

DOI: 10.15514/ISPRAS-2026-38(6)-1



Поиск множественного перечисления коллекций в исходном коде на языке C# методами статического анализа

^{1,2} К.А. Ханевская, ORCID: 0009-0000-8162-8243 <k.hanevskaja@ispras.ru>^{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 Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.² Московский физико-технический институт,
Россия, 141700, Московская область, г. Долгопрудный, Институтский пер., 9.³ Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация: Статья посвящена проблеме множественного перечисления *отложено инициализированных коллекций* в языке C#. Множественные перечисления таких коллекций могут приводить к существенному снижению производительности и непредсказуемому поведению программы. В статье предлагается метод обнаружения ошибок множественного перечисления, реализованный в рамках промышленного статического анализатора SharpChecker. Алгоритм основан на символьном выполнении, чувствительном к потоку управления и путям выполнения, и поддерживает межпроцедурный анализ. Детектор выявляет как повторные перечисления в пределах одного метода, так и возникающие через цепочки вызовов методов. Экспериментальная оценка на 20 проектах с открытым исходным кодом показала высокую долю истинных предупреждений (около 95%). Результаты демонстрируют эффективность и практическую ценность предложенного подхода. Это позволяет включить его в число основных детекторов и предоставить функциональность пользователям.

Ключевые слова: статический анализ; символьное выполнение; поиск ошибок; множественное перечисление; LINQ-методы; отложенная инициализация; работа с коллекциями.

Для цитирования: Ханевская К.А., Тяжкороб У.В., Игнатьев В.Н. Поиск множественного перечисления коллекций в исходном коде на языке C# методами статического анализа. Труды ИСП РАН, том 38, вып. 6, часть 1, 2026 г., стр. 9–22. DOI: 10.15514/ISPRAS-2026-38(6)-1.

Благодарности: Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П.Иванникова РАН – ЦКП ИСП РАН

Detection of Multiple Enumeration in the C# Source Code Using Static Analysis

^{1,2} K.A. Khanevskaya, ORCID: 0009-0000-8162-8243 <k.hanevskaja@ispras.ru>^{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 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: This article is dedicated to the problem of multiple enumeration of lazy collections in C#, an antipattern that can lead to significant performance degradation and unpredictable program behavior. A static detector is proposed, implemented within the industrial static analyzer SharpChecker. The algorithm is based on symbolic execution sensitive to control flow and execution paths and supports interprocedural analysis. The detector identifies both repeated enumerations within a single method and those occurring through chains of called methods. Experimental evaluation on 20 open-source projects demonstrated a high true positive rate (approximately 95%). The results demonstrate the effectiveness and practical value of the proposed approach, allowing its integration as a core detector and providing the functionality to users.

Keywords: static analysis; symbolic execution; bug detection; multiple enumeration; LINQ methods; deferred execution; working with collections.

For citation: Khanevskaya K.A., Tsiazhkorob U.V., Ignatiev V.N. Detection of Multiple Enumeration in the C# Source Code Using Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 6, part 1, 2026, pp. 9-22 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-1.

Acknowledgements: The results were obtained using the equipment of the Shared Research Facility «Shared Research Center of the Ivannikov Institute for System Programming of the Russian Academy of Sciences (SRC ISP RAS)»

1. Введение

Производительность и надёжность кода – одни из ключевых аспектов качества программного обеспечения. Множественное перечисление *отложено инициализированных коллекций* может оказывать существенное негативное влияние на оба этих аспекта. *Отложено инициализированными коллекциями* в языке C# называются коллекции, элементы которых вычисляются непосредственно при обращении к ним и не сохраняются в памяти. Хотя само по себе множественное перечисление не является синтаксической или логической ошибкой, повторный обход одной и той же последовательности способен многократно замедлять выполнение программы, а в ряде случаев – приводить к непредсказуемым результатам. При этом, при множественном перечислении генерация исключений не происходит, что делает его сложным для обнаружения при ручном тестировании. В то же время, существующие инструменты статического анализа способны выявлять лишь самые простые случаи такой проблемы, оставляя не покрытыми сложные межпроцедурные сценарии.

Согласно рейтингу популярности языков программирования TIOBE [1], C# стабильно входит в пятёрку лидеров, что обуславливает высокую востребованность средств анализа для программ, написанных на языке C#. Коллекции в языке C# являются широко используемыми благодаря существованию множества продвинутых механизмов взаимодействия с ними. Это подтверждает актуальность разработки детектора множественного перечисления для языка C#.

