DOI: 10.15514/ISPRAS-2026-38(1)-4

Поиск ошибок в исходном коде на C# на основе статического анализа помеченных данных

М.В. Беляев, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>
П.И. Рагозина, ORCID: 0000-0003-4219-7203 <pragozina@ispras.ru>
В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

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

Аннотация. Ошибки в исходном коде, часто являющиеся причиной уязвимостей, могут быть найдены с помощью разных методов статического анализа. Однако методы на основе анализа помеченных данных имеют ряд преимуществ. В работе предложен комплекс методов, которые в совокупности с алгоритмом на основе IFDS для распространения помеченных данных позволяют превзойти результаты существующих промышленных инструментов. Комплекс методов реализован в промышленном статическом анализаторе SharpChecker и протестирован как на наборе тестов для статических анализаторов, таких как Juliet и WebGoat, для оценки полноты, так и на реальных проектах для оценки точности и производительности. Приведены результаты сравнения с популярными инструментами InferSharp, Security Code Scan, а также коммерческими анализаторами. Показаны относительно высокие значения полноты и точности анализатора SharpChecker. Результаты сравнения позволяют заключить, что предложенный комплекс методов имеет высокую практическую значимость при поиске уязвимостей.

Ключевые слова: статический анализ; анализ помеченных данных; язык C#; поиск уязвимостей; задача анализа потоков данных IFDS; символьное выполнение.

Для цитирования: Беляев М.В., Рагозина П.И., Игнатьев В.Н. Поиск ошибок в исходном коде на C# на основе статического анализа помеченных данных. Труды ИСП РАН, том 38, вып. 1, 2026 г., стр. 45–60. DOI: 10.15514/ISPRAS–2026–38(1)–4.

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

Search for Errors in C# Source Code Based on Static Taint Analysis

M.V. Belyaev, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>
P.I. Ragozina, ORCID: 0000-0003-4219-7203 <pragozina@ispras.ru>
V.N. Ignatyev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

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

Abstract. Various static analysis methods can detect errors in source code that often lead to vulnerabilities. Tainted data analysis, in particular, offers several advantages. This paper proposes a set of methods that, when combined with an IFDS-based taint propagation algorithm, can outperform existing industrial tools. The method set has been implemented in the industrial-grade SharpChecker analyzer and evaluated on both static analyzer test suites – such as Juliet and WebGoat – to assess completeness, and on real-world projects to measure accuracy and performance. A comparison with popular analyzers, including InferSharp, Security Code Scan, and others, demonstrates SharpChecker's comparatively high completeness and accuracy. These results lead to the conclusion that the proposed method set is of high practical value for vulnerability detection.

Keywords: static analysis; taint analysis; C# language; vulnerabilities detection; IFDS; symbolic execution.

For citation: Belyaev M.V., Ragozina P.I., Ignatyev V.N. Search for errors in C# source code based on static taint analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 1, 2026, pp. 45-60 (in Russian). DOI: 10.15514/ISPRAS-2026-38(1)-4.

Acknowledgements. The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

1. Введение

При разработке программ неизбежно появление ошибок и уязвимостей. Их поиск требует применения специальных инструментов. Существуют различные методы поиска уязвимостей, например, тестирование, статический и динамический анализ, фаззинг. Тестирование зачастую не подходит для обнаружения уязвимостей, так как они проявляются на нестандартных, не предусмотренных программистом данных. Поэтому более оправдан поиск уязвимостей при помощи динамического и статического анализа.

Динамический анализ требует запуска программы, а вместе с ним – настройки окружения и подготовки входных данных. Статический анализ не требует запуска и позволяет проверить все возможные пути выполнения программы. Для поиска уязвимостей могут применяться различные методы статического анализа: проверка файлов конфигурации, сигнатурный поиск, методы на основе символьного анализа и анализа помеченных данных (АПД). В работе [1] сравнивались последние два подхода и была отмечена перспективность методов на основе помеченных данных, которые, однако, обладают рядом недостатков. В данной работе предлагается дополнить базовую реализацию комплексом вспомогательных алгоритмов, обеспечивающих уверенное превосходство над другими решениями, в том числе и на основе АПД.

Анализ помеченных данных – один из широко распространенных методов поиска уязвимостей и дефектов безопасности. Он может быть как статическим [1], так и динамическим [2], и нацелен на отслеживание путей распространения чувствительных данных от *истока* – метода или инструкции, помечающей данные, до *стока* – метода или инструкции, использование помеченных данных в которой является уязвимостью или утечкой. Например, если пользовательский ввод без проверки попадает в вызов метода семейства `exec`, то возможна уязвимость *внедрение команд* (Command Injection [3]). В то время как при динамическом АПД требуется при необходимости отследить опасный путь, которым будут распространяться помеченные данные во время выполнения, задача статического – обнаружение всевозможных таких путей.

Для оценки результатов практически любого метода статического анализа используются следующие метрики: *полнота* – отношение числа верно найденных ошибок к их общему количеству в анализируемом проекте; *точность* – отношение числа верно найденных к общему количеству найденных ошибок; *производительность*. Однако, помимо них, промышленный анализатор кода должен соответствовать еще трем ключевым требованиям: детерминизм, масштабируемость и конфигурируемость [4].

