DOI: 10.15514/ISPRAS-2022-34(6)-2

Повышение точности статического анализа за счет учета значений полей класса, имеющих единственное константное значение

[1,2] *В.С. Карцев, ORCID: 0000-0001-7482-0835 <karcev.vs@ispras.ru>*
[1,3] *В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@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 статье описывается подход, позволяющий с помощью предварительного анализа полей с единственным возможным значением повысить точность работы детекторов статического анализатора уровня символьного выполнения. Помимо этого, детектор предупреждает программиста о забытых модификаторах readonly и о неиспользуемых полях. Детектор был реализован в рамках промышленного статического анализатора SharpChecker для языка C#. Анализ проводится на уровне абстрактного синтаксического дерева, что позволяет снизить затраты времени и ресурсов. Сохраненные значения для ряда полей позволяют использовать конкретные константы вместо символьных значений на этапе символьного выполнения, тем самым повышая точность. В результате удалось заметно улучшить качество некоторых анализаторов, таких как UNREACHABLE_CODE (улучшение на 7.57%) или DEREF_OF_NULL (улучшение на 1.33%), и получить новые срабатывания в случаях с забытым readonly или неиспользуемыми полями. Хорошие результаты позволили включить детектор в основную ветку SharpChecker и предоставить его пользователям. В работе подробно рассмотрен алгоритм работы детектора и приведены примеры срабатываний на наборе ПО с открытым исходным кодом.

Ключевые слова: статический анализ; C#; неиспользуемое поле; забытый readonly; повышение точности; анализ использования полей; единожды инициализированное поле

Для цитирования: Карцев В.С., Игнатьев В.Н. Повышение точности статического анализа за счет учета значений полей класса, имеющих единственное константное значение. Труды ИСП РАН, том 34, вып. 6, 2022 г., стр. 29-40. DOI: 10.15514/ISPRAS-2022-34(6)-2

Improving the accuracy of static analysis by accounting for the values of class fields that can have only one constant value

[1,2] *V.S. Karcev, ORCID: 0000-0001-7482-0835 <karcev.vs@ispras.ru>*
[1,3] *V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@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 describes the approach for the improvement of the accuracy of general purpose static symbolic execution analysis of C# sources based on the accounting for the values of class fields that can have only one possible value. In addition, we propose the detector of forgotten readonly modifiers and unused fields, that use data collected by the main analysis. The approach and detectors were implemented as part of the industrial static analyzer SharpChecker. The main analysis is performed at the AST level to reduce time and resource costs. Collected values of the fields are used during symbolic execution phase allowing it to use concrete value instead of symbolic for the subset of class fields. As a result, we managed to noticeably improve the accuracy of some analyzers, such as UNREACHABLE_CODE (improved by 7.57%) or DEREF_OF_NULL (improved by 1.33%) and get new results in cases with forgotten **readonly** or unused fields. Achieved results allow to use analysis and detectors in the main branch of the SharpChecker and make it available to users. The paper considers in detail the algorithm of the detector and provides examples of results on the set of open source software.

Keywords: static analysis; C#; unused field; forgotten readonly; improving accuracy; field usage analysis; once-initialized field

For citation: Karcev V. S., Ignatiev V.N. Improving the accuracy of static analysis by accounting for the values of class fields that can have only one constant value. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 29-40 (in Russian). DOI: 10.15514/ISPRAS-2022-34(6)-2

1. Введение

В языке C#, помимо привычного модификатора `const`, используется также модификатор `readonly`. В отличие от константных полей, поля с таким модификатором можно инициализировать любым, даже неконстантным значением. Однако инициализация возможна только в конструкторе, что гарантируется компилятором, а далее такое поле будет доступно только для чтения. Так, если такое поле будет инициализироваться конкретным константным значением при любых сценариях использования, то на протяжении всего выполнения это значение останется неизменным. Информация о единственном константном значении такого поля может пригодиться в процессе статического анализа.

Часто встречаются ситуации, когда какое-либо поле, не помеченное `const` или `readonly`, единожды инициализируется каким-либо константным значением, или же все конструкторы присваивают ему единственное значение, которое больше нигде не меняется. Такие поля могут быть причиной некоторых ошибок, так как можно легко забыть, что поле может иметь лишь одно возможное значение. Помимо этого, наличие единственно возможного значения может сигнализировать об ошибках при инициализации такого поля.

