DOI: 10.15514/ISPRAS-2025-37(6)-8

Аннотирование исходного кода для статического анализа

[1] *В.О. Афанасьев, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *Е.А. Велесевич, ORCID: 0009-0007-7850-6860 <evel@ispras.ru>*
[1,2] *Б.В. Орлов, ORCID: 0009-0009-1602-6014 <boris.orlov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

Аннотация. В статье описывается аннотирование исходного кода для статического анализа. Рассмотрены атрибуты C/C++ и аннотации JVM-языков. Приведены основные цели и причины аннотирования исходного кода для статического анализа. Описаны основные аспекты реализации поддержки пользовательских аннотаций в анализаторе Svace.

Ключевые слова: статический анализ; поиск ошибок; уязвимости; анализатор Svace; компиляторная инфраструктура LLVM; виртуальная машина Java; языки программирования C/C++, Java, Kotlin.

Для цитирования: Афанасьев В.О., Бородин А.Е., Велесевич Е.А., Орлов Б.В. Аннотирование исходного кода для статического анализа. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 135–148. DOI: 10.15514/ISPRAS–2025–37(6)–8.

Source Code Annotation for Static Analysis

[1] *V.O. Afanasyev, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *E.A. Velesevich, ORCID: 0009-0007-7850-6860 <evel@ispras.ru>*
[1,2] *B.V. Orlov, ORCID: 0009-0009-1602-6014 <boris.orlov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. This paper describes source code annotations for static analysis. C/C++ attributes and JVM annotations are considered. Primary goals and reasons for source code annotation for static analysis are given. Main implementation aspects of annotations in Svace static analyzer are described.

Keywords: static analysis; search for defects; vulnerabilities; Svace; LLVM; JVM; C/C++; Java; Kotlin.

For citation: Afanasyev V.O., Borodin A.E., Velesevich E.A., Orlov B.V. Source Code Annotation for Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 135-148' (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-8.

1. Введение

Инструменты статического анализа позволяют находить ошибки в исходном коде программ без их запуска и, во многих случаях, без предварительной подготовки программы. Тем не менее, не все свойства программ могут быть выведены автоматически. Для таких ситуаций решением будет добавление подсказок для анализатора об особенностях исходного кода.

Во многих языках есть встроенные средства по предоставлению дополнительной семантики. Например, язык Java предлагает аннотации к исходному коду, которые добавляют семантику программы. Пример такой аннотации – `Nullable`, которая подсказывает, что переменная может иметь нулевое значение.

Аннотации находятся вместе с кодом и не будут потеряны при рефакторинге, либо другом изменении кода. Также они видны программисту, который может поправить их. Поэтому логично будет использовать встроенные средства языка для подсказок статическому анализатору.

В данной работе мы описываем использование пользовательских аннотаций для языков C, C++, Java и Kotlin для улучшения статического анализа программ. Все описанные работы выполнены в инструменте Svace [1, 2]. Аннотации добавляют информацию для анализатора, при этом не являются обязательными. В разделе 2 описываются возможности аннотирования кода в языках C, C++, Java, Kotlin. В разделе 3 приводятся возможные способы использования аннотаций для статических анализаторов. Реализация в инструменте Svace находится в разделе 4. Результаты тестирования приведены в разделе 5.

2. Способы аннотирования кода в языках программирования

2.1 Атрибуты C/C++

Атрибуты – устоявшийся способ передачи компилятору C/C++ дополнительной информации. Так, существуют несколько языковых расширений, поддерживающих синтаксис атрибутов в коде: GNU `__attribute__((...))` [3] и MSVC `__declspec(...)` [4]. Помимо расширений, современные стандарты языков C и C++ описывают атрибуты `[[...]]`, которые можно добавлять практически к любой сущности в коде.

С помощью атрибутов программист может влиять на различные аспекты работы компилятора: на вывод диагностики, например с помощью атрибутов `deprecated` и `fallthrough`, или непосредственно на генерацию машинного кода (с помощью атрибута `omp::parallel`). Таким образом, в зависимости от типа атрибута, компилятор может как сохранять информацию о нем в используемом им промежуточном представлении, так и не сохранять ее. Например, GNU-атрибут `nonnull` в компиляторе Clang непосредственно сохраняется как `nonnull`-атрибут параметра функции в LLVM IR [5].