- Детерминизм означает, что анализ должен выдавать идентичные результаты при повторных запусках на одном и том же коде.
- Масштабируемость – это способность анализатора эффективно работать с большими кодовыми базами в миллионы строк кода без экспоненциального роста времени анализа, потребления памяти и количества выданных предупреждений.
- Конфигурируемость – это возможность адаптировать анализатор под нужды конкретного проекта или пользователя, что в контексте АПД означает наличие интерфейса для описания собственных наборов истоков, стоков и других параметров поиска ошибок.

В работе описан метод анализа, реализованный в рамках разрабатываемого в ИСП РАН промышленного статического анализатора языка C# SharpChecker. Этот метод позволяет, учитывая указанные выше требования, достичь результатов, превосходящих другие известные решения, такие как Security Code Scan, InferSharp и другие.

Существуют различные методы реализации анализа помеченных данных. Решение, рассмотренное в данной работе, основано на представлении задачи анализа помеченных данных ***IFDS**-задачи* [1], то есть задачи межпроцедурного (***I**nterprocedural*) анализа потока данных с конечным (***F**inite*) числом фактов и дистрибутивными (***D**istributive*) передаточными функциями, оперирующей подмножествами (***S**ubset*). Алгоритм решения такой задачи, называемый алгоритмом табуляции, был описан уже в 1995 году [5]. Он позволяет свести IFDS-задачу к задаче достижимости на расширенном межпроцедурном графе потока управления (ГПУ). Однако для использования с целью поиска уязвимостей описанный алгоритм требует решения следующих проблем:

- IFDS-решение крайне трудно масштабируется из-за необходимости полного построения и хранения в памяти межпроцедурного ГПУ;
- на определенных шаблонах кода IFDS-алгоритм не может завершить анализ;
- алгоритм не имеет чувствительности к путям выполнения, что снижает точность анализа;
- алгоритм требует дополнительного анализа косвенных и виртуальных вызовов, иначе снижаются полнота и точность.
- алгоритм требует дополнительного моделирования библиотечных функций, иначе помеченность теряется при обработке таких вызовов.

В связи с этим необходима разработка комплекса вспомогательных методов анализа для повышения показателей анализа на основе IFDS. В работе предложен такой комплекс, описаны его преимущества и недостатки. Оценивается вклад как отдельных компонентов комплекса, так и сравнение итоговых результатов с другими инструментами статического анализа – InferSharp, Security Code Scan и другими.

В разделе 2 данной работы будут более подробно рассмотрены конкретные решения этих проблем, выбранные для анализатора помеченных данных в SharpChecker.

Некоторые из элементов этого комплекса, реализованные в рамках статического анализатора, уже были упомянуты в статьях ([6-7]), однако данная работа впервые охватывает их все и даёт совокупное представление о полученной технологии статического анализа помеченных данных.

2. Схема работы анализатора SharpChecker

Многие из перечисленных выше алгоритмов актуальны не только для анализа помеченных данных, но и для других методов статического анализа. В связи с этим, многие из них уже реализованы в промышленном анализаторе для символьного анализа, а их результаты могут повторно использоваться в АПД без дополнительных накладных расходов.

Работа SharpChecker основывается на компиляторной платформе Microsoft .NET Compiler Platform (Roslyn) [8]. Roslyn предоставляет абстрактное синтаксическое дерево (АСД), таблицу символов и основу для графа потока управления, а SharpChecker использует их для построения графов потока управления всех методов и дальнейшего анализа.

Данный процесс включает следующие этапы [9]:

1) Анализ синтаксического дерева: выполняется разбор дерева, выявление явных и неявных вызовов, построение иерархии классов и поиск синтаксических ошибок.
2) Построение статического графа вызовов.
3) Чувствительный к путям и контексту вызова межпроцедурный анализ на основе символьного выполнения. Производится анализ каждой функции в обратном топологическом порядке по графу вызовов с помощью движка символьного выполнения. Масштабируемый межпроцедурный анализ реализуется с использованием резюме методов для хранения собранной информации. Резюме используются при обработке инструкции вызова вместо анализа кода вызываемого метода. Для каждой функции выполняется:
    a) построение графа потока управления;
    b) обход базовых блоков в порядке выполнения, в процессе которого вычисляется состояние памяти программы и генерируются события анализа;
    c) обработка этих событий детекторами для сбора данных и выполнения проверок.
4) Анализ помеченных данных: выполняется на основе описанного выше алгоритма;
5) Формирование отчета: на заключительном этапе обработчик выдает предупреждения на основе собранной информации.

Третий этап анализа реализует символьное выполнение с объединением состояний. Каждая функция рассматривается как точка входа в программу. Все переменные, используемые функцией, параметризуются в начальной точке символьными значениями. Затем все инструкции ГПУ символьно исполняются, обновляя символьное состояние памяти программы. В точках слияния путей выполняется объединение состояний памяти с учетом предиката каждого пути, что в некоторых случаях может привести к потере точности. Циклы обходятся три раза: моделируется первая итерация и две «произвольные», перед анализом которых всем измененным в цикле переменным присваиваются новые символьные значения. Такой подход позволяет выполнить анализ большинства возможных путей выполнения функции, не запуская её. В анализаторе SharpChecker метод символьного выполнения и анализ помеченных данных могут запускаться как независимо, так и вместе, а при совместном запуске данные, получаемые с помощью первого метода, могут быть использованы для повышения точности и полноты второго. Подробнее эта интеграция будет описана в дальнейших разделах.