При анализе значения таких полей не учитываются, так как анализатор не проверяет, могут ли их значения быть перезаписаны. Это снижает точность результатов. Отсюда вытекает возможность выявлять такие поля и использовать полученные данные при дальнейшем анализе исходного кода.

Кроме того, поиск полей с единственно возможными значениями включает в себя поиск полей, у которых, скорее всего, был забыт модификатор `readonly`. О таких полях

необходимо предупреждать программиста, чтобы он мог исправить возможную ошибку и не допустить некорректного использования поля в дальнейшем.

1.1 Актуальность

В настоящее время язык программирования C# является пятым по популярности согласно рейтингу TIOBE [1]. Он используется для широкого спектра задач, начиная с прикладных программ и заканчивая веб-приложениями. Так, объем проектов на данном языке неизменно растет, что влечет повышение сложности анализа. Таким образом, возникает необходимость производить максимально качественный анализ за минимальное время.

Предлагаемый в статье подход позволяет повысить точность анализа, практически не увеличивая его ресурсоемкость, а также является абсолютно консервативным. В процессе анализа исследуются только те поля, которые никак не могут изменяться во время работы программы. Авторам неизвестно о других инструментах, реализующих указанный алгоритм повышения точности за счет анализа полей, имеющих лишь одно константное значение.

1.2 Мотивационный пример

В качестве ошибочной ситуации, обнаруживаемой предлагаемым подходом, можно привести следующий упрощенный пример.

Допустим, в некоторой библиотеке создан класс логгера, приведенный на листинге 1. Разумно было бы объявить поле `Log.LogFileName` с модификатором `readonly`, так как изменение данного поля во время работы программы не подразумевается в логике библиотеки.

```
1 public class Log
2 {
3     private string LogFileName = "log.txt";
4
5     public string GetName()
6     {
7         return LogFileName;
8     }
9 }
```

Листинг 1. Пример класса с забытым readonly
Listing 1. A class example with forgotten readonly

На листинге 2 приведен пример использования данного класса сторонним разработчиком в своем проекте. Так как пользователь не осведомлен о том, что поле не может меняться, он добавляет бесполезную проверку, что приводит к проблеме недостижимого кода.

```
1 public class User
2 {
3     var Logger = new Log();
4
5     public Register()
6     {
7         if (Logger.GetName() != "log.txt")
8         {
9           ...
10        }
11
12        else
13        {
14          ...
```



```
15        }
16    }
17 }
```

Листинг 2. Ошибочное использование класса
Listing 2. Wrong class usage

В приведенном примере опущенный модификатор `readonly` привел к тому, что сторонний программист написал код, который никогда не будет исполнен. Нахождение таких полей, и выяснение их значений могут повысить точность поиска ошибок различных типов, обнаруживаемых методом символьного исполнения, например, `UNREACHABLE_CODE`, `DEREF_OF_NULL`.

1.3 Цели и задачи

Цели данной работы перечислены ниже:

1) повышение точности анализа за счет использования неизменных значений полей классов;
2) определение значений полей, которые инициализируются единожды и не меняются;
3) поиск полей с забытым `readonly`.

Результатом работы должен стать анализатор, который будет

- искать поля классов, в которых, вероятно, был забыт модификатор доступа `readonly` и предупреждать о них программиста;
- выяснять, существует ли единственно возможное значение этого поля при любых вариантах использования класса и запоминать это единственное значение;
- предоставлять другим анализаторам информацию о полях, имеющих единственное возможное значение.

Помимо прочего, так как анализатор будет строить результат на основе статистики использования этого поля, можно будет дополнительно генерировать предупреждения о забытых `readonly` и неиспользуемых полях.

1.4 Существующие решения

В большинстве промышленных инструментов, таких как Svace [2-4] или SonarQube [5], значения полей, инициализированных единожды, не используются в процессе анализа. Срабатывания, найденные с помощью информации, собранной новым анализатором, не удалось воспроизвести в других статических анализаторах. Таким образом, можно предположить, что в других продуктах значения единожды инициализированных полей не учитываются в анализе.

В основном статические анализаторы предполагают, что поля классов, не объявленные как `const`, не могут иметь константное значение или же анализируют поведение лишь локальных переменных, не исследуя точное поведение объектов. Так, статический анализатор Pagai [6] анализирует только промежуточное представление кода LLVM, в связи с чем не может обнаружить поле с единственным константным значением.