2.2 JVM-аннотации

Аннотации в языках на основе JVM (Java Virtual Machine), в том числе в Java и Kotlin, позволяют предоставлять метаданные для сущностей в коде. Они применяются к использованиям типов, а также к определениям (в частности, к определениям типов, методов, полей, переменных и параметров) [6].

Аннотации могут присутствовать в исходном коде и удаляться после этапа компиляции – такие аннотации могут обрабатываться самим компилятором, его плагинами [7] или процессорами аннотаций [8] для обнаружения ошибок или генерации дополнительного кода. Также аннотации могут сохраняться в JVM-байткоде и быть доступны во время выполнения программы при помощи инструментов Reflection API [9], например, для средств метапрограммирования.

В языке Kotlin, в отличие от Java, аннотации можно применять и к произвольным выражениям. Но такие аннотации доступны только в момент компиляции, в байткоде они не сохраняются. Единственный способ их обработать – плагины компилятора или процессоры аннотаций [10-11].

Система типов языка Kotlin имеет разделение типов на те, значениями которых может являться `null` (nullable типы) и те, значениями которых `null` быть не может (non-nullable типы) [12]. Однако байткод JVM такого разделения не имеет – все переменные ссылочных типов могут иметь значения `null`, поэтому для предоставления данной информации о типах другим программам, компилятор языка Kotlin сохраняет её в байткоде в виде JVM-аннотаций типов.

3. Использование аннотаций в статическом анализаторе

Аннотирование исходного кода может использоваться с разными целями. Мы рассмотрели следующие случаи:

1. Выступать источником ошибки в том случае, если реальный источник неизвестен (или труден для получения) статическому анализатору.
2. Предоставлять дополнительную семантику о программе, которая может в явном виде не присутствовать в исходном коде.
3. Быть некоторыми инвариантами программы, которые требуется дополнительно проверить статическому анализатору.

Далее подробно опишем каждый из сценариев использования.

3.1 Аннотации как источник ошибки

Популярным подходом к статическому анализу является анализ на основе резюме, при котором выполняется обход функций программы по графу вызовов снизу-вверх начиная с листьев графа. При этом анализатору доступна информация о вызываемых функциях, но ничего не известно о вызывающих.

Помимо этого, функции могут быть публичной частью какой-либо библиотеки и не вызываться нигде в исходном коде. В таком случае информации о контексте вызова вовсе нет.

Пользовательские аннотации позволяют предоставить дополнительную информацию о входных параметрах функций, фактически о требованиях, предъявляемых к контексту вызова функции. Аннотация `Nullable` из листинга 1 является примером такой аннотации. В данном случае сообщается, что в контексте вызова допустимо передавать нулевую ссылку. Поэтому статический анализатор может считать, что функция вызывается с передачей значения `null` в качестве аргумента. Если функция разыменовывает параметр без проверки, то будет выдано сообщение об ошибке.

```
1  public class Example {
2    private final Map<Long, String> idToUserName;
3
4    public void store(long id, @Nullable User user) {
5      idToUserName.put(
6        id,
7        user.getName() // Потенциальное разыменование нулевой ссылки
8      );
9    }
10 }
```

Листинг 1. Пример аннотации Nullable в Java.
Listing 1. Example of Nullable annotation for Java.

3.2 Предоставление дополнительной семантики о программе

Значения полей структур или классов могут быть связаны друг с другом, при этом не всегда такие зависимости могут быть выражены средствами языка программирования.

В листинге 2 показан пример структуры, имеющей два поля: `len` и `buf`. В поле `len` хранится размер массива, на который указывает поле `buf`. Функция `test`, заполняющая массив, содержит ошибку: цикл выходит за границы массива. Проверку `i <= s->len` необходимо заменить на `i < s->len`.

```
1  typedef struct {
2      unsigned int len;
3      char* buf __attribute__((buflen(len)));
4  } S;
5
6  void test(S* s) {
7      for (int i = 0; i <= s->len; ++i) {
8          s->buf[i] = i; // ERROR: Выход за границы при i = len
9      }
10 }
```

Листинг 2. Массив и его длина в качестве полей структуры.
Listing 2. Array and its length as structure fields.

