Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Строганов Никита Сергеевич

Выпускная квалификационная работа

Разработка статического анализатора на основе символьного исполнения

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель:
доцент, факультет математики и компьютерных наук, к.ф.-м.н., Д.С. Шалымов

Рецензент:
ассистент, Институт компьютерных наук и технологий СПбПУ, В. Соболь

Санкт-Петербург
2023 г.

Содержание

Введение

Сложно представить себе разработку программного обеспечения в современном мире без применения инструментов анализа кода. По мере того как программы становятся сложнее, возрастает и сложность их тестирования, а значит возрастает и потребность в автоматизации поиска ошибок или дефектов. Одним из подходов к решению этой задачи является статический анализ.

Статический анализ — это анализ программного обеспечения, производимый (в отличие от динамического анализа [1]) без реального выполнения исследуемых программ. Чаще всего анализ производится над исходным кодом продукта, хотя иногда задействуется и какой-нибудь вид объектного кода, например, байт-код языка программирования Java. Таким образом, методы статического анализа помогают найти ошибки в коде ещё на этапе компиляции, что значительно ускоряет разработку программ, снижая потраченное программистом время на отладку и тестирование.

В связи с высокой востребованностью темы автоматического обнаружения дефектов в программах, на рынке существует огромное количество различных статических анализаторов. Список из наиболее известных инструментов, позволяющих анализировать Java код включает в себя SpotBugs [2], Coverity [3], CodeQL [4], Infer [5], а также многие другие. Перечисленные инструменты неплохо справляются с поиском простых ошибок или опечаток, однако в большинстве своём, не выполняют глубокого исследования кода, то есть не обнаруживают редкие сценарии исполнения, приводящие к некорректному поведению программы. Кроме того, часто допускается большое количество ложноположительных срабатываний. Другими словами, инструмент может выдать предупреждение о найденной ошибке, которой на самом деле не существует. Это, конечно же, негативно сказывается на производительности программиста, который теперь вынужден тратить время на рассмотрение выданного предупреждения.

Таким образом, существует запрос на разработку инструмента, который бы выполнял глубокий анализ и находил серьезные проблемы в коде, приводящие к неожиданному поведению программы. В достижении этой цели может

помочь символьное исполнение.

Техника символьного исполнения заключается в том, чтобы исполнять программу не на конкретных значениях входных данных, а на так называемых символьных переменных. Благодаря этому, для каждого пути исполнения можно получить логические ограничения на значения входных параметров, при выполнении которых этот путь достигается. После этого при помощи SMT[1]-решателя такие значения могут быть найдены, и сгенерирован тестовый случай, реализующий данный путь. Описанный механизм обладает таким свойством, как теоретически полное покрытие кода, что позволяет обнаруживать редкие сценарии исполнения, которые приводят к неправильному результату работы программы. Ещё одно немаловажное достоинство символьного исполнения заключается в сравнительно небольшом количестве ложных срабатываний относительно других методов статического анализа [6].

Для оценки качества анализатора кода обычно используется реестр самых опасных уязвимостей программного обеспечения [7]. Список включает в себя такие ошибки как запись или чтение вне границ массива, SQL-инъекция, использование непроверенного пользовательского ввода, разыменование нулевого указателя, а также многие другие уязвимости, позволяющие злоумышленникам полностью захватить систему, украсть данные или помешать работе приложений. Символьное исполнение способно находить далеко не все из перечисленных ошибок, однако его возможности могут быть существенно расширены с помощью метода taint-анализа. Метод заключается в отслеживании распространения непроверенных внешних данных по программе и позволяет обнаруживать нарушения в безопасности, например, инъекции в базу данных, операционную или файловую систему. Таким образом, если встроить taint-анализ непосредственно в ядро символьной виртуальной машины, то можно объединить достоинства каждого из подходов, а именно, получить анализатор с низким уровнем ложноположительных срабатываний, который, помимо всего прочего, умеет находить проблемы в безопасности.

В рамках данной работы в качестве символьной виртуальной машины был взят UnitTestBot [8] — инструмент, который по исходному коду на языке

[1]Satisfiability Modulo Theory, задача разрешимости для логических формул с учётом лежащих в их основе теорий

Java автоматически генерирует тесты, пытаясь максимизировать покрытие. UnitTestBot может также находить тестовые случаи, приводящие к таким дефектам в программном обеспечении, как выбрасывание необработанного исключения, переполнения примитивных численных типов, зависания методов и некоторые другие. Однако сами по себе найденные значения входных параметров тестируемой функции ещё не являются приемлемым результатом статического анализа, поскольку качество анализатора измеряется не только в его способности находить неочевидные сценарии исполнения, но и в удобстве использования для программиста. Следовательно, выдаваемое предупреждение о найденной ошибке, помимо краткого описания самой ошибки, должно содержать конкретное место в коде, а так же всю ту информацию, которая может помочь пользователю разобраться в проблеме как можно быстрее.

Немаловажным фактором удобства использования инструмента является простота его установки и запуска. В современном мире большинство программистов используют интегрированные среды разработки (англ. IDE), которые дают возможность встраивать в себя сторонние модули для решения каких-либо задач. Например, UnitTestBot имеет собственный плагин для IntelliJ IDEA [9] — одной из наиболее популярных IDE для языка Java. В связи с этим было решено добавить функциональность выполнения статического анализа, а также отображения его результатов непосредственно в плагин UnitTestBot.

Постановка задачи

Целью данной работы является реализация статического анализатора Java кода на основе инструмента для автоматической генерации тестов UnitTestBot. Для достижения цели были сформулированы следующие задачи.

- Провести обзор предметной области и подходов существующих статических анализаторов с целью выявления их достоинств и недостатков.
- Реализовать модуль, который на основе тестов, сгенерированных инструментом UnitTestBot, будет предоставлять пользователю отчёт с найденными ошибками.
- Разработать пользовательский интерфейс для просмотра отчёта статического анализа и встроить его в плагин UnitTestBot для IntelliJ IDEA.
- Модифицировать ядро символьной виртуальной машины, встроив в него технику taint-анализа, которая позволит обнаруживать нарушения безопасности в коде.
- Протестировать разработанный статический анализатор для определения эффективности нахождения ошибок и уязвимостей в различных программах.

1. Обзор предметной области

1.1. Существующие решения

В данном разделе будут описаны существующие инструменты статического анализа, их возможности, преимущества и недостатки.

1.1.1 SpotBugs

Популярный анализатор исходного кода для языка Java. Распространяется в виде плагина для IntelliJ IDEA, задач для систем сборки Ant, Gradle и Maven, а также интерфейса командной строки. Внутри него реализованы различные эвристики, основанные на синтаксическом анализе текста программы. Хорошо справляется с поиском простых ошибок или опечаток, а также плохих практик в написании кода.

Возможности инструмента существенно расширяются при установке дополнения FindSecurityBugs [10], которое реализует некоторые алгоритмы для поиска уязвимостей, в том числе и taint-анализ. Вместе с этим модулем SpotBugs может находить ошибки в безопасности, например использование непроверенного пользовательского ввода в критических секциях программы (HTTP-запросы, работа с файловой системой и так далее).

Основным недостатком данного продукта является большое количество ложных срабатываний (от 40 до 60 процентов [11]), которые возникают из-за применения методов, основанных на исследовании исходного текста программы.

1.1.2 Coverity

Пакет программного обеспечения, состоящий из статического и динамического анализаторов кода на C++, Java, C#, Scala и некоторых других языках. Внутри Coverity реализовано несколько подходов к поиску ошибок, в том числе и taint-анализ, который, прежде всего, используется для нахождения случаев нарушения безопасности, таких как SQL-инъекции или XSS («межсайтовый скриптинг»).

Продукт активно используют для автоматической проверки критически важного кода, например, лаборатория реактивного движения НАСА тестировала с помощью него исходные коды марсохода Curiosity [12].

Как и у многих рассматриваемых инструментов, существенный недостаток Coverity — частые ложноположительные срабатывания. В разных публикациях на тему измерения доли ошибочных предупреждений Coverity приводятся разные данные, но в среднем получается от 10 до 35 процентов [13, 14, 15].

1.1.3 CodeQL

Инструмент статического анализа с открытым исходным кодом, а также поставляемый вместе с ним язык запросов, похожий на SQL, позволяющий проверять некоторые гипотезы относительно кода или идущих через него данных. Например, можно составить запросы на этом декларативном языке, которые проверят есть ли путь в графе потока данных от места пользовательского ввода до небезопасного участка кода, где эти данные применяются. Таким образом удаётся обнаружить такие уязвимости, как SQL-инъекция, XSS, разыменование нулевого указателя и некоторые другие.

Простейший сценарий использования состоит в том, чтобы просто запустить сканер с набором стандартных, написанных создателями, запросов, а затем просматривать результаты, среди которых в основном будут проблемы с качеством кода и проблемы с безопасностью. Однако более продуктивный сценарий включает в себя написание собственных запросов, исходя из специфики конкретного приложения. В этом заключается одновременно и основное преимущество, и главный недостаток данного продукта. С одной стороны пользователю предоставляется большая гибкость в написании запросов под собственные нужды. С другой стороны, чтобы начать эффективно пользоваться инструментом, нужно не просто придумать и написать релевантные для конкретного проекта запросы, а ещё и выучить непростой декларативный язык вместе со всем его спектром возможностей.

Корректно оценить частоту ложноположительных срабатываний может быть сложно, ведь результат напрямую зависит от набора пользовательских

запросов, с которыми запускается CodeQL. Тем не менее существуют исследования, где проводятся подобные сравнения, например, в статье [16] указана доля ошибочных сообщений в 50 процентов, что довольно много.

1.1.4 Infer

Статический анализатор программ для Java, C и Objective-C, который способен отслеживать проблемы, вызванные разыменованиями нулевого указателя, утечками ресурсов и памяти, гонками данных в многопоточной среде и некоторыми другими ошибками.

Infer основан на сепарационной логике [17] и технике bi-abduction [18], что позволяет ему рассуждать о мутациях в памяти компьютера и выполнять глубокий анализ кода на предмет некорректного обращения с памятью. Отметим, что использование формальных подходов способствует меньшему количеству выдаваемых ложноположительных результатов по сравнению с описанными выше инструментами, а именно около 2 процентов [19]. Тем не менее алгоритмы, реализованные внутри Infer, не обладают таким свойством, как теоретически полное отсутствие ложноположительных срабатываний. Другими словами, при любой реализации они будут, в отличие от, например, символьного исполнения, у которого такое свойство есть.

