Санкт-Петербургский государственный университет

***Бучин Вячеслав Андреевич***

Выпускная квалификационная работа

# Статический анализ программ на платформе .NET

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н. Мордвинов Д. А.

Консультант:
ведущий разработчик ООО «ИИнтелКод» Костицын М. П.

Рецензент:
старший академический консультант ООО «КоулмэнТех» Соболь В.

Санкт-Петербург
2025

Saint Petersburg State University

***Viacheslav Buchin***

Bachelor's Thesis

# Static analysis for .NET programs

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2021 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D.A. Mordvinov

Consultant:
lead developer at LLC «IIntelCode» M.P. Kosticyn

Reviewer:
senior academic advisor at LLC «CoulmenTech» V. Sobol

Saint Petersburg
2025

# Оглавление

# Введение

В современном мире неуклонно растет количество различных приложений, а также их сложность. В связи с этим выполнение требований обеспечения качества для них становится все более трудоемким. Эти требования зачастую включают в себя обеспечение информационной безопасности. Согласно государственному стандарту Российской Федерации[1] одной из целей разработки безопасного программного обеспечения является нахождение участков исходного кода, которые могут быть потенциально использованы для реализации угроз информационной безопасности. Также данный стандарт включает в себя перечень процессов разработки безопасного программного обеспечения. В него входит *статический анализ* [9] исходного кода.

Главная особенность данного вида анализа — возможность производить исследование программы без ее выполнения. В отличие от динамического анализа, который выполняется во время работы программы, статический анализ позволяет обнаруживать проблемы на ранних этапах разработки. Одним из видов статического анализа является статическое тестирование безопасности приложения (англ. Static Application Security Testing, SAST) [16]. Данный вид анализа направлен на выявление различных уязвимостей в исходном коде, таких как SQL-инъекции, утечки памяти, некорректное использование API, а также многих других.

Платформа .NET, в рамках которой выполнена данная работа, продолжает динамично развиваться. Языки программирования данной платформы [2, 5] обладают большим количеством синтаксических конструкций. Кроме того, .NET поддерживает множество функциональных особенностей. Среди них, например, строгая статическая типизация и богатая система типов, позволяющая создавать обобщенные типы — те, что содержат в своем определении параметры, заместо которых разработчик в месте использования может подставить другой тип, в том числе и обобщенный. Также на параметры обобщенных типов могут

[1] ГОСТ Р 56939-2024

быть наложены различные ограничения, что снижает количество подходящих вместо них типов. Кроме того, .NET реализует механизм виртуальных вызовов, вследствие чего до момента исполнения программы неизвестно, какой код будет выполнен на самом деле. Вдобавок, .NET поддерживает передачу значений по ссылке при вызове метода. Перечисленное выше затрудняет разработку статического анализатора для платформы .NET, потому что большое влияние на его качество работы оказывает поддержка им различных особенностей языка программирования, на котором разработано приложение.

Проект USVM [18], в рамках которого выполнена данная работа, предназначен для создания средств автоматического анализа программ для разных языков программирования. В нем используются различные техники анализа: символьное исполнение, фаззинг, статический анализ. В настоящее время на базе проекта USVM реализованы средства анализа программ для следующих языков программирования: Java, Python, TypeScript. Такая универсальность достигается за счет использования абстрактной модели исходного кода и реализации техники анализа на ее основе. Вследствие чего, для поддержки нового языка программирования нужно построить данное представление, а для поддержки новой техники анализа — реализовать ее зависимые от языка программирования аспекты.

# 1 Постановка задачи

Целью данной работы является повышение безопасности ПО, разработанного на платформе .NET, с помощью обнаружения потенциальных уязвимостей статическим анализом во время разработки. Для достижения обозначенной цели были поставлены следующие задачи:

- провести обзор подходов анализа потока данных, а также инструментов статического анализа для платформы .NET;
- спроектировать решение, учитывающее особенности платформы .NET;
- реализовать спроектированное решение в инфраструктуре проекта USVM;
- провести экспериментальное сравнение разработанной системы с другими инструментами статического анализа для .NET.

# 2 Обзор

В данной главе рассмотрен анализ потока данных, а также его подходы: анализ помеченных данных (англ. taint-analysis), задачи IFDS. В завершении главы приводится обзор проекта USVM, в рамках которого выполнена эта работа.