В некоторых случаях статический анализатор может отследить заполнение полей структур и выдать ошибку в месте использования. Но код заполнения может находиться в программе далеко от места его использования, и статическому анализатору будет сложно отследить путь в программе между этими точками, а также проверить выполнимость пути. Наличие атрибута `buflen(len)` позволило бы найти эту ошибку.

В общем случае размер массива может определяться более сложным способом. Размер может быть на единицу больше, чем `len`, если `len` не учитывает завершающий ноль строки. Либо

размер может быть произведением длины массива и размера отдельного элемента. В листинге 3 показан пример более сложной структуры, в которой размер массива определяется по формуле `header + height * width`.

```
typedef struct {
    unsigned int header;
    unsigned int height;
    unsigned int width;
    char* buf __attribute__((buflen(header + height * width)));
} S;

void fill(S* s, unsigned int row, unsigned int col, char val) {
    unsigned int index = s->header + row * s->width + col;
    if (index <= s->header + s->height * s->width) {
        s->buf[index] = val; // ERROR: Выход за границы буфера
    }
}
```

Листинг 3. Формула над полями.
Listing 3. Formula over fields.

Другим примером может служить аннотация `GuardedBy` из Android API [13]. Данная аннотация позволяет указывать, что обращение к члену класса должно происходить только при синхронизации на поле с соответствующим именем. Статический анализатор может выдать ошибку, если обращение к полю производится без синхронизации.

```
public class Example {
    private final Object lock = new Object();

    @GuardedBy("lock")
    private boolean field;

    public void test_ok() {
        synchronized (lock) {
            field = false; // OK
        }
    }

    public void test_bad() {
        field = true; // ERROR: Отсутствует синхронизация
    }
}
```

Листинг 4. Аннотация `GuardedBy`.
Listing 4. `GuardedBy` annotation.

3.3 Проверка корректности инвариантов

Также полезным будет реализация проверок, что переменным, помеченным аннотациями, присваиваются допустимые значения. В этом случае пользователь программы получит предупреждение от анализатора, если присваивающий код нарушает инварианты, заданные аннотациями. В листинге 5 показан пример кода, где поле, аннотированное

интервалом `[-10; 10]`, получает значение за пределами этого интервала. Выдача предупреждения позволит обнаружить ошибку в коде.

```
1  typedef struct {
2      int value __attribute__((value_interval(-10, 10)));
3  } S;
4
5  void test(S* s) {
6      s->value = 200; // ERROR: Выход за границы интервала
7  }
```

Листинг 5. Нарушение инварианта структуры.
Listing 5. Structure invariant violation.

4. Реализация

4.1 Clang

Первым этапом мы определили, какую дополнительную информацию было бы неплохо иметь в анализаторе, и составили соответствующий список атрибутов, приведённый в таблице 1 (разумеется, позже могут быть добавлены и другие атрибуты).

Табл. 1. Атрибуты C/C++.
Table 1. C/C++ attributes.

Название атрибута	Семантика
`buflen(x)`	поле структуры хранит буфер, размер которого задаётся выражением `x`
`value_interval(l, r)`	число принадлежит интервалу `[l; r]`
`taint`	адрес памяти может контролироваться злоумышленником
`taint_int/taint_int(l, r)`	целое число в интервале `[l; r]` может контролироваться злоумышленником
`not_null`	указатель не может быть нулевым
`possible_null`	указатель может быть нулевым
`possible_negative`	целое число может быть отрицательным
`target_dependent`	значение зависит от используемой платформы (например, размер типа `int`)

Можно заметить, что `not_null` очень похож на GNU-атрибут `nonnull`, но новый атрибут применим не только к функциям, но и к переменным и полям структур и классов.

Компилятор Clang, используемый в Svace, уже модифицирован для того, чтобы распознавать различные языковые расширения, поддерживаемые другими компиляторами. Таким образом, мы модифицировали Clang и для того, чтобы он распознавал наши новые атрибуты.

Большинство атрибутов довольно просты в реализации, так как имеют аргументы-константы или не имеют аргументов совсем. Тем не менее, даже для простых атрибутов все же были сделаны проверки правильности использования: корректность интервалов, правильность типов сущностей, к которым указаны атрибуты, а также отсутствие противоречивых атрибутов (`possible_null` и `not_null` не могут быть использованы для одной сущности).