1.1.5 Выводы

Проведённый обзор существующих инструментов статического анализа показал, что многие популярные продукты, во-первых, допускают большое количество ложноположительных срабатываний, а во-вторых, проводят лишь поверхностный анализ, не обнаруживая редкие сценарии исполнения, приводящие к некорректному поведению программы. Одним из способов решения обозначенных недостатков является использование формальных методов математической логики. В данной работе предлагается применить технику символьного исполнения.

1.2. Символьное исполнение

Символьное исполнение [20, 21] — это широко распространенная техника анализа программного обеспечения, которая позволяет для каждого из возможных путей исполнения найти соответствующий ему набор входных данных. В частности, метод способен обнаружить пути, ведущие к неожиданному результату работы программы, например, к выбрасыванию исключения.

1.2.1 Описание алгоритма

Символьное исполнение работает в условиях неопределенности входных данных. Неопределённые данные, другими словами *символы* или *символьные* переменные, являются абстракцией конкретных объектов или переменных, которыми оперирует код во время обычного исполнения. Символы, в отличие от конкретных переменных, могут принимать сразу целое множество значений, которые допускает соответствующий им тип данных. Эти абстракции хранятся в специальной *символьной* памяти, состояние которой поддерживает алгоритм символьного исполнения в процессе своей работы. Помимо символьной памяти `memory`, в поддерживаемое состояние входят следующие данные:

- текущая инструкция программы `stmt`, которую обрабатывает алгоритм.
- условие пути `pathCondition` в виде логической формулы, выражающее ограничения на символьные переменные для текущего пути исполнения.
- путь исполнения `path`, по которому мы пришли в это состояние.

В общих чертах опишем алгоритм символьного исполнения.

Изначально есть одно состояние, которое соответствует состоянию программы при её запуске.

Далее, если алгоритм встречает инструкцию без ветвления, то он просто обновляет текущее состояние с учётом полученной информации. Например, инструкция присваивания $\texttt{x} := \texttt{1}$ вызовет обновление ячейки символьной памяти, которая соответствует переменной `x`.

Если же алгоритм встречает инструкцию с ветвлением, например, $\texttt{if}(\xi)$, то он дублирует текущее состояние state, получая два новых состояния $\texttt{state}_{\texttt{if}}$ и $\texttt{state}_{\texttt{else}}$. В состоянии $\texttt{state}_{\texttt{if}}$ обновляется условие пути так, будто бы ξ выполняется, а в состоянии $\texttt{state}_{\texttt{else}}$ условие пути обновляется так, будто бы ξ не выполняется. Таким образом, получаются два состояния, соответствующие разветвлению. После чего оба условия проверяются на выполнимость, и, в случае невыполнимости какого-то из полученных условий, соответствующее состояние удаляется из рассмотрения.

Работа алгоритма продолжается до тех пор, пока состояние не придёт в конечную инструкцию или исполнение не завершится в результате нахождения ошибки в программе.

После завершения исполнения символьная память и условие пути конечного состояния $\texttt{state}_{\texttt{f}}$ определяют ограничения на входные данные программы. Для того чтобы найти конкретные входные данные, которые удовлетворяют составленным ограничениям, используется SMT-решатель [22]. В итоге программа, запущенная на найденных входных данных, пройдёт тот же путь исполнения, что и конечное состояние $\texttt{state}_{\texttt{f}}$, из которого они были получены.

1.2.2 Пример работы

Для лучшего понимания концепции, рассмотрим пример работы символьной виртуальной машины на функции example.

```
1  int example(int x, int y) {
2      int a = 1;
3
4      if (x > 27) {
5          a += y;
6      } else {
7          a += 12;
8      }
9
10     return y / a;
11 }
```

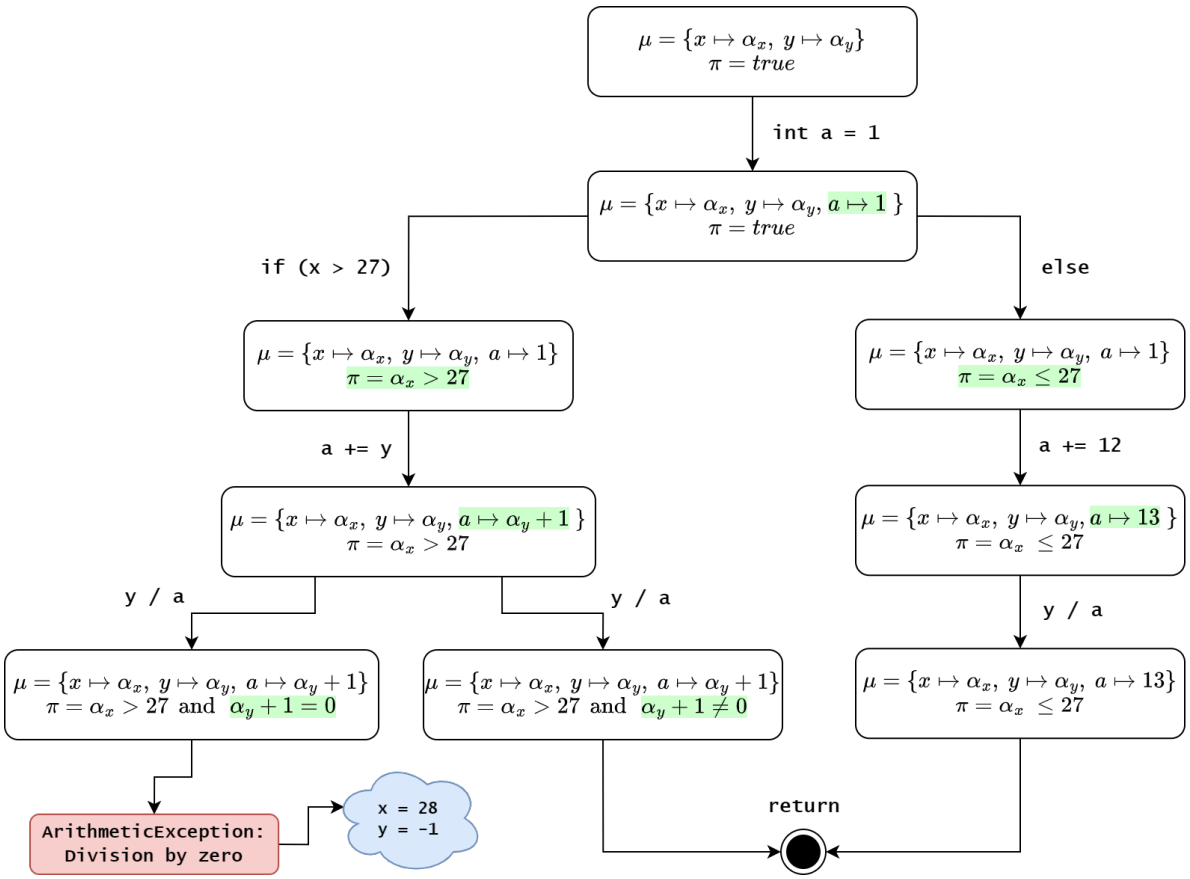

Рис. 1: Схема символьного исполнения функции `example`.

По итогам анализа мы хотим ответить на вопрос, существуют ли какие-нибудь конкретные входные аргументы функции `example`, при которых в строке 10 выбрасывается исключение `ArithmeticException: / by zero`, и, если существуют, то хотим найти их.

В схеме, представленной на рис. 1, прямоугольник — это одно состояние символьной машины. Оно хранит в себе символьную память и условие пути, которые для краткости обозначены μ и π соответственно. Память μ в данном примере хранит для каждой переменной её символьное значение, то есть для x это α_x, а для y — α_y. Условие пути π — это логическая формула со значениями символьных переменных, выполняемая на текущем пути исполнения. Переход от состояния к состоянию происходит при обработке некоторой инструкции исходного кода, записанной на соответствующем ребре графа состояний.

Изначально в μ лежат символьные переменные, на которые ещё не наложено ни одно ограничение, а π — это всегда выполняющаяся формула, то

есть просто $true$.

Обрабатывая инструкции исходного кода, символьная виртуальная машина поддерживает значения μ и π в актуальном состоянии. Например, после строки `int a = 1`, в μ запомнилась переменная a и её значение 1. А после обработки условного оператора `if (x > 27)`, исполнение разветвилось на два пути, в каждом из которых обновилось π: в ветке `if` оно равно $\alpha_x > 27$, а в ветке `else` равно $\alpha_x \leq 27$.

В некоторый момент алгоритм обнаруживает состояние, которое ведёт к выбрасыванию исключения. После этого он обращается к SMT-решателю с запросом найти конкретные значения α_x и α_y, которые выполняют формулу $\pi = \alpha_x > 27 \wedge \alpha_y + 1 = 0$, или же сообщить, что решений не существует. В данном примере формула оказывается выполнима, и SMT-решатель возвращает набор $\alpha_x = 28,\ \alpha_y = -1$. Таким образом, удалось автоматически сгенерировать тестовый случай `example(28, -1)`, который приводит к падению программы из-за необработанного исключения.

1.3. UnitTestBot

UnitTestBot — инструмент, который по исходному коду на языке Java автоматически генерирует модульные тесты, пытаясь при этом максимизировать количество покрытых инструкций. UnitTestBot использует для работы собственную символьную виртуальную машину, что даёт возможность ожидать от сгенерированных тестов теоретически полного покрытия кода.

Продукт поставляется как плагин для среды разработки IntelliJ IDEA, что позволяет запускать генерацию тестов, используя удобный пользовательский интерфейс. Инструмент поддерживает популярные системы сборки Gradle и Maven, поэтому его достаточно легко применить на любом проекте, написанном на языке Java. Помимо всего прочего, UnitTestBot предоставляет пользователям множество конфигурируемых параметров своей работы, гибко настраиваясь под нужды тестирования конкретного проекта. В частности, программист может задать количество времени в секундах, которое будет доступно инструменту для генерации тестов.

1.3.1 Архитектура

В данном подразделе будет проведён обзор общей архитектуры проекта, а также некоторых деталей реализации символьной виртуальной машины, которую использует UnitTestBot.

