкции, к общему виду, не зависящему от типа данных и локальных переменных, позволяет в несколько раз сократить количество используемых признаков. Затем отбрасываются самые часто и редко встречающиеся N-граммы и производится стандартизация признаков с целью усреднения влияния N-грамм на выявление аномалий. В заключении к данным применяется метод главных компонент для уменьшения размерности с минимальной потерей информации.

2.4. Условные аномалии

В дипломных работах [17, 18] подробно рассматривались кодовые аномалии на синтаксических деревьях, то есть сложные или необычные с точки зрения синтаксиса фрагменты кода. При этом 7 самых крупных изученных классов аномалий также содержали в себе примеры с аномальным байт-кодом. Поэтому в данной работе было принято решение подробнее изучить условные аномалии, фрагменты кода, имеющие нормальное синтаксическое дерево и аномальный байт-код. Процесс их нахождения изображён на диаграмме 6: из репозиториев GitHub, содержащих релизы проектов, извлекаются синтаксические деревья из

исходного кода и байт-код из выложенных .jar и .apk архивов при помощи инструментов, реализованных в диссертации [17]. Затем для обоих представлений детектируются выбросы и из аномалий, найденных в байт-коде, удаляются те примеры, что имеют аномальное синтаксическое дерево.

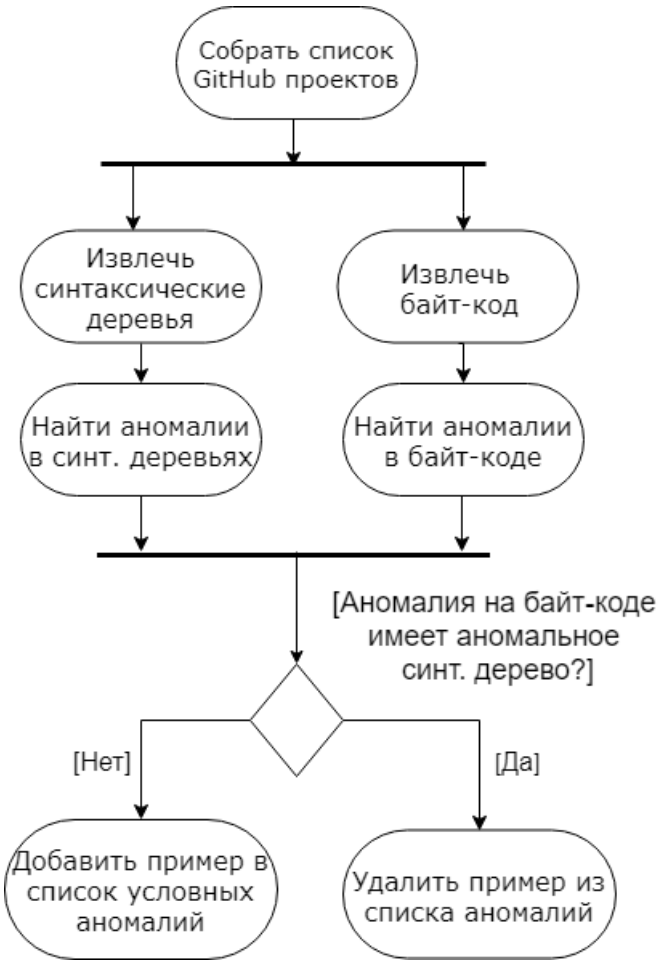

Рис. 6: Диаграмма деятельности для поиска условных аномалий

3. Аномалии в Kotlin/Native проектах

Для поиска аномалий в Kotlin/Native проектах в рамках данной работы реализован инструмент для сбора и векторного представления LLVM IR. Алгоритм его работы представлен на рисунке 7.

Рис. 7: Алгоритм сбора LLVM IR

3.1. Сбор репозиториев

С помощью модуля сбора репозиториев с сервиса GitHub, реализованного в работе [18], был получен список проектов, основным языком которых является Kotlin. Для сборки Kotlin/Native проектов используется система автоматиечской сборки Gradle, поэтому далее с помощью GitHub API[9] осуществляется скачивание файлов сборки проектов (build.gradle файлов) и в них отыскиваются специфичные для Kotlin/Native проектов плагины и этапы сборки. Если такие конструкции найдены, то производится скачивание исходного кода проекта.

3.2. Извлечение LLVM IR

Следующим этапом является сборка Kotlin/Native проектов с добавлением дополнительных параметров к компилятору Kotlin для вывода сгенерированного LLVM IR в заданный файл. Для этого реализован плагин-обёртка для kotlin-multiplatform плагина, который во время

[9]https://developer.github.com/v3/

сборки при вызове компилятора и добавляет к нему аргумент '-Xprint-bitcode', позволяющий выводить LLVM IR в стандартный поток ошибок, который перенаправляется в файл. В результате, каждому проекту соответствует один файл, содержащий в себе несколько модулей LLVM IR. С помощью стандартных утилит LLVM IR преобразуется в биткод, который требуется для более подробного изучения примеров аномалий.