Целью данной работы является разработать метод поиска множественного перечисления коллекций и реализовать в виде детектора для языка C# в рамках промышленного статического анализатора SharpChecker. Обнаружение таких ошибок осложняется необходимостью межпроцедурного анализа для обнаружения дефектов в цепочках вызовов методов и чувствительности к путям выполнения программы для учета условий перечислений. Для достижения поставленной цели создан набор синтетических тестов, покрывающих различные случаи возникновения данной проблемы, разработан и реализован алгоритм работы детектора, а также проведено тестирование на реальных проектах с открытым исходным кодом. В работе приводятся результаты тестирования, сравнение с существующими решениями, оценка эффективности предложенного подхода и производительности детектора.

1.1 Методы LINQ для работы с коллекциями (LINQ to objects)

Коллекция в языке C# – абстракция для хранения и управления группами объектов. Интерфейс `IEnumerable` предоставляет возможность последовательного перебора элементов коллекции, определяя метод `GetEnumerator()`, и реализуется большинством стандартных типов коллекций.

LINQ (Language Integrated Query) – это технология, предоставляемая платформой .NET, позволяющая выдавать запросы к различным структурам данных, в частности, к коллекциям [2]. Использование LINQ методов при работе с коллекциями всегда приводит к их перечислению при вычислении результата, так как операция запроса включает в себя инструкцию `foreach` [3].

Существуют два вида методов LINQ: немедленного и отложенного выполнения [4]. Методы немедленного выполнения производят фактическое перечисление коллекции в момент вызова. Результат вычисляется сразу и может быть как атомарным значением (число, элемент, булево значение), так и коллекцией типов `Array`, `List` или `Dictionary`. Примерами таких методов являются `Contains`, `Count` или `Sum`, `ToList`. Методы отложенного выполнения возвращают объект типа `IEnumerable<T>`, и вычисление результата производится только при перечислении коллекции. К таким методам относятся, например, методы `Where`, `Select` или `Reverse`.

Таким образом, коллекция в языке C# может быть полностью вычисленной и храниться в памяти или представлять собой отложенный запрос, результат которого не хранится в памяти, а вычисляется заново при каждом обращении. Так, например, в Листинге 1 коллекция `sortNumbers`, которая является результатом цепочки отложенных запросов, будет вычислена только в момент вызова метода `ToList()`. В случае последующих использований полученной коллекции `sortNumbers`, все отложенные запросы будут выполняться повторно.

```
1 var sortNumbers = numbers.Where(...).Select(...).GroupBy(...);
2 var sortNumbersList = sortNumbers.ToList();
```

Листинг 1. Использование отложенного метода.
Listing 1. Using lazy method.

Помимо LINQ, ещё одним способом реализовать отложенное вычисление результата является использование ключевого слова `yield return` в теле метода, возвращающего `IEnumerable<T>`. Такой метод не вычисляет результирующую последовательность сразу, а возвращает перечислитель. Фактический перебор элементов произойдёт только тогда, когда возвращённый объект `IEnumerable<T>` будет перечислен (например, в цикле `foreach` или при вызове немедленного LINQ-метода). Таким образом, метод `yield return` также создаёт *отложенно инициализированную коллекцию*, многократное перечисление которой может привести к повторным вычислениям.

1.2 Проблема множественного перечисления

Каждый раз, когда *отложенно инициализированная коллекция* перечисляется, вся цепочка LINQ-методов, создавшая эту коллекцию, выполняется заново. Это означает, что фильтрация, сортировка, преобразование элементов и другие дорогие операции будут произведены повторно. Более того, если коллекция является базой данных, будет выполнено повторное обращение к серверу, что является лишней тратой ресурсов. Соответственно, множественное перечисление такой коллекции может привести к значительным потерям производительности, особенно при больших объёмах данных.

Это не единственная проблема, к которой может привести множественное перечисление. В случае, если операция перечисления имеет побочные эффекты (чтение и модификация глобальных переменных, выполнение операций ввода-вывода и другие операции), повторное её перечисление может привести к сложно обнаруживаемым ошибкам в логике работы программы. Так, например, в Листинге 2, дважды перечисляя *отложенно инициализированную коллекцию* `seq`, являющуюся результатом вызова метода `GetWithCounter()`, мы неявно меняем поле `counter`, так как её инкрементирование – побочный эффект перечисляющего метода. Таким образом, если разработчик рассчитывает, что после вызова `seq.Count()` и `seq.First()` поле `counter` сохранит своё исходное значение, равное нулю, он столкнётся с ошибкой в логике работы программы.

```
01 public class ClassA
02 {
03     public static int counter = 0;
04
05     public static IEnumerable<int> GetWithCounter()
06     {
07         for (int i = 0; i < 3; i++)
08         {
09             counter++; // изменение поля
10             yield return i;
11         }
12     }
13 }
14
15 public class Program
16 {
17     public static void Main()
18     {
19         var seq = ClassA.GetWithCounter();
20         seq.Count(); // counter == 3
21         seq.First(); // counter == 4
22     }
23 }
```

Листинг 2. Операция перечисления с побочными эффектами.
Listing 2. Enumeration operation with side effects.