На рис. 2 представлена высокоуровневая диаграмма ключевых модулей проекта. Рассмотрим каждую компоненту подробнее.

Плагин для IntelliJ IDEA задаёт представление интерфейса в указанной среде разработки. Модуль собирает с пользовательского проекта все нужные для генерации тестов данные, например структуру классов или файлы с Java байт-кодом. Другими словами, является точкой входа в приложение, позволяя настраивать и запускать генерацию тестов.

Пакет `UTBot API` отвечает за формирование входных данных для генерации тестов, запуск символьной виртуальной машины и возвращение результата её работы в удобном для вызывающего кода виде. Типичный сценарий использования данного модуля выглядит следующим образом. Плагин для IntelliJ IDEA совершает запрос на генерацию тестов, передавая байт-код

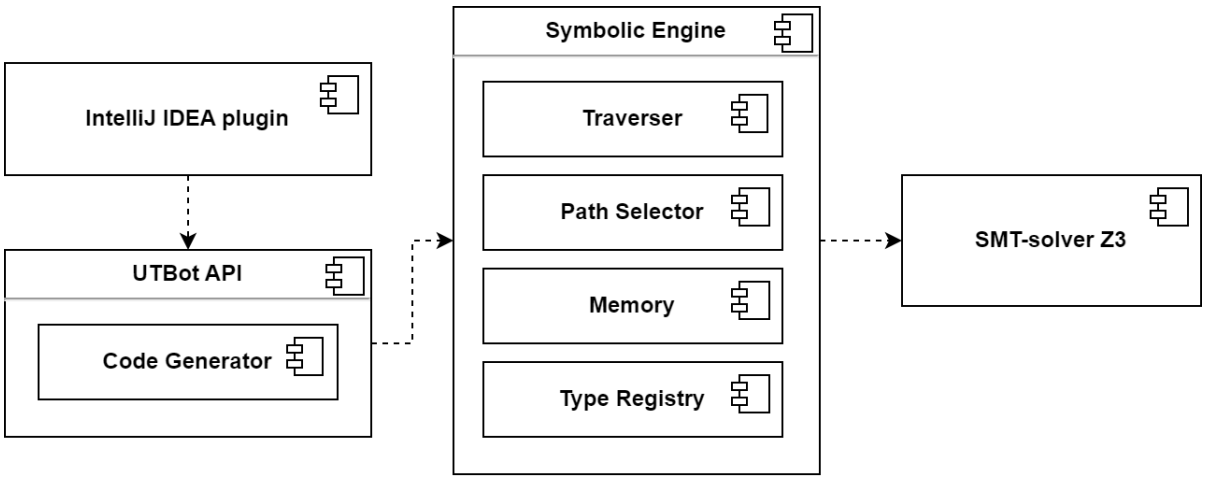

Рис. 2: Диаграмма основных модулей проекта UnitTestBot.

пользовательских классов и все остальные необходимые данные. После чего описываемый компонент преобразует пришедшие данные в нужный формат и запускает на них символьное исполнение. Результат работы символьной машины поступает в `Code Generator`, который генерирует код тестов на языке Java, соответствующий найденным тестовым случаям. В итоге полученный файл с тестами возвращается в модуль плагина и отображается на экране пользователя.

Модуль `Symbolic Engine` — это символьная виртуальная машина внутри UnitTestBot, центральный и самый сложный компонент проекта. В качестве входных данных, он ожидает байт-код одного метода[2], а результатом исполнения является множество тестовых случаев, то есть наборов входных параметров метода. В процессе своей работы модуль делает запросы к SMT-решателю Z3 [23], как было описано в общем алгоритме символьного исполнения (в разделе 1.2.1).

Работа модуля высокоуровнево выглядит следующим образом. На каждом шаге текущее состояние извлекается из `Path Selector`, затем обрабатывается в `Traverser`, который возвращает несколько новых состояний. Для каждого из полученных состояний проверяется, является ли оно терминальным. Если да, то оно запоминается как результат исполнения, а иначе кладётся в очередь состояний внутри `Path Selector`.

Теперь рассмотрим обязанности наиболее важных классов данного мо-

[2]На самом деле символьная машина работает не с обычным Java байт-кодом, а с упрощённым трёхадресным кодом. Однако для простоты, эта деталь реализации опускается.

дуля подробнее.

- `Path Selector` — это класс, принимающий решение о том, какая инструкция программы должна быть обработана следующей. Хранит внутри себя некоторый список доступных состояний (при старте только одно начальное), позволяет добавлять туда новое состояние, а также извлекать некоторое следующее. Какое именно состояние будет извлечено из списка, определяется на основании реализованной внутри эвристики.

- `Traverser` обрабатывает поданное на вход состояние. Он содержит информацию о графе потока управления, иерархии классов в программе и системе символьных типов. `Traverser`, в зависимости от состояния и текущей инструкции, создаёт набор новых состояний, в которых обновлена символьная память и добавлены необходимые логические ограничения для соответствующего пути исполнения. Описанное множество состояний и является результатом работы класса.

- `Memory` отвечает за символьное представление памяти состояния в программе. Высокоуровнево работает ровно так, как было описано в главе про символьное исполнение (1.2.1).

- `Type Registry` — важный компонент, который содержит информацию о системе типов в анализируемом коде. Даёт возможность узнать тип символьного объекта во время исполнения.

1.3.2 Пример работы

Рассмотрим пример работы инструмента UnitTestBot на уже знакомой функции `example`, которая теперь является методом класса `Main`.

```
1  public class Main {
2
3      int example(int x, int y) {
4          int a = 1;
5
6          if (x > 27) {
7              a += y;
8          } else {
9              a += 12;
10         }
11
12         return y / a;
13     }
14 }
```

Программист запускает генерацию тестов, используя интерфейс плагина для IntelliJ IDEA, при этом указав количество отведённого на генерацию тестов времени 10 секунд.

Конфигурация, Java байт-код, структура классов, а также вся остальная необходимая информация преобразуется в модуле `UTBot API` и передается в `Symbolic Engine` в ожидаемом для него виде. Тот, в свою очередь, с помощью символьного исполнения находит 2 набора входных параметров функции.

1. Набор $x = 27,\ y = -255$ покрывает все строчки, кроме седьмой, и возвращает -19 в качестве результата метода `example`.

2. Набор $x = 28,\ y = -1$ покрывает оставшуюся непокрытой строку номер 7, а также инициирует выброс необработанного исключения `ArithmeticException: / by zero` в строке 12.

Результат работы `Symbolic Engine` передаётся в `Code Generator`, который для данного примера генерирует следующий код.

```
1  import org.junit.jupiter.api.Test;
2
3  import static org.junit.jupiter.api.Assertions.assertEquals;
4
5  public final class MainTest {
6
7      @Test
8      public void testExample1() {
9          Main main = new Main();
10         int actual = main.example(27, -255);
11         assertEquals(-19, actual);
12     }
13
14     @Test
15     public void testExample2() {
16         Main main = new Main();
17         main.example(28, -1);
18     }
19 }
```

Если запустить полученные тесты, то второй завершится с ошибкой. Это является ожидаемым поведением инструмента, ведь программист забыл обработать возможное исключение. Об этом ему сообщается в виде теста, который не проходит.

1.4. Taint-анализ

Taint-анализ [24] — это технология, позволяющая отслеживать распространение непроверенных внешних данных по программе. Попадание таких данных в критические секции кода может привести к различным уязвимостям, включая SQL-инъекцию, межсайтовый скриптинг (XSS) и многие другие. Злоумышленники могут использовать эти уязвимости для нарушения корректной работы системы, получения конфиденциальных данных или проведения других несанкционированных операций. Taint-анализ помогает находить описанные ошибки ещё на этапе компиляции. Отметим, что методика может быть применена и во время непосредственного выполнения программы [25], однако данная работа не будет затрагивать эту область.

1.4.1 Описание алгоритма

Ключевая идея подхода заключается в том, что любая переменная, которая может быть изменена внешним пользователем, представляет потенциальную угрозу безопасности. Если эта переменная используется в некотором выражении, то значение этого выражения также становится подозрительным. Далее, алгоритм отслеживает и оповещает о ситуациях, когда какая-либо из этих *помеченных* переменных используется для выполнения опасных команд, например, прямых запросов в базу данных или в операционную систему компьютера.

Для своей работы, taint-анализ требует конфигурацию, в которой методам программы можно назначить одну из следующих ролей.

- `Taint source` — *источник* непроверенных данных. К примеру, это может быть метод считывания пользовательского ввода или же метод получения параметра пришедшего HTTP-запроса. Переменные, в которые записывается результат выполнения `Taint source`, называются *помеченными*. Семантика метода определяет, какая именно метка будет наложена на переменную. Причём имя метки может быть совершенно произвольным, так как его выбирает тот, кто пишет конфигурацию.
  Например, может быть задано, что функция `getPassword()` накладывает метку `"sensitive-data"` на своё возвращаемое значение.

- `Taint sink` — *приёмник*, некоторая критическая секция приложения. Это может быть метод прямого обращения к базе данных или файловой системе, а также любая другая потенциально опасная операция. Для `Taint sink` можно задать набор меток, переменные с которыми не должны в него просочиться.
  К примеру, если значение с меткой `"sensitive-data"` передаётся в функцию журналирования, которая печатает свои аргументы напрямую в консоль, то это ошибка разработчика, так как происходит утечка данных.

Помимо источников и приёмников, есть ещё две вспомогательные сущности, которые нужны для расширения возможностей алгоритма.

- `Taint pass` — это функция, которая помечает возвращаемое значение с учётом меток в её аргументах. В зависимости от реализации, метки могут накладываться не только на результат метода, но и на объект `this`, и на входные параметры.
  Например, в конфигурации может быть установлено, что метод конкатенации строк `concat(String a, String b)` помечает свой результат всеми метками из `a` и из `b`.

- `Taint cleaner` — это функция, которая удаляет заданный набор меток из переданных аргументов. Чаще всего, это некоторый метод валидации, который проверяет, что пользователь ввёл данные в ожидаемом формате.
  К примеру, метод `validateEmail(String email)` удаляет метку `XSS` при успешном прохождении проверок, так как теперь в строке `email` точно нет непроверенных данных, которые могут привести к уязвимости межсайтового скриптинга.