Наибольшую сложность для реализации по нескольким причинам представлял атрибут buflen. Первой причиной было то, что в аргументе атрибута должно быть возможно использовать поля, которые будут объявлены в теле класса позднее (как в листинге 6). То есть атрибут должен обрабатываться после того, как компилятор проведёт синтаксический разбор класса до конца. Как известно, так же должны обрабатываться C++ inline-методы, определенные в теле класса, однако в нашем случае атрибут может быть использован и в C-коде, для которого Clang не поддерживал отложенный разбор. Мы распространили C++ механизм отложенного разбора на язык C, модифицировав функции синтаксического разбора структур так, чтобы во время анализа аргументов `buflen` было доступно полное определение класса.

```
1 struct B {
2     char* buf [[buflen(len)]];
3     size_t len;
4 };
```

Листинг 6. Отложенный синтаксический разбор.
Listing 6. Delayed parsing.

Вторая причина также происходила из правил языка C. В отличие от C++, C не поддерживает использование полей структур без явного указания объекта, к которому относятся поля [14], но в контексте атрибута не существует никакого объекта, который можно использовать. Таким образом, было необходимо реализовать поддержку генерации абстрактного синтаксического дерева в компиляторе Clang для прямого использования полей в аргументах атрибута, что и было проделано.

Третья причина следовала из особенностей работы анализатора Svace. Svace разработан так, чтобы перехватывать обычную сборку проекта [1], следовательно новые атрибуты не должны нарушать работу оригинальной сборки. В случае со стандартными атрибутами (`[[...]]`) никаких проблем нет, так как стандарт говорит, что неизвестные атрибуты должны игнорироваться, а аргументом атрибута может быть произвольная последовательность токенов [14]. Для GNU-атрибутов (`__attribute__`) правила другие – неизвестные атрибуты по-прежнему игнорируются, однако аргументы неизвестных атрибутов должны иметь одну из допустимых форм: идентификатор; идентификатор, за которым следует несколько выражений, разделенных запятой; несколько или ноль выражений, разделенных запятой [3]. Аргументы `buflen` могут быть сложными выражениями (как было показано в листинге 3) и могут вызывать ошибки в оригинальной сборке, которая обрабатывает `buflen` как неизвестный атрибут. Для избежания таких ошибок аргумент `buflen` может быть экранирован двойными кавычками (см. листинг 7), чтобы оригинальный компилятор обрабатывал его как строковую константу (которая, конечно, является и выражением).

```
1 struct B {
2     char* buf __attribute__((buflen("2*len*len+1")));
3     size_t len;
4 };
```

Листинг 7. Экранированный аргумент.
Listing 7. Escaped argument.

4.2 JVM

Анализ JVM-программ в Svace реализован на основе байткода JVM, генерируемого при компиляции [15-17].

С одной стороны, поддержка пользовательских аннотаций при таком подходе довольно проста – для аннотаций должна быть указана опция (Retention), при которой информация об их использовании сохраняется в байткод. Модификаций в компилятор при этом вносить не требуется, в отличии от случая с атрибутами для C/C++.

С другой стороны, невозможно получить информацию об аннотациях, которые существуют только в момент компиляции, но не добавляются в байткод. В частности, как сказано в разделе 2.2, аннотации на произвольных выражениях из Kotlin сохранить в байткод невозможно. Тем не менее, для нашего подхода мы считаем такое ограничение несущественным – интересные для анализа в Svace свойства имеет смысл указывать при помощи аннотаций только для локальных переменных, параметров функций, их возвращаемых значений, а также для полей классов.

Для JVM были поддержаны аннотации, приведённые в табл. 2.

Табл. 2. Аннотации JVM.
Table 2. JVM annotations.

Название аннотации	Семантика
`Interval(l, r)`	число принадлежит интервалу `[l; r]`
`TaintedPtr`	ссылочное значение может контролироваться злоумышленником
`TaintedInterval(l, r)`	целое число в интервале `[l; r]` может контролироваться злоумышленником
`Trusted`	значение используется в чувствительной операции
`TrustedInterval(l, r)`	целочисленное значение используется в чувствительной операции при принадлежности интервалу `[l; r]`
`NotNull`	ссылочное значение не может принимать значение `null`
`Nullable`	ссылочное значение может принимать значение `null`

Как было сказано в разделе 2.2, информация о типах из Kotlin, допускающих значение `null`, тоже сохраняется в байткод JVM. Поэтому их поддержка эквивалентна поддержке аннотаций `NotNull` и `Nullable`.