Для работы символьного выполнения, как и для АПД, требуется моделирование библиотечных функций. В анализаторе SharpChecker реализовано сразу несколько методов [7]: база данных аннотаций библиотечных функций и их спецификации на C#. Эти методы и созданные спецификации могут использоваться при поиске уязвимостей с помощью АПД. На этапе символьного анализа для каждой инструкции вызова вычисляется список возможных кандидатов с помощью методов девиртуализации, анализа интерфейсов и

делегатов с учетом информации о типах созданных объектов, вычисляемых при символьном исполнении. Данная информация также доступна на следующем этапе. Кроме этого, резюме методов могут быть сериализованы на диск для использования как на более поздних этапах, так и при следующем запуске.

3. Анализ помеченных данных в SharpChecker

Базовая реализация АПД в SharpChecker описана в статьях [4], [9] и [10]. Подсистема анализа состоит из движка анализа, основанного на алгоритме IFDS, описания набора искомых ошибок, которые задаются в виде правил, описанных ниже, и вспомогательных подсистем, необходимых для эффективного сбора данных для движка.

Правило безопасности для АПД – утверждение о программе, нарушение которого означает наличие потенциального дефекта, связанного с безопасностью программы, – например, утечки пароля, SQL-инъекции. Правила безопасности задают наборы истоков и стоков, а задача проверки таких правил заключается в поиске путей от истоков до стоков путем межпроцедурного анализа помеченных данных.

В общем случае существует два направления анализа путей распространения помеченных данных: прямое и обратное. Нарушения разных правил безопасности могут обнаруживаться более эффективно при выборе того или иного направления. Например, SQL-инъекцию (CWE-89 [11]) проще находить в прямом направлении, отслеживая распространение данных из истока – функции чтения пользовательского ввода – до одного или нескольких SQL-запросов, количество которых в проекте обычно больше. В то же время при поиске константных паролей (CWE-798 [12]) эффективнее использовать обратное направление – от стока (использование строки в качестве пароля) к истоку (создание константы), которых в программе на несколько порядков больше.

Также правило безопасности может быть **составным** – в таком случае для его проверки требуется исследование нескольких связанных между собой путей распространения данных в программе. Такие правила задаются как комбинации простых правил безопасности помеченных данных: в качестве истока или стока в одном правиле используется другое правило.

Формализованное описание правил безопасности помеченных данных в SharpChecker задается в текстовом формате (JSON) и для каждого отдельного правила включает в себя:

- название дефекта;
- направление анализа (прямое или обратное);
- множества истоков и стоков;
- множество *пропагаторов* – специальных передаточных функций, определяющих особенности распространения помеченных данных через внешние функции для данного типа ошибки;
- множество *санитайзеров* – функций, прекращающих распространение помеченных данных, например, экранирование или удаление специальных символов, порождение исключений при обнаружении дефекта.

Анализ начинается при построении графа вызовов. На этом этапе в программе ищутся *начальные точки* анализа – инструкции в коде, соответствующие истокам одного из правил безопасности, с которых начинается распространение. В случае обратного направления анализа начальные точки вычисляются на основе стоков правил.

На основном этапе для каждой найденной точки строятся все возможные пути распространения данных из нее. Для этого анализируется семантика каждой инструкции промежуточного представления кода и сопоставляется с заданным для правила набором передаточных функций, санитайзеров и пропагаторов. При обнаружении новой точки

программы, куда могут попасть помеченные данные, она добавляется в очередь анализа. После завершения обработки текущей точки из очереди берется новая. Для сокращения объемов повторного анализа применяются резюме.

При обнаружении для некоторого правила пути от начальной точки до конечной в список предупреждений добавляется сообщение о нарушенном правиле.

В связи с неограниченной длиной путей в программе вводятся **пределы** – это максимальные значения глубины анализа помеченных данных, необходимые для отсечения слишком длинных и маловероятных путей. Пределы также могут при необходимости использоваться для сокращения времени анализа.

3.1 Дополняющие методы анализа

Как было отмечено ранее, без моделирования окружения, анализа виртуальных и других косвенных вызовов результаты подхода на основе IFDS не удовлетворяют требованиям, предъявляемым к современным статическим анализаторам. В данном разделе будут подробнее описаны изменения и расширения IFDS-алгоритма анализа помеченных данных, внесенные с целью повышения масштабируемости, точности, полноты и конфигурируемости.

3.1.1 Анализ библиотечных функций

В анализаторе SharpChecker реализовано несколько методов моделирования библиотечных функций, описанных в статье [7]. Анализ помеченных данных использует подход на основе спецификаций. Спецификация библиотечной функции – это «сокращенная» реализация этой функции, имеющая идентичный прототип, код которой содержит только важные для анализа инструкции и не содержит ничего избыточного, как например, вызовов других функций, циклов и т. д. Такие спецификации обычно создаются вручную для часто используемых методов, а также с помощью больших языковых моделей [13]. Они позволяют описать все возможные пути распространения помеченных данных в результате выполнения метода. Отсутствие избыточного кода в спецификациях обеспечивает высокую скорость анализа по сравнению с анализом настоящей реализации, если она доступна, и повышает полноту анализа по сравнению с пропуском библиотечных функций.