Алгоритм taint-анализа сканирует граф потока данных, пытаясь обнаружить маршрут между методом из множества `Taint sources` и методом из `Taint sinks`. Основной проблемой классического алгоритма, как и у всех методов неточного анализа, является большое количество ложноположительных срабатываний. Однако существуют научные публикации [26], в которых

данная проблема решается с помощью применения символьного исполнения. Другими словами, если встроить taint-анализ непосредственно внутрь символьной виртуальной машины, то удаётся практически полностью избавиться от ложных результатов, а также заодно получить конкретные входные данные, реализующие найденный путь от источника к приёмнику. Последний подход и будет реализован в данной работе.

1.4.2 Пример работы

Рассмотрим пример простой функции, в которой есть уязвимость вида SQL-инъекция: если злоумышленник введёт в переменную `name` строку `"name'); DROP TABLE Employees; --"`, то у него получится удалить таблицу `Employees` вместе со всеми данными, которые в ней хранились.

```
1  void example(Connection connection) throws SQLException {
2      Scanner sc = new Scanner(System.in);
3      String name = sc.nextLine();
4
5      Statement stmt = connection.createStatement();
6      String query =
7          "INSERT INTO Employees(name) VALUES('"
8          .concat(name)
9          .concat("')");
10     stmt.executeUpdate(query);
11 }
```

Для работы taint-анализа нужно сначала задать правильную конфигурацию.

- `Taint source` — метод `nextLine` класса `java.util.Scanner`, который добавляет к возвращаемому значению метку `"user-input"`.
- `Taint pass` — метод `concat` класса `java.lang.String`, который пропускает через себя метку `"user-input"`, полученную либо у первого аргумента, либо у объекта, на котором этот метод вызывается (то есть у `this`).

- `Taint sink` — метод `executeUpdate` класса `java.sql.Statement`, который отслеживает переменные, помеченные `"user-input"`.

Любая корректная реализация taint-анализа должна найти в этом коде ошибку: переменная `query` с меткой `"user-input"` передаётся в приёмник `executeUpdate`.

Важно отметить, что в обязанности алгоритма не входит нахождение конкретных данных, которые бы смог ввести злоумышленник, чтобы навредить программе. Требуется лишь обнаружить маршрут, соединяющий источник и приёмник.

2. Создание отчёта по сгенерированным тестам

Результатом работы любого статического анализатора является некоторый отчёт. Иначе говоря, список обнаруженных ошибок вместе со всей дополнительной информацией, которая может помочь программисту разобраться в проблеме быстрее.

Таким образом, первая задача, которая решается в данной работе — создание такого отчёта по имеющимся данным, а именно по сгенерированным инструментом UnitTestBot тестам. Причём наибольший интерес вызывают тестовые случаи, приводящие к падению программы, то есть к выбрасыванию необработанных исключений. Следовательно, сообщение об ошибке должно содержать конкретное место в коде, где возникает исключение, а также путь исполнения, который к нему приводит.

2.1. Формат SARIF

В качестве формата отчёта был выбран SARIF (Static Analysis Results Interchange Format) [27]. Это формат на основе JSON [28], созданный специально для единообразного описания результатов статического анализа. Принят как стандарт OASIS[3] и используется во многих популярных инструментах.

Рассмотрим структуру формата, а также основные возможности для описания ошибок, которые он предоставляет. Легче всего это сделать на примере небольшого отчёта, который приведён ниже (для краткости все незначащие детали заменены на многоточие).

[3]Organization for the Advancement of Structured Information Standards) — глобальный консорциум, который управляет разработкой, конвергенцией и принятием промышленных стандартов электронной коммерции в рамках международного информационного сообщества.

```
{
  "schema" : "https://.../sarif-schema-2.1.0.json",
  "version" : "2.1.0",
  "runs" : [
    "tool" : { ... },
    "results" : [ {
      "ruleId" : "...",
      "level" : "error",
      "message" : { "text" : "..." },
      "locations" : [ ... ],
      "relatedLocations" : [ ... ],
      "codeFlows" : [ ... ]
    } ]
  ]
}
```

На самом верхнем уровне формат определяет `runs` — список результатов различных запусков анализатора.

Каждый запуск содержит в себе `tool` — некоторая информации об инструменте и `results` — непосредственно список обнаруженных ошибок.

Один результат описывается ключами

- `ruleId` — уникальный идентификатор для данного типа ошибки.
- `level` — уровень серьезности дефекта, например, `error` или `warning`.
- `message` — сообщение, которое будет показано пользователю.
- `locations` — список мест в коде, где была найдена ошибка. Каждый отдельный `location` имеет собственное внутреннее устройство, о котором рассказано ниже.
- `relatedLocations` — список мест в коде, каким-либо образом связанных с ошибкой. Элементами списка являются те же объекты `location`.
- `codeFlows` — список путей, по которым прошла программа, прежде чем наткнуться на обнаруженный дефект.

Объект `location` состоит из пути до файла `artifactLocation` и позиции внутри файла `region`. Пример объекта `location` представлен ниже.

```
"physicalLocation" : {
  "artifactLocation" : {
    "uri" : "src/examples/Main.java"
  },
  "region" : {
    "startLine" : 91,
    "startColumn" : 9,
    "endLine": 91,
    "endColumn": 20
  }
}
```

Как уже отмечалось ранее, SARIF имеет неплохую поддержку в сфере анализа программ. В частности, смотреть отчёт в удобном виде, с навигацией по коду и другими полезными функциями, можно в редакторе Visual Studio Code [29]. Также есть интеграция с CI/CD платформы GitHub [30], что может быть удобно для больших команд разработчиков.

2.2. Извлечение информации из тестов

Теперь опишем, как именно разработанный модуль формирования отчётов получает нужную информацию об ошибках по результатам символьного исполнения.

После того как UnitTestBot отработал на пользовательском коде и сгенерировал для него тесты, модуль смотрит на набор найденных терминальных состояний, другими словами, на тестовые случаи. При этом он будет рассматривать только те состояния, которые приводят к выбрасыванию исключения.

В терминальном состоянии `state` содержится достаточно много полезной информации. Во-первых, в там хранится результат исполнения, в нашем случае это тип возникающего исключения. Во-вторых, конкретные входные данные анализируемого метода, полученные из SMT-решателя. Зная это, модуль формирует сообщение об ошибке по шаблону ниже.

```
"Unexpected ArithmeticException: / by zero.
 Test case: example(28, -1)."
```

В данном случае было найдено деление на ноль при запуске метода `example` с аргументами 28 и -1.

В `state` также содержится путь из инструкций `path`, по которому машина пришла в это состояние. Она работает с трёхадресным Java байт-кодом, однако внутри UnitTestBot есть механизм, который по любой инструкции может вернуть номер соответствующей строчки в исходном коде.

Чтобы восстановить номер строки с ошибкой `startLine`, достаточно взять последний элемент из списка `path`. Далее, модуль считывает строку с номером `startLine` из файла с кодом пользователя и вычисляет `startColumn` и `endColumn`, обрезая пробельные символы с начала и с конца. Таким образом, формируется поле `location` в отчёте.

Поле `codeFlows` также восстанавливается из значения `path`. Модуль анализирует граф вызовов методов на пути `path` и назначает для каждого уровня вложенности последнюю на этом уровне инструкцию. Полученный список номеров строк оказывается равен настоящей трассировке стека, как если бы мы просто запустили код с найденными входными аргументами, то есть ровно то, что и требовалось.

Последнее, что фиксируется в отчёте — это поле `relatedLocations`. В него помещается ссылка на соответствующий данному результату тест в файле со сгенерированными тестами. Когда стартует разработанный модуль, UnitTestBot уже записал тесты в выходной файл, а значит можно просто найти позицию нужного теста по его имени в этом файле.

В итоге вся собранная информация структурируется и сериализуется в JSON.

2.3. Пример работы

Модуль поддерживает все стандартные исключения Java, в том числе и `AssertionError`, поэтому, один из типичных способов использования инструмента выглядит следующим образом. Программист сначала расставляет

assert проверки в коде, если хочет гарантировать выполнение некоторых инвариантов. После чего он запускает разработанный статический анализатор, который выявляет все места, где проверки не проходят.

Рассмотрим пример его работы на функции abs, которая, как понятно из названия, должна возвращать модуль числа. Её возвращаемое значение обязано быть не меньше нуля, и, чтобы это гарантировать, в методе присутствует строчка assert x >= 0.

```
1 public class Main {
2     int abs(int x) {
3         if (x < 0) {
4             x *= -1;
5         }
6         assert x >= 0;
7         return x;
8     }
9 }
```

Однако UnitTestBot обнаружил, что передача числа -2^{32} приводит к AssertionError (происходит переполнение типа int), что и отражено в сформированном отчёте SARIF. Он получился достаточно большим (около 100 строк), поэтому для краткости я перечислю в здесь только значения важных полей без сохранения структуры формата SARIF.

- Сообщение об ошибке:

```
"message" : {
  "text" : "Unexpected AssertionError.
            Test case: abs(-2147483648)"
}
```

- Местоположение ошибки:

```
"uri" : "src/Main.java",
"region" : {
  "startLine" : 6,
  "startColumn" : 9
}
```

- Местоположение сгенерированного теста:

```
"uri" : "utbot_tests/MainTest.java",
"region" : {
  "startLine" : 29,
  "startColumn" : 5
}
```

- Трассировка стека:

```
[
    "MainTest.testAbs3(MainTest.java:34)",
    "Main.abs(Main.java:6)"
]
```

3. Плагин для IntelliJ IDEA

Абсолютное большинство программистов используют различные интегрированные среды разработки. Ключевой особенностью современных IDE является возможность подключения в них плагинов — сторонних модулей, выполняющих определённые задачи. Таким образом, каждый пользователь может установить себе нужный набор плагинов и решать возникающие при программировании задачи прямо в IDE, что часто удобнее, чем пользоваться отдельными сервисами.

Многие статические анализаторы имеют собственные плагины под ту или иную среду разработки. В данной работе было принято решение реализовать плагин для IntelliJ IDEA, который бы позволял запускать статический анализ и просматривать его результаты, то есть отчёт SARIF. Разработка, однако, велась не с нуля, а на основе уже существующего плагина для автоматической генерации модульных тестов — UnitTestBot.

3.1. Детали реализации

Платформа IntelliJ предоставляет сторонним разработчикам широкие возможности по созданию новых модулей [31]. В частности, существует стандартный механизм `Inspections`, который позволяет отображать список найденных проблем на панели `Problems View`. Для того чтобы воспользоваться этой возможностью, были выполнены следующие шаги.

