DOI: 10.15514/ISPRAS-2018-1(2)-33

Static analysis for languages with exception handling

[1,2]*Afanasyev V. O., ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1,3]*Dvortsova V. V., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>*
[1]*Borodin A. E., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia
[2]*National Research University Higher School of Economics,*
20, Myasnitskaya Str., Moscow, 101000, Russian Federation
[3]*Moscow State University,*
Leninskie gory 1, Moscow, Russian Federation

Abstract. This paper describes static anslysis for the languages with exception handling. A low-level intermediate representation, which supports exceptions, is proposed in this study. Data flow analyses for unreachable code detection were described. A general scheme of static analysis that takes exception related paths into account was given. The algorithms were implemented as a part of the static analysis tool *Svace* for C++, Java and Kotlin languages.

Keywords: static analysis; search for defects; vulnerabilities; Java; Kotlin; C++; *Svace*; JVM

For citation: Afanasyev V. O., Dvortsova V. V., Borodin A. E. Static analysis for languages with exception handling. Trudy ISP RAN/Proc. ISP RAS, 2022, vol. 1, issue 2, pp. 3–4. 10.15514/ISPRAS-2018-1(2)-33

Статический анализатор для языков с обработкой исключений

[1,2]*Афанасьев В. О., ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1,3]*Дворцова В. В., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>*
[1]*Бородин А. Е., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1]*Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2]*Национальный Исследовательский Университет Высшая Школа Экономики, 101000, Россия, г. Москва, ул. Мясницкая, д. 20*
[3]*Московский государственный университет им. М.В. Ломоносова , Россия, г. Москва, Ленинские горы д.1*

Аннотация. В статье описывается статический анализ для языков с обработкой исключений. В данной работе предложено низкоуровневое промежуточное представление для поддержки исключений; описаны анализы потока данных для поиска недостижимого кода, связанного с исключениями; приведена общая схема для статического анализа с учётом возможных путей, возникающих при использовании исключений. Алгоритмы реализованы в анализаторе *Svace* для языков C++, Java, Kotlin.

Ключевые слова: статический анализ; поиск ошибок; обработка исключений; Java; Kotlin; C++; *Svace*; JVM

Для цитирования: Афанасьев В. О., Дворцова В. В., Бородин А. Е. Статический анализатор для языков с обработкой исключений. Труды ИСП РАН, 2022, том 1 вып. 2, с. 3–4. 10.15514/ISPRAS-2018-1(2)-33

1. Введение

Обработка исключений — это широко используемый метод управления ошибками в программах, он поддерживается в том числе такими языками как C++, Java и Kotlin. Специальные языковые конструкции позволяют бросить исключение, создав объект специального типа, а также поймать исключение по его типу, выполнив действия для обработки исключительной ситуации.

Исключения позволяют в некоторых случаях сделать код проще, позволяя избежать постоянной обработки кодов возврата. Кроме этого, выполнение части действий может гарантироваться семантикой языка. Например, в языке C++ будут вызваны деструкторы объектов во всех случаях выхода из функции. Последнее позволяет не забыть про освобождение ресурсов при возникновении исключительных ситуаций. В языках Java и Kotlin есть возможность автоматического вызова функции `close` с помощью конструкции «`try` с ресурсами». На листинге 1 показан код двух семантически эквивалентных функций. Функция `readConfigEx` значительно компактнее

и проще читается.

```
1  void readConfig(String fileName, Reader reader) {
2    InputStream is = null;
3    try {
4      is = new FileInputStream(fileName);
5      if (reader.isEnd(is)) {
6        is.close();
7        return;
8      }
9    } catch (Throwable e) {
10     if (is != null)
11       is.close();
12     throw e;
13   }
14 }
15
16 void readConfigEx(String fileName, Reader reader) {
17   try (InputStream is = new FileInputStream(fileName)) {
18     if (reader.isEnd(is)) {
19       return;
20     }
21   }
22 }
```

Листинг 1: Пример автоматического вызова close

Listing 1: Example of try-with-resource

Многие уязвимости из перечня CWE (Common Weakness Enumeration) описывают ситуации, связанные с неправильной обработкой исключений, ошибочных ситуаций и т. п. [1]. Примеры таких ошибок: неправильное закрытие ресурсов при возникновении исключения, что может привести к их утечке; отсутствие конструкции, обрабатывающей исключение, которое возникает где-либо в программе, из-за чего данное исключение может аварийно завершить программу.

В статье мы рассматриваем задачу статического анализа исходного кода, содержащего инструкции обработки исключений, с целью поиска ошибок в нём. Мы не ограничиваемся лишь видами ошибок, напрямую связанными с исключениями, а ставим задачу создания анализатора для поиска произвольных ошибок в программах, использующих исключения.

Использование исключений в коде программ означает наличие специфических путей выполнения, возникающих при их бросании и обработке. Для успешного анализа статический анализатор должен уметь учитывать эти пути. Мы предлагаем подход, основанный на создании низкоуровневого промежуточного представления, явно содержащего переходы на возможные пути выполнения, связанные с обработкой исключений. Таким образом, задача разбивается на генерацию промежуточного представления, содержащего в явном виде все необходимые переходы, и на создание анализатора, который оперирует этим промежуточным представлением.

Описываемый подход был реализован в инструменте статического анализа

Svace [2—4] для языков C++, Java, Kotlin, требующих поддержку исключений.

Статья построена следующим образом. В главе 2 описываются ситуации в исходном коде, которые требуют анализа исключений. В главе 3 описывается анализируемый язык — низкоуровневое представление, предлагаемое для анализа исключений. Глава 4 описывает анализ. Глава 5 посвящена генерации промежуточного представления для языков C++, Java и Kotlin. Результаты тестирования на проектах с открытым исходным кодом представлены в главе 6. В главе 7 проведено сравнение с другими работами.

2. Исключения в исходном коде

В данном разделе мы приведём примеры ошибок, связанных с исключениями, и опишем необходимые свойства, требуемые от статического анализатора для поиска подобных ошибок в коде.

2.1 Пути выполнения, связанные с исключениями

Одна из самых часто встречающихся ошибок при работе с исключениями — неправильное освобождение ресурсов [5]. В листинге 2 приведён пример неправильного закрытия потока ввода. Код на строке 5 закрывает объект типа `FileInputStream`, но при этом в строке 4 может возникнуть исключение, что приведёт к тому, что поток не будет закрыт. В данном случае, ошибка может быть исправлена переносом строки 5 в `finally`-блок и объявлением переменной `is` перед началом `try`-блока.

```
1 try {
2   InputStream is = new FileInputStream(fileName);
3   byte b[] = new byte[is.available()];
4   is.read(b);
5   is.close();
6 } catch (Throwable t) {
7   System.err.println(t.getMessage());
8 }
```

Листинг 2: Пример неправильного освобождения ресурсов

Listing 2: Example of an incorrect resource release

Другой пример подобного рода уязвимости — некорректная обработка исключительной ситуации, приводящая к неправильному состоянию объекта. В листинге 3 приведён пример такой ошибки. Если исключение возникнет при вызове метода на строке 7, то объект `lock` всегда будет находиться в «захваченном» состоянии, что может привести к взаимной блокировке в другой части программы.

```
1 public class Foo {
2   private final Lock lock = ...;
3
4   public void foo() {
5     try {
```

```
6         lock.lock();
7         doSomething();
8         lock.unlock();
9       } catch (Throwable t) {
10        System.err.println(t.getMessage());
11      }
12    }
13
14    private void doSomething() { ... }
15  }
```

Листинг 3: Пример возможной взаимной блокировки

Listing 3: Example of a possible deadlock

В обоих случаях инструкция вызова функции может бросить исключение, которое приведёт к появлению другого пути выполнения программы. Игнорирование таких путей приводит к ошибкам в программе. Фактически выполнение инструкции вызова функции может создавать две группы путей: нормальное выполнение и выполнение с брошенным исключением.