Многие функции, например, реализующие работу с коллекциями, такие как `Add` и `Union`, имеют большое количество различных реализаций для разных типов данных, однако сохраняют общую семантику. В коде их вызовы представлены соответствующими интерфейсными методами. Поэтому вместо того, чтобы задавать спецификации для конкретных реализаций интерфейсного метода, для анализа помеченных данных реализована возможность указать путь распространения данных через функции-пропагаторы (которыми могут являться и интерфейсные методы) непосредственно в файле описания правил безопасности. Это также позволяет задавать пути распространения, специфичные для конкретного типа дефекта. Например, в случае поиска деления на ноль, метод `int.Parse` является пропагатором, а для SQL-инъекции – нет, так как угрозу представляют только строки.

Таким образом, моделирование окружения является наиболее важным средством, обеспечивающим полноту, так как в противном случае анализ будет страдать от недопомеченности в результате потери информации в многочисленных вызовах внешних функций.

Наряду с пропагаторами реализована возможность указывать санитайзеры – функции или инструкции, снимающие пометку. Это позволяет запретить отдельные пути распространения для специфичных правил безопасности, что необходимо для поддержки корректной проверки данных перед использованием. Например, при поиске ошибок деления на ноль сравнение помеченного значения с константой может снимать с него пометку.

3.1.2 Девиртуализация

Обработка вызовов, в которых вызываемый метод точно не известен на этапе компиляции, может приводить и к недостаточной, и к избыточной помеченности. В таких вызовах могут находиться как стоки, для которых необходимо выдавать предупреждение, так и санитайзеры, обеспечивающие проверку данных, а также возможно их копирование и распространение пометок.

При анализе кода на C#, для которого характерно широкое использование виртуальных и других косвенных вызовов – например, через делегаты и интерфейсы – эта проблема особенно актуальна. Для ее решения в рамках SharpChecker используются преимущества комплексного анализа кода различными методами: на этапе построения графа вызовов работают простые алгоритмы девиртуализации, которые по иерархии классов сокращают множество допустимых кандидатов виртуального вызова. На этапе символьного выполнения вычисляется еще более точная информация о возможных значениях и типах переменных, на основе которой выбираются наиболее подходящие кандидаты и информация о них сохраняется для последующего анализа помеченных данных. Аналогично выполняется уточнение методов, вызываемых из делегатов и интерфейсов. В случае полного отсутствия анализа или в результате менее точных подходов, характерных для других инструментов, список кандидатов может быть как пустым, так и содержать лишние методы, негативно сказываясь на показателях анализатора.

3.1.3 Объединение начальных точек

Многие правила безопасности имеют общие истоки и похожие списки санитайзеров и пропагаторов, а различаются только набором стоков. Например, пользовательский ввод рассматривается в качестве истока помеченных данных многих правил безопасности, т. к. без должной проверки не должен использоваться ни в SQL-запросах, ни при запуске команд. Количество правил безопасности в SharpChecker на данный момент превосходит 25 и непрерывно растет. Независимое выполнение анализа правил приводит к снижению масштабируемости при увеличении числа детекторов из-за избыточных повторных вычислений. Для повышения масштабируемости с ростом числа детекторов был реализован алгоритм объединения анализа общих для нескольких правил начальных точек. При этом у совместного выполнения есть слабые стороны, например, усложнение реализации и трудности в достижении одинаковых результатов между совместным и раздельным анализом – например, в случае, когда правила имеют одинаковые истоки, но различные наборы пропагаторов и санитайзеров.

В качестве примера правил с общими начальными точками можно рассмотреть вышеупомянутые SQL-инъекцию и инъекцию команды ОС. И в первом, и во втором случае начальными точками (истоками) будут команды пользовательского ввода, также похожим образом будет производиться распространение пометок. Разница состоит только в стоках (SQL-запрос в первом случае и выполнение команды во втором). Реализация совместной обработки истоков ускорила этап анализа помеченных данных более чем в 1.5 раза при тестировании на наборе проектов из 3.4 млн. строк кода. Кроме того, эта оптимизация обеспечила возможность добавления новых детекторов ошибок без существенного влияния на время работы. Это особенно важно для поддержки созданных пользователями детекторов.

3.1.4 Фильтрация недостижимого кода

Не весь код программы, поступающей на вход статическому анализатору, является достижимым при каких-либо входных данных программы. Например, сбор отладочной информации может включаться при определенных параметрах сборки программы, а наличие кода, реализующего сохранение отладочной информации, может быть причиной ложного предупреждения об утечке конфиденциальных данных. Алгоритм IFDS не способен

обеспечить чувствительность к путям выполнения и не позволяет избавиться от ложных предупреждений в недостижимом коде. Однако использование информации о достижимости базовых блоков графа потока управления, собранной на этапе символьного выполнения, помогает фильтровать такие ошибки. Проверка достижимости анализируемого кода не только повышает точность, и обеспечивает чувствительность к путям, но и сокращает затраты ресурсов и времени, если завершать анализ в недостижимых блоках.

На этапе символьного выполнения идентификаторы недостижимых базовых блоков сохраняются в резюме методов. Далее ядро анализа помеченных данных завершает распространение пометок при обработке инструкций недостижимых блоков.