1. Зарегистрирован новый `InspectionTool` в конфигурации плагина.
2. Переопределён метод `GlobalSimpleInspectionTool.checkFile`, который добавляет обнаруженные ошибки для конкретного переданного файла `psiFile` в общий список проблем для `psiFile`.
3. Добавлен автоматический запуск механизма `Inspections` сразу после генерации тестов и создания отчёта SARIF, чтобы отобразить найденные дефекты в IDE.

Ниже представлен сценарий использования модифицированного плагина.

1. Пользователь запускает генерацию тестов в специальном режиме, который предполагает дальнейшее отображение результатов статического анализа.

2. После окончания работы символьной машины и создания отчёта SARIF, на экране открывается вкладка `Problems View`, где перечислены обнаруженные инструментом ошибки и уязвимости в виде списка сообщений (поле `message` в SARIF).

3. При нажатии на любой из элементов этого списка, IDE показывает соответствующую ему строчку кода (поле `location`).

4. В окошке справа появляется кнопка для перехода к нужному сгенерированному тесту (поле `relatedLocation`).

5. Также, есть возможность посмотреть трассировку стека найденного исключения (поле `codeFlows`). Причём эта функциональность поддерживает навигацию по коду для каждого представленного кадра стека.

Отдельно отметим, что перед разработкой самого интерфейса, было исследовано несколько альтернативных путей решения задачи по визуализации результатов. Например, для этого можно было написать отдельный плагин или сверстать собственную вкладку. В итоге было решено, что способ с использованием уже готовых `Inspections` и `Problems View` будет наиболее прост в реализации и, при этом, привычен для конечного пользователя, поэтому был выбран именно он.

Помимо описанной функциональности, плагин имеет дополнительные возможности для повышения удобства использования. Например, существует возможность запускать анализ не только на одном выбранном классе, но и на нескольких, а также на всём проекте сразу.

3.2. Пример работы

В приложении на Рис. 4 представлен снимок экрана среды IntelliJ IDEA с разработанным пользовательским интерфейсом визуализации результатов статического анализа. В данном случае, инструмент был запущен на методах

Main.example и Util.abs, реализации которых уже появлялись в примерах ранее, а также на методе Main.sumAbs.

```
1 int sumAbs(int a, int b) {
2     return Util.abs(a) + Util.abs(b);
3 }
```

При нажатии на кнопку View generated test, происходит переход к соответствующему тесту, как показано в приложении на Рис. 5.

При нажатии на кнопку Analyze stack trace, открывается вкладка, представленная в приложении на Рис. 6.

4. Taint-анализ

Как уже было упомянуто, taint-анализ позволяет находить уязвимости в безопасности, отслеживая распространение непроверенных пользовательских данных по программе. Алгоритм действительно очень известный, ведь те или иные его модификации реализованы во многих популярных анализаторах. В частности, в некоторых из тех, что были рассмотрены во введении.

Одной из задач данной работы является реализация taint-анализа поверх символьной виртуальной машины UnitTestBot, что избавит описываемую технику от ложноположительных срабатываний. Таким образом, в результате модификации, единственным источником неточностей останется только само символьное исполнение.

4.1. Конфигурация

В процессе изучения темы не было найдено ни одного универсального формата конфигурации taint-анализа, а все известные мне статические анализаторы описывают свой собственный способ настройки. Таким образом, первая подзадача, которую нужно было решить ещё до модификации ядра символьной машины, — придумать схему конфигурации, где бы можно было описать правила (`sources`, `passes`, `cleaners` и `sinks`), с которыми будет вестись работа в дальнейшем. Формат должен быть удобен для пользователя, но в то же время предоставлять достаточно широкие возможности по настройке алгоритма. Исходя из перечисленных требований, в качестве базового языка разметки был взят YAML [32].

Теперь рассмотрим созданный формат в подробностях. Общая структура конфигурационного документа представлена ниже.

```
sources:
  - <rule-1>
  - <rule-2>
  - < ... >
passes: [ ... ]
cleaners: [ ... ]
sinks: [ ... ]
```

То есть это просто списки правил, относящиеся к конкретному типу. Каждое правило имеет некоторый набор характеристик.

- Уникальный идентификатор метода, которого описывает данное правило. Состоит из полного имени метода, включающего имя пакета и имя класса, а также из сигнатуры метода — набора типов его аргументов (ключ `signature` в YAML).
- Некоторые условия `conditions`, которые должны выполняться во время исполнения, чтобы правило сработало.
- Множество меток `marks`, которыми оперирует правило.
- Набор конкретных действий по управлению метками, которые происходят при срабатывании правила (`add-to`, `get-from`, `remove-from` или `check` в зависимости от семантики правила).

Таким образом, одно правило, например `taint source`, может выглядеть следующим образом.

```
com.abc.ClassName.methodName:
  signature: [ <int>, _, <java.lang.Object> ]
  conditions:
    arg1:
      not: [ -1, 1 ]
  add-to: [ this, arg2, return ]
  marks: user-input
```

Данное правило определяется для метода с именем `methodName` из класса `ClassName`, который находится в пакете `com.abc`. Метод принимает ровно 3 аргумента, первый из которых имеет тип `int`, второй может быть совершенно любым, а последний имеет тип `java.lang.Object`. Ключ `signature` может быть не указан, тогда подходящей считается любая перегрузка `methodName`.

Срабатывание происходит только в том случае, когда первый аргумент (`arg1`) не равен ни -1, ни 1 , что и указано по ключу `conditions` (список интерпретируется как логическое ИЛИ). Данный параметр является необязательным, в случае его отсутствия не будут проверены никакие условия.

Описанный источник накладывает метку `user-input` к переменным, соответствующим `this`, `arg2` и `return`. Другими словами, к объекту класса `ClassName`, на котором вызывается `methodName`, второму аргументу функции и возвращаемому значению. Причём по ключам `add-to` и `marks` может лежать как список, так и одиночное значение — это сделано для удобства использования.

Остальные типы правил имеют такой же синтаксис, как и `source`, за исключением ключа `add-to`.

- `Taint pass` передаёт метки с одного множества объектов на другое, поэтому для него определяются два ключа: `get-from` и `add-to` соответственно. Причём на `add-to` накладываются все указанные в `marks` метки, если в `get-from` есть хотя бы одна из них.

- `Taint cleaner` удаляет метки с множества объектов, поэтому его ключ называется `remove-from`.

- `Taint sink` проверяет наличие некоторых меток в переменных, множество которых лежит по ключу `check`.

Разработанный формат, с одной стороны, является лаконичным и читабельным для конечного пользователя, а с другой стороны, достаточно мощным для описания сложных правил или условий срабатывания.

Для описанной схемы конфигурации был реализован парсер, а также произведена необходимая интеграция с плагином для IntelliJ IDEA, чтобы программист мог настраивать taint-анализ под свой конкретный проект.

Итоговый инструмент можно применять, и не составляя новых правил, для этого были добавлены распространённые источники и приёмники непроверенных данных. Приведём примеры для ясности.

- `java.util.Scanner.nextLine` — это `source`, накладывающий метку `user-input` на возвращаемое значение.

- `java.lang.String.concat` — это `pass`, который добавляет к возвращаемому значению все метки, полученные из `this` и `arg1`.

- `java.lang.String.isEmpty` — это `cleaner`, который удаляет все метки из объекта `this` в том случае, если `return` равняется `true` (пустая строка не может содержать вредоносных данных).

- `java.sql.Statement.execute` — это `sink` для меток `user-input`.

4.2. Детали реализации

В данном подразделе будут описаны общие концепции интеграции taint-анализа и символьного исполнения, а также детали реализации в UnitTestBot.

4.2.1 Общие концепции

Ключевая идея состоит в том, чтобы для каждой символьной переменной поддерживать набор меток, наложенных на неё в процессе работы символьной машины (подробнее о символьном исполнении описано в разделе 1.2.1). Эти наборы могут изменяться только при вызове метода, для которого существует правило в конфигурации (`source`, `pass`, `cleaner` или `sink`).

Общий вид обработки источника представлен в алгоритме 1. Для каждого заданного в конфигурации правила `rule`, которое относится к текущему методу, и для каждой сущности `entity` внутри него (`this`, `return` или аргумент метода) алгоритм получает соответствующую ему символьную переменную `symbol`, после чего добавляет к её набору меток `marks[symbol]` метки из `rule`.

Algorithm 1: Обработка источника помеченных данных

ProcessTaintSource $(config, stmt)$
- **Data:** $config$ — набор правил taint-анализа
- **Data:** $stmt$ — текущая инструкция (вызов метода)
- $rules = config.getSourcesBy(stmt)$;
- **foreach** $rule \in rules$ **do**
  - **foreach** $entity \in rule.entities$ **do**
    - $symbol \leftarrow getSymbolicValue(stmt, entity)$;
    - $marks[symbol] = marks[symbol] \cup rule.marks$;
  - **end**
- **end**

Алгоритм обработки правил `taint sink` (2) выглядит также, с той лишь разницей, что вместо добавления символьная машина проверяет наличие запрещённых меток `rule.marks` во множестве `marks[symbol]`, и, если пересечение множеств не пусто, то сообщает о найденной уязвимости.

Algorithm 2: Обработка приёмника помеченных данных

```
ProcessTaintSink (config, stmt)
    Data: config — набор правил taint-анализа
    Data: stmt — текущая инструкция (вызов метода)
    rules = config.getSinksBy(stmt);
    foreach rule ∈ rules do
        foreach entity ∈ rule.entities do
            symbol ← getSymbolicValue(stmt, entity);
            if marks[symbol] ∩ rule.marks ≠ ∅ then
                reportProblem();
            end
        end
    end
```

Алгоритмы для правил `taint pass` и `cleaner` аналогичны представленным выше, за исключением строки с изменением `marks` (в соответствии со своей семантикой), поэтому для краткости в тексте работы не приводятся.

4.2.2 Модификация символьной машины

Теперь подробно рассмотрим детали встраивания taint-анализа в выбранную символьную виртуальную машину.