3.3. Извлечение признаков

Для анализа LLVM IR существует интерфейс, предоставляющий доступ как к собранным модулям приложения, так и к отдельным его функциям и переменным. С помощью данного интерфейеса производится векторизация LLVM IR: из модулей последовательно извлекаются функции, для каждой из которой строятся N-граммы. Алгоритм их построения аналогичен алгоритму для представления байт-кода, рассмотренному в главе 2.2. Такой способ векторизации был выбран, так как он был успешно применён для анализа байт-кода в предыдущих работах.

Таким образом, входными данными для поиска аномалий являются функции, представленные с помощью N-грамм в виде векторов. Поиск аномалий производится алгоритмами, рассмотренными в главе 1.2. Результатом их работы являются списки потенциальных аномалий, содержащих сигнатуры функций в специфическом для LLVM IR виде.

3.4. Постобработка

Для каждой из полученных ранее сигнатур происходит поиск исходного кода на языке программирования Kotlin, версия LLVM IR, содержащая определение искомой функции и используемых в ней переменных, а также соответсвующий LLVM биткод. Каждое из этих представлений по-своему полезно для классификации и оценки аномалий. LLVM IR позволяет подробно рассмотреть, каким образом компилятор преобразовал потенциально аномальный исходный код. Однако, чтение LLVM IR является достаточно сложным процессом, требующим боль-

шого количества времени. Вместо этого эффективнее использовать инструменты, использующие LLVM биткод в качестве входных данных и позволяющие получить информацию о скомпилированном коде в автоматическом режиме [14].

4. Эксперименты

4.1. Метод оценки

Основной задачей, поставленной в данной работе, является предоставление разработчикам языка Kotlin и проекта Kotlin/Native отчёта, содержащего набор кодовых аномалий. Для отбора аномалий, представляющих наибольший интерес для разработчиков, из примеров, полученных применением алгоритмов поиска выборосов, предлагается произвести следующие шаги:

1. Каждому найденному примеру сопоставить пару из его исходного кода и .class-файла или LLVM IR, в зависимости от анализируемых данных.

2. Вручную разделить полученные примеры на классы потенциальных аномалий.

3. Из полученных классов выбрать потеницальные аномалии, наиболее ярко описывающие свой класс или имеющие специфические языковые конструкции, для оценки экспертами.

4. Предоставить экспертам из состава разработчиков компилятора Kotlin отчёт и получить оценки для найденных классов аномалий.

4.2. Аномалии в байт-коде

Для поиска аномалий в байт-коде функций были рассмотрены релизы проектов на языке Kotlin, выложенные на сервис GitHub с 2009 года по март 2019 года. Полученный набор данных состоял из 487 проектов, из которых были извлечены 40,352 классов и 169,199 функций. Однако, после удаления функций, содержащих менее 20 инструкций, выборка сократилась до 33,840 примеров. В ходе векторного представления функций было извлечено 72,238 уникальных N-грамм (N=3), которые с помощью эвристик были преобразованы в 7,851 признак. Далее были удалены самые часто и редко встречаемые N-граммы и применён

метод главных компонент, после чего размерность векторов уменьшилась до 550. В результате применения алгоритмов Isolation Forest, LOF и автокодировщика к данным были отобраны 117 примеров аномалий, вручную разделённые автором на 15 классов.

4.2.1. Экспертная оценка

Из 117 найденных аномалий для каждого класса были отобраны от 1 до 4 примеров аномалий. Отобранные 39 аномалий были направлены двум экспертам, разработчикам компилятора Kotlin, для оценки каждого примера по шкале от 1 до 5. В таблице 1 перечислены классы условных аномалий, количество примеров N, использованных для оценки класса, и оценка класса Q. Для 5 из 15 классов были получены оценки 3, 4 и 5.

№	Описание класса аномалий	N	Q
1	**Использование рефлексии**	**3**	**5**
2	**Много аргументов конструктора**	**4**	**4**
3	**Безопасные вызовы функций**	**2**	**3**
4	**Последовательности nop-инструкций**	**3**	**3**
5	**Много вложенных вызовов функций**	**3**	**3**
6	Сгенерировано много байт-кода (от-но объёма исходного кода)	2	2
7	Арифметические выражения	2	2
8	Много аргументов функций	1	2
9	Длинные/сложные логические выражения	3	2
10	Вложенные циклы	2	2
11	Много return-операторов с аннотациями	3	1
12	Частое использование float-литералов	2	1
13	Длинные цепочки вызовов методов	4	1
14	Лямбда-функции в качестве аргументов	2	1
15	Много похожих вызовов методов или функций	3	1

