DOI: 10.15514/ISPRAS-2022-34(6)-3

Поиск использований освобожденного ресурса в исходном коде на языке C# методами статического анализа

[1,2] *У.В. Тяжкороб, ORCID: 0000-0002-2375-6842 <tsiazhkorob@ispras.ru>*
[1,3] *В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*
[1,3] *А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9*
[3] *Московский государственный университет им. М.В. Ломоносова, Россия, г. Москва, Ленинские горы д.1*

Аннотация. В данной работе описывается масштабируемый детектор для поиска использований освобожденного ресурса в исходном коде на основе статического символьного выполнения. Данный детектор выполняет межпроцедурный анализ, чувствительный к потоку управления и контексту вызовов. Детектор реализован в рамках промышленного инструмента SharpChecker, его точность (около 70% истинных срабатываний) позволяет включить его в число основных детекторов и предоставить функционал конечным пользователям. В работе рассматривается алгоритм детектора, адаптированный для SharpChecker. Также представлены результаты тестирования детектора на наборе ПО с открытым исходным кодом и примеры срабатываний на реальных проектах.

Ключевые слова: статический анализ; символьное выполнение; поиск ошибок; освобожденный ресурс

Для цитирования: Тяжкороб У.В., Игнатьев В.Н., Белеванцев А.А. Поиск использований освобожденного ресурса в исходном коде на языке C# методами статического анализа. Труды ИСП РАН, том 34, вып. 6, 2022 г., стр. 41-50. DOI: 10.15514/ISPRAS-2022-34(6)-3

Благодарности. Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект №20-01-00581 А.

Detection of uses of disposed resources in C\# source code using static analysis

[1,2] *U.V. Tsiazhkorob, ORCID: 0000-0002-2375-6842* <tsiazhkorob@ispras.ru>
[1,3] *V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*
[1,3] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the RAS, 25 Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. The paper is devoted to the scalable approach for the detection of uses of disposed resources in C# source code, that is based on static symbolic execution. The resulting detector is implemented as a part of an industrial SharpChecker, that performs a scalable inter-procedural path-, and context-sensitive analysis. The evaluation of the developed detector shows 70% true positive ratio allowing it to include to the standard set of detectors and provide functionality to users. The paper describes a detection algorithm that takes into account the limitations imposed by the existing infrastructure of SharpChecker, its evaluation on the set of open-source programs containing 6 mln LOC and some examples of found errors in real projects.

Keywords: static analysis; symbolic execution; use-after-free; bug detection; disposed resource

For citation: Tsiazhkorob U.V., Ignatiev V.N., Belevantsev A.A. Detection of uses of disposed resources in C# source code using static analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 41-50 (in Russian). DOI: 10.15514/ISPRAS-2022-34(6)-3

Acknowledgments. This work was supported by the Russian Foundation for Basic Research, project №20-01-00581 A

1. Введение

1.1 Ресурсы в языке C#

В языке программирования C# существует два типа ресурсов: управляемые и неуправляемые. Память, выделяемая под управляемые ресурсы, контролируется сборщиком мусора.

Когда количество ссылок на такой ресурс становится нулевым, сборщик мусора автоматически освобождает занимаемую этим ресурсом память. Неуправляемые ресурсы – все, за чем не следит сборщик мусора. Такие ресурсы необходимо явно освобождать после использования. Наиболее распространенные из них: файлы, сетевые подключения, подключения к базам данных и прочее.

`IDisposable` – встроенный .NET интерфейс – механизм высвобождения неуправляемых ресурсов, предлагаемый Microsoft. Он предназначен для детерминированного освобождения неуправляемых ресурсов любых типов, владеющих как управляемыми, так и неуправляемыми ресурсами, а также типов, унаследованных от класса, реализующего `IDisposable`.

Данный интерфейс предоставляет метод `Dispose()`, с помощью которого производится освобождение ресурса. Также в языке C# существует конструкция `using`, которая сама будет обрабатывать вызов `Dispose()`.