В свою очередь, поиск забытых `readonly` и неиспользуемых полей, являющийся побочным результатом проекта, реализован во многих решениях. Детекторы неиспользуемых полей и забытых модификаторов `readonly` включены во многие IDE, включая, например, Visual Studio [7, 8]. Однако существующие решения в основном являются составляющими частями небольших продуктов, предназначенных в основном для рефакторинга кода [9] и выдачи предупреждений «на лету». Так, существующие решения могут обнаруживать лишь простейшие ошибки из-за специфики реализации.

2. Реализация

Анализатор был разработан в рамках промышленного инструмента SharpChecker [10], предназначенного для поиска ошибок в объемных проектах методами статического анализа.

2.1 Схема работы SharpChecker

В первую очередь SharpChecker, используя инфраструктуру Roslyn [11], перехватывает компиляцию и собирает решение. Далее запускается этап анализа, состоящий, по большей части, из анализа абстрактного синтаксического дерева и символьного исполнения.

На этапе анализа синтаксического дерева можно сделать выводы о корректности исходного кода, найти синтаксические ошибки.

На этапе символьного исполнения производится масштабируемый межпроцедурный анализ, чувствительный к потоку управления [12]. Символьное исполнение в SharpChecker допускает ложные срабатывания и пропущенные ошибки. Его цель – найти максимальное количество ошибок за минимальное время при заданном проценте истинных срабатываний. На данном этапе анализируется граф потока управления, возможные значения переменных и выполнимость условий. С помощью этого метода можно найти ошибки, связанные, например, с недостижимым кодом или использованием удаленного ресурса.

Так как результаты работы предлагаемого анализатора должны использоваться на уровне символьного исполнения, то они вычисляются раньше - на уровне анализа синтаксического дерева. В SharpChecker при анализе синтаксического дерева используются механизмы Roslyn. Конкретно, для поиска информации о полях необходимо иметь «идентификатор» каждого поля. В Roslyn для этого используется `IFieldSymbol`, являющийся представлением поля в таблице символов. Однако в различных проектах одно и то же поле будет иметь различные `IFieldSymbol`. Для этих целей можно использовать реализованный в SharpChecker `IFieldSymbolId`, представляющий собой «полное имя» поля, включающее в себя пространство имен и класс, которому принадлежит поле. В отличие от встроенного в Roslyn `IFieldSymbol`, он остается одним и тем же в различных проектах, что позволяет производить анализ решений, состоящих из множества проектов и учитывать использования поля в каждом из них.

Используя эти возможности, можно реализовать анализатор использования полей классов, запоминающий, при наличии, единственное константное значение, возможное для этого поля.

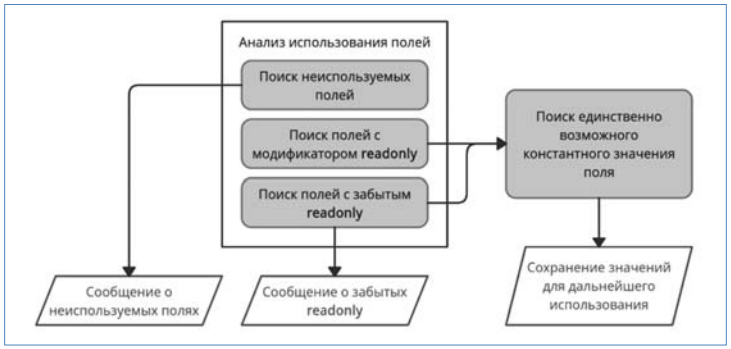

Рис. 1: Схема работы анализа
Fig.. 1: Scheme of the analysis

2.2 Схема работы алгоритма

Несмотря на то, что все этапы анализа полей происходят параллельно, формально их можно разбить на последовательные шаги. Тогда принцип работы анализатора можно представить схемой на рис. 1.

Фактически механизм реализован с помощью механизмов Roslyn, позволяющих создать обработчики для каждого типа узлов синтаксического дерева. Схематически алгоритм изображен на рис. 2.