Общий вид работы UnitTestBot выглядит следующим образом. На вход инструменту поступает программа в виде инструкций байт-кода. Объект `PathSelector` определяет порядок обхода символьных состояний, `Traverser` отвечает за обработку одной инструкции, а также поддерживает символьную память `Memory` и систему типов `TypeRegistry` в актуальном виде. В терминальном состоянии делается запрос к SMT-решателю с целью определить выполнимость пройденного пути и найти для него конкретные входные данные. Более подробное описание ключевых мест в процессе работы UnitTestBot представлены в разделе 1.3.1.

Основная идея реализованного подхода состоит в том, что каждой символьной переменной сопоставляется `taint vector` — 64-битное значение, каждый бит `i` которого отвечает за наличие метки с номером `i` в этом объекте. После чего, в процессе работы символьной машины, эти сопоставления поддерживаются и обновляются в соответствии с классическим алгоритмом taint-анализа.

Сделанные в данной работе изменения в большинстве своём затрагивают классы `Traverser` и `Memory`. Также, были добавлены новые компоненты `TaintModel`, `TaintMarkRegistry` и `TaintMarkManager`. На рис. 3 представлена высокоуровневая диаграмма разработанного модуля. Разберём зону ответственности каждой сущности подробнее.

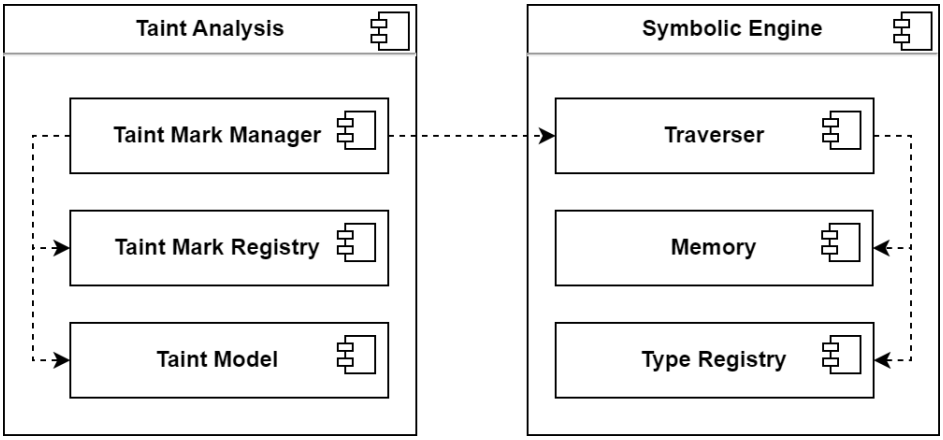

Рис. 3: Архитектура модуля Taint-анализа.

Объект класса `TaintMarkRegistry` хранит сопоставление между именем метки и её порядковым номером от 0 до 63. Видим, что количество меток, которые могут быть одновременно задействованы в анализе, ограничено числом 64. Однако, во-первых, этого хватает для почти любого разумного примера, а во-вторых, решение было обусловлено вопросами производительности — операции с типом данных `Long` совершаются намного быстрее, чем если бы был использован битовый массив произвольного размера.

Компонент `TaintModel` отвечает за предоставление доступа к конфигурации. В частности, он определяет способ преобразования условий (значение по ключу `conditions` в YAML-документе) в логические выражения над символьными переменными. Отметим, что сам `TaintModel` не выполняет никаких действий, а лишь предоставляет процедуру, которую можно запустить, уже

имея на руках Memory и TypeRegistry.

Memory занимается хранением значений taint vector для символьных переменных. В этом же классе были реализованы простые функции для обновления векторов и их получения по адресу символьного объекта. Вся сложная логика по добавлению и удалению меток, опирающаяся на теорию taint-анализа, была написана в отдельной сущности TaintMarkManager. Другими словами, этот класс оборачивает низкоуровневую работу с памятью в понятные с точки зрения предметной области операции.

Обновление информации о помеченных переменных происходит в процессе работы Traverser. Перед каждой из invoke инструкций, которые соответствуют запуску некоторого метода в пользовательском коде, вызывается специальный обработчик processTaintSink, а уже после инструкции invoke вызываются обработчики processTaintSource, processTaintPass и processTaintCleaner. Такой порядок связан с тем, что всем правилам, кроме приёмников, нужен результат работы функции. При этом факт передачи заражённых данных происходит в момент запуска функции-приёмника, поэтому сообщить о найденной уязвимости можно ещё до её выполнения.

Перечисленные обработчики правил обращаются к TaintModel, чтобы проверить, есть ли информация про рассматриваемый метод в конфигурации, и, если есть, то получить её. Функция processTaintSink запрашивает у TaintMarkManager данные об уже выставленных метках и добавляет ограничения в SMT-решатель, выполнение которых соответствует обнаружению дефекта. Остальные обработчики изменяют символьную память Memory через тот же TaintMarkManager, добавляя и удаляя метки у выбранных символьных объектов.

В результате удаётся не только обнаружить возможные уязвимости, но и сгенерировать конкретные входные данные, запуск программы на которых покажет, по какому маршруту заражённые переменные могут проникнуть в критические секции кода.

4.2.3 Модификация генератора кода

Результатом работы UnitTestBot, помимо добавленного в данной работе отчёта SARIF, являются тесты. Другими словами, на каждом найденном тестовом случае запускается `CodeGenerator`, который должен оформить тест в виде кода на языке Java. Причём, если тест ведёт к выбрасыванию необработанного исключения, то он не должен проходить.

Ошибки, которые получаются в результате работы модуля taint-анализа, не являются настоящими с точки зрения языка, так как это не исключения. Однако хотелось бы всё равно выделить такие тесты как провальные, поэтому генератор кода был изменён таким образом, чтобы в конце каждого теста добавлялась искусственная ошибка, которая бы и обеспечивала падение.

```
1 Assertions.fail(
2     "'java.lang.String' marked 'user-input'
3      was passed into 'Statement.execute' method"
4 );
```

Предоставленное решение оказалось очень удачным, ведь оно позволило автоматически получить интеграцию с отчётами SARIF и визуализацией результатов во вкладке `Problems view` в IntelliJ IDEA. Найденный тестовый случай теперь трактуется как настоящее исключение, а для них уже написана вся нужная логика.

4.3. Пример работы

Рассмотрим пример работы реализованного модуля на классе `User`.

```java
public class User {

    String getLogin() { /* some logic */ }

    String getPassword() { /* some logic */ }

    String userInfo(String login, String password) {
        return login + "#" + password;
    }

    void printUserInfo() {
        var login = getLogin();
        var password = getPassword();
        var info = userInfo(login, password);
        System.out.println(info);
    }
}
```

Метод `getPassword` возвращает чувствительные данные, которые ни в коем случае не должны утечь из приложения, а программист печатает их в стандартный поток вывода, что является серьёзной ошибкой.

Сначала напишем конфигурацию, которая выражает сказанную мысль, и сохраним её в файл `./.idea/utbot-taint-config.yaml`, откуда её сможет прочесть анализатор.

```yaml
sources:
  - User.getPassword:
      add-to: return
      marks: sensitive-data
passes:
  - User.userInfo:
      get-from: [ arg1, arg2 ]
      add-to: return
      marks: [] # all
sinks:
  - java.io.PrintStream.println:
      check: arg1
      marks: sensitive-data
```

Теперь запускаем UnitTestBot с помощью плагина для IntelliJ IDEA и

смотрим на полученный результат.

Во-первых, сгенерировался тест, по которому понятно, что модуль taint-анализа нашёл уязвимость.

```
1 @Test
2 public void testPrintUserInfo1() {
3     User user = new User();
4     user.printUserInfo();
5     fail(
6         "'java.lang.String' marked 'sensitive-data'
7          was passed into 'PrintStream.println' method"
8     );
9 }
```

Во-вторых, появилась панель `Problems view` (рис. 7), где отображена найденная проблема в виде результата статического анализа.

Разберём чуть подробнее внутренности работы модуля на данном примере.

После выполнения метода `getPassword` символ, соответствующий переменной `password`, помечается как `sensitite-data` (возводится нулевой бит в его `taint vector`).

После вызова `userInfo` помечается также переменная `info`, так как `userInfo` — это `taint pass`, который добавляет к возвращаемому значению все метки, собранные из обоих своих аргументов.

Перед печатью `info` на консоль, функция-обработчик `processTaintSink` добавляет ограничение в SMT-решатель, выполнение которого соответствует выбрасыванию нашего фиктивного исключения. Логическая формула для данного пути оказывается выполнима, поэтому анализатор сообщает о найденной ошибке, что мы в итоге и наблюдаем.

5. Тестирование и эксперименты

В данной главе будут описаны результаты тестирования разработанного статического анализатора на различных проектах. Эксперименты проводились с целью проверки качества полученного программного продукта. Причём под качеством подразумевается не только количество и серьёзность обнаруживаемых им дефектов, но и простота настройки и запуска, а также удобство использования инструмента в целом.

Замеры производительности не осуществлялись и не были целью тестирования, однако для лучшей воспроизводимости полученных результатов, укажем технические характеристики ноутбука, на котором оно проводилось.

- `OS` — `Windows 10, 64-bit`
- `CPU` — `Intel(R) Core(TM) i5-8665U CPU @ 1.90GHz`
- `RAM` — `16 ГБ`

Всё взаимодействие с настройкой, запуском и результатами анализатора происходило исключительно средствами модифицированного в данной работе плагина для IntelliJ IDEA. Для всех примеров было выставлено ограничение по времени на анализ кода в 60 секунд на один класс.

5.1. Тестирование taint-анализа

В ходе экспериментов сравнивалась эффективность нахождения уязвимостей между инструментом SpotBugs и техникой taint-анализа, реализованной в данной работе. Конкретнее, для сравнения были выбраны следующие метрики.

- Общее количество найденных проблем.
- Доля обнаруженных ошибок от общего числа дефектов в программе.
- Доля ложноположительных срабатываний от общего числа обнаруженных проблем.

В качестве тестов, на которых проводились замеры, была взята часть проекта Juliet Java [33] — набора синтетических программ, в которых содержится более 100 различных типов уязвимостей CWE. Все программы распределены по разделам в соответствии с названием допущенной в них ошибки. Для тестирования алгоритмов taint-анализа был выбран раздел с SQL-инъекциями, в котором находится 3660 классов и 2220 специально сделанных ошибок.