Тестирование на наборе из проектов с открытым исходным кодом показало, что использование данных о недостижимом коде, получаемых при помощи символьного выполнения, приводит не только к исчезновению срабатываний, относящихся к недостижимому коду (не играющих роли в проекте и, как правило, ложных), но и к обнаружению новых уязвимостей, достигнуть которых ранее не позволяло достижение пределов.

3.1.5 Фильтрация методов, не взаимодействующих с помеченными данными

Алгоритм IFDS предполагает последовательную обработку всех инструкций всех методов на пути выполнения. Даже если метод никак не взаимодействует с помеченными данными, приходится обходить все его инструкции, чтобы в этом убедиться. Например, метод, принимающий объект с помеченным полем, может не использовать это поле. К подобным случаям также относятся методы класса, не использующие его помеченные поля. Так, в примере, приведенном на рис. 1, `Bad()` – опасный метод, так как может привести к SQL-инъекции. `Good()` – пример метода, анализ которого может быть пропущен без ущерба для результата.

Использование предварительного простого анализа, например, только абстрактного синтаксического дерева, позволяет заранее определить все используемые методом поля и параметры, чтобы было можно пропустить его обработку целиком, когда он не может повлиять на распространение помеченных данных. Подход, реализованный в SharpChecker, состоит из трех этапов:

1) На этапе анализа абстрактного синтаксического дерева выполняется сбор информации обо всех использованных методом переменных и их сохранение в резюме.
2) Далее выполняется распространение данных по графу вызовов, что обеспечивает перенос информации из резюме всех вызываемых методов в вызывающий.
3) При обработке вызова метода на этапе анализа помеченных данных список используемых полей из резюме сопоставляется с помеченными данными. Если никакие помеченные переменные им не используются, анализ этого метода пропускается.

Эта оптимизация позволяет находить больше дефектов, что вызвано более поздним достижением пределов, завершающих анализ. Так, на описанном наборе проектов удалось получить 149 новых истинных срабатываний. Кроме того, пропуск этих функций повышает масштабируемость.

3.2 Пользовательские правила безопасности

Для адаптации анализа к конкретной предметной области и кодовой базе полезной является возможность настройки параметров анализа помеченных данных на стороне пользователя. Помимо редактирования существующих в анализаторе правил безопасности, поставляемых в текстовом формате JSON, пользователи могут создавать собственные правила для отслеживания возможных путей распространения чувствительных данных в своем коде.

В рамках SharpChecker это реализовано путем предоставления пользователю формата описания правил безопасности на основе API, реализованного в ядре анализа помеченных данных. Полнота доступного API подтверждается наличием более 25 существующих детекторов, обладающих высокими показателями точности и полноты. Реализованные в SharpChecker оптимизации ядра анализа обеспечивают масштабируемость и точность работы даже при большом числе точек начала анализа.

```
1  class C {
2      SqliteCommand c = new();
3      string taintedStr;
4      int i;
5      public void Bad() {
6          string query = "SELECT* FROM table WHERE name = '" + taintedStr + "'";
7          // переменная query помечена
8          c.CommandText += query; // помеченная переменная query попала в сток
9      }
10     public void Good(int arg) => i = arg;
11     public void F() {
12         taintedStr = Console.ReadLine(); // (!SQL_INJECTION)
13         // переменная this.taintedStr помечена
14         Good(3);
15         Bad(); // c.CommandText помечен
16         c.ExecuteReader(); // выполнение опасного запроса
17     }
18 }
```

Рис. 1. Метод Bad(), распространяющий помеченные данные, и метод Good(), не влияющий на их распространение.

Fig. 1. The Bad() method that propagates the tainted data and the Good() method that does not affect its propagation.

3.3 Поддержка Visual Basic

Язык программирования Visual Basic (VB.NET), несмотря на свою популярность (7-е место в рейтинге TIOBE [14]), остается слабо охваченным промышленными анализаторами, которые фокусируются на C#. Это создает пробел в инструментах для анализа legacy-проектов на VB.NET, которые часто содержат скрытые ошибки и уязвимости. Кроме того, основной компилятор языков C# и VB.NET Roslyn содержит большое количество кода на VB.NET и требует проведения полного анализа для возможности его применения в ответственных проектах, например, при сертификации. В связи с этим, поддержка языка реализована в SharpChecker [15], включая подсистему анализа помеченных данных.

Внедрение поддержки VB.NET в SharpChecker основано на использовании общей компиляторной инфраструктуры Roslyn, поддерживающей оба языка. Это упростило добавление поддержки VB.NET для большей части детекторов уязвимостей, за исключением относящихся к символьному выполнению: для них потребовалось учитывать различия в представлении символов между языками, такие как организация свойств и их внутренних полей.

Поскольку анализ помеченных данных в SharpChecker реализован на унифицированных структурах данных Roslyn, как например IOperation, его адаптация к VB.NET не потребовала значительных изменений ни ядра, ни правил безопасности. После доработки построения графа вызовов подсистема автоматически начала корректно работать с кодом на VB.NET. Провести сравнительное тестирование для этого языка программирования оказалось трудно ввиду его мало распространённой поддержки анализаторами C#. Тем не менее на проектах с открытым исходным кодом он показывает высокую точность. Так, на проекте EVE-IPH [16] (игровая утилита с открытым исходным кодом) обнаружено 342 срабатываний, все они истинные.