2.2 Недостижимый код

Помимо создания новых путей, исключения могут сделать часть путей невыполнимыми. В листинге 4 приведён пример подобного кода. Строка 14, в которой значение входного аргумента y равно нулю, недостижима, так как в начале этой функции находится соответствующая проверка, где бросается исключение при нулевом значении.

```
1  void error(const char* msg) {
2      throw std::logic_error(msg);
3  }
4
5  int rem(int x, int y) {
6      if (y == 0) error("Division by zero");
7
8      int sign = (x < 0) ? -1 : 1;
9      if (y > 0) {
10         return sign * (std::abs(x) % y);
11     } else if (y < 0) {
12         return -sign * (std::abs(x) % -y);
13     } else {
14         return sign * INT_MAX;
15     }
16 }
```

Листинг 4: Пример недостижимого пути, вызванного брошенным исключением

Listing 4: Example of an unreachable execution path caused by thrown exception

Анализ, не умеющий отсеивать такие пути, будет менее точным и может выдавать малополезные предупреждения.

Более того, часть источников недостижимого кода может быть непосредственно связана с обработкой исключений. Например, код на языке C++, приведённый в листинге 5, содержит `catch`-блок, который никогда не может быть достигнут — так как исключение типа `std::logic_error` наследуется от `std::exception`, а обработчик исключения типа `std::exception` стоит выше, то второй обработчик никогда не будет достигнут во время выполнения вне зависимости от реализации функции `foo`. Стоит отметить, что подобная проблема актуальна для языков C++ и Kotlin, но не для языка Java — компилятор этого языка посчитает подобный код некорректным.

```
1 try {
2     foo();
3 } catch (std::exception) {
4     // Catch std::exception
5 } catch (std::logic_error) {
6     // Catch std::logic_error
7 }
```

Листинг 5: Пример недостижимого `catch`-блока

Listing 5: Example of an unreachable `catch`-block

Таким образом, в ходе анализа важно уметь определять, что исключения могут быть подтипами друг друга.

2.3 Бесполезные предупреждения

В некоторых случаях разработчик может полагаться на наличие конструкции, обрабатывающей исключения.

В листинге 6 приведён пример кода, содержащего потенциальное разыменование нулевой ссылки. Метод на строке 2 может вернуть нулевую ссылку. Дальнейшее выполнение программы приведёт к разыменованию этой ссылки на строке 3.

```
1 try {
2   var ref = mayReturnNull();
3   ref.calc();
4 } catch (NullPointerException e) {
5   ...
6 }
```

Листинг 6: Пример `catch`-блока, обрабатывающего NPE

Listing 6: Example of a `catch`-block that handles NPE

Статический анализатор может найти эту ошибку и сообщить о ней. Но в данном случае предупреждение будет бесполезным, так как программист обрабатывает исключение типа `NullPointerException`.

3. Анализируемый язык

Для описания нашего подхода мы используем в данной статье следующий язык. Для краткости изложения мы ограничиваем наш язык лишь конструкциями, представляющими интерес с точки зрения обработки исключений.

Язык содержит следующие типы данных:

- целые числа;
- указатели[1] — поддерживаются указатели на целые числа и другие указатели. Для простоты мы не стали рассматривать указатели на функции.
- исключения — имеют уникальное имя, представляют собой тип объектов, создаваемых и обрабатываемых в исключительных ситуациях, могут находиться в отношении наследования между собой.

Программы в этом языке будут представлять собой последовательность функций, последняя из которых является точкой входа. Каждая функция описывается графом потока управления (ГПУ), содержащим инструкции следующих видов:

- `assume(cond)` — выполнение завершается, если `cond` ложное;
- `!cond` — логическое отрицание;
- `a == b` — сравнение на равенство;
- `a != b` — сравнение на неравенство;
- `a = *b` — разыменование;
- `*a = b` — запись в память;
- `return a` — единственный возврат из функции;
- `a = alloca()` — выделение памяти на стеке;
- `r, e = func(a, b, ...)` — вызов функции, которая может бросить исключение. Здесь `e` — код исключения, который может иметь значение $\emptyset$, если функция не бросает исключение. Для функции, которая точно не бросит исключение, также будем использовать обозначение `r = func(a, b, ...)`;
- `throw e` — возврат из функции с исключением;
- `try_enter(e1, e2, ...)` — вход в блок, обрабатывающий исключения `e1, e2, ...`;
- `try_exit(e1, e2, ...)` — выход из блока, обрабатывающего исключения `e1, e2, ...`;
- `e1` $\leq$: `e2` — возвращает истину, если `e1` является подтипом `e2` или типы `e1` и `e2` эквивалентны.

Условные конструкции в нашем языке представляют собой недетерминированное

[1] Здесь и далее, когда мы будем упоминать несколько языков одновременно, мы будем использовать термин «указатель», подразумевая под ним указатели в языке C++ и ссылки в языках Java и Kotlin

ветвление и линейную инструкцию `assume(cond)`, имеющую следующую семантику: инструкция завершает выполнение программы, если условие `cond` ложно. Инструкция `assume` является линейной, то есть на графе потока управления имеет одно входное ребро и одно выходное, что значительно упрощает анализ.

В данной работе мы не будем рассматривать случаи, когда выполнение произвольной инструкции, кроме инструкций вызова и оператора `throw`, может привести к исключению (так, в Java разыменование нулевой ссылки приведёт к исключению типа `NullPointerException`). С одной стороны, это бы затруднило анализ, так как увеличило бы размер промежуточного представления и потребовало бы адаптации каждого детектора. С другой стороны, ошибки, возникающие при выполнении подобных инструкций, можно обнаружить при помощи специализированных анализов, которые не требуют поддержки анализа исключений. Так, ошибка выхода за границы массива может искаться отдельным детектором (например, при помощи подхода, описанного в статье [6]), поэтому поддержка путей выполнения, создаваемых исключениями типа `ArrayIndexOutOfBoundsException` в языках Java и Kotlin, была бы избыточной.

Для простоты будем считать, что в языке нет глобальных переменных. Они легко моделируются с помощью параметров функции. Кроме этого мы не рассматриваем программы с циклами, так как с точки зрения обработки исключений циклы не вносят существенных изменений по сравнению с условными выражениями.