Если не освободить неуправляемые ресурсы, ошибка может не проявиться при тестировании и долгое время может не давать о себе знать во время работы программы, но в некоторый момент она проявится, например, при попытке записать данные в освобожденный ресурс, и тогда информация может быть безвозвратно утеряна.

1.2 Актуальность

В наше время проекты становятся больше, поэтому в них сложнее заметить ошибки, некоторые из которых могут привести к проблемам во время исполнения программы. Кроме того, чем раньше на этапе разработки будет обнаружена ошибка, тем дешевле обойдется ее исправление. Чтобы систематически находить ошибки до запуска программы, используются статические анализаторы. Учитывая то, что ряд ошибок сложно обнаружить тестированием, преимущество статического анализа [2, 3] в том, что ошибки находятся сразу, еще до запуска программы, не требуя подбора входных данных для воспроизведения. Это достигается за счет того, что методы статического анализа позволяют обрабатывать сразу все пути выполнения программы.

Ошибка использования освобожденного ресурса, как и многие другие, сложно обнаруживается на тестах, может проявляться по-разному в разных окружениях, зачастую приводя к аварийному завершению программы или потере данных. Для поиска таких ошибок в исходном коде может использоваться статический анализ. В данной работе предлагается детектор использования неуправляемых ресурсов после их освобождения на основе статического символьного выполнения [1] для исходного кода на языке C#. Метод символьного выполнения обеспечивает чувствительность к путям и контексту вызовов, что важно для снижения доли ложных срабатываний. Необходимость реализации данного детектора объясняется высокой популярностью и востребованностью языка C#. Согласно рейтингу TIOBE [4], этот язык является пятым по популярности в мире.

1.3 Мотивационный пример

Рассмотрим упрощенный пример, основанный на реальной проблеме в коде на языке C# (листинг 1).

```
1  class Test
2  {
3      public abstract class ImgClassAbs : IDisposable
4      {
5          Bitmap Img = new Bitmap(100, 100);
6
7          public abstract void Abort();
8
9          public void Dispose()
10         {
11             Img.Dispose();
12         }
13         public void Use()
14         {
15             Img.Clone(); // использование ресурса
16         }
17     }
18     public class ImgClassDisp : ImgClassAbs
19     {
20         public override void Abort()
21         {
22             Dispose(); // освобождение ресурса
23         }
24     }
25     public class ImgClassNotDisp : ImgClassAbs
26     {
```



```
27         public override void Abort() { }
28     }
29     public abstract class TestClass
30     {
31         public void Func(ImgClassAbs elem)
32         {
33             if (elem != null)
34             {
35                 elem.Abort(); // освобождение ресурса
36                 elem.Use(); // USE_AFTER_DISPOSE:
       Resource <elem.Img> was already disposed
37             }
38         }
39     }
40 }
```

Листинг 1. Мотивационный пример
Listing 1. Motivational example

Проблема проявляется в методе `Func()`. Метод `Abort()`, вызванный для ресурса `elem` является абстрактным методом, соответственно он имеет несколько реализаций. Можно не заметить, что в одной из этих реализаций вызывается метод `Dispose()`, как показано в примере. Таким образом, ресурс `elem.Img` может быть освобожден в методе `Abort()`. Далее следует вызов метода, использующей `elem.Img`. В таком случае в процессе исполнения возникает исключение. Именно такие ситуации и находит реализованный детектор.

1.4 Существующие решения

Для языков программирования с ручной работой с памятью существуют детекторы для поиска уязвимостей, связанных с некорректным использованием динамической памяти в процессе работы программы [5-8]. Более того, существуют и статические `USE-AFTER-FREE` детекторы [9-11], выполняющие ту же задачу, что и описываемый, однако для других языков программирования.