4.3 Анализатор

В Svace анализ основан на отслеживании интересующих свойств на символьных переменных. Реализация детекторов заключается в проверке свойств для соответствующих инструкций (например, проверка свойства указателя быть нулевым для инструкции разыменования). Реализация большинства атрибутов является тривиальной. Для входных параметров функции она заключается в установке нужных свойств в начале анализа каждой функции. Для полей структур свойства устанавливаются при чтении соответствующего поля.

Нарушение инвариантов проверяется при присваивании полям структур и при передаче аргументов в функции некорректных (на основе соответствующих анализируемых свойств) значений.

Отдельный интерес представляет реализация `buflen`. При генерации LLVM IR для структуры, если её поле помечено атрибутом `buflen`, создается функция `struct_name.buf_name()`, которая в качестве аргумента принимает указатель на структуру `struct_name`, а возвращает результат выражения, указанного в качестве аргумента атрибута `buflen`. Идентификатор этой функции добавляется в отладочную информацию для поля `buf_name`. После чтения отладочной информации анализатор сопоставляет указатель на

функцию `struct_name.buf_name()` полю структуры. В листинге 8 показан пример такой кодогенерации, соответствующий исходному коду из листинга 3.

```
1  typedef struct {
2      unsigned int header;
3      unsigned int height;
4      unsigned int width;
5      char* buf __attribute__((buflen(S.buf)));
6  } S;
7
8  unsigned int S.buf(S* s) { // Сгенерировано компилятором
9      return s->header + s->height * s->width;
10 }
11
12 void fill(S* s, unsigned int row, unsigned int col, char val) {
13     unsigned int index = s->header + row * s->width + col;
14     if (index <= s->header + s->height * s->width) {
15         s->buf[index] = val;
16     }
17 }
```

Листинг 8. Формула над полями с созданной функцией.
Listing 8. Formula over fields with created function.

Идея заключается в следующем. Изначально выполняется анализ всех служебных функций типа `S.buf`. Проведенный анализ позволяет установить, что возвращаемое значение является результатом арифметического выражения, зависящего от полей структуры входного параметра: `s->header + s->height * s->width`. Далее анализ выполняется на основе резюме стандартным образом, но при анализе функции `fill` в её начало добавляется вызов функции `S.buf`. Его обработка заключается в применении резюме этой функции и трансляции символьного выражения для возвращаемого значения в контекст `fill`. Результирующее символьное выражение сопоставляется с полем структуры, помеченным атрибутом `buflen`, то есть с `s->buf`.

При обращении к данному полю структуры по индексу (как на строке 15 в примере) или при обращении к полю с помощью функций `memcpy` и `memcmp` проверяется возможность выхода за указанную длину буфера в данной точке программы. Для этого составляется формула ошибки для SMT-решателя:

$$max(index) \geq len \wedge len \geq 0$$

где $\square\square\square$ – идентификатор значения из анализируемого свойства, а $\square\square\square(\square\square\square\square\square)$ – максимальное значение индекса из интервала допустимых значений, по которому обращаются к буферу (или число элементов для копирования или сравнения в случае с `memcpy` и `memcmp` соответственно). Если формула выполнима, то выдаётся соответствующее предупреждение о возможности выхода за границы буфера.

5. Результаты

Помимо аннотаций, добавленных специально как часть Svace, анализатор также учитывает все аннотации с названием `NotNull`, `NonNull` и `Nullable` из любой библиотеки как специальные. Мы проанализировали часть исходного кода операционной системы Android-14, написанной на Java, и исходный код компилятора Kotlinc версии 1.9.22, написанного на

Java и Kotlin, в которых повсеместно используются подобные аннотации (в частности, и неявно для типов из Kotlin, допускающих значение `null`). В таблице 3 приведены результаты тестирования этих аннотаций.

До изменений на Android-14 выдавалось суммарно 276822 предупреждения. После поддержки перечисленных аннотаций появилось 2904 новых срабатывания и пропало 21 срабатывание. Из пропавших срабатываний 8 было истинными и 13 было ложными. Из 100 случайных новых срабатываний 13 являются ложными, 54 – истинными и 33 – не попадают ни в одну категорию.