Кроме того, при работе с *отложенно инициализированными коллекциями* существует ещё одна проблема: между перечислениями источник данных может быть изменен, что приведёт к неожиданным результатам. Пример в Листинге 3 демонстрирует такую ситуацию. Содержание *отложенно инициализированной коллекции* `query` меняется без непосредственного обращения к ней, так как элементы её источника `numbers` были изменены, а каждое перечисление коллекции `query` заново выполняет фильтрацию `numbers`.

```
1 List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
2 var query = numbers.Where(x => x > 2); // отложенный запрос
3 int firstCount = query.Count(); // firstCount == 3
4 numbers.Add(10);
5 numbers.Remove(1);
6 int secondCount = query.Count(); // secondCount == 4
```

Листинг 3. Изменение источника отложено инициализированной коллекции.
Listing 3. Changing the source of a lazy collection.

1.3 Мотивационный пример

Рассмотрим пример ошибки, основанной на реальной проблеме в коде (Листинг 4).

```
01 void Foo(List<int> initValues)
02 {
03     // создание отложено инициализированной коллекции values
04     IEnumerable<int> values = initValues.Where(x => x % 2 == 0);
05     foreach (var nv in values) // первое перечисление values
06         Console.WriteLine(nv);
07     int counter = CountCollection(values, false);
08 }
09
10 int CountCollection(IEnumerable<int> values, bool flag)
11 {
12     int counter = 0;
13     if (flag) // проверка значения переменной flag
14         foreach (var nv in values) // перечисление не выполняется
15             counter++;
16     else
17         counter = values.Count(); // второе перечисление values
18     return counter;
19 }
```

Листинг 4. Мотивационный пример, демонстрирующий множественное перечисление.
Listing 4. Motivational example demonstrating multiple enumeration.

В данном примере коллекция `values` является результатом отложенного метода `Where`, поэтому каждое обращение к ней заново выполняет фильтрацию коллекции-источника `initValues`. Первое перечисление происходит в цикле `foreach` в методе `Foo`. Затем коллекция `values` передаётся в метод `CountCollection` со значением `flag == false`. В этом случае условие `if (flag)` ложно, соответственно перечисление не выполняется. Однако в ветке `else` вызывается `Count()`, который перечисляет `values` повторно (поскольку `Count()` – метод немедленного выполнения). Таким образом, происходит два перечисления одной и той же *отложено инициализированной коллекции*, что приводит к потере производительности из-за повторной фильтрации источника `initValues`.

Этот пример построен путем упрощения реальной ошибки в коде для демонстрации последствий множественного перечисления и необходимости поддержки детектором межпроцедурного анализа и чувствительности к путям выполнения.

1.4 Существующие решения

Проблема множественного перечисления может встретиться не только в коде на языке C#, однако, для него она является особенно актуальной. В большинстве других языков повторное перечисление одной и той же последовательности либо невозможно, либо приводит к очевидной ошибке. Так, в Java Stream API существуют отложенные и терминальные

операции: отложенные задают правила преобразования данных, а терминальные иницируют их выполнение и закрывают поток – то есть всю цепочку отложенных преобразований – после чего повторное использование потока невозможно [5]. В Python генераторы – итераторы, реализующие стратегию ленивых вычислений – являются одноразовыми: после исчерпания повторная итерация возвращает пустую последовательность [6]. В языке Rust итераторы обладают такими же свойствами. В языке C# интерфейс `IEnumerable` для работы с коллекциями допускает многократное перечисление одной и той же *отложено инициализированной коллекции*, причём каждый раз последовательность породивших её LINQ-методов выполняется заново. Это создаёт условия для возникновения трудноуловимых ошибок, которые могут критически сказаться на производительности и результатах выполнения программы.

Существующие статические анализаторы для языка C# предлагают средства для обнаружения множественного перечисления, однако их возможности ограничены.

ReSharper (JetBrains) [7-8] может генерировать предупреждения типа «Possible multiple enumeration of `IEnumerable`» [9], однако алгоритм работы данного детектора не поддерживает межпроцедурный анализ. В связи с этим, ReSharper генерирует предупреждения при любой передаче коллекции в метод в качестве аргумента, вне зависимости от того, происходит ли в нём фактическое перечисление. Это повышает долю ложных предупреждений.

Реализация встроенного в .NET SDK [10] детектора множественного перечисления коллекции `CA1851` [11] также ограничена внутрипроцедурным анализом и не учитывает цепочки вызовов.

Существует также `NetFabric.Hyperlinq.Analyzer` [12] – специализированный Roslyn-анализатор, содержащий ряд правил для оптимизации работы с перечислимыми коллекциями в C#. Однако, как показывает анализ его правил, ни одно из них не направлено на выявление множественного перечисления одной и той же *отложено инициализированной коллекции*.