Табл. 1. Иллюстрация связи дополнений к классическому IFDS-анализу и аспектов оценки статического анализатора, которые они повышают. Поля, которые соответствуют связанным дополнению и аспекту, отмечены символом «+».
Table 1. An illustration of the relationship between the additions to the classical IFDS analysis and the aspects of the static analyzer evaluation improved by them. Fields that correspond to the associated addition and aspect are marked with a "+" symbol.

	Масштабируемость	Точность	Полнота	Конфигурируемость	Чувствительность к путям
Анализ библиотечных функций	+		+		
Девирту-ализация		+	+		
Объединение начальных точек	+				
Фильтрация недостижимого кода		+	+		+
Фильтрация методов, не взаимодействующих с помеченными данными	+		+		
Пользовательские детекторы				+	
Поддержка Visual Basic			+		

4. Сравнение с существующими решениями

Основными метриками качества анализатора являются полнота и точность. Если точность можно оценить после трудоемкой разметки предупреждений, выданных анализатором, то для оценки полноты необходима информация о всех ошибках в коде, которая доступна только для тестов. Поэтому для оценки полноты используются специальные наборы тестов для статических анализаторов. Для языка C# существует версия Juliet [17], состоящая из множества различных небольших тестовых программ с размеченными уязвимостями. Помимо этого, известны проекты с внедренными уязвимостями WebGoat.NET [18], PumaPrey [19] и другие. Для оценки полноты на реальных проектах можно выполнить сравнение с другими инструментами анализа. Наиболее известные анализаторы, заявляющие поддержку поиска ошибок, реализованных в SharpChecker на основе анализа помеченных данных, описаны ниже.

4.1 InferSharp

InferSharp [20-21] – это статический анализатор кода для C#. Он специализируется на межпроцедурном анализе потоков данных для обнаружения уязвимостей безопасности и включает в себя в том числе и анализ помеченных данных, однако он не использует напрямую алгоритмы решения IFDS-задач. Авторы разработали собственную архитектуру статического анализа помеченных данных, которая решает схожие проблемы другими методами. Анализируется не исходный код, а скомпилированные бинарные файлы.

Описанные в данной работе особенности присутствуют в InferSharp в следующем виде:

- поддерживается девиртуализация;
- недостижимый код пропускается и не обрабатывается анализом помеченных данных, при этом используются сложные межпроцедурные алгоритмы его поиска;
- архитектура InferSharp не предполагает добавления пользовательских правил безопасности для анализа помеченных данных.

4.2 Security Code Scan

Security Code Scan [22] – специализированный для платформы .NET инструмент статического анализа кода на C# и VisualBasic.NET. Как и SharpChecker, он использует возможности компилятора Roslyn для анализа исходного кода. Анализатор поддерживает поиск уязвимостей и проблем безопасности, в том числе с помощью анализа помеченных данных. Однако анализ выполняется без использования IFDS, вместо этого производится анализ на основе символьного выполнения, чувствительный к путям, но с ограниченной контекстной чувствительностью. Среди основных особенностей можно отметить отсутствие девиртуализации и учета недостижимого кода. Инструмент позволяет пользователям с помощью XML-конфигурации создавать собственные правила с конфигурируемыми истоками, стоками и санитайзерами, но, в отличие от SharpChecker, не позволяет настраивать пропагаторы или создавать более сложную логику распространения.

4.3 SonarQube

SonarQube [23] – платформа с открытым исходным кодом для непрерывного анализа и измерения качества программного кода для большого набора языков, в том числе и C#. Существует также коммерческая версия анализатора, реализующая более глубокий анализ при поиске ошибок безопасности. Анализ помеченных данных входит в набор возможностей анализатора, с алгоритмами, также основанными на IFDS-подходе.

- Девиртуализация отсутствует, межпроцедурность очень ограничена.
- Проверки на недостижимость при анализе помеченных данных не производятся, недостижимые истоки/стоки/пути анализируются.
- Анализатор предоставляет пользователю возможность модифицировать некоторые из существующих правил под свои нужды, а также создавать собственные. При создании детекторов предоставляются очень широкие возможности, в том числе полный контроль над источниками, стоками, пропагаторами, расширенные паттерны распространения и сложные сценарии.

4.4 Анализатор CP

Анализатор CP – это пакет коммерческого программного обеспечения, состоящий из статического и динамического анализаторов кода, производящий поиск ошибок и уязвимостей в исходных кодах программ, написанных на Си, C++, Java, C# и JavaScript.

- Активно используется девиртуализация.
- Код может проверяться на достижимость, но недостижимый код тоже анализируется.
- Анализатор предоставляет широкие возможности для создания пользовательских детекторов ошибок.

4.5 Результаты тестирования

Из-за различий в реализациях между SharpChecker и другими анализаторами произвести абсолютно точное сравнение между ними не представляется возможным. Так, независимый запуск анализа помеченных данных и анализа методом символьного выполнения, реализованный в SharpChecker, невозможен в других анализаторах, что делает сравнение времени выполнения непоказательным. Кроме того, было решено сравнивать только результаты анализа помеченных данных, однако при этом возникла проблема отсутствия четкого обособления в некоторых детекторах этой категории ошибок. Так, в SonarQube ввиду ограниченной межпроцедурности при поиске уязвимости вида *SQL-инъекция* обнаруживаются не пути от истока к стоку, а только возможные стоки (использование поступившей извне метода переменной в SQL-запросе). Аналогично встречаются такие

«упрощенные» предупреждения и для нескольких других типов уязвимостей. Анализ помеченных данных при этом не обособляется как отдельная категория.