До изменений на Kotlinc-1.9.22 выдавалось суммарно 3682 предупреждения. После поддержки перечисленных аннотаций появилось 2879 новых срабатываний и пропало 94 срабатывания. Из пропавших срабатываний 12 было истинными и 82 ложными. Из 100 случайных новых срабатываний 15 являются ложными, 25 – истинными и 60 – не попадают ни в одну категорию.

Табл. 3. Результаты для аннотаций, связанных с null, на JVM-проектах.
Table 3. Results for null-related annotations in JVM projects.

Проект	Всего предупреждений		Появилось			Пропало		
	До	После	Всего	Истин. (из 100)	Ложн. (из 100)	Всего	Истин.	Ложн.
Android-14	276822	279705	2904	54	13	21	8	13
Kotlinc-1.9.22	3682	6467	2879	25	15	94	12	82

33 новых срабатывания на Android-14 и 60 новых срабатываний на Kotlinc-1.9.22, которые не попадают в категории истинных или ложных, являются истинными формально, но фактически разыменования нулевой ссылки в таком коде никогда не происходит. Причина этому в том, что не всегда аннотация `@Nullable` может являться источником ошибки. Например, в листинге 9 приведён пример, когда метод, помеченный данной аннотацией, возвращает нулевую ссылку только при определённом условии. Если это условие предварительно проверяется, дефект с разыменованием нулевой ссылки не проявляется. Использование аннотации `@Nullable` в подобных контекстах довольно распространено, что может приводить к большому количеству срабатываний, которые с точки зрения анализатора являются корректными, т.к. явная проверка на `null` отсутствует, но не релевантны для пользователя. Для избежание таких случаев полезным было бы введение аннотации, подобной `@Nullable`, но хранящей и условие, при котором возможно нулевое значение, как, например, в стандартной библиотеке языка C# [18].

6. Похожие работы

Среда разработки IntelliJ IDEA [19] предоставляет возможности для статического анализа кода с учётом наиболее распространённых аннотаций. Помимо этого, для IntelliJ IDEA предоставляются отдельные специальные аннотации [20]. Наиболее интересной из них является аннотация `@Contract`, позволяющая задавать контракт, которому следует и аннотированный метод, и все его вызовы: например, что функция является чистой; или же что функция бросает исключение при выполнении условия на входные параметры. Пример использования приведён в листинге 10.

```
1 interface List<T> {
2     boolean isEmpty();
3
4     @Nullable // Возвращает null только для пустого списка
```

```
    T getFirstOrNull();
}

class Example {
    public void test(List<User> users) {
        if (users.isEmpty()) return;
        User firstUser = users.getFirstOrNull();
        String name = firstUser.getName(); // Нет ошибки
        ...
    }
}
```

Листинг 9. Отсутствие разыменования нулевой ссылки при наличии `@Nullable`.
Listing 9. Absent null dereference issue in presence of `@Nullable`.

```
public class Example {
  // Чистая функция, результат которой обязан использоваться
  @Contract(pure = true)
  public static boolean not(boolean value) {
    return !value;
  }

  // Функция, которая завершается исключением,
  //   если значение входного параметра ложно
  @Contract("false -> fail")
  public static require(boolean condition) {
    if (!condition) throw new IllegalStateException();
  }
}
```

Листинг 10. Пример использования аннотации `Contract`.
Listing 10. Example of `Contract` annotation usage.

Язык Python начиная с версии 3.5 предоставляет возможность указания типовых аннотаций [21-22]. Данные аннотации не являются обязательными, но могут быть использованы средами разработки, линтерами и другими инструментами для статической проверки корректности типов. Они стали основой для статического анализатора Муру [23]. В листинге 11 на строке 1 `int` – это не тип переменной `b`, а аннотация, которая сообщает, что переменная `b` должна иметь тип `int`. На строке 5 переменная получает строковый тип. С точки зрения языка это корректный код, и он будет успешно исполнен. Но анализатор Муру выдаст ошибку на этой строке, так как переменная получает тип, который противоречит аннотации. Подобные аннотации позволяют добавить проверку типов для языка с динамической типизацией и обнаружить часть ошибок во время написания программы. Согласно исследованию [24], 7% проектов на Python используют аннотации типов. На наш взгляд это довольно высокий показатель, так как язык Python занимает нишу для быстрого создания прототипов, и часто его используют из-за отсутствия строгой типизации.

```
b: int
if f:
    b = int(input())
else:
```

```
5     b = input()
```

Листинг 11. Пример типовых аннотаций в Python.
Listing 11. Python type annotations example.