Таблица 1: Экспертная оценка условных аномалий в байт-коде

4.2.2. Сравнение с другими работами

В ходе предыдущих работ по поиску кодовых аномалий выбирались различные структурные единицы программ, представления исходного

кода и способы векторизации данных. В таблице 2 представлена сравнительная характеристика четырёх работ: W_1 - данная работа, W_2 - магистерская диссертация из Университета ИТМО [17], в которой применяется неявное представление байт-кода и синтаксических деревьев, W_3 - выпускная квалифицкационная работа из СПбГУ[18], в которой для поиска аномалий применялись метрики (49 кодовых метрики и 2 метрики синтаксического дерева), W_4 - магистерская диссертация из СПбГУ, выполняемая одновременно с данной работой, в которой для векторного представления используется токенизация.

	$\mathbf{W_1}$	$\mathbf{W_2}$	$\mathbf{W_3}$	$\mathbf{W_4}$
Структурная единица	функция	класс	функция	файл
Представление	байт-код	байт-код синтакс. деревья	исходный код синтакс. деревья	исходный код
Способ векторизации	N-граммы	N-граммы	кодовые метрики	токенизация
Условные аномалии	+	+	-	-

Таблица 2: Сравнение работ по поиску кодовых аномалий

В таблице 3 символом "+" в соответсвующем столбце отмечено, в каких из работ были представлены классы аномалий, исследуемые в данной работе, в столбце Q указана экспертная оценка, поставленная классу аномалий в рамках данной работы . Из 15 рассмотренных классов 8 не встречались в аналогичных работах, остальные встречались только в одной из работ.

Таким образом, благодаря применению различных методов поиска аномалий, удалению из выборки потенциальных аномалий, имеющих сложное синтаксическое дерево, и анализу байт-кода на уровне функций, что значительно упрощает классификацию аномалий, были найдены 8 новых классов аномалий и новые примеры для классов, которые были представлены одним или двумя примерами в прошлых работах.

Описание класса аномалий	$\mathbf{W_1}$	$\mathbf{W_2}$	$\mathbf{W_3}$	$\mathbf{W_4}$
Использование рефлексии	+			
Много аргументов конструктора	+			
Безопасные вызовы функций	+	+		
Последовательности nop-инструкций	+			
Много вложенных вызовов функций	+		+	
Сгенерировано много байт-кода	+			
Арифметические выражения	+			+
Много аргументов функций	+			
Длинные/сложные логические выражения	+	+		
Вложенные циклы	+	+		
Много return-операторов с аннотациями	+			
Частое использование float-литералов	+			
Длинные цепочки вызовов методов	+	+		
Лямбда-функции в качестве аргументов	+			
Много похожих вызовов методов или функций	+		+	

Таблица 3: Сравнение результатов с аналогичными работами

4.3. Аномалии в LLVM IR

Для поиска аномалий в Kotlin/Native проектах с помощью анализа LLVM IR были рассмотрены проекты, опубликованные на сервисе GitHub с сентября 2018 года по апрель 2019 года. Это обусловлено тем, что в сентябре 2018 года были выпущены достаточно стабильные плагины для сборки проектов, а в октябре 2018 года была выпущена бета-версия Kotlin/Native. В результате удалось собрать LLVM IR из 63 проектов, содержащих 9,498 функций. Из полученных функций в процессе векторизации были получены 6109 уникальных N-грамм (N=3), 250 из которых были отобраны в качестве признаков для обнаружения аномалий. В результате, были получены 73 аномалии, разделённый автором работы на 9 классов. Каждому примеру соответствовал исходный код, LLVM IR и LLVM биткод.

4.3.1. Экспертная оценка

Аналогично эксперименту с байт-кодом, для оценки экспертами были отобраны от двух до четырёх примеров аномалий. В результате для

Описание класса аномалий	N	Q
Cinterop	3	3
Использование обобщённых типов	3	2
Много побитовых операций	2	2
Арифметические выражения	2	2
Длинные цепочки вызовов методов	2	
Сложные определения объектов	2	
Coroutines	4	
Сложные when-выражения	3	
Сложный поток управления	4	

Таблица 4: Экспертная оценка аномалий в LLVM IR

оценки экспертом были направлены 25 примеров. Окончательные оценки классов аномалий приведены в таблице 4.

5. Заключение

В ходе данной работы были получены следующие результаты:

- На наборе данных, состоящем из 487 проектов, были собраны 16 классов аномалий на основе байт-кода, 8 из которых не встречались в предыдущих исследованиях.
- Реализованы инструменты для сбора LLVM IR из Kotlin/Native проектов и его векторного предствления.
- С помощью разработанных инструментов был проведён поиск аномалий в Kotlin/Native проектах. Были собраны 73 аномалии, разделённые на 9 классов.
- Предоставлен отчёт разработчикам языка программирования Kotlin, содержащий примеры аномалий, получивших высокую экспертную оценку.