Поскольку в реальных проектах с открытым исходным кодом множественные перечисления встречаются в цепочках вызовов методов, наличие межпроцедурного анализа является критическим аспектом у детекторов данной проблемы. Существующие решения поиска множественного перечисления в языке C# не поддерживают межпроцедурный анализ и, соответственно, не обеспечивают качественного полноценного анализа. Предлагаемый в данной работе детектор призван восполнить этот пробел и обеспечить высокую точность поиска множественных перечислений *отложено инициализированных коллекций* даже в сложных межпроцедурных сценариях.

2. Описание схемы работы детектора

2.1 Общая схема работы SharpChecker

Детектор, описанный в данной работе, реализован в рамках промышленного статического анализатора SharpChecker [13]. Анализатор работает с исходным кодом на языке C# и Visual Basic .NET [14], используя часть инфраструктуры Roslyn [15]. Roslyn – это платформа с открытым исходным кодом, которая обеспечивает доступ к компиляторам и программному интерфейсу, позволяющим анализировать код на языке C#.

Процесс анализа SharpChecker состоит из нескольких стадий. На первой стадии происходит построение абстрактного синтаксического дерева (АСД) кода. Анализ АСД может выявить синтаксические ошибки. Затем строится статический граф вызовов, обход которого выполняется в обратном топологическом порядке – от листьев к корню. В ходе этого обхода для каждого метода формируется резюме – структура данных, накапливающая необходимую для анализа информацию о методе [13]. Поскольку обход графа вызовов выполняется от листьев к корню, резюме каждого метода содержит информацию не только о нём самом, но

и о вызываемых им методах. Это и обеспечивает возможность межпроцедурного анализа [16].

Следующим этапом для каждого метода отдельно строится граф потока управления (ГПУ). Узлами этого графа являются базовые блоки (ББ), каждый из которых содержит в себе последовательный набор инструкций. Рёбра этого графа представляют собой логические переходы от одного блока инструкций к другому. Таким образом структура графа представляет собой все возможные пути выполнения программы. Состояние программы на пути выполнения описывается контекстом – структурой данных, которая передаётся по этому пути от блока к блоку. В точках слияния контексты, пришедшие по разным путям, объединяются в один.

Для параметризации состояния используются символьные значения: каждому параметру присваивается уникальный идентификатор значения, что позволяет представлять переменные с помощью символьных выражений. Этот способ анализа называется символьным выполнением [17]. Чувствительность к путям выполнения обеспечивается тем, что контекст, описывающий состояние программы, передаётся вдоль каждого пути независимо, а в точках слияния контексты объединяются по заранее заданным правилам – это позволяет учитывать условия ветвлений при анализе.

2.2 Общая схема работы детектора

Детектор множественного перечисления основан на поиске вызовов методов, которые приводят к перечислению *отложено инициализированных коллекций*. При первом обнаружении такого вызова для конкретной коллекции в контексте анализируемого пути выполнения сохраняется информация о факте её перечисления: уникальный идентификатор коллекции, вызванный перечисляющий метод, место вызова и условие, при котором он выполняется. При каждом последующем вызове перечисляющего метода анализатор проверяет, не была ли коллекция с таким идентификатором уже перечислена ранее. Если была, проверяется совместность условий обоих перечислений. При совместных условиях формируется диагностическое сообщение, которое сохраняется во временном резюме метода; в противном случае информация о перечислении сохраняется для сравнения с последующими перечислениями той же коллекции. По завершении анализа метода все накопленные диагностики выдаются в виде предупреждений.

2.3 Сценарии возникновения множественного перечисления

Как уже отмечалось, множественное перечисление представляет проблему только для *отложено инициализированных коллекций*. Детектор отслеживает перечисления коллекции в трёх случаях:

- Коллекция является результатом выполнения отложенного LINQ-метода, как показано в Листинге 1.
- Коллекция является результатом метода, использующего `yield return` для возвращаемых значений.
- Коллекция передаётся в метод в качестве аргумента с типом, унаследованным от `IEnumerable<T>`.

Нельзя точно определить, какой именно объект будет передан в метод – конкретный список или результат отложенного запроса. Поэтому любое перечисление аргументов типа `IEnumerable<T>`, переданных в метод, считается потенциально опасным.

Если говорить непосредственно о самом перечислении, то помимо очевидных случаев нескольких последовательных вызовов перечисляющих методов (немедленные LINQ-методы или обход в цикле `foreach`, как на Листинге 5 в табл. 1), детектор также

отслеживает вызовы перечисляющих методов внутри циклов, где одно и то же перечисление будет повторяться многократно при каждой итерации (Листинг 6 в табл. 1).