В тестовых данных все допущенные ошибки размечены, то есть для каждого класса известна находящаяся в нём уязвимость, а также её местоположение в коде. Таким образом, требовалось запустить анализаторы и сравнить полученные наборы предупреждений с правильным.

Результаты запусков представлены в таблице 1.

	UnitTestBot	SpotBugs
Обнаружено проблем	1023	5280
Верно найденных ошибок	46%	100%
Ложных срабатываний	0%	58%

Таблица 1: Результаты запуска на тестовом наборе Juliet Java.

Модифицированный UnitTestBot смог найти 1023 SQL-инъекции, что составляет 46% из всех допущенных в наборе ошибок. При этом не было выдано ни одного ложноположительного срабатывания. Оставшиеся 54% не были найдены, в основном, по той причине, что инструменту не удалось сгенерировать ни одного теста за предоставленное ему время. В частности, использование массивов или длинных строк в анализируемых программах значительно замедляет реализованное в UnitTestBot символьное исполнение. Ускорение его работы является одним из наиболее приоритетных направлений для исследования в будущем.

SpotBugs обнаружил все 100% уязвимостей, однако вместе с этим нашёл ещё 3060 несуществующих проблем, что почти в полтора раза больше, чем общее число реальных ошибок в программах (2220). Таким образом, его частота ложных срабатываний составила 58%.

Полученные результаты показывают, что разработанный модуль taint-анализа позволил инструменту UnitTestBot находить уязвимости вида SQL-

инъекция практически в половине случаев на соответствующем тестовом наборе. Отметим, что до модификации UnitTestBot был в принципе не применим к данной задаче.

5.2. Запуск на реальных проектах

С целью показать способность инструмента анализировать не только синтетические программы, но и большие объёмы реального кода, было проведено тестирование на нескольких популярных проектах на языке Java, код которых находится в открытом доступе.

Перед тем, как запустить статический анализ на каждом из кандидатов, проекты были собраны, то есть их классы были скомпилированы в Java байт-код, который требуется для работы инструмента. Результаты экспериментов описаны в соответствующих подразделах ниже.

5.2.1 Tape

Tape [34] — быстрая транзакционная файловая очередь для Java. Проект не очень большой — всего пару тысяч строк кода, однако этого вполне достаточно для целей тестирования.

В результате запуска анализатора был сгенерирован отчёт SARIF размером более 5 тысяч строк, в котором было подробно описано 50 обнаруженных проблем.

- `NullPointerException` — возникал 37 раз.
- `IOException` — 10 раз.
- `NegativeArraySizeException` — 2 раза.
- `NoSuchElementException` — только 1 раз.

Важно отметить, что некоторые найденные ошибки, особенно, большое количество разыменований нулевого указателя, могут никогда не проявиться при реальном использовании Tape. Проблема возникает, если передавать в публичные методы классов «некорректные» значения аргументов, то есть не

соблюдать негласные договорённости между пользователем и разработчиком библиотеки. Основной недостаток таких неявных соглашений заключается в том, что пользователи, а то и сам разработчик через какое-то время после написания кода, могут случайно забыть их выполнить и, как следствие, получить неправильное поведение программы. Статический анализатор отлавливает такие ситуации и сообщает о них.

Например, в проекте есть метод `void add(T entry)`, разработчик которого неявно подразумевал, что в его метод никто не передаст значение `null`, ведь иначе будет выброшено исключение `NullPointerException`. Однако UnitTestBot сгенерировал тестовый случай, где `entry` равняется `null`, а также добавил в отчёт статического анализа этот тест как ошибку. В данном случае, разработчику стоило пометить `entry` аннотацией `@NotNull`, что решило бы описанную проблему.

Похожий пример возник в методе `void remove(int n)`. UnitTestBot обнаружил, что его запуск с аргументом `n = 131072` приводит к выбрасыванию исключения `NoSuchElementException`. Здесь возможны два варианта. Первый состоит в том, что автор библиотеки не знал о таком сценарии, а значит инструмент действительно нашёл допущенную ошибку. Второй — он знал, но по какой-то причине не воспользовался стандартным способом документирования потенциальных исключений и не добавил строку `throws NoSuchElementException` в объявление функции. В этом случае поведение анализатора тоже ожидаемо — выданным сообщением об ошибке он подсказал разработчику задокументировать поведение метода во избежание проблем в будущем.

5.2.2 Fastjson

Fastjson [35] — высокопроизводительная библиотека, которая используется для преобразования объектов Java в их JSON представление. Проект имеет гораздо более внушительную кодовую базу по сравнению с предыдущим — около 50 тысяч строк.

По итогам анализа, было обнаружено 1121 проблема, среди которых 830 разыменований нулевого указателя и 193 выхода за границы массива. Также

было найдено 75 `JSONException` и 10 `JSONPathException`, что показывает способность инструмента работать с пользовательскими исключениями, определёнными непосредственно в анализируемом проекте.

5.3. Запуск на задачах Codeforces

На больших проектах сложно оценить серьёзность находимых ошибок, ведь для этого нужно, во-первых, хорошо разбираться в самом проекте, а во-вторых, вручную проанализировать каждый выдаваемый результат, что очень трудозатратно. Поэтому, чтобы оценить насколько неочевидные проблемы способен находить анализатор, были проведены дополнительные эксперименты на решениях задач с CodeForces [36].

В ходе работы было рассмотрено более 100 задач по программированию и алгоритмам. Для каждой из них, были изучены отправленные в систему решения на языке Java, которые не прошли тестирование по причине выброшенного исключения (вердикт «Ошибка исполнения»). Из всех посылок были отобраны только 23, на которых и планировалось запустить статический анализ. В выборку включались программы, требующие глубокого анализа, так как допущенная в них ошибка не была очевидна. Отметим также, что на сайте CodeForces не раскрывается конкретный тестовый случай, на котором провалилось решение, поэтому для программиста может быть полезно найти его с помощью UnitTestBot.

В условиях задач обычно устанавливаются ограничения на входные данные, на которые может опираться отправляемая в систему программа. Поэтому, чтобы эффективно анализировать код, нужно сначала привести его к следующему виду.

```
1  public class Main {
2
3      static /* тип ответа */ solve(/* ... */) {
4          assume(/* ограничения в виде предиката */);
5          // само решение
6      }
7
8      public static void main(String[] args) {
9          // ввод данных
10         var result = solve(/* входные данные */);
11         // вывод ответа
12     }
13 }
```

Статический анализ запускался только на функции `solve`, которая уже не занимается чтением входных данных и печатью результата. В ней вызывается специальный метод `assume` инструмента UnitTestBot, позволяющий наложить определённые ограничения на символьные переменные. В данном случае это нужно как раз для учёта ограничений из условия задачи.

Описанное преобразование кода нужно было вручную проделать для каждой из 23 выбранных посылок. В результате анализа, для 10 из 23 программ удалось обнаружить тестовый случай, приводящий к выбрасыванию исключения.

Отметим, что неправильные посылки специально подбирались так, чтобы допущенные в них ошибки не повторяли друг друга. Таким образом, получилось проверить анализатор на способность находить различные дефекты. Были выявлены такие проблемы, как деление на ноль, разыменование нулевого указателя, переполнение стека и выход за границы массива или строки — часто возникающие исключения в коде на Java.

Рассмотрим одну из успешно найденных ошибок. Соответствующая посылка, а точнее, уже отредактированная по шаблону функция `solve`, представлена в листинге 1. Решение было отправлено по задаче B соревнования «Codeforces Round 805» [37]. Смотря на код решения, совершенно неочевидно, какая в нём есть проблема, где она возникает и как её воспроизвести. Однако статический анализатор даёт ответы на все эти вопросы, возвращая

предупреждение на строку 24, в котором указан искомый тестовый случай — `solve("ac")`.

```
"Unexpected StringIndexOutOfBoundsException:
    String index out of range: 2.
 Test case: solve(ac)."
```

Заключение

Основным результатом работы является готовый к использованию инструмент статического анализа кода на языке Java. В процессе достижения этой цели были выполнены следующие задачи.

- Проведён обзор существующих на рынке решений, в ходе которого были выявлены их ключевые недостатки — поверхностность проводимого анализа, а также большое количество ложноположительных срабатываний.
- Реализован модуль создания отчётов в формате SARIF на основе тестовых случаев, сгенерированных инструментом UnitTestBot.
- Разработан удобный пользовательский интерфейс для просмотра отчётов SARIF в среде разработки IntelliJ IDEA.
- В символьную виртуальную машину встроена техника taint-анализа, расширяющая её возможности. Реализованный алгоритм позволяет находить нарушения безопасности, а именно, случаи использования непроверенных данных в критических секциях программы.
- Разработанный статический анализатор был протестирован на нескольких известных проектах с открытым исходным кодом, а также на решениях задач с сайта Codeforces. Проведённые эксперименты показали возможность применения продукта на практике.

Список литературы

[1] B. Thomas. The concept of dynamic analysis. ACM SIGSOFT Software Engineering Note, 1999, pp. 216–234.

[2] SpotBugs repository. URL: https://github.com/spotbugs/spotbugs (дата обр. 17.05.2023).

[3] Coverity. URL: https://scan.coverity.com/ (дата обр. 12.04.2023).

[4] CodeQL repository. URL: https://github.com/github/codeql/ (дата обр. 12.04.2023).

[5] FB Infer. URL: https://fbinfer.com/ (дата обр. 12.04.2023).

[6] H. Li, T. Kim, M. Bat-Erdene, H. Lee. Software Vulnerability Detection Using Backward Trace Analysis and Symbolic Execution. International Conference on Availability, Reliability and Security, 2013, pp. 446-454.

[7] CWE Top 25 Most Dangerous Software Weaknesses, 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html (дата обр. 12.04.2023).

[8] UnitTestBot repository. URL: https://github.com/UnitTestBot/UTBotJava/ (дата обр. 12.04.2023).

[9] Jetbrains IntelliJ IDEA repository. URL: https://github.com/JetBrains/intellij-community (дата обр. 12.04.2023).

[10] Find Security Bugs repository. URL: https://github.com/find-sec-bugs/find-sec-bugs (дата обр. 17.05.2023).

[11] S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller, D. Yao. Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks. IEEE Transactions on Software Engineering, 2023, vol. 49, no. 2, pp. 485-497.

[12] Mars Rover Curiosity's 'Space Doctors' On Bug Hunting In Space. URL: https://www.huffingtonpost.co.uk/2012/09/27/curiositys-doctors-mars-rover-coverity_n_1919115.html (дата обр. 19.04.2023).