Clang Static Analyzer [25] позволяет аннотировать исходный код при помощи атрибутов GCC [3], таким образом помогая анализатору находить новые дефекты и уменьшать количество ложных срабатываний. Пример приведён в листинге 12. Атрибут `nonnull(1,3)` показывает, что первый и третий параметр никогда не должны принимать нулевое значение. Но на строке 4 в качестве последнего аргумента передаётся нулевой указатель, о чём сообщит анализатор.

```
1 int bar(int*p, int q, int *r) __attribute__((nonnull(1,3)));
2
3 int foo(int *p, int *q) {
4    return !p ? bar(q, 2, p)
5              : bar(p, 2, q);
6 }
```

Листинг 12. Пример аннотаций в CSA.
Listing 12. CSA source annotations example.

В инструменте Svace реализована реакция на комментарии в коде. Плюсы – универсальность реализации, подходит для множества языков. Минусы – реализация основана на синтаксическом разборе исходного кода, компилятор не используется, в сложных случаях возможны ошибки. Пример приведён в листинге 13.

```
1 typedef struct {
2   int* ptr; // svace: possible_null
3 } S;
4
5 int example(S* s) {
6   return *s->ptr; // Потенциальное разыменование нулевого
7 указателя
  }
```

Листинг 13. Пример комментария-аннотации.
Listing 13. Example of comment annotation.

Другой пример аннотаций в виде комментариев реализован в легковесном статическом анализаторе Splint [26], в котором используются аннотации для параметров функций, возвращаемых значений, глобальных переменных и полей структур в виде комментариев к исходному коду. Например, аннотация для параметра `/*@notnull@*/` означает, что он не может принимать нулевое значение. Инструмент выдаст предупреждения во всех случаях, когда параметр может получить нулевое значение. При анализе таких функций анализатор считает, что значение параметра ненулевое. При этом аннотировать можно не только определения функций, но и их объявления, что позволяет аннотировать библиотечные функции с отсутствующим кодом. В Svace мы не вводили такую возможность, т.к. в инструменте уже присутствует механизм, позволяющий моделировать поведение библиотечных функций.

Работа [27] посвящена поиску уязвимостей форматной строки в C с помощью вывода типов. Помеченные данные реализованы в виде расширения системы типов C дополнительными квалификаторами типов. Квалификатор tainted используется подобно уже существующему квалификатору const. Предупреждение выдаётся, если выражение с таким квалификатором

используется как форматная строка. Предложенная система похожа на атрибуты для C/C++ в Svace, отличия заключаются в том, что они используются разными видами анализа, а также тем, что дополнительные квалификаторы вызовут ошибки компиляции и поэтому требуют специальной подготовки кода.

7. Заключение

Статический анализатор, способный проводить проверки без специальной подготовки программы, обладает очевидными преимуществами. Тем не менее, возможность добавления пользователем дополнительной информации о программе также представляет ценность. Ключевым достоинством аннотаций является автоматическая проверка анализатором указанных свойств при их нарушении, что не только упрощает анализ программы, но и повышает читаемость кода.

В данной работе рассмотрены возможности аннотирования исходного кода для языков C, C++, Java и Kotlin, а также особенности их реализации в статическом анализаторе. Пользовательские аннотации позволяют существенно улучшить процесс анализа, однако их эффективность во многом зависит от корректности и полноты предоставляемых пользователем данных.

Список литературы / References

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

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

[3]. GNU Compiler Collection. Attribute Syntax. Английский. URL: https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html (дата обращения 16.04.2025).

[4]. __declspec | Microsoft Learn. Английский. URL: https://learn.microsoft.com/en-us/cpp/cpp/declspec?view=msvc-170 (дата обращения 16.04.2025).

[5]. LLVM Compiler Infrastructure. Английский. URL: https://llvm.org/docs/LangRef.html (дата обращения 16.04.2025).