Существуют разные подходы к поиску использований освобожденных ресурсов, имеющие различные преимущества и недостатки. Описываемый в статье `USE-AFTER-DISPOSE` детектор реализует межпроцедурный, чувствительный к контексту, путям и потокам анализ на основе символьного выполнения. Близким аналогом к нему является реализация `USE-AFTER-FREE` детектора в SVACE [5]. Также существует подход на основе анализа помеченных данных, например, IRBIS [12], преимуществом которого является возможность обнаруживать больше ошибок, ценой значительной доли ложных предупреждений. Оба инструмента поддерживают языки программирования C и C++. В свою очередь, у языка C#, например, нет указателей, и, как следствие, появляются проблемы в анализе псевдонимов.

В C-подобных языках освобождение памяти происходит в основном при помощи функции стандартной библиотеки `free()`, применимой только для указателей и имеющей поведение, строго определенное реализацией аллокатора. В случае языка C# рассматривается метод `Dispose()`, предоставляемый интерфейсом `IDisposable` и имеющий различные реализации для различных классов.

Для языка C# существуют детекторы только для обнаружения возможных утечек ресурсов [13]. В связи с этим был создан детектор, исследующий исходный код на наличие использований освобожденных ресурсов.

2. Описание схемы работы детектора

2.1 Общая схема работы SharpChecker

SharpChecker, в рамках которого реализован рассматриваемый детектор, использует часть инфраструктуры Roslyn [14], отвечающую за разбор файлов проектов, решений, для работы с файлами системы сборки, а также компиляции исходного кода программы.

В начале анализа происходит построение АСД (абстрактного синтаксического дерева) с помощью Roslyn. При этом также строится статический граф вызовов и производится поиск синтаксических ошибок.

Далее, при обходе графа вызовов от листьев к корню, часть собираемой информации сохраняется в резюме анализируемого метода для дальнейшего ее использования при анализе методов, вызывающих данный.

Затем происходит построение графа потока управления для каждого метода с помощью графа вызовов, АСД и таблицы символов. Вершинами графа потока управления являются базовые блоки (ББ) – последовательности следующих одна за другой инструкций промежуточного представления кода. После, используя собранную информацию, обработчик завершения анализа выдает предупреждения.

И, наконец, работает чувствительный к путям и контексту анализ. При обработке вызовов методов он использует информацию, полученную из их резюме, при этом объединяет состояния в точке слияния. Состояние в точке входа в анализируемый метод параметризуется с помощью символьных переменных. То есть для каждого параметра метода вводится уникальный идентификатор `ValueId`. Таким образом, есть возможность представить результаты выполнения метода в виде выражений над символьными переменными.

В случае, когда использование какого-либо объекта достигается несколькими его определениями, вводится функция объединения значений или ϕ-функция. В этом случае его `ValueId` также будет представлен функцией объединения `ValueIds` всех достигающих этого использования значений объекта.

Для хранения информации в рамках анализа базового блока используется контекст. При переходе от анализа одного ББ к анализу следующего ББ, содержимое контекста первого вливается в контекст второго. Правила, по которым происходит перенос информации, задаются атрибутом `Join`.

2.2 Внутрипроцедурный анализ

Существует множество ситуаций, воспроизводящих описываемую проблему. Рассмотрим сначала самую простую из них. Пусть в коде не предполагалось дальнейшего использования определенного ресурса и его освободили в целях сохранения памяти, но потом все же он где-то понадобился, а про его освобождение забыли.

Сохраняя `ValueId` освобожденных ресурсов, при последующих использованиях можно будет сделать вывод о наличии ошибки. Если мы рассматриваем только внутрипроцедурный анализ, то хранить `ValueId` ресурсов нужно только до конца метода. Для этого будем использовать контекст базового блока ГПУ.

Далее рассмотрим ситуацию, когда объект удаляется и позднее используется при определенных условиях, например, в ветвлении `if-else`. В таком случае ошибка произойдет тогда и только тогда, когда условия освобождения и использования пересекаются, то есть после освобождения непременно произойдет использование ресурса. Соответственно, в контексте, помимо `ValueId`, будем сохранять условия освобождения ресурса.