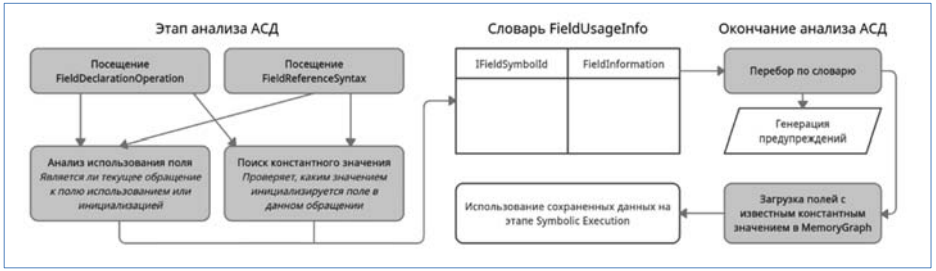

Рис. 2: Схема работы алгоритма
Fig. 2: Scheme of the algorithm

Как видно из схемы, каждый из обработчиков вызывает две функции-анализатора, первая из которых определяет, является обращение к полю использованием или инициализацией. Этот анализатор собирает информацию, необходимую для вывода о необходимости добавить к объявлению поля модификатор `readonly` и о том, что поле не используется. Второй анализатор занимается сбором данных о том, какое значение присваивается полю в каждой из инициализаций, переданных на обработку.

Оба анализатора хранят информацию в словаре, ключом в котором является `IFieldSymbolId` анализируемого поля, а все информация о нем содержится в объекте класса `FieldInformation`. Данный класс предназначен для удобного хранения информации об использовании поля, местах инициализаций и возможных значениях.

В конце этапа анализа абстрактного синтаксического дерева вызывается метод, создающий диагностики для всех полей, которые должны быть `readonly` и для полей, которые не используются. Параллельно вся информация о полях, имеющих лишь одно возможное значение, переносится в `MemoryGraph`, из которого на этапе символьного исполнения можно будет извлечь их значение.

2.2.1 Выбор кандидатов

Для начала необходимо отсеять поля, которые гарантированно не могут иметь единственного константного значения. В первую очередь это поля, доступные для изменения извне. Такое поле, не имея ни одного присваивания в видимой области кода, может быть изменено сторонним разработчиком в его коде. Помимо этого, не могут иметь единственное значение те поля, которые получают различные значения при различных условиях.

Подытожим: наличие единственно возможного константного значения возможно для полей, удовлетворяющих одному из следующих условий:

- поле имеет модификатор доступа readonly, а значит, не может иметь инициализаций вне конструкторов, что гарантируется компилятором;
- поле недоступно для изменения вне анализируемого кода, а все инициализации в области видимости располагаются в конструкторах.

Отдельно необходимо рассмотреть второй вид кандидатов. Недоступными для внешнего редактирования будут поля, имеющие модификатор доступа `private` или `internal`.

Помимо этого, недоступны извне будут и поля классов, имеющих соответствующий уровень доступа или же содержащиеся внутри приватных классов. Можно заметить, что такие поля, недоступные для внешнего редактирования и имеющие инициализации только внутри конструкторов, полностью удовлетворяют определению `readonly`-полей.

Так, о схожести такого поля с `readonly` необходимо сообщить программисту. Для этого был создан новый тип ошибки – `FORGOTTEN_READONLY`.

2.2.2 Определение полей с забытым readonly

Для поиска полей с забытым `readonly` необходимо проанализировать каждое обращение к исследуемому полю и подсчитать количество инициализаций внутри и вне конструкторов. Так, если вне конструкторов не найдено ни одной инициализации и выполняется одно из условий:

- поле имеет модификатор доступа private или internal;
- класс, к которому принадлежит поле, является приватным;
- класс, к которому принадлежит поле, вложен в приватный класс,

то данное поле не может изменяться извне и у него мог быть забыт модификатор `readonly`.

Вывод о необходимости объявить поле как `readonly` возможно сделать только после окончания анализа всего исходного кода. Таким образом, необходимо хранить информацию по всем полям, в том числе и по тем, для которых достоверно известно, что они меняются вне конструкторов. Таким образом, после завершения анализа всего исходного кода для каждого поля можно будет сделать вывод о необходимости добавления модификатора `readonly`.

2.2.3 Поиск неиспользуемых полей

Поиск неиспользуемых полей аналогичен поиску полей с забытым `readonly`, однако необходимо подсчитать количество использований этого поля, результат которых используется в дальнейшем. Так, например, использование поля для изменения собственного значения (пример приведен на листинге 3) можно не считать использованием.

```
1 class Counter
2 {
3     private int CValue = 0;
4
5     private void Increase(int value)
6     {
7         CValue = CValue + value;
8     }
9 }
```

Листинг 3. Использование поля для изменения собственного значения
Listing 3. Using a field to change its own value

Этим примером можно проиллюстрировать использование поля для изменения его же значения.