На рисунке 1 приведён пример ГПУ, который может быть сгенерирован для кода из листинга 7.

```
1 r = -1;
2 try {
3   r = foo(a, b, ...);
4 } catch (e1) {
5   // Catch-block e1
6 } catch (e2) {
7   // Catch-block e2
8 }
9 return r;
```

Листинг 7: Пример кода, обрабатывающего исключения

Listing 7: Example of code with exception handling

На приведённом ГПУ на рисунке 1 есть четыре группы путей:

- путь для выполнения без исключений;
- путь, обрабатывающий исключение `e1`;
- путь, обрабатывающий исключение `e2`. Заметим, что если исключение `e2` является подтипом `e1`, то этот путь будет невыполнимым;
- путь, соответствующий другим исключениям, отличным от `e1` и `e2`.

В зависимости от реализации функции `foo`, любой из этих путей может быть невыполнимым.

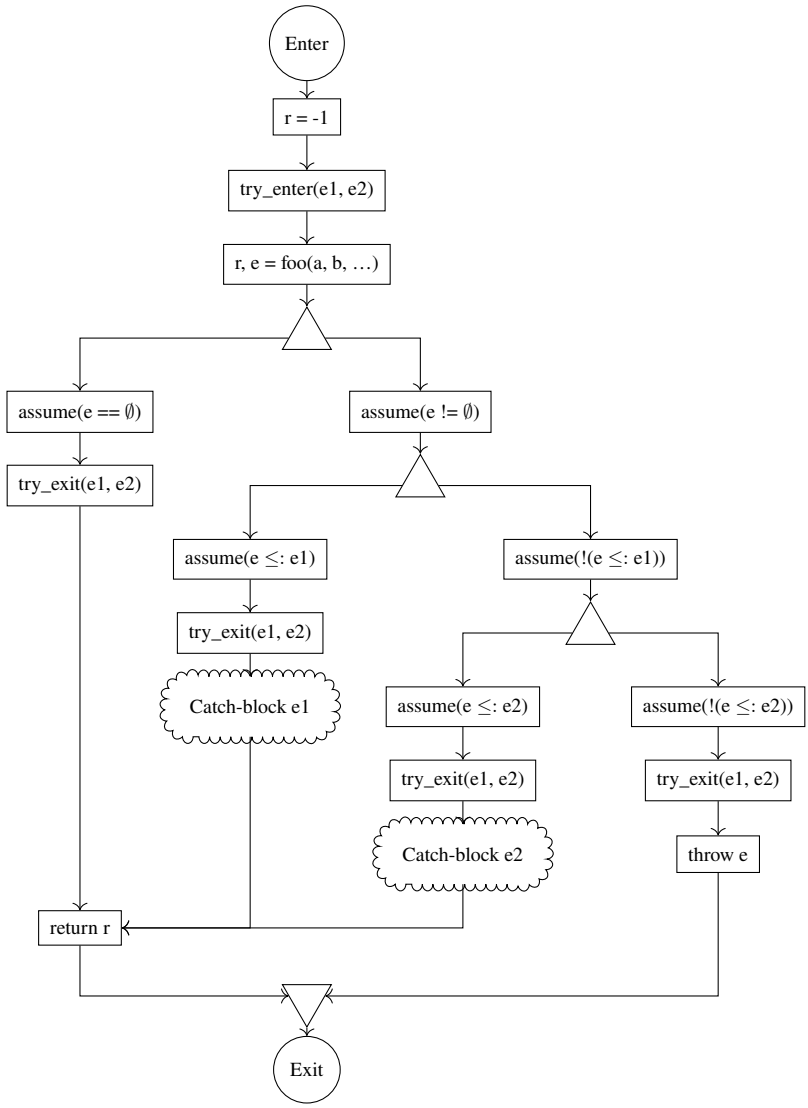

Рис. 1: Пример ГПУ для кода из листинга 7
Fig. 1: Example of a CFG for the code from listing 7

4. Анализ

4.1 Общая схема анализа

Для поддержки межпроцедурного анализа мы используем анализ на основе резюме. При таком подходе после завершения анализа функции создаётся её резюме, которое используется в точках вызова функции. Таким образом, межпроцедурный анализ состоит из двух операций: создание резюме и применение резюме в контексте вызова. Подобный анализ является *контекстно-чувствительным*, так как одно и то же резюме для некоторой функции применяется по-разному в контексте каждого её вызова.

Для анализа отдельной функции используется символьное выполнение с объединением состояний в точках слияния путей. Подробно данный анализ описан в [4]. Здесь отметим, что этот анализ имеет чувствительность к путям, выполняет моделирование полей структур, указываемых ячеек памяти и элементов массивов. На его базе реализовано множество детекторов, в том числе ошибки разыменования нулевых указателей, деления на ноль, переполнения буфера, утечки ресурсов.

Перед запуском символьного выполнения для каждой функции запускается дополнительный движок анализа, основанный на анализе потока данных (DFA-engine) [7], целью которого является поиск недостижимого кода. Движок помечает недостижимые рёбра, которые он сможет обнаружить. При анализе объединений на ГПУ учитываются только достижимые входные рёбра. Таким образом, анализ недостижимого кода может улучшить точность основного анализа.

В данной работе необходимо дополнить анализ путями выполнения, связанными с обработкой исключений, что обеспечивается использованием описанного промежуточного представления. Также требуется расширить DFA-движок для определения новых источников недостижимого кода.

4.2 Анализ потока данных

Общая схема анализов в DFA-движке выглядит следующим образом:

1. Для каждой инструкции создаётся передаточная функция. Передаточная функция для входного состояния для некоторой полурешётки возвращает выходное состояние.
2. Запускается итеративный анализ по ГПУ с использованием передаточных функций.
3. При необходимости, информация сохраняется в резюме функции.
4. Выполняется обход ГПУ, который проверяет состояния на входных рёбрах и саму инструкцию. Если можно сделать вывод о недостижимости, то выходное ребро помечается как недостижимое.
5. Информация о недостижимости распространяется на последующие инструкции базового блока. Кроме того, если все входные рёбра в точке объединения

путей недостижимы, то выходное ребро так же помечается как недостижимое.

Различные анализы выполняются поочерёдно и дополняют результаты друг друга. Если ребро помечено как недостижимое, то все последующие анализы не будут его учитывать.

Для анализа исключений мы расширили DFA-движок анализом функций, которые бросают исключения только определенных типов (раздел 4.2.2). В дополнение к основному анализу был реализован вспомогательный анализ, сохраняющий обрабатываемые исключения (раздел 4.2.1). Реализованные анализы имеют потоковую чувствительность без чувствительности к путям выполнения.

4.2.1 Анализ, сохраняющий обрабатываемые исключения

Целью данного анализа является определение для каждого ребра на ГПУ функции типов исключений, которые в этой точке обрабатываются. Данный анализ в DFA-движке выполняется первым.

Анализ реагирует на инструкции `try_enter(e1, e2, …)` и `try_exit(e1, e2, …)`: при входе в `try-catch`-конструкцию обрабатываемые типы исключений добавляются в соответствующее множество, а при выходе из неё типы исключений, обработка которых заканчивается в этой точке, удаляются из множества. В точке сбора происходит пересечение множеств.

Данный анализ является вспомогательным. Некоторые детекторы используют его результаты для подавления неактуальных предупреждений. Например, они используются, чтобы подавлять неактуальные предупреждения о разыменовании нулевой ссылки, если исключение типа `NullPointerException` обрабатывается (см. листинг 6).