Табл. 1. Сценарии множественного перечисления.

Table 1. Multiple enumeration scenarios.

1	<code>void Foo(IEnumerable<int> obj)</code>	1	<code>void Foo(IEnumerable<int> obj)</code>
2	<code>{</code>	2	<code>{</code>
3	<code>obj.Count();</code>	3	<code>while (...)</code>
4	<code>obj.First();</code>	4	<code>{</code>
5	<code>foreach (var o in obj)</code>	5	<code>obj.Count();</code>
6	<code>{ ... }</code>	6	<code>...</code>
7	<code>}</code>	7	<code>}</code>
		8	<code>}</code>

Листинг 5. Вызов нескольких LINQ-методов.

Listing 5. Calling multiple LINQ methods.

Листинг 6. Вызов LINQ-метода в теле цикла.

Listing 6. Calling a LINQ method in a loop body.

2.4 Внутрипроцедурный анализ

Анализ кода внутри одного метода выполняется с помощью обработчиков событий (таких как вызов метода, обращение к полю класса и т. д.), вызываемых из ядра механизма символьного анализа при обходе ГПУ [18].

Для обнаружения множественных перечислений ключевыми являются следующие типы событий:

- Передача *отложено инициализированной коллекции* в немедленный LINQ-метод. Такое событие означает, что коллекция будет перечислена для получения результата.
- Вызов метода `GetEnumerator()` для *отложено инициализированной коллекции*. Этот вызов означает получение перечислителя, необходимого для последующего обхода коллекции. В большинстве случаев он происходит неявно, в цикле `foreach`, тогда как инструкция `foreach` при компиляции преобразуется в последовательность вызовов `GetEnumerator()` и `MoveNext()` [19]. Однако `GetEnumerator()` может быть вызван и явно. Поскольку фактическое перечисление начинается в момент первого вызова `MoveNext()`, именно этот вызов фиксируется как перечисление коллекции. Вызов `GetEnumerator()` при этом сохраняется во временном резюме метода как вспомогательная информация – для того, чтобы в дальнейшем связать перечислитель с исходной коллекцией.
- Передача коллекции в отложенный LINQ-метод. В отличие от первых двух случаев, здесь перечисления не происходит, но результатом вызова становится новая *отложено инициализированная коллекция*. Детектор должен запомнить эту связь, чтобы в дальнейшем отслеживать перечисления полученной коллекции.

Информация о всех трёх типах событий сохраняется в контексте базового блока.

При анализе графа потока управления в SharpChecker циклы разворачиваются на три итерации. Благодаря этому вызовы LINQ-методов внутри циклов автоматически распознаются как многократные, поэтому отдельно обрабатывать этот случай не требуется.

2.5 Межпроцедурный анализ

Проблема множественного перечисления коллекции не ограничивается случаями, когда LINQ-методы вызываются непосредственно в анализируемом методе. Перечисление может происходить внутри других методов, вызываемых из текущего. Для обнаружения таких ситуаций необходим межпроцедурный анализ.

Для реализации межпроцедурного анализа информация об обнаруженных перечислениях, сохранённая в контексте, по завершении анализа каждого метода переносится в его резюме. Таким образом, в резюме метода накапливается информация обо всех перечислениях коллекций, которые он совершает.

При обработке события передачи *отложено инициализированной коллекции* в какой-либо метод в качестве параметра, анализатор обращается к резюме вызываемого метода. Если резюме указывает, что коллекция будет перечислена внутри него, то этот факт фиксируется в контексте.

По завершении анализа базового блока, содержащего вызов, зафиксированный факт перечисления переносится из контекста в резюме вызывающего метода. Благодаря такому механизму информация о перечислениях распространяется вверх по цепочке вызовов – от вызываемых методов к вызывающим. Это позволяет выявлять неявные перечисления, скрытые за несколькими уровнями вызовов, и корректно диагностировать множественное перечисление даже в сложных сценариях взаимодействия методов.

2.6 Формальное описание алгоритма

Для систематизации предложенного алгоритма все отслеживаемые события и соответствующие им модификации структур данных обобщены в табл. 2.

3. Тестирование

Тестирование проводилось на наборе из 20 проектов с открытым исходным кодом, общий объём которого составил 7 млн. строк кода. Анализ выполнялся на машине с 20-ядерным процессором Intel® Core™ i7-14700, имеющей 64 GB оперативной памяти.

Всего детектор сгенерировал 213 предупреждений. После ручной проверки было выяснено, что 203 из них являются истинными, а 10 – ложными. Таким образом, доля истинных предупреждений составила около 95%. При этом включение детектора множественного перечисления незначительно повлияло на общую производительность анализатора SharpChecker (время работы возросло приблизительно на 1%).