2.2.4 Поиск единственного константного значения

Одновременно с анализом на забытое readonly для каждого поля необходимо проводить анализ на возможное константное значение. Так, каждый раз, встречая инициализацию поля, необходимо сверять новое значение с сохраненным для данного поля. Возможны следующие случаи.

- **Для поля не сохранено никакого значения.** В таком случае необходимо запомнить найденное значение в качестве известного для данного поля.
- **Для поля уже сохранено отличное от текущего значение.** В данном случае можно понять, что поле принимает различные значения в разных сценариях использования. Для такого поля невозможно судить о его значении.
- **Для поля уже сохранено такое же значение.** В данном случае не нужно делать каких-либо дополнительных действий.

Помимо этого, если встречена инициализация неконстантным значением, можно сделать вывод, что поле не может иметь единственно возможного константного значения.

В случае, если поле не инициализируется при объявлении, в качестве известного константного значения будет запомнено значение по умолчанию для данного типа данных. Таким образом можно будет обработать ситуации, когда в некоторых конструкторах отсутствует инициализация поля, а в других оно инициализируется значением по умолчанию для типа.

3. Результаты

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

После интеграции анализатора время анализа уменьшилось на 0.2% и составило 1 час, 16 минут и 51 секунду. Незначительное уменьшение времени анализа можно объяснить случайной погрешностью. Таким образом, созданный анализатор не влияет на итоговую производительность SharpChecker.

3.1 Краткий обзор

После интеграции решения в SharpChecker было получено улучшение в работе других анализаторов, в частности:

- заметное улучшение (+7.57%) работы `UNREACHABLE_CODE`: из 68 новых срабатываний 66 оказались истинными, 1 ложным; при этом ушло 10 ложных срабатываний.
- некоторое улучшение (+1.33%) в работе `DEREF_OF_NULL`: были получены 3 новых истинных срабатывания.
- незначительное улучшение в работе анализатора `DEREF_AFTER_AS`.

Помимо улучшения работы прочих анализаторов, было найдено множество ошибок `FORGOTTEN_READONLY` и `UNUSED_FIELD`.

Улучшение работы связано с тем, что значения полей, имеющих единственное константное значение, начали учитываться, даже если неизвестно, каким образом создавался объект.

3.2 Демонстрация улучшения работы сторонних анализаторов. Примеры новых срабатываний

3.2.1 UNREACHABLE_CODE

Рассмотрим срабатывание, обнаруженное в проекте OpenSim версии 0.9.0.0 в файле AnimationSet.cs в строке 135 (листинг 4).

```
1 public class AnimationSet
2 {
3     private bool m_parseError = false;
4     ...
5
6     public Byte[] ToBytes()
7     {
8         if (m_parseError) // UNREACHABLE_CODE
9         {
```

```
10              ...
11          }
12          ...
13      }
14  }
```

Листинг 4. Недостижимый код в проекте OpenSim
Listing 4. Unreachable code in the OpenSim project

Здесь приватное поле `AnimationSet.m_parseError` при объявлении инициализируется значением `false`, однако больше нигде не меняется внутри класса. Кроме того, из-за того, что поле объявлено как приватное, его нельзя изменить вне класса `AnimationSet`. Таким образом, данное поле может иметь лишь одно значение – `false`.

Далее данное поле используется в методе `ToBytes()`, который меняет свое поведение в зависимости от значения этого поля. Однако имея информацию о том, какое значение единственно возможно для данного поля, можно сделать вывод, что блок кода, обрамленный условием, будет недостижим.

3.2.2 DEREF_OF_NULL

Приведенное на листинге 5 срабатывание было найдено в проекте Lucene.Net [13] версии 4.8.0 в строке 535 файла TestRuleSetupAndRestoreClassEnv.cs.

```
1  internal sealed class TestRuleSetupAndRestoreClassEnv
2  {
3      ...
4
5      internal HashSet<string> AvoidCodecs;
6
7      ...
8
9      private bool ShouldAvoidCodec(string codec)
10     {
11         return AvoidCodecs.Count > 0 && // DEREF_OF_NULL
12             AvoidCodecs.Contains(codec);
13     }
14 }
```

Листинг 5. Разыменование null в проекте Lucene.Net
Listing 5. Null dereferencing in the Lucene.Net project

В данном примере приватное поле `AvoidCodecs` ни разу не инициализировано внутри области видимости, а значит, имеет значение по умолчанию для ссылочного типа – `null`.