[13] N. Imtiaz, L. Williams. A synopsis of static analysis alerts on open source software. HotSoS '19: Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security, 2019, pp. 1–3.

[14] S. Afrose, S. Rahaman, D. Yao. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. IEEE Cybersecurity Development (SecDev), 2019, pp. 49-61.

[15] S. Li, Z. Su. Finding Unstable Code via Compiler-Driven Differential Testing. ASPLOS 2023: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 3, 2023, pp.238–251

[16] W. Kang, B, Son, K, Heo. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), 2022, pp, 1695–1708.

[17] D. Distefano, P. O'Hearn, H. Yang. A Local Shape Analysis Based on Separation Logic. TACAS, 2006, pp. 287–302.

[18] C. Calcagno, D. Distefano, P. O'Hearn, H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. Journal of the ACM, 2011, Vol. 58, pp 1–66.

[19] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, D. Lucanu. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2017, pp. 161-168.

[20] C. Cadar, K. Sen. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013, Vol. 56, no. 2, pp. 82–90.

[21] R. Baldoni, E. Coppa, D'elia Daniele Cono, C. Demetrescu, I. Finocchi. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 2018. Vol. 51, no. 3, pp. 1–39.

[22] D. Monniaux. A Survey of Satisfiability Modulo Theory. Computer Algebra in Scientific Computing, 2016, pp. 1–24.

[23] L. De Moura, N. Bjørner. Z3: An efficient SMT solver. International conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008. pp. 337–340.

[24] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, M. McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, EECS Department, 2010, pp. 1–20.

[25] M. Kang, S. McCamant, P. Poosankam, D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. Proceedings of the Network and Distributed System Security Symposium, NDSS, 2011, pp. 1–14.

[26] E. J. Schwartz, T. Avgerinos, D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). IEEE Symposium on Security and Privacy, 2010, pp. 317–331.

[27] Static Analysis Results Interchange Format (SARIF) Version 2.0. URL: `https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html` (дата обр. 26.04.2023).

[28] JSON format. URL: `https://www.json.org/json-ru.html` (дата обр. 26.05.2023).

[29] Visual Studio Code repository. URL: `https://github.com/microsoft/vscode` (дата обр. 26.04.2023).

[30] SARIF support for code scanning. URL: `https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning` (дата обр. 26.04.2023).

[31] The IntelliJ Platform. URL: https://plugins.jetbrains.com/docs/intellij/intellij-platform.html (дата обр. 27.04.2023).

[32] The Official YAML Web Site. URL: https://yaml.org/ (дата обр. 02.05.2023).

[33] Juliet Test Suite 1.3. URL: https://samate.nist.gov/SARD/test-suites/111 (дата обр. 17.05.2023).

[34] Tape repository. URL: https://github.com/square/tape (дата обр. 03.05.2023).

[35] Fastjson repository. URL: https://github.com/alibaba/fastjson (дата обр. 07.05.2023).

[36] CodeForces. URL: https://codeforces.com/ (дата обр. 03.05.2023)

[37] Codeforces Round 805, задача B. URL: https://codeforces.com/contest/1702/problem/B (дата обр. 03.05.2023)

Приложение

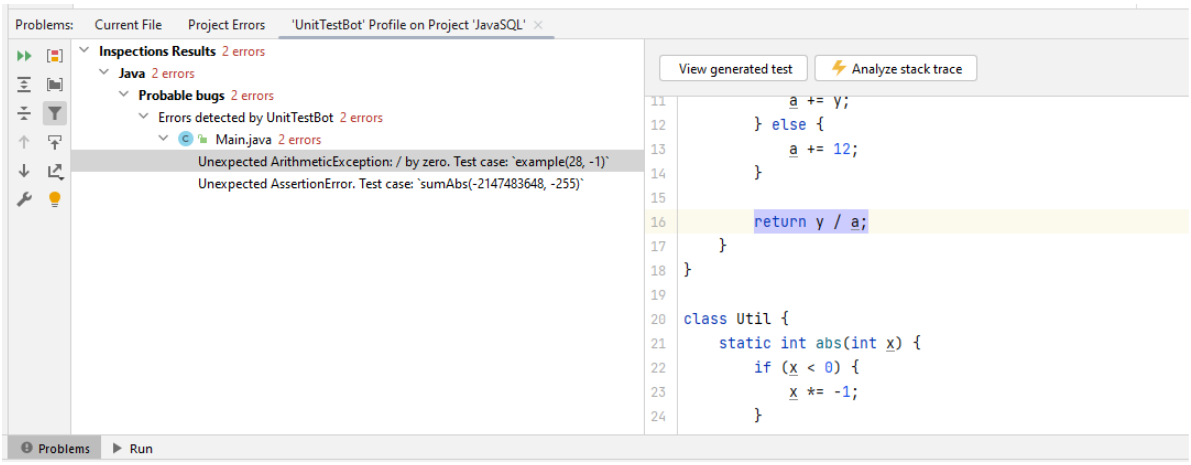

Рис. 4: Интерфейс просмотра результатов статического анализа.

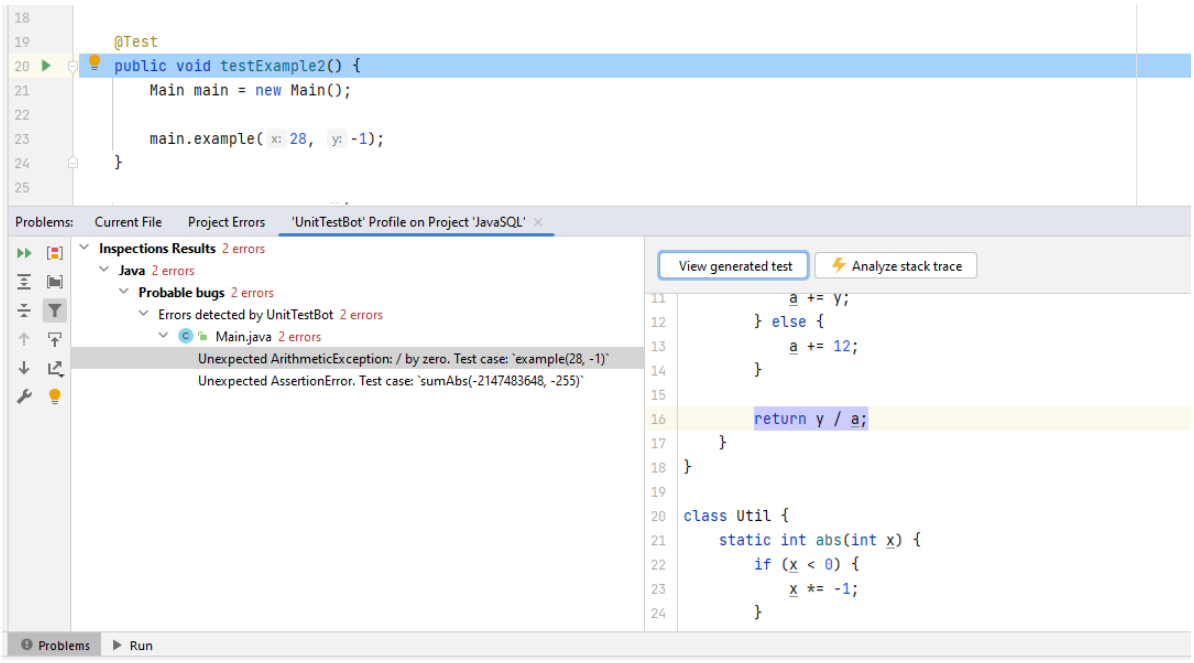

Рис. 5: Ссылка на сгенерированный тест.

Рис. 6: Просмотр трассировки стека.

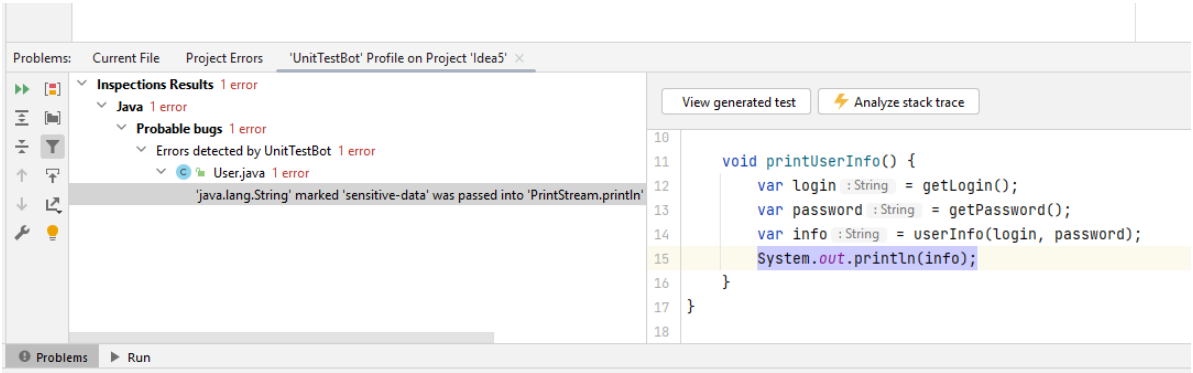

Рис. 7: Обнаруженная taint-анализом проблема.

```
1  public static int solve(String s) {
2      for (int i = 0; i < s.length(); ++i) {
3          assume('a' <= s.charAt(i) && s.charAt(i) <= 'z');
4      }
5
6      int counter = 0;
7      int count = 0;
8      while (count < s.length()) {
9          counter++;
10         char mem1 = s.charAt(count++);
11         while (count < s.length() && s.charAt(count) == mem1) {
12             count++;
13         }
14         if (count >= s.length()) {
15             break;
16         }
17         char mem2 = s.charAt(count++);
18         while (
19             count < s.length() && (s.charAt(count) == mem1 ||
20             s.charAt(count) == mem2)
21         ) {
22             count++;
23         }
24         char mem3 = s.charAt(count++);
25         while (
26             count < s.length() && (s.charAt(count) == mem1 ||
27             s.charAt(count) == mem2 || s.charAt(count) == mem3)
28         ) {
29             count++;
30         }
31     }
32
33     return counter;
34 }
```

Listing 1: Решение задачи Codeforces-805B, в котором допущена ошибка.