4.2.2 Анализ бросаемых исключений

Элементами полурешётки является множество типов бросаемых исключений. Нижним элементом является пустое множество. Также выделен специальный элемент, являющийся верхним элементом, означающий все возможные варианты. Оператором сбора является объединение множеств бросаемых исключений. С целью производительности, мы ограничили размер множества десятью элементами. Если размер превышен, то множество заменяется на верхний элемент.

Полурешётка имеет следующую семантику: если на некотором ребре значением полурешётки является множество Exc, то это значит, что при выполнении программы могут быть брошены только перечисленные исключения. Невозможна ситуация, когда бросается исключение $e \notin Exc$.

Передаточные функции анализа:

$$TF[throw\ e] \ = \ Exc \rightarrow \{e\}$$

$$TF[r,\ e\ =\ foo(...)]\ =\ Exc \to Exc \cup summary(foo)$$

$$TF[r,\ e\ =\ foo(...); summary(foo) = \emptyset]\ =\ Exc \to \top$$

$$TF[r =\ foo(...)]\ =\ Exc \to Exc$$

$$TF[assume\ e\ ==\ \emptyset;\ r,\ e\ =\ call]\ =\ Exc \to \emptyset$$

$$TF[assume\ e \leq: type;\ r,\ e\ =\ call]\ =\ Exc \to \emptyset$$

$$TF[assume\ !(e \leq: type);\ r,\ e\ =\ call]\ =\ Exc \to Exc \setminus \{type\}$$

Для инструкции бросания исключения `throw e` результирующее множество содержит только это исключение. Для инструкции вызова функции `r, e = foo(...)` ко множеству возможных исключений добавляется множество из резюме этой функции. Если по каким-то причинам для вызова функции нет резюме ($summary(foo) = \emptyset$), то мы используем верхний элемент решётки, предполагая, что может быть брошено любое исключение. Для функций, не бросающих исключение, используется передаточная функция, которая ничего не меняет.

Если функция не бросила исключение (`assume e == 0; r, e = call`), то используется нижний элемент решётки. Инструкция `assume e` $\leq:$ `type; r, e = call` означает, что было перехвачено исключение определённого типа. При выполнении данного кода никакие другие исключения не могут быть брошены, поэтому мы также используем нижний элемент. И для случая, когда какое-то исключение не было поймано (`assume !(e` $\leq:$ `type); r, e = call`), из множества возможных исключений убирается этот тип.

Результаты данного анализа могут быть использованы для пометки исключений для двух случаев:

1. `assume e` $\leq:$ `type` — выходное ребро помечается как недостижимое, если множество исключений не содержит данный тип: $type \notin Exc$. Этот случай соответствует ребру входа в `catch`-блок;
2. `assume !(e` $\leq:$ `type)` — выходное ребро помечается как недостижимое, если множество исключений содержит только данный тип: $Exc = \{type\}$. Данный случай соответствует ребру обхода `catch`-блока.

Описанный анализ позволяет понять, что в листинге 8 блок `catch(...)` недостижим. Последний факт позволяет анализу ошибок деления на ноль не выдать ложное предупреждение, так как на всех остальных путях переменной `x` присваивается ненулевое значение.

```
1 void foo(int a) {
2   if (a <= 0) throw 8;
3 }
4 int bar(int a) {
5   int x = 0;
6   try {
7     foo(a);
8     x = 1;
```

```
 9     } catch(int e) {
10       x = 2;
11     } catch(...) { }
12     return 100 / x;
13   }
```

Листинг 8: Недостижимый `catch`-блок

Listing 8: Unreachable `catch`

Частный, но важный случай, находимый этим анализом, — это функции, которые точно не бросают исключения. Для вызова таких функций, все рёбра, соответствующие обработке исключений, помечаются как недостижимые.

4.2.3 Анализ точно брошенных исключений

Некоторые функции всегда бросают исключения при выполнении. Для анализа таких функций был разработан несложный анализ, задача которого — поместить в резюме информацию, что нормальное выполнение функции невозможно и функция всегда бросает исключение.

Рёбра ГПУ, соответствующие нормальному выполнению таких функций, помечаются как недостижимые.

4.3 Детектор необработанных исключений

Одна из часто встречающихся уязвимостей при обработке исключительных ситуаций — необработанные исключения [8]. Данная уязвимость может привести к аварийному завершению программы, что может быть использовано злоумышленниками при атаке типа «отказ в обслуживании».

В листинге 9 приведён пример необработанного исключения. В строке 5 производится вызов метода `parseInt`, который может бросить исключение типа `NumberFormatException`, если передаваемая строка не является числом. Так как исключение данного типа не обрабатывается, а передаваемый аргумент не проверяется, то программа может завершиться аварийно при некорректных значениях.

```
1  public static void main(String[] args) {
2    int threadsNum = 1;
3    for (int i = 0; i < args.length; ++i) {
4      if (args[i].startsWith("-j")) {
5        threadsNum =
             Integer.parseInt(args[i].substring("-j".length()));
6      }
7    }
8    System.out.println("Number of threads: " + threadsNum);
9  }
```

Листинг 9: Пример необработанного исключения

Listing 9: Example of an unhandled exception

Для поиска необработанных исключений был разработан специальный детектор.

Детектор использует межпроцедурный анализ для поиска исключений, которые бросаются в программе, но не обрабатываются и могут аварийно завершить эту программу.

При анализе используются три множества типов исключений[2]:

- $Expected$ — типы исключений, которые ожидаются в данной точке программы (то есть, обрабатываются `try-catch` конструкцией). Формируется при помощи результатов вспомогательного анализа, описанного в разделе 4.2.1;
- $Uncaught$ — типы исключений, которые не были обработаны в `catch`-блоке. В точке сбора происходит объединение данных множеств;
- $Processed$ — типы исключений, которые были обработаны в `catch`-блоке. В точке сбора происходит пересечение данных множеств.

При достижении вызова функции исключения, которые могут быть брошены в результате этого вызова, разделяются на пойманные и не пойманные, формируя множества $Processed$ и $Uncaught$ соответственно. При этом трассы предупреждений расширяются меткой `call` с информацией о вызываемой функции.

При достижении инструкции промежуточного представления, соответствующей оператору `throw`, тип исключения ищется во множествах $Processed$ и $Uncaught$. Возможны три следующих ситуации:

- Если исключение находится во множестве $Processed$, то оно было поймано `catch`-блоком, но в итоге было брошено заново — тогда трасса для этого типа исключения расширяется меткой `rethrow`, тип удаляется из множества $Processed$ и добавляется во множество $Uncaught$;
- Если исключение находится во множестве $Uncaught$, то оно не было поймано `catch`-блоком, но достигло `finally`-блока и после его завершения было брошено заново — в этом случае трасса для этого типа исключения остаётся прежней, тип остаётся во множестве $Uncaught$;
- Если исключения нет ни в одном из множеств, то такая ситуация соответствует созданию и бросанию нового типа исключения через оператор `throw` — в таком случае создаётся новая трасса для этого типа исключения с меткой `throw` и информация добавляется во множество $Uncaught$.

При достижении выхода из функции, её сигнатура сверяется с функцией `main` — если текущая функция является точкой входа в программу, то для всех исключений из множества $Uncaught$ должно быть выдано соответствующее предупреждение. В ином случае, если функция не является функцией `main`, то все исключения из множества $Uncaught$ добавляются в резюме анализируемой функции и используются при обработке вызовов.