[6]. Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D., Bierman G. The Java® Language Specification. Java SE 21 Edition. 2023 г.

[7]. Java SE 21. Interface Plugin. Английский. URL: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.compiler/com/sun/source/util/Plugin.html (дата обращения 15.04.2025).

[8]. Java SE 21. Interface Processor. Английский. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.compiler/javax/annotation/processing/Processor.html (дата обращения 15.04.2025).

[9]. The Java™ Tutorials. Trail: The Reflection API. Английский. URL: https://docs.oracle.com/javase/tutorial/reflect/ (дата обращения 15.04.2025).

[10]. Kotlin Symbol Processing API. Английский. URL: https://kotlinlang.org/docs/ksp-overview.html (дата обращения 11.04.2025).

[11]. kapt compiler plugin. Английский. URL: https://kotlinlang.org/docs/kapt.html (дата обращения 11.04.2025).

[12]. Kotlin. Null safety. Английский. URL: https://kotlinlang.org/docs/null-safety.html (дата обращения 24.04.2025).

[13]. Android API Reference. GuardedBy. Английский. URL: https://developer.android.com/reference/androidx/annotation/GuardedBy (дата обращения 22.04.2025).

[14]. International Standard ISO/IEC 14882:2024(E) – Programming Language C++. 2024 г.

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

[16]. Афанасьев В. О., Поляков С. А., Бородин А. Е., Белеванцев А. А. Kotlin с точки зрения разработчика статического анализатора. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр.67-82. DOI: 10.15514/ISPRAS-2021-33(6)-5.
[17]. Афанасьев В. О., Бородин А. Е., Белеванцев А. А. Статический анализ для языка Scala. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 9-20. DOI: 10.15514/ISPRAS-2024-36(3)-1.
[18]. Attributes for null-state static analysis interpreted by the C# compiler. Conditional post-conditions: NotNullWhen, MaybeNullWhen, and NotNullIfNotNull. Английский. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis#conditional-post-conditions-notnullwhen-maybenullwhen-and-notnullifnotnull (дата обращения 14.05.2025).
[19]. IntelliJ IDEA. Annotations. Английский. URL: https://www.jetbrains.com/help/idea/annotating-source-code.html (дата обращения 15.04.2025).
[20]. IntelliJ IDEA. Annotations. JetBrains annotations dependency. Английский. URL: https://www.jetbrains.com/help/idea/annotating-source-code.html#jetbrains-annotations (дата обращения 15.04.2025).
[21]. The Python Standard Library. typing – Support for type hints. Английский. URL: https://docs.python.org/3/library/typing.html (дата обращения 15.04.2025).
[22]. Van Rossum G., Lehtosalo J., Langa L. Python Enhancement Proposals. PEP 484 – Type Hints. Английский. URL: https://peps.python.org/pep-0484/ (дата обращения 15.04.2025).
[23]. Mypy: Optional Static Typing for Python. Английский. URL: https://mypy-lang.org/ (дата обращения 15.04.2025).
[24]. Di Grazia L., Pradel M. The Evolution of Type Annotations in Python: An Empirical Study. Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022 г. DOI: 10.1145/3540250.3549114.
[25]. Clang 21.0.0git documentation. Source Annotations. Английский. URL: https://clang.llvm.org/docs/analyzer/user-docs/Annotations.html (дата обращения 16.04.2025).
[26]. Evans D., Larochelle D. Improving security using extensible lightweight static analysis. IEEE software, том 19, вып. 1, 2002 г., стр. 42-51. DOI: 10.1109/52.976940.
[27]. Shankar U., Talwar K., Foster J. S., Wagner D. Detecting format string vulnerabilities with type qualifiers. 10th USENIX Security Symposium (USENIX Security 01), 2001 г.

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ – аспирант ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM языки.

Vitaly Olegovich AFANASYEV – postgraduate student at ISP RAS. Research interests: compiler technologies, static analysis, JVM languages.

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

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

Евгений Александрович ВЕЛЕСЕВИЧ – научный сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии.

Evgeny Alexandrovich VELESEVICH – researcher at ISP RAS. Research interests: compiler technologies.

Борис Викторович ОРЛОВ – студент ВМК МГУ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ.

Boris Viktorovich ORLOV – student at the Faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University, researcher at ISP RAS. Research interests: compiler technologies, static analysis.