Если после освобождения в переменную присваивается ссылка на новый или существующий объект, то при поиске в списке освобожденных ресурсов `ValueId` рассматриваемого ресурса, проверяем, является ли он освобожденным. Если объект «обновляется» при всех условиях, при которых он удаляется, то его необходимо убрать из списка освобожденных ресурсов.

2.3 Межпроцедурный анализ

Если освобождение некоторого ресурса произошло в вызванном методе, то его нельзя обнаружить последовательным внутрипроцедурным анализом. Тогда, будем получать информацию об освобожденных ресурсах из результатов анализа вызываемого метода. Однако контекст удаляется в конце метода, значит в этот момент нужно переносить информацию в резюме. А при вызове метода переносим содержимое его резюме в контекст вызывающего метода.

Возможны ситуации, в которых использование объекта так же происходит в вызове какого-либо метода. Тогда для анализа требуется иметь возможность получить информацию об использованиях в вызываемом методе. Таким образом, помимо удалений, в контекст будем сохранять и использования, чтобы потом, перенеся их в резюме, иметь к ним доступ из вызывающего метода.

Аналогично будем поступать в случае «обновления» ресурса.

Более того, при переносе из резюме вызываемого метода в контекст вызывающего необходимо учитывать условие, при котором происходит вызов рассматриваемого метода.

2.4 Представление ресурсов в детекторе

Стоит упомянуть, что при анализе кода имеет смысл сохранять в контекст и резюме только объекты классов, унаследованных от интерфейса `IDisposable`, так как только они являются интересующими нас неуправляемыми ресурсами.

Рассмотрим освобождение ресурса. Метод `Dispose()` может быть как библиотечным, так и заданным пользователем. В случае библиотечного метода, когда реализация недоступна при проведении анализа, или метода, в резюме которого нет сохраненных освобожденных ресурсов, запоминаем как освобожденные `IDisposable` поля объекта класса, к которому применен метод `Dispose()`, так как применяя метод `Dispose()` к какому-либо объекту, программист предполагает удаление этого объекта, то есть освобождение его полей. В случае заданного пользователем метода с непустым резюме освобожденных ресурсов – в контекст вызывающего метода переносится резюме заданного пользователем метода.

В случае вызова любого метода, отличного от `Dispose()`, используя его резюме, получаем информацию об освобожденных в нем ресурсах, и записываем ее в контекст вызывающего метода.

Также следует учесть, что значение объекта может быть ϕ-функцией, следовательно, его `ValueId` также будет ϕ-функцией. В этом случае проделываем все то же для всех достигающих это использование значений объекта, учитывая условия достижения использования.

2.5 Использование ресурсов и вызов диагностики

Следует рассмотреть различные сценарии использования объектов. В данной работе рассматривается два из них: вызов метода, и разыменование, так как только они могут породить неопределенное поведение. Встречая любой из них при последовательном внутрипроцедурном анализе или при переносе резюме какого-либо вызванного метода, из контекста мы извлекаем список освобожденных ресурсов.

Если ресурс был освобожден, и условия его использования и освобождения совместны, то есть могут выполняться одновременно, тогда создается диагностика. Если диагностика добавлена, выходим из обработки использования. Если же ресурс не был освобожден, или диагностика не добавилась, сохраняем это использование в контекст анализируемого метода вместе с его условием (а также достигающие значения в случае ϕ-функции) для дальнейшего анализа.

Чтобы исправить найденную детектором ошибку, следует понимать, в чем конкретно она заключается, где находятся освобождение и использование ресурса. Таким образом, для упрощения анализа истинности предупреждения используются возможности SharpChecker, позволяющие, зная расположение удаления и использования в коде, отобразить их. Так, в контексте и резюме, помимо условий освобождения и использования ресурса, будем хранить еще и локации.