Для языка C++ помимо функции `main` проверяются все деструкторы, так как де-

[2]Все три множества являются частью множества B состояния программы

структор может быть вызван во время раскрутки стека при уже брошенном исключении, и если в этот момент будет брошено ещё одно исключение, то будет вызвана функция `std::terminate`, завершающая программу [9].

Также нужно отметить, что в языке C++ исключение может завершать программу аварийно и при использовании так называемых «exception specifications»: если функция определяет, что из неё могут быть брошены исключения типов `e1`, …, `eN`, но при этом бросает исключение типа `u` такое, что `!(u <: ei)` для всех `i`, то будет произведён вызов функции `std::terminate`. Поэтому, предупреждение о необработанном исключении будет выдано и в том случае, если множество $Uncaught$ содержит типы исключений, которые не описаны в объявлении функции.

Для описания свойств поведения библиотечных функций, важных для детекторов, анализатор *Svace* использует так называемые спецификации, которые моделируют эти свойства при помощи вызовов специальных функций. Мы добавили спецфункции `sf_could_throw` и `sf_could_throw_pedantic`, означающие, что из функции может быть брошено исключение определённого типа. Версия `pedantic` используется для случаев, когда возникновение исключений маловероятно.

Отметим, что не все типы исключений одинаково критичны — некоторые возникают в очень редких случаях, а другие могут указывать о наличии серьёзных проблем в работе программ. Например, исключение типа `OutOfMemoryError` в JVM вряд ли должно обрабатываться приложением, так как оно не сможет корректно восстановиться после данного исключения. Соответственно, можно разделить типы предупреждений на те, которые относятся к критичным типам исключений, и те, которые относятся к исключениям, которые могут быть не обработаны или не являются критичными. Для пометки типов исключений, как те, которые обязаны быть обработаны, в спецификациях *Svace* для Java и Kotlin была добавлена аннотация `MustBeCaught`, при помощи которой можно пометить необходимый тип исключения. Этот признак наследуется: если класс `A` имеет аннотацию `MustBeCaught` и является родителем класса `B`, то исключение типа `B` тоже должно быть поймано.

Также стоит отметить, что для некоторых типов исключений предупреждение выдавать и вовсе бессмысленно. Например, исключение типа `AssertionError` в JVM бросается только при проверке инвариантов в программе при помощи ключевого слова `assert`, поэтому пользовательская программа не должна обрабатывать исключения такого типа. К исключениям, для которых предупреждение выдавать бессмысленно, мы точно так же отнесли `ExceptionInInitializerError`, возникающее при брошенном исключении в статическом инициализаторе класса, и `LinkageError`, которое возникает только при загрузке классов и при специфичных случаях использования Reflection API [10].

В коде на C++ исключение типа `std::bad_alloc` может возникнуть в большом количестве мест, поэтому было решено выделить отдельный подтип предупреждения для этого типа исключений.

В некоторых случаях программист может ожидать, что некоторые типы исключений будут завершать программу аварийно. Такое возможно, например, при первичной

обработке входных аргументов — исключение, которое будет брошено при встрече с некорректным аргументом, не вызовет никаких критических ошибок в работе приложения, если не будет поймано. В таких случаях, программист может пометить в функции `main` те исключения, которые он ожидает (при помощи ключевого слова `throws` в случае Java, при помощи аннотации `Throws` в случае Kotlin или при помощи «exception specification» в случае C++). В этом случае, детектор не будет выдавать предупреждения для указанных типов исключений.

5. Генерация промежуточного представления

Инструмент *Svace* осуществляет поиск ошибок в два этапа:

1. С помощью модифицированного компилятора для соответствующего языка создаётся низкоуровневое промежуточное представление;
2. Полученное промежуточное представление подаётся на вход анализатору, который преобразует его в унифицированное промежуточное представление (см. главу 3), для которого выполняется анализ.

Наше промежуточное представление генерируется разными компиляторами: Javac OpenJDK [11], Koltinc [12] и Clang [13].

5.1 Информация об исключениях в промежуточном представлении для C++

Используемый в *Svace* для C++ компилятор Clang использует разное ABI при генерации промежуточного представления для ОС Linux и Windows, в результате промежуточное представление для них существенно отличается.

Промежуточное представление для Linux имеет Itanium ABI [14]. В анализаторе *Svace* мы полностью поддерживаем все инструкции, генерируемые подобным представлением. Для анализа нам важны следующие инструкции:

- `invoke` — чтобы знать, для каких вызовов исключения обрабатываются;
- `landingpad` — для определения типов обрабатываемых исключений[3];
- `resume`, `__cxa_rethrow`, `__cxa_throw` — чтобы знать о брошенном исключении;
- `__cxa_call_unexpected` — чтобы знать, в каких случаях программа может завершиться вызовом `std::terminate`;
- `llvm.eh.typeid.for` — для получения типа исключения, на соответствующий обработчик которого сделан переход.

[3]Мы не проводим явной трансформации этой инструкции в описанные ранее `try_enter(e1, e2, …)` и `try_exit(e1, e2, …)`. Это можно сделать разными способами, например, расставляя возле каждой инструкции `invoke` пару из `try_enter(e1, e2, …)` и `try_exit(e1, e2, …)`, используя типы исключений, обрабатываемые соответствующим `landingpad`. Но мы решили оставить данную инструкцию в нашем представлении, необходимым образом модифицировав движок анализа.

Для ОС Windows в LLVM используется представление, отличающееся от Itanium ABI [15]. На момент написания статьи, из всех специфичных инструкций данного промежуточного представления в *Svace* поддерживается только инструкция `__CxxThrowException`.

Таким образом, мы решили расширить существующее промежуточное представление *Svace* инструкциями LLVM для обработки исключений. Создание нового промежуточного представления является нетривиальной задачей и не является необходимостью в данном случае.

5.2 Информация об исключениях в JVM-байт-коде

В JVM, в отличие от LLVM, нет специализированных инструкций, относящихся к обработке исключений. Для хранения информации об обрабатываемых исключениях в JVM используется структура данных, именуемая «exception table», которая содержит: тип обрабатываемого исключения; отрезок инструкций, на котором данное исключение обрабатывается; смещение инструкции, на которую должен быть осуществлён переход при пойманном исключении.

Мы используем библиотеку ASM [16] для чтения JVM-class-файлов, поэтому у нас нет проблем с корректным извлечением данной информации. Тем не менее, большую проблему составляет построение промежуточного представления анализатора на основе этой информации. Так как библиотека ASM читает байт-код сверху вниз, то построение ГПУ осуществляется в аналогичном порядке (при этом смещение каждой метки в момент чтения неизвестно). Это приводит к тому, что невозможно для каждой инструкции в момент её обработки определить, приводит ли её выполнение к выходу из текущего обработчика исключений.