Список литературы

[1] Bimodal Modelling of Source Code and Natural Language / Miltos Allamanis, Daniel Tarlow, Andrew Gordon, Yi Wei // Proceedings of the 32nd International Conference on Machine Learning / Ed. by Francis Bach, David Blei. — Vol. 37 of Proceedings of Machine Learning Research. — Lille, France : PMLR, 2015. — 07–09 Jul. — P. 2123–2132. — URL: http://proceedings.mlr.press/v37/allamanis15.html.

[2] Bugram: Bug Detection with N-gram Language Models / Song Wang, Devin Chollak, Dana Movshovitz-Attias, Lin Tan // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. — ASE 2016. — New York, NY, USA : ACM, 2016. — P. 708–719. — URL: http://doi.acm.org/10.1145/2970276.2970341.

[3] Building Program Vector Representations for Deep Learning / Hao Peng, Lili Mou, Ge Li et al. // Knowledge Science, Engineering and Management / Ed. by Songmao Zhang, Martin Wirsing, Zili Zhang. — Cham : Springer International Publishing, 2015. — P. 547–553.

[4] Chandola Varun Banerjee Arindam Kumar Vipin. Anomaly Detection: A Survey. ACM Comput. Surv. — ACM Computing Surveys (CSUR), 2009. — URL: https://dl.acm.org/10.1145/1541880.1541882.

[5] DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones / L. Jiang, G. Misherghi, Z. Su, S. Glondu // 29th International Conference on Software Engineering (ICSE'07). — 2007. — May. — P. 96–105.

[6] David Goldberg Yinan Shan. The Importance of Features for Statistical Anomaly Detection. — USENIX HotCloud workshop, 2015.

[7] De-anonymizing Programmers via Code Stylometry / Aylin Caliskan-Islam, Richard Harang, Andrew Liu et al. // Proceedings of the 24th USENIX Conference on Security Symposium. — SEC'15. — Berkeley,

CA, USA : USENIX Association, 2015. — P. 255–270. — URL: http://dl.acm.org/citation.cfm?id=2831143.2831160.

[8] Deep Learning Similarities from Different Representations of Source Code / Michele Tufano, Cody Watson, Gabriele Bavota et al. // Proceedings of the 15th International Conference on Mining Software Repositories. — MSR '18. — New York, NY, USA : ACM, 2018. — P. 542–553. — URL: http://doi.acm.org/10.1145/3196398.3196431.

[9] Hinneburg Alexander, Aggarwal Charu C., Keim Daniel A. What Is the Nearest Neighbor in High Dimensional Spaces? // Proceedings of the 26th International Conference on Very Large Data Bases. — VLDB '00. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000. — P. 506–515. — URL: http://dl.acm.org/citation.cfm?id=645926.671675.

[10] Hodge V., Austin J. A survey of outlier detection methodologies. Artificial Intelligence Review. 2004 ; Vol. 22, No. 2. pp. 85-126.

[11] Jerone T. A. Andrews Edward J. Morton, Griffin Lewis D. Detecting Anomalous Data Using Auto-Encoders. — 2016.

[12] Liu Fei Tony, Ting Kai Ming, Zhou Zhi-Hua. Isolation-Based Anomaly Detection // ACM Trans. Knowl. Discov. Data. — P. 3:1–3:39. — URL: http://doi.acm.org/10.1145/2133360.2133363.

[13] Markus M. Breunig Hans-Peter Kriegel. LOF: identifying density-based local outliers. — ACM SIGMOD international conference on Management of data.

[14] Schubert Philipp Dominik, Hermann Ben, Bodden Eric. PhASAR: An Inter-procedural Static Analysis Framework for C/C++ // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by Tomáš Vojnar, Lijun Zhang. — Cham : Springer International Publishing, 2019. — P. 393–410.

[15] Zhou Chong, Paffenroth Randy C. Anomaly Detection with Robust Deep Autoencoders // Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. — KDD '17. — New York, NY, USA : ACM, 2017. — P. 665–674. — URL: http://doi.acm.org/10.1145/3097983.3098052.

[16] Zhou Fei Tony Liu ; Kai Ming Ting ; Zhi-Hua. Isolation Forest. — Eighth IEEE International Conference on Data Mining, 2008.

[17] В.А. Петухов. Магистерская диссертация "Обнаружение проблем производительности в программах на языке программирования Kotlin с использованием статического анализа кода". — Университет ИТМО, 2018.

[18] К.П. Смиренко. Выпускная квалификационная работа "Детектор аномалий в программах на языке Kotlin". — СПбГУ, 2018.