3. Результаты

Тестирование производилось на наборе из 21 проекта с открытым исходным кодом, имеющих размер 6 млн. строк кода. Анализ выполнялся на машине с 8-ядерным процессором Intel(R) Core(TM) i7-6700, имеющей 32 Gb RAM.

Было выдано 119 предупреждений из которых все были проанализированы вручную. 83 из них были классифицированы как истинные, 36 – как ложные. В результате доля истинных срабатываний составляет 70%.

Рассмотрим некоторые из этих случаев в качестве иллюстрации.

3.1 Пример 1

Ошибка (листинг 2) найдена на проекте Roslyn версии 2.8.2, JsonWriterTests.cs: 92–97.

```
1 var stringWriter = new StringWriter();
2 using (var jsonWriter = new JsonWriter(stringWriter))
3 {
4       ...
5 }
6 // ресурс stringWriter освобожден после блока using
7 return stringWriter.ToString();// USE_AFTER_DISPOSE
    Resource <stringWriter> was already disposed
```

Листинг 2. Место срабатывания
Listing 2. Location of the error

```
1  internal sealed class JsonWriter : IDisposable
2  {
3      private readonly TextWriter _output;
4      ...
5      public JsonWriter(TextWriter output)
6      {
7          _output = output;
8      ...
9      }
10      ...
11     public void Dispose()
12     {
13          _output.Dispose();
14     }
15     ...
16 }
```

Листинг 3. Класс JsonWriter
Listing 3. JsonWriter class

В данном случае `stringWriter` и `jsonWriter` являются `IDisposable` объектами, более того, в конструкторе класса `JsonWriter` (3) не создаются новые объекты, а присваиваются переданные. Таким образом, одно из полей объекта `jsonWriter` (конкретно `_output`) является ссылкой на объект `stringWriter`, и когда в конце конструкции `using` освобождается ресурс `jsonWriter`, освобождается и `stringWriter`. Далее вызывается метод `ToString()` для освобожденного `stringWriter`. Эту проблему достаточно легко не заметить в процессе написания кода, и на этапе выполнения могло бы возникнуть неопределенное поведение.

3.2 Пример 2

Ошибка (листинг 4) найдена на проекте netmq версии 3.3.0.12, XSub.cs: 196–227.

```
1  while (true)
2  {
3      bool isMessageAvailable = m_fairQueueing.Recv(ref msg);
         // USE_AFTER_DISPOSE Resource
       <m_fairQueueing.m_atomicCounter> was already disposed
4      ...
5      while (msg.HasMore)
6      {
7          isMessageAvailable = m_fairQueueing.Recv(ref msg);
            // освобождение ресурса
8          ...
9      }
10 }
```

Листинг 4. Место срабатывания
Listing 4. Location of the error

```
1  public bool Recv(ref Msg msg)
2  {
3      return RecvPipe(null, ref msg);
        // использование ресурса, освобождение ресурса
4  }
5  public bool RecvPipe(Pipe[] pipe, ref Msg msg)
6  {
7       msg.Close();
8       ...
9  }
10
11 public void Close()
12 {
13     if (... || m_atomicCounter.Decrement() == 0)
14     {
15      ...
16     }
17     m_atomicCounter.Dispose();
18     ...
19 }
```

Листинг 5. Реализация методов
Listing 5. Method Implementation

В этом примере использование стоит раньше удаления в коде, однако это все происходит в цикле. Таким образом, когда в конце одной итерации объект удаляется, на следующей итерации он используется и возникает неопределенное поведение. Видно, что за использование и освобождение отвечает один и тот же метод `Recv(ref Msg msg)`, однако если посмотреть на его реализацию и реализации последовательно вызываемых методов

(листинг 5), видно, что сначала идет использование объекта, а потом удаление, и если два раза подряд вызвать `Recv`, то получим использование освобожденного ресурса.

При включении детектора использований освобожденных ресурсов время работы SharpChecker возрастает с 1 часа и 19 минут до 1 часа и 38 минут. Увеличение времени на 23% связано с количеством информации, которую необходимо хранить в резюме методов, копировать при анализе вызовов и обрабатывать для создания диагностик. В связи с замедлением работы планируется провести анализ накопленных данных с целью минимизации их объема.