Мы решили данную проблему следующим образом. Когда анализатор встречает начало отрезка, обрабатывающего исключения, то он генерирует искусственное ветвление из инструкций `Try (e1, e2, …)` и `Catch (e1, e2, …)`: первая инструкция просто ведёт на следующую инструкцию байт-кода, а вторая — на первую инструкцию `catch`-блока, обрабатывающего типы исключений `e1, e2, ...`. Первая соответствует ранее описанной инструкции `try_enter(e1, e2, …)`. Вторая инструкция всегда помечается недостижимой и не является путём, через который может пойти программа. Её наличие необходимо лишь для того, чтобы показать анализу точки, в которых заканчивается обработка типов исключений `e1, e2, …` — этими точками являются непосредственные постдоминаторы базовых блоков, в которых находятся инструкции `Try (e1, e2, …)` и `Catch (e1, e2, …)`, а также все выходы из метода. Добавляя во все эти точки инструкцию `try_exit(e1, e2, …)` получим представление, описанное ранее[4].

[4]Стоит оговориться, что мы не проводим данную трансформацию явно в нашем промежуточном представлении. Вместо этого мы модифицировали анализ из раздела 4.2.1, чтобы передаточная функция для инструкции `Catch (e1, e2, …)` не добавляла типы `e1, e2, …` во множество обрабатываемых исключений. Таким образом, пересечение множеств в точке сбора позволяет добиться того же эффекта, который бы дала обработка инструкции `try_exit(e1, e2, …)`.

5.3 Представление вызовов в ГПУ

LLVM-биткод использует инструкции `call` и `invoke` для вызовов функций. Инструкция `call` используется в тех случаях, когда не требуются переходы на базовые блоки, обрабатывающие исключения. Для остальных случаев используется инструкция `invoke`, которая может передавать управление в различные точки программы, в зависимости от того, возникло ли исключение при вызове [17].

Так как в JVM-байт-коде нет подобного рода инструкций вызова, которые бы генерировались компиляторами, то мы решили использовать похожий подход при построении ГПУ внутри анализатора. Если в `try`-блоке присутствуют инструкции вызова, то мы считаем, что эти вызовы могут сгенерировать любое отслеживаемое `catch`-блоками исключение (а также любое исключение, описанное в объявлении метода при помощи ключевого слова `throws`) — вместо обычной инструкции вызова генерируется ветвление, одна ветвь которого соответствует нормальному завершению вызова, а другая предполагает, что возникло исключение, и ведёт в один из `catch`-блоков (если соответствующего `catch`-блока найдено не было, то ветвь ведёт на выход из метода по исключению).

Мы генерируем переходы только для вызовов, исключения от которых обрабатываются в вызывающей функции. Например, в листинге 10 функция `bar`, так же, как и функция `foo`, может бросить исключение. Мы не генерируем явных переходов для этого случая и оставляем задачу точного определения выполненных инструкций анализу.

```
1  try {
2    r = foo(a, b, ...);
3    return r;
4  } catch (e1) {
5    // Catch-block e1
6  } catch (e2) {
7    // Catch-block e2
8  }
9  r = bar(a, b, ...);
10 return r;
```

Листинг 10: Пример кода, обрабатывающего исключения

Listing 10: Example of code with exception handling

5.4 Представление `finally`-блоков в JVM

Компиляторы языков Java и Kotlin при генерации `try-catch-finally` конструкций генерируют дублирующийся код, соответствующий коду из `finally`-блока, и добавляют его после всех инструкций, после которых происходит выход из `try-catch-finally` конструкции — это требуется самой семантикой `finally`-блока, так как код из него должен выполняться всегда. Но дублирование `finally`-блоков неудобно тем, что исходный код программы перестаёт взаимно-однозначно отображаться на байт-код — каждой строке исходного кода может соответствовать

сразу несколько инструкций из промежуточного представления.

Чтобы избежать ложных срабатываний, вызванных подобной кодогенерацией, мы используем модифицированные компиляторы языков Java и Kotlin, в которых используем старую схему генерации `finally`-блоков при помощи подпрограмм [18] (и их инструкций `jsr` и `ret`), позволяющих генерировать код из `finally`-блока лишь единожды: инструкция `jsr` осуществляет переход в `finally`-блок, сохраняя адрес возврата на стек, а инструкция `ret` возвращается по сохранённому адресу при завершении `finally`-блока. Подробнее о деталях реализации можно прочитать в [19].

Стоит также отметить один интересный нюанс. При использовании конструкции `try-finally` без блоков `catch`, вследствие эвристик, описанных ранее, считается, что никакое исключение не бросается внутри `try`-блока. Поэтому, для более консервативного анализа мы считаем, что при наличии в коде `finally`-блока, инструкции вызова в соответствующем ему `try`-блоке могут бросить исключение самого общего типа — типа `Throwable` (но только при условии, что нет более конкретного типа исключения, бросаемого из `try`-блока).

6. Результаты

Для оценки влияния анализа исключений на результат мы взяли старую версию анализатора *Svace*, в которой была упрощённая обработка исключений. Во-первых, в ней отсутствовали специализированные DF-анализы для исключений. Во-вторых, для JVM использовалась упрощённая схема построения промежуточного представления: считалось, что метод может бросить только те исключения, которые указаны в его определении при помощи ключевого слова `throws`, поэтому соответствующие пути в ГПУ строились только для этих исключений.

Для оценки результатов были выбраны следующие проекты с открытым исходным кодом:

- JVM:
  - Операционная система Android 11 [20]. Проект содержит более 33 миллионов строк кода на языке Java.
  - Компилятор языка Kotlin версии 1.4.10 [12]. На данный момент, это самый крупный проект с открытым исходным кодом на языке Kotlin. Проект содержит приблизительно 2 миллиона строк Kotlin кода и 1.1 миллиона строк Java кода.
- C++:
  - Kodi версии 17.0a2-Krypton [21]. Проект содержит около 650 тысяч строк кода на C++.
  - Операционная система Tizen версии 2.3 [22] размером более 6.6 миллионов строк кода, из которых 1.7 миллионов написаны на C++.

В таблицах 1 и 2 приведены группы детекторов, результаты анализа для которых су-

щественно изменились. Строка «Суммарно» учитывает предупреждения, выданные всеми детекторами (в том числе, не перечисленными в таблице отдельно), кроме детектора необработанных исключений — его результаты представлены в таблице 3.

В таблицах 1 и 2 представлены только результаты для проектов на языках Java и Kotlin, так как нами не было замечено существенных изменений на проектах на языке C++: на проекте Tizen стало на 19 ложных предупреждений меньше и на 1 истинное предупреждение меньше (при общем количестве в 49777 предупреждений), на проекте Kodi стало на 6 ложных предупреждений больше (при общем количестве в 25104 предупреждения).

Как видно из результатов, использование специального анализа с учётом исключений повышает качество результатов. Такой анализ позволяет существенно сократить количество ложных предупреждений, а также найти немного больше истинных.