В InferSharp и Security Code Scan удалось явно выделить типы ошибок, связанные с анализом помеченных данных, а для анализатора SonarQube было решено использовать для сравнения типы ошибок, как правило относящиеся к анализу помеченных данных. При этом «упрощенное» предупреждение считалось истинным, если отмеченный сток в действительности может быть достигнут из какого-либо истока.

Упомянутое выше отсутствие в части анализаторов девиртуализации оказало влияние на количество срабатываний, так как интерфейсы и абстрактные классы часто встречаются в проектах на C#.

Анализатор InferSharp лучше работает с более простыми программами, а на проектах с открытым кодом и тестах Juliet его анализ помеченных данных не обнаружил ни одной ошибки несмотря на то, что на небольших синтетических тестах он зачастую показывает более точный результат по сравнению с другими описанными детекторами за счет применения девиртуализации и поиска недостижимых участков кода.

Все анализаторы на приведенных проектах показывают детерминированный результат.

Табл. 1 и 2 демонстрируют, что, хотя точность SharpChecker не абсолютна, она превышает точность большинства других детекторов на обоих представленных проектах. Кроме того, покрытие данного анализатора наиболее широкое из рассматриваемых. Табл. 3 демонстрирует результаты анализа рассматриваемых анализаторов на размеченном наборе синтетических тестов Juliet test suite (версия 1.3), что позволило получить предположительные значения точности и полноты анализа помеченных данных. Можно заметить, что SharpChecker продемонстрировал по совокупности сравнительно высокие показатели точности и полноты, а также широкий набор правил анализа. При этом Security Code Scan более точен, но проигрывает в полноте. Кроме того, для некоторых анализаторов, в частности в случае анализатора CP, малое количество или отсутствие срабатываний для ряда правил связано с более узким набором стоков для них, нежели предполагается тестами.

Табл. 1. Результаты анализа намеренно небезопасного проекта WebGoat [18].
Table 1. Results of analysis of the intentionally insecure WebGoat [18] project.

	Всего	TP	Точность
SharpChecker	131	103	78,6%
InferSharp	0	0	–
Security Code Scan	36	18	50%
SonarQube	42	29	69%
Анализатор CP	121	105	87%

Табл. 2. Результаты анализа проекта OpenSimulator [24], содержащего наибольшее количество ошибок из тех рассматриваемых проектов с открытым кодом, которые удалось запустить без ошибок на всех четырёх анализаторах.
Table 2. The results of the analysis of the OpenSimulator project [24], which contains the highest number of errors among the considered open-source projects that were successfully run without errors on all four third-party analyzers.

	Всего	TP	Точность
SharpChecker	598	418	70%
InferSharp	0	0	–

Security Code Scan	0	0	–
SonarQube	144	92	64%
Анализатор CP	397	112	28%

Табл. 3. Подробные результаты анализа помеченных данных на тестах Juliet.
Table 3. Detailed results of the taint analysis of the Juliet tests.

Тип дефекта	SharpChecker			Security Code Scan			SonarQube			Анализатор CP		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Code injection	570	540	0	–			–			0	0	570
Command injection	570	0	0	186	0	384	–			208	0	362
Cross site redirect	513	0	0	186	0	327	–			208	2	305
Division by zero	2128	2820	266	–			–			60	72	2234
Format string injection	798	28	0	–			–			0	0	798
Hardcoded password	57	0	57	17	0	97	4	0	110	83	0	31
Information exposure	51	0	51	–			–			–		
Insecure randomness	0	0	34	–			–			–		
Key management	74	0	0	–			0	17	74	44	3	30
LDAP injection	570	0	0	186	0	384	–			364	7	206
Log injection	0	0	570	–			–			0	0	570
Missing encryption	121	526	1398	–			–			168	1894	1391
Path traversal	870	0	270	322	0	818	–			348	8	792
Process control	0	0	17	–			–			–		
XSS	1824	0	228	558	0	1494	–			0	0	2052
Setting manipulation	570	0	0	–			–			–		
SQL injection	1539	54	0	420	0	1119	684	864	855	510	10	1029

Тип дефекта	SharpChecker			Security Code Scan			SonarQube			Анализатор CP		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Uncontrolled memory allocation	1824	0	245	–			–			–		
Uncontrolled resource consumption	2242	80	0	–			–			–		
Unsafe cookie usage	–			–			17	263	17	17	98	17
Unsafe reflection	513	0	0	–			–			–		
XPath injection	570	18	0	210	308	360	–			204	213	366
Итого	15404	4066	3136	2085	308	4983	705	1144	1056	2214	2307	10753
Полнота	83%			30%			40%			17%		
Точность	79%			87%			38%			49%		

5. Заключение

Разработанный подход позволил достичь высоких относительно других существующих решений полноты и точности, что подтверждено экспериментальными результатами. Это было достигнуто благодаря сочетанию IFDS-анализа в качестве основы с рядом новых подходов, заполняющих пробелы IFDS-абстракции. Кроме того, сделаны шаги для повышения детерминизма, масштабируемости, конфигурируемости.