4. Заключение

В коде больших проектов встречаются ошибки, из-за которых может произойти аварийное завершение работы программы или потеря данных из-за использования освобожденного ресурса. Некоторые из этих ошибок могут привести к непоправимым последствиям. Рассмотрев большинство случаев, где возникает использование освобожденного ресурса, получилось создать алгоритм обнаружения такого рода ошибок, являющийся основой описанного детектора с точностью около 70% истинных срабатываний.

Анализ двадцати одного проекта с открытым исходным кодом показал, что ошибки, связанные с использованием освобожденного ресурса, нередко допускаются, что свидетельствует о прикладной ценности реализованного детектора.

Список литературы / References

[1] Baldoni R., Coppa E. et al. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), vol. 51, issue 3, 2018, pp. 1-39.

[2] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2 / Koshelev V.K., Ignatiev V.N. et al. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, issue 4, 2017, pp. 268−276.

[3] Аветисян А., Белеванцев А. и др. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, 2011 г., стр. 23-38 / Avetisyan A., Belevantsev A. et al. Using static analysis for finding security vulnerabilities and critical errors in source code. Trudy ISP RAN/Proc. ISP RAS, vol. 21, 2011, pp. 23-38 (in Russian).

[4] Tiobe index for ranking the popularity of programming languages, 2022. URL: https://www.tiobe.com/tiobe-index.

[5] Иванников В.П., Белеванцев А.А. и др. 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 / Ivannikov V.P., Belevantsev A.A. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, issue 5, 2014, pp. 265-275.

[6] Аветисян А., Бородин А. Механизмы расширения системы статического анализа svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011 г., стр. 39-54 / Avetisyan A., Borodin A. Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors. Trudy ISP RAN/Proc. ISP RAS, vol. 21, 2011, pp. 39-54 (in Russian).

[7] Henry J., Monniaux D., Moy M. Pagai: a path sensitive static analyser. Electronic Notes in Theoretical Computer Science, vol. 289, 2012, pp. 15-25.

[8] Несов В. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции РусКрипто, 2009.

[9] Bai J.-J., Lawall J. et al. Effective static analysis of concurrency {use-after-free} bugs in linux device drivers. In Proc. of the USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 255-268.

[10] van der Kouwe E., Nigade V., Giuffrida C. DangSan: scalable use-after-free detection. In Proc. of the Twelfth European Conference on Computer Systems, 2017, pp. 405-419.

[11] Ye J., Zhang C., Han X. UAFChecker: scalable static detection of use-after-free vulnerabilities. In Proc. of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 1529-1531.

[12] Shimchik N., Ignatyev V., Belevantsev A. Improving accuracy and completeness of source code static taint analysis. In Proc. of the 2021 Ivannikov ISPRAS Open Conference (ISPRAS), 2021, pp. 61-68.

[13] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2 / Koshelev V.K., Ignatiev V.N. et al. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, issue 4, 2017, pp. 268−276.

[14] dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. Available at: https://github.com/dotnet/roslyn, accessed 23.10.2021.

Информация об авторах / Information about authors

Ульяна Владимировна ТЯЖКОРОБ – студентка бакалавриата Физтех-школы Радиотехники и Компьютерных Технологий Московского физико-технического института, сотрудник ИСП РАН. Научные интересы: статический анализ программ, статическое символьное выполнение, алгоритмы поиска дефектов в исходном коде.

Uljana Vladimirovna TSIAZHKOROB is a bachelor's student at the Department of Radio Engineering and Computer Technologies of the Moscow Institute of Physics and Technology, an employee of the ISP RAS. Research interests: static program analysis, static symbolic execution, algorithms for finding defects in source.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV, PhD in computer sciences, senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.

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

Andrey Andreevich BELEVANTSEV – Dr.Sc., Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.