Почти 38% истинных предупреждений указывают на неявные множественные перечисления – на те, которые возникают в цепочках вызовов методов. Эти случаи были выявлены исключительно благодаря поддержке межпроцедурного анализа. Существующие детекторы, не поддерживающие его, либо полностью пропустили бы подобные ошибки, либо сгенерировали бы большое количество ложных предупреждений, так как передача коллекций типа IEnumerable<T> в качестве аргументов методов очень часто встречается в коде реальных проектов. Избыток ложных предупреждений потребовал бы значительных временных затрат на их верификацию.

Результаты анализа проекта Roslyn статическим анализатором ReSharper продемонстрировали описанную проблему. В то время, как SharpChecker указал на 113 случаев множественного перечисления коллекции, ReSharper сгенерировал 368 предупреждений. Основной причиной большого количества предупреждений является рассмотрение как перечисления каждого случая передачи коллекции типа IEnumerable<T> в качестве аргумента или же присваивания такой коллекции полю класса. Пример такого предупреждения продемонстрирован на Листингах 7 и 8. Проект Roslyn, CSharpSmartTokenFormatter.cs: 29-42.

В примере предупреждения видно, что в методе ThrowIfNull не происходит перечисления его аргументов, таким образом, передача в него коллекции formattingRules не должна рассматриваться, как перечисление этой коллекции.

Однако, благодаря такому подходу, ReSharper обнаружил случай множественного перечисления коллекции, который описываемый в работе детектор пропустил (Листинг 9).

Табл. 2. Правила обработки событий множественного перечисления.
Table 2. Rules for processing multiple enumeration events.

Анализируемое событие	Значимость для анализа			Сохраняемая информация	Структура данных, куда сохраняется информация
Вызов метода GetEnumerator() для <i>отложено инициализированной коллекции</i>	Получение перечислителя коллекции			Связь уникальных символьных значений коллекции и ее перечислителя	Контекст ББ
Вызов метода MoveNext() (применительно к перечислителю <i>отложено инициализированной коллекции</i>)	Перечисление коллекции	Н о м е р п е р е ч и с л е н и я	1	Уникальное символьное значение коллекции, условие перечисления, место перечисления	Контекст ББ, Резюме метода (по завершении анализа всего метода)
Передача <i>отложено инициализированной коллекции</i> в немедленный LINQ-метод	Перечисление коллекции		2	Создание диагностики о множественном перечислении (включает себя информацию о каждом из перечислений). Диагностика создаётся только, если условия перечислений совместны	Временное резюме метода
Передача <i>отложено инициализированной коллекции</i> в любой другой метод (в том случае, если в резюме метода зафиксировано перечисление)	Перечисление в теле метода		>2	Добавление информации о новых перечислениях в уже созданную диагностику	Временное резюме метода
Передача коллекции в отложенный LINQ-метод	Создание новой отложено инициализированной коллекции			Связь уникальных символьных значений первоначальной коллекции и новой отложено инициализированной	Контекст ББ

```
1 public CSharpSmartTokenFormatter(OptionSet optionSet,
2     IEnumerable<AbstractFormattingRule> formattingRules,
3     CompilationUnitSyntax root)
4 {
5     ...
6     Contract.ThrowIfNull(formattingRules); //первое перечисление
7     _formattingRules = formattingRules; //второе перечисление
8     ...
9 }
```

Листинг 7. Место предупреждения.
Listing 7. Location of the error.

```
1 public static void ThrowIfNull<T>([NotNull] T value,
2     [CallerLineNumber] int lineNumber = 0) where T : class{
3 {
4     if (value is null)
5         Fail("Unexpected null", lineNumber);
6 }
```

Листинг 8. Реализация методов.
Listing 8. Methods implementation.

```

01 public static
02 ImmutableArray<ParameterName> GenerateParameterNames(...,
03 IEnumerable<Argument<TEExpressionSyntax>> arguments,
04 IList<string> reservedNames)
05 {
06     ...
07     var isFixed = reservedNames.Select(s => true)
08     .Concat(arguments.Select(a => a.NameColon != null))
09     .ToImmutableArray();
10
11     var parameterNames = reservedNames
12     .Concat(arguments
13     .Select(a => semanticModel
14     .GenerateNameForArgument(a, cancellationTokens))
15     ).ToImmutableArray();
16     ...
17 }

```

Листинг 9. Место предупреждения.
Listing 9. Location of the error.