Таблица 1: Результаты сравнения версий *Svace* для Android 11

Группа детекторов	Предупр.		Истинных		Ложных		Процент изменений
	До	После	Появ.	Исч.	Появ.	Исч.	
Разыменование нулевой ссылки	6267	6097	35	86	34	136	0.81%
Утечка ресурсов	636	782	74	9	42	41	10.06%
Избыточное сравнение	1377	1342	4	11	9	37	1.53%
Недостижимый код	1244	1749	41	18	36	132	9.57%
Суммарно	18103	18289	205	73	197	565	2.76%

Таблица 2: Результаты сравнения версий *Svace* для Kotlinc-1.4.10

Группа детекторов	Предупр.		Истинных		Ложных		Процент изменений
	До	После	Появ.	Исч.	Появ.	Исч.	
Разыменование нулевой ссылки	2188	1534	11	10	14	667	29.89%
Утечка ресурсов	16	28	7	2	7	0	-12.5%
Избыточное сравнение	31	32	0	0	2	1	-3.23%
Недостижимый код	87	136	30	0	26	7	12.64%
Суммарно	6351	5723	55	7	63	709	10.93%

Таблица 3: Результаты детектора необработанных исключений

Проект	Всего	Классифицировано	Истинных	Ложных	Процент истинных
Android 11	155	152	87	65	57.24%
Kotlinc-1.4.10	51	46	34	12	73.91%
Kodi	175	155	151	4	97.42%
Tizen-2.3	422	184	183	1	99.46%

7. Похожие работы

В [23] описывается подход расширения статического анализатора Java для Kotlin кода. Использовался инструмент `SECUCHECK` [24]. Авторы явным образом указывают, что не знают ни одного статического анализатора, поддерживающего глубокий анализ потока данных для языка Kotlin, что косвенно свидетельствует об актуальности нашей работы, так как мы описываем именно такой анализ. Кроме этого, низкоуровневое представление, предлагаемое в нашем подходе, позволяет анализировать Java и Kotlin схожим образом. Хотя наш подход и предполагает существенные усилия на поддержку языка Kotlin, однако они локализованы в генерации промежуточного представления.

Clang Static Analyzer [25] включает в свой состав анализатор на основе символьного выполнения. К сожалению, мы не смогли получить значимые результаты на реальных проектах, которые можно было бы сравнить с нашим анализатором. Просмотренные срабатывания не относились к коду, связанному с исключениями. Поэтому мы создали синтетические тесты для детектора деления на ноль, чтобы выяснить возможности анализатора. Нам не удалось найти отличия, обусловленные обработкой исключений, кроме ситуации, когда обрабатывается базовый тип от бросаемого. Для таких тестов наша реализация ошибалась, так как мы не поддержали отношения подтипов для классов C++ (см. листинг 11).

В C++ есть множество правил преобразования типов, например исключение типа `long` будет перехватываться блоком `catch(long& e)`, то есть ссылочный тип является подтипом основного типа. Поэтому реализация операции `(e` $\leq$`: type)` для C++ является нетривиальной задачей.

```
1  class A {}
2
3  class B : public A {}
4
5  void foo() {
6    throw B();
7  }
8
9  int test() {
10   int x = 0;
11   try {
12     foo();
13   } catch (A &a) {
14     x = 1;
15   }
16   return 100 / x;
17 }
```

Листинг 11: Пример кода, где необходим анализ, учитывающий подтипы

Listing 11: Example of a code that requires subtype aware analysis

В работе [26] описывается подход к анализу исключений для C++. Там так же используется предварительная трансформация программы в представление без ис-

ключений. В отличии от их подхода, мы используем более низкоуровневое промежуточное представление, которое подходит для анализа не только C++, но и других языков.

Анализатор SharpChecker [27] для языка C# реализован на основе компилятора Roslyn и использует возможности компилятора для построения ГПУ. Реализация анализа исключений [28] в анализаторе имеет не только потоковую-чувствительность, но и чувствительность к путям. На наш взгляд чувствительность к путям не интересна для реализации анализов недостижимого кода, так как обработка исключений редко содержит условные выражения. Но добавление чувствительности к путям может быть полезно для детектора необработанных исключений. Например, в примере из листинга 12 на строке 16 не может возникнуть исключение при вызове конструктора, так как передаваемый в него аргумент заведомо не равен `null`. При этом наш детектор выдаёт здесь ложное срабатывание, так как не обладает чувствительностью к путям.

```
1  class Printer {
2      private final String s;
3
4      public Printer(String s) {
5          if (s == null) throw new IllegalArgumentException();
6          this.s = s;
7      }
8
9      public void print() {
10         System.out.println(s);
11     }
12 }
13
14 class Main {
15     public static void main(String[] args) {
16         new Printer("Hello, world!").print();
17     }
18 }
```

Листинг 12: Ложное срабатывание детектора необработанных исключений

Listing 12: False positive warning by unhandled exceptions detector

Также мы провели сравнение со статическими анализаторами с открытым исходным кодом SpotBugs [29] и Infer [30]. Оба анализатора были запущены на проекте Android 11. При помощи SpotBugs был проанализирован Kotlinc-1.4.10.

SpotBugs обнаруживает следующие типы дефектов, которые ищутся и при помощи *Svace*:

- Разыменование нулевого указателя на пути с брошенным исключением;
- Утечка ресурсов из-за необработанного исключения;
- Объект типа `Lock` захватывается, но не освобождается в случае брошенного исключения.

Результаты сравнения для данных типов предупреждений приведены в таблице 4[5]. В анализаторе Infer присутствует только один тип предупреждения, интересный в данной работе, который имеет и наш инструмент — утечка ресурсов из-за брошенного исключения. Результаты сравнения представлены в таблице 5[5].

Таблица 4: Результаты сравнения *Svace* со SpotBugs

Истинных дефектов найдено	SpotBugs и *Svace*	49
	Только SpotBugs	2
	Только *Svace*	106
Ложных выдано	SpotBugs и *Svace*	10
	Только SpotBugs	57
	Только *Svace*	95

Таблица 5: Результаты сравнения *Svace* с Infer

Истинных дефектов найдено	Infer и *Svace*	3
	Только Infer	1
	Только *Svace*	112
Ложных выдано	Infer и *Svace*	1
	Только Infer	5
	Только *Svace*	73

Только один из трёх пропущенных истинных дефектов является проблемой анализа исключений. Проблема заключается в том, что *Svace* считает, что нативный метод в языках Java и Kotlin не может бросить исключение, хотя в общем случае это неверно. Это приводит к тому, что в примере из листинга 13 наш инструмент не находит проблему с неосвобождённым объектом типа `Lock` из-за брошенного исключения. Тем не менее, SpotBugs, находящий этот дефект, выдаёт большое количество ложных предупреждений, так как считает, что любой метод может кинуть любое исключение. Также недостатком можно назвать то, что наш инструмент выдаёт большое количество ложных предупреждений типа утечка ресурсов, но это проблема не анализа исключений, а самого детектора, которую в будущем планируется исправить. Несмотря на описанные проблемы, наш подход позволяет находить гораздо больше истинных дефектов, чем инструменты SpotBugs и Infer.

```
1 class Foo {
2   private final ReentrantReadWriteLock lock = new
        ReentrantReadWriteLock();
3   private final ReentrantReadWriteLock writeLock = lock.writeLock();
4
```

[5]Результаты для *Svace*, представленные в этих таблицах, не являются до конца полными, так как мы не смогли классифицировать абсолютно все предупреждения, выданные нашим инструментом, из-за большого их количества. Таким образом, реальное количество истинных и ложных предупреждений может быть выше, чем указано в таблице.

```
5     private final native void nativeFoo();
6
7     public void foo() {
8       writeLock.lock();
9       nativeFoo();
10      writeLock.unlock();
11    }
12  }
```

Листинг 13: Пример пропущенного дефекта с вызовом нативного метода

Listing 13: Example of the false negative warning with the native method call

8. Заключение

В статье описан подход к статическому анализу программ на языках с обработкой исключений. В рамках работы было предложено промежуточное представление, позволяющее анализировать программы на разных языках программирования, содержащих конструкции для работы с исключениями. Приведены описания легковесных анализов, работа которых позволяет отсеять множество недостижимых путей, возникающих при работе с исключениями. Описан алгоритм работы детектора для поиска необработанных исключений.