Таким образом, при обращении к данному полю возникает проблема разыменования `null`.

3.3 Примеры срабатываний

3.3.1 FORGOTTEN_READONLY

В качестве примера можно рассмотреть фрагмент кода 6 из проекта BobBuilder версии 1.0.0.42 (листинг 6). Ошибка была обнаружена в файле Actions.cs в строке 15.

```
1  namespace BobBuilder.Actions
2  {
3      class CloseTag : AbstractEditAction
4      {
5          IEditAction _OldAction = null; // FORGOTTEN_READONLY
6
```



```
7          public CloseTag( IEditAction oldAction )
8          {
9              _OldAction = oldAction;
10         }
11         ...
12         }
13     ...
14 }
```

Листинг 6. Забытое readonly в проекте BobBuilder
Listing 6. Forgotten readonly in the BobBuilder project

В данном примере анализатор определил поле `_OldAction` как поле с забытым модификатором `readonly`. Действительно, данное поле присваивается лишь дважды – при объявлении и внутри конструктора. Так как данное поле имеет уровень доступа по умолчанию (`private`), то оно недоступно для редактирования вне анализируемого кода.

Поэтому данное поле, вероятно, должно быть помечено как `readonly`.

3.3.2 UNUSED_FIELD

Срабатывание в проекте Spartacus версии 0.45.1, обнаруженное в файле ChainSubRule.cs в строке 7 (листинг 7).

```
1  using System;
2
3  namespace PDFjet.NET {
4      class ChainSubRule {
5       ...
6
7       int inputGlyphCount; // UNUSED_FIELD
8       ...
9      }
10 }
```

Листинг 7. Неиспользуемое поле в проекте Spartacus
Listing 7. Unused field in the Spartacus project

В результате работы поле `inputGlyphCount` было помечено как неиспользуемое, так как класс, которому оно принадлежит, имеет уровень доступа `internal`, а внутри сборки использований поля не было найдено. Соответственно, данное поле не используется в анализируемом коде и не может быть использовано в коде, использующем данную библиотеку.

4. Заключение

Изначально значения полей с единственно возможным константным значением не учитывались в анализе. С помощью реализованного анализатора удалось добиться улучшения работы SharpChecker, были найдены новые ошибки. Таким образом, результаты анализа единожды инициализированных полей могут использоваться для повышения точности других анализаторов. В результате, реализацией было доказано, что учет таких полей возможен без увеличения затрат и полезен при анализе больших проектов.

Помимо прочего, в процессе удалось найти поля с, вероятно, забытым модификатором доступа readonly и неиспользуемые поля.

В связи с улучшением результатов работы SharpChecker патч был включен в основную ветку инструмента, и его результаты теперь могут использоваться у конечных пользователей.

Список литературы / References

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

[2] 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.

[3] 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).

[4] 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).

[5] Lenarduzzi V., Lomio F. et al. Are SonarQube rules inducing bugs? In Proc. of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 501−511.

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

[7] Detecting forgotten readonly in visual studio. Available at: https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/style-rules/ide0044, accessed 20.08.2022.

[8] Detecting unused class members in visual studio. Available at: https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/style-rules/ide0051, accessed 20.08.2022.

[9] Atkinson D.C., King T. Lightweight detection of program refactorings. In Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC′05), 2005, 8 p.

[10] 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.

[11] 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.

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

[13] Lucene.NET 4.8.0. Available at: https://lucenenet.apache.org, accessed 23.10.2021.

Информация об авторах / Information about authors

Вадим Сергеевич КАРЦЕВ – студент бакалавриата Физтех-школы Радиотехники и Компьютерных Технологий МФТИ, сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, статическое символьное выполнение, поиск дефектов в исходном коде.

Vadim Sergeevitch KARCEV is a bachelor’s student at the Department of Radio Engineering and Computer Technologies of MIPT, an employee of the ISP RAS. Research interests: compiler technologies, static program analysis, static symbolic execution, defect search 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.