ReSharper обнаружил здесь множественное перечисление коллекции arguments. Это истинное предупреждение: при вызове ToImmutableArray() по цепочке выполняются отложенные запросы – в первом случае сначала вычисляются результаты reservedNames.Select и arguments.Select, затем вычисляется результат их слияния (Concat); аналогично и во втором случае. Поскольку коллекция arguments передана в метод в качестве аргумента типа IEnumerable<T>, она потенциально является результатом отложенного запроса. В этом случае при вычислении isFixed и parameterNames она перечисляется дважды. SharpChecker не находит дефект такого вида: он связывает вызов немедленного метода ToImmutableArray() только с коллекцией reservedNames. Детектор не распознаёт, что передача arguments в качестве аргумента в Concat, вызванный для другой коллекции, повлечёт за собой перечисление arguments при материализации результата выполнения метода Concat (слияния двух коллекций).

3.1 Пример 1

Рассмотрим ошибку (Листинг 10), найденную в проекте Wcell в DBCDirComparer.cs на строках 38-58.

```

01 public void Compare(TextWriter writer)
02 {
03     var newFiles = new HashSet<string>(...);
04     ...
05     var addedFiles = newFiles.Except(oldFiles); // addedFiles
06     // результат отложенного метода
07
08     if (addedFiles.Count() > 0) // первое перечисление (Count())
09     {
10         writer.WriteLine("{0} files have been added:",
11         addedFiles.Count()); // второе перечисление (Count())
12         foreach (var file in addedFiles) // третье перечисление
13         // (в цикле foreach)
14         {
15             ...
16         }
17     }
18 }

```

Листинг 10. Место предупреждения.
Listing 10. Location of the error.

В данном примере коллекция addedFiles является результатом метода Except с отложенным выполнением, тем самым, она вычисляется каждый раз заново четыре раза: дважды при вызове метода и еще дважды в цикле. Чтобы избежать потерь в производительности, было бы целесообразно вычислить её с помощью newFiles.Except(oldFiles).ToList(). Этот пример является иллюстрацией наиболее распространённого случая некорректной работы с отложенно инициализированными коллекциями.

3.2 Пример 2

Данный пример фрагмента кода из проекта Lucene.Net, Lucene42DocValuesConsumer.cs строки 314-345 (Листинги 11-12) не содержит множественного перечисления. Здесь он приведён, чтобы показать необходимость чувствительности к потоку управления и путям выполнения программы в целях уменьшения количества ложных предупреждений.

```

01 public override void AddSortedField(FieldInfo field,
02     IEnumerable<BytesRef> values, IEnumerable<long?> docToOrd)
03 {
04     ...
05     foreach (long? n in docToOrd) // первое перечисление docToOrd
06     // в цикле foreach
07     {
08         ...
09     }
10     AddNumericField(field, docToOrd, false); // передача docToOrd
11     // и значения false в метод AddNumericField
12 }

```

Листинг 11. Потенциальное место ложного предупреждения.
Listing 11. Potential false warning location.

```

01 internal virtual void AddNumericField(FieldInfo field,
02     IEnumerable<long?> values, bool optimizeStorage)
03 {
04     ...
05     if (optimizeStorage) // проверка флага optimizeStorage
06     {
07         ...
08         foreach (long? nv in values) // второе перечисление
09         docToOrd в цикле foreach, которое не происходит
10         из-за невыполнения условия
11         {
12             ...
13         }
14     }
15 }

```

Листинг 12. Реализация методов.
Listing 12. Methods implementation.

Можно ошибочно предположить, что коллекция docToOrd будет перечислена дважды. Первый раз – в цикле foreach, и второй раз в теле метода AddNumericField. Однако цикл в вызываемом методе будет выполнен только в случае истинности условия optimizeStorage == true, но в метод было передано значение false, а значит, повторного перечисления произведено не будет. Это ложное предупреждение удалось устранить благодаря работе механизма чувствительности к путям выполнения программы. В данном случае другой статический анализатор, не имеющий полной поддержки

межпроцедурного анализа, например ReSharper, сгенерировал бы ложное предупреждение из-за факта передачи коллекции в метод как аргумента.

4. Заключение

Множественное перечисление коллекций в языке C# – часто встречающаяся проблема, приводящая к ухудшению производительности программ. При этом данную проблему очень сложно обнаружить при отладке. В связи с этим статический анализ является одним из самых эффективных способов её выявления. Таким образом, необходима полноценная поддержка поиска множественного перечисления в инструментах статического анализа. Существующие решения обладают ограниченной полнотой анализа и способны выявлять лишь тривиальные случаи множественного перечисления, не охватывая межпроцедурные сценарии.

Реализованный на основе символьного выполнения алгоритм с поддержкой межпроцедурного, чувствительного к потоку управления и путям выполнения программы механизма продемонстрировал высокую эффективность при тестировании на реальных проектах.