Все перечисленные выше результаты были реализованы и протестированы в инструменте статического анализа *Svace* для языков C++, Java и Kotlin. Для каждого языка в статье были указаны особенности генерации промежуточного представления.

Результаты работы позволили повысить общее качество анализа. Также было проведено сравнение наших результатов с другими работами. Предложенное нами решение, в отличие от рассмотренных нами работ, позволяет использовать единое представление для анализа нескольких языков.

В будущем планируется улучшение реализованных анализов, в частности, поддержка вычисления отношения подтипов для языка C++.

Список литературы / References

[1]. Common Weakness Enumeration. CWE-703: Improper Check or Handling of Exceptional Conditions. URL: https://cwe.mitre.org/data/definitions/703.html (дата обращения 28.01.2022).

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

[3]. A. Borodin, A. Goremykin, S. P. Vartanov и A. Belevantsev. Searching for Taint Vulnerabilities with Svace Static Analysis Tool. *Programming and Computer Software*, 47(6):466—481, 2021.

[4]. A. Borodin и I. Dudina. Intraprocedural Analysis Based on Symbolic Execution for Bug Detection. *Programming and Computer Software*, 47(8):858—865, 2021.

[5]. Common Weakness Enumeration. CWE-460: Improper Cleanup on Thrown Exception. URL: https://cwe.mitre.org/data/definitions/460.html (дата обращения 28.01.2022).

[6]. И. А. Дудина, В. К. Кошелев и А. Е. Бородин. Поиск ошибок доступа к буферу в программах на языке c/c++. *Труды Института системного программирования РАН*, 28(4):149—168, 2016.

[7]. R. R. Mulyukov и A. E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. *Proceedings of the Institute for System Programming of the RAS*, 28(5):145—158, 2016.

[8]. Common Weakness Enumeration. CWE-248: Uncaught exception. URL: https://cwe.mitre.org/data/definitions/248.html (дата обращения 28.01.2022).

[9]. Working Draft, Standard for Programming Language C++. 18.5.1 The std::terminate() function. 27 нояб. 2017. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf (дата обращения 12.08.2022).

[10]. Oracle® Java™ Documentation: The Java™ Tutorials. Trail: The Reflection API. URL: https://docs.oracle.com/javase/tutorial/reflect/index.html (дата обращения 17.05.2022).

[11]. OpenJDK. The Java programming language Compiler Group. URL: https://openjdk.org/groups/compiler/ (дата обращения 12.08.2022).

[12]. JetBrains/kotlin. The Kotlin Programming Language. URL: https://github.com/JetBrains/kotlin (дата обращения 22.04.2022).

[13]. Clang: a C language family frontend for LLVM. URL: https://clang.llvm.org/ (дата обращения 12.08.2022).

[14]. LLVM Compiler Infrastructure. Exception Handling in LLVM. Itanium ABI Zero-cost Exception Handling. 11 авг. 2022. URL: https://llvm.org/docs/ExceptionHandling.html#itanium-abi-zero-cost-exception-handling (дата обращения 11.08.2022).

[15]. LLVM Compiler Infrastructure. Exception Handling in LLVM. Exception Handling using the Windows Runtime. 11 авг. 2022. URL: https://llvm.org/docs/ExceptionHandling.html#exception-handling-using-the-windows-runtime (дата обращения 11.08.2022).

[16]. ASM. URL: https://asm.ow2.io/ (дата обращения 18.08.2022).

[17]. LLVM Compiler Infrastructure. Exception Handling in LLVM. Try/Catch. 11 авг. 2022. URL: https://llvm.org/docs/ExceptionHandling.html#try-catch (дата обращения 11.08.2022).

[18]. T. Lindholm, F. Yellin, G. Bracha и A. Buckley. The Java® Virtual Machine Specification. Java SE 8 Edition. Exceptions and finally. 13 февр. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.10.2.5 (дата обращения 20.12.2021).

[19]. А. П. Меркулов, С. А. Поляков и А. А. Белеванцев. Анализ программ на языке Java в инструменте Svace. *Труды Института системного программирования РАН*, 29(3):57—74, 2017.

[20]. Android Open Source Project. URL: https://source.android.com/ (дата обращения 22.04.2022).

[21]. xbmc/xbmc. Kodi is an award-winning free and open source home theater/media center software and entertainment hub for digital media. URL: https://github.com/xbmc/xbmc (дата обращения 18.08.2022).

[22]. Tizen Docs. URL: https://docs.tizen.org/ (дата обращения 18.08.2022).

[23]. R. Krishnamurthy, G. Piskachev и E. Bodden. To what extent can we analyze Kotlin programs using existing Java taint analysis tools? *arXiv preprint arXiv:2207.09379*, 2022.

[24]. G. Piskachev, R. Krishnamurthy и E. Bodden. SecuCheck: Engineering configurable taint analysis for software developers. В *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, страницы 24—29. IEEE, 2021.

[25]. Clang Static Analyzer. URL: https://clang-analyzer.llvm.org/ (дата обращения 17.08.2022).

[26]. P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić и A. Gupta. Interprocedural exception analysis for C++. В *European Conference on Object-Oriented Programming*, страницы 583—608. Springer, 2011.

[27]. V. Koshelev, V. Ignatiev, A. Borzilov и A. Belevantsev. SharpChecker: Static analysis tool for C# programs. *Programming and Computer Software*, 43(4):268—276, 2017.

[28]. M. Belyaev и V. Ignatyev. Exception Analysis for Errors Detection in the SharpChecker Static Analyzer for C#. В *2021 Ivannikov Ispras Open Conference (ISPRAS)*, страницы 8—16. IEEE, 2021.

[29]. SpotBugs. Find bugs in Java Programs. URL: `https://spotbugs.github.io/` (дата обращения 22.04.2022).

[30]. Infer. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. URL: `https://fbinfer.com/` (дата обращения 22.04.2022).

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ — студент магистратуры факультета компьютерных наук НИУ ВШЭ, сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM языки.

Vitaly Olegovich AFANASYEV — graduate student at the Faculty of Computer Science, NRU HSE, employee of Institute for System Programming of the RAS. Research interests: compiler technologies, static analysis, JVM languages.

Варвара Викторовна ДВОРЦОВА — студент магистратуры факультета вычислительной математики и кибернетики МГУ имени М. В. Ломоносова, сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ, анализ Golang.

Varvara Viktorovna DVORTSOVA — graduate student at the Faculty of Computational Mathematics and Cybernetics of Moscow State University (MSU), employee of Institute for System Programming of the RAS. Research interests: compiler technologies, static analysis, Golang analysis.

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

Alexey Evgenevich BORODIN — PhD, researcher of Institute for System Programming of the RAS since 2007. Research interests: static analysis for finding errors in source code.