Перспективы дальнейших исследований видятся в расширении поддерживаемых языков программирования, углублении межпроцедурного анализа и разработке новых адаптивных механизмов настройки параметров анализа. Реализация этих направлений позволит еще больше повысить полноту и точность метода, а также расширить область его практического применения.

Список литературы

[1]. Belyaev M. V. et al. Comparative analysis of two approaches to static taint analysis //Programming and Computer Software. – 2018. – Т. 44. – №. 6. – С. 459-466.

[2]. Кулямин В. В. Обзор методов динамического анализа программного обеспечения //Труды Института системного программирования РАН. – 2023. – Т. 35. – №. 4. – С. 7-44.

[3]. The MITRE Corporation, CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'), MITRE CWE. Available at: https://cwe.mitre.org/data/definitions/78, accessed 10.10.2025.

[4]. Ignatyev V. N. A Holistic Static Analysis for Finding Errors in Source Code //Программная инженерия. – 2025. – Т. 16. – №. 10. – С. 517-531.

[5]. Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability //Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1995. – С. 49-61. DOI: 10.1145/199448.199462.

[6]. Shimchik N. V., Ignatyev V. N., Belevantsev A. A. Improving accuracy and completeness of source code static taint analysis //2021 Ivannikov Ispras Open Conference (ISPRAS). – IEEE, 2021. – С. 61-68.
[7]. Biktimirov M. G., Ignatyev V. N., Belyaev M. V. Improving the accuracy of library function modeling in the static analyzer //2023 Ivannikov Ispras Open Conference (ISPRAS). – IEEE, 2023. – С. 26-32.
[8]. The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. Available at: https://github.com/dotnet/Roslyn, accessed 10.10.2025.
[9]. Кошелев В. К., Игнатьев В. Н., Борзилов А. И. Инфраструктура статического анализа программ на языке С //Труды Института системного программирования РАН. – 2016. – Т. 28. – №. 1. – С. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2.
[10]. Koshelev V. K. et al. SharpChecker: Static analysis tool for C# programs //Programming and Computer Software. – 2017. – Т. 43. – №. 4. – С. 268-276.
[11]. The MITRE Corporation, CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), MITRE CWE. Available at: https://cwe.mitre.org/data/definitions/89, accessed 10.10.2025.
[12]. The MITRE Corporation, CWE-798: Use of Hard-coded Credentials, MITRE CWE. Available at: https://cwe.mitre.org/data/definitions/798, accessed 10.10.2025.
[13]. Ignatyev V. N. et al. Large language models in source code static analysis //2024 Ivannikov Memorial Workshop (IVMEM). – IEEE, 2024. – С. 28-35.
[14]. TIOBE Index - TIOBE. Available at: https://www.tiobe.com/tiobe-index/, accessed 08.11.2025.
[15]. Карцев В. С., Игнатьев В. Н. Поддержка Visual Basic. NET в статическом анализаторе SharpChecker //Труды Института системного программирования РАН. – 2024. – Т. 36. – №. 3. – С. 49-62. DOI: 10.15514/ISPRAS-2024-36(3)-4
[16]. Code for the EVE Isk per Hour program. Available at: https://github.com/EVEIPH/EVE-IPH, accessed 27.11.2025.
[17]. Black P. E. Juliet 1.3 test suite: Changes from 1.2. – Gaithersburg, MD, USA : US Department of Commerce, National Institute of Standards and Technology, 2018.
[18]. WebGoat.NET is a deliberately insecure application. Available at: https://github.com/jerryhoff/WebGoat.NET, accessed 25.11.2025.
[19]. Puma Prey contains vulnerable .NET applications, which provide a target for running secure coding challenges, CTFs, and testing the Puma Scan analyzers. Available at: https://github.com/pumasecurity/puma-prey, accessed 25.11.2025.
[20]. Infersharp. Available at: https://github.com/microsoft/infersharp, accessed 10.10.2025.
[21]. Calcagno C., Distefano D. Infer: An automatic program verifier for memory safety of C programs //NASA Formal Methods Symposium. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – С. 459-465. DOI: 10.1007/978-3-642-20398-5_33
[22]. security-code-scan: Vulnerability Patterns Detector for C# and VB.NET. Available at: https://github.com/security-code-scan/security-code-scan, accessed 15.10.2025
[23]. SonarQube. Available at: https://github.com/SonarSource/sonarqube, accessed 15.10.2025.
[24]. OpenSimulator is an open source multi-platform, multi-user 3D application server. Available at: http://opensimulator.org/wiki/Main_Page, accessed 01.11.2025.

Информация об авторах / Information about authors

Михаил Владимирович БЕЛЯЕВ – младший научный сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, анализ помеченных данных.

Mikhail Vladimirovich BELYAEV is a Junior Researcher at ISP RAS. Research interests: compiler technologies, static code analysis, taint analysis.

Полина Ильинична РАГОЗИНА – старший лаборант ИСП РАН, исследователь в отделе компиляторных технологий. Научные интересы: статический анализ программ, символьное выполнение, анализ помеченных данных.

Polina Ilyinichna RAGOZINA is a Senior Laboratory Assistant at ISP RAS, a researcher at the Department of Compiler Technologies. Scientific interests: static analysis of programs, symbolic execution, taint analysis.

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. His research interests include program analysis techniques for error detection in program source code using classical static analysis and machine learning.