Экспериментальная оценка на 20 проектах с открытым исходным кодом подтвердила практическую ценность решения: было сгенерировано 213 предупреждений, 95% из которых являются истинными. При этом почти 38% истинных предупреждений связаны с неявными перечислениями в цепочках вызовов методов. Такие дефекты не могли быть обнаружены инструментами без поддержки межпроцедурного анализа. Сравнение с существующими решениями показало необходимость полноценной поддержки межпроцедурного чувствительного к путям и потокам выполнения анализа для обнаружения ошибок множественного перечисления *отложено инициализированных коллекций* в сложных сценариях.

Список литературы / References

- [1]. Tiobe index for ranking the popularity of programming languages. Available at: <https://www.tiobe.com/tiobe-index>, accessed 03.07.2026.
- [2]. Meijer E. The world according to LINQ. Communications of the ACM, 2011, vol. 54, no. 10, pp. 45-51.
- [3]. LINQ (Language Integrated Query). Available at: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/>, accessed 03.07.2026.
- [4]. Introduction to LINQ Queries in C#. Available at: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/get-started/introduction-to-linq-queries>, accessed 03.07.2026.
- [5]. Nostas J. et al. How do developers use the Java Stream API? International Conference on Computational Science and Its Applications. Cham: Springer International Publishing, 2021, pp. 323-335.
- [6]. Lott S. F. Functional Python programming: Discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads. Packt Publishing Ltd, 2018.
- [7]. ReSharper: The Visual Studio Extension for .NET Developers. Available at: <https://www.jetbrains.com/resharper/>, accessed 03.07.2026.
- [8]. Firouzi E., Sami A. Visual studio automated refactoring tool should improve development time, but resharpener led to more solution-build failures //2019 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT). IEEE, 2019, pp. 2-6.
- [9]. Code inspection: 'Possible multiple enumeration'. ReSharper Documentation. Available at: <https://www.jetbrains.com/help/resharper/PossibleMultipleEnumeration.html>, accessed 03.07.2026.
- [10]. .NET SDK: Set of libraries and tools that developers use to create .NET applications and libraries. Available at: <https://learn.microsoft.com/en-us/dotnet/core/sdk>, accessed 03.07.2026.
- [11]. CA1851: Possible multiple enumerations of IEnumerable collection. Available at: <https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1851>, accessed 03.07.2026.
- [12]. NetFabric.Hyperlinq.Analyzer: A Roslyn Analyzer that contains several rules to help improve enumeration performance when using C#. Available at: <https://www.nuget.org/packages/NetFabric.Hyperlinq.Analyzer/>, accessed 03.07.2026.

- [13]. Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, 2016, том 28, вып. 1, стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2. / Koshchev V., Ignatyev V., Borzilov A. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 1, pp. 21-40 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-2.
- [14]. Карцев В. С., Игнатьев В. Н. Поддержка Visual Basic .NET в статическом анализаторе SharpChecker. Труды ИСП РАН, 2024, том 36, № 3, с. 49-62 (на английском языке). DOI: 10.15514/ISPRAS-2024-36(3)-4. / Karcev V.S., Ignatyev V.N. Support of Visual Basic .NET in SharpChecker static analyzer. Trudy ISP RAN/Proc. ISP RAS, 2024, vol. 36, issue 3, pp. 49-62. DOI: 10.15514/ISPRAS-2024-36(3)-4.
- [15]. 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 03.07.2026.
- [16]. Koshchev 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.
- [17]. Baldoni R., Coppa E. et al. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 2018, vol. 51, issue 3, pp. 1-39.
- [18]. Аветисян А., Белеванцев А. и др. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, 2011, том 21, 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, 2011, vol. 21, pp. 23-38 (in Russian).
- [19]. Ханевская К.А., Тяжкороб У.В., Игнатьев В.Н. Поиск модификаций коллекции во время её перечисления в исходном коде на языке C# методами статического анализа. Труды ИСП РАН, 2026, том 38, вып. 3, часть 1, стр. 129-142. DOI: 10.15514/ISPRAS-2026-38(3)-7. / Khanevskaya K.A., Tsiashkorob U.V., Ignatiev V.N. Detection of collection modifications during its enumeration in the C# source code using static analysis. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 3, part 1, pp. 129-142 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-7.

Информация об авторах / Information about authors

Ксения Андреевна ХАНЕВСКАЯ – 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.

Ksenia Andreevna KHANEVSKAYA – 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. Her research interests are static program analysis, static symbolic execution, algorithms for finding defects in source.

Ульяна Владимировна ТЯЖКОРОБ – студентка аспирантуры 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.

Ul'jana Vladimirovna TSIASHKOROB is a postgraduate student at the Department of Radio Engineering and Computer Technologies of the Moscow Institute of Physics and Technology, an employee of the ISP RAS. Her research interests are 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 исходных текстах программ на основе статического анализа.

Valery Nikolayevich IGNATYEV – Cand. Sci. (Phys.-Math.) 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.