## 2.1 Статический анализ

*Статический анализ* [6] — метод исследования программного кода без его выполнения, направленный на выявление ошибок, уязвимостей и потенциальных улучшений. Этот метод играет важную роль в обеспечении качества и безопасности программного обеспечения, особенно в условиях растущей сложности современных приложений. Статический анализ применяется для решения широкого спектра задач: от поиска синтаксических ошибок и нарушений стиля кода до выявления уязвимостей, таких как SQL-инъекции, утечки памяти или некорректное использование API. Данная глава посвящена рассмотрению одного из подходов статического анализа — анализа потока данных.

### 2.1.1 Анализ потока данных

*Анализ потока данных* (англ. *Data-flow analysis*) [8, 13] — это один из фундаментальных подходов статического анализа, который используется для изучения того, как данные перемещаются и преобразуются внутри программы.

Исследование с помощью данного подхода происходит следующим образом. Сначала программа разделяется на базовые блоки, а также строится граф потока управления, где вершины — базовые блоки, а ребра — переходы между ними. Далее для каждого блока $b$ необходимо решить следующие уравнения:

$$\begin{cases} out_b = effect(in_b), \\ in_b = merge_{out_p \in pred_b}(out_p), \end{cases}$$

где $in_i$, $out_i$ — входные и выходные состояния для блока $i$, $pred_i$ — вершины, предшествующие $i$ в графе потока управления. Функция перехода *effect* позволяет получить новое выходное состояние по входному, а функция $merge$ отвечает за получение входного состояния на основании всех выходных состояний предшествующих блоков. Каждый конкретный анализ определяет семантику функций *effect* и $merge$. Рассмотренные уравнения отвечают прямому анализу (англ. forward analysis), однако существует и обратный анализ (англ. backward analysis), в котором исследование графа потока управления происходит в обратном порядке: от конца к началу. Уравнения для такого вида анализа строятся аналогичным образом.

Рассмотрим важные характеристики анализа потока данных [10].

- *Чувствительность к потоку* — способность анализа учитывать порядок выполнения инструкций в программе.
- *Чувствительность к пути исполнения* — способность анализа учитывать ограничения, накладываемые на переменные при ветвлениях и условных переходах.
- *Чувствительность к контексту вызова* — способность учитывать информацию о месте вызова при исследовании метода, а также использовать результаты исследования в том же месте вызова, а не во всех возможных.

### 2.1.2 Задачи IFDS

Задачи *Interprocedural, Finite, Distributive, Subset* (IFDS) [1, 15] являются подмножеством задач, решаемых с помощью анализа потока данных. Они имеют следующие ограничения.

1. Множество состояний базовых блоков конечно.
2. Дистрибутивность *effect* относительно $merge$, т.е.
   $\forall s_1, s_2 \;\; merge(\mathit{effect}(s_1), \mathit{effect}(s_2)) = \mathit{effect}(merge(s_1, s_2))$

Эти ограничения позволили разработать эффективный полиномиальный алгоритм решения такого класса задач [15, 14]. Этого удалось достичь за счет сведения задачи IFDS к задаче поиска путей в специальном графе. Данный алгоритм оперирует следующими понятиями.

- *Факт* — специфичная для задачи информация в конкретной точке программы.
- *Нулевой* факт — факт, обозначающий достижимость инструкции.
- *Путь* факта — перемещение информации между инструкциями программы.
- *Flow-функции* — правила, определяющие пути факта, в зависимости от семантики инструкции исследуемой программы.

```
void writeZeroTo(out int p) {
    p = 0;
}

int foo() {
    int x;
    int y;
    writeZeroTo(x);
    return x + y;
}
```

Рис. 1: Программа для построения графа для IFDS

Подробнее рассмотрим приведенные выше понятия на примере программы с рис. 1, проведя анализ неинициализированных переменных, в ходе которого требуется понять, возвращает ли функция `foo()` неинициализированные данные. В данном случае фактами является информация о том, находятся ли в переменной или аргументе функции неинициализированные данные. Для рассмотрения путей фактов и flow-функций обратимся к графу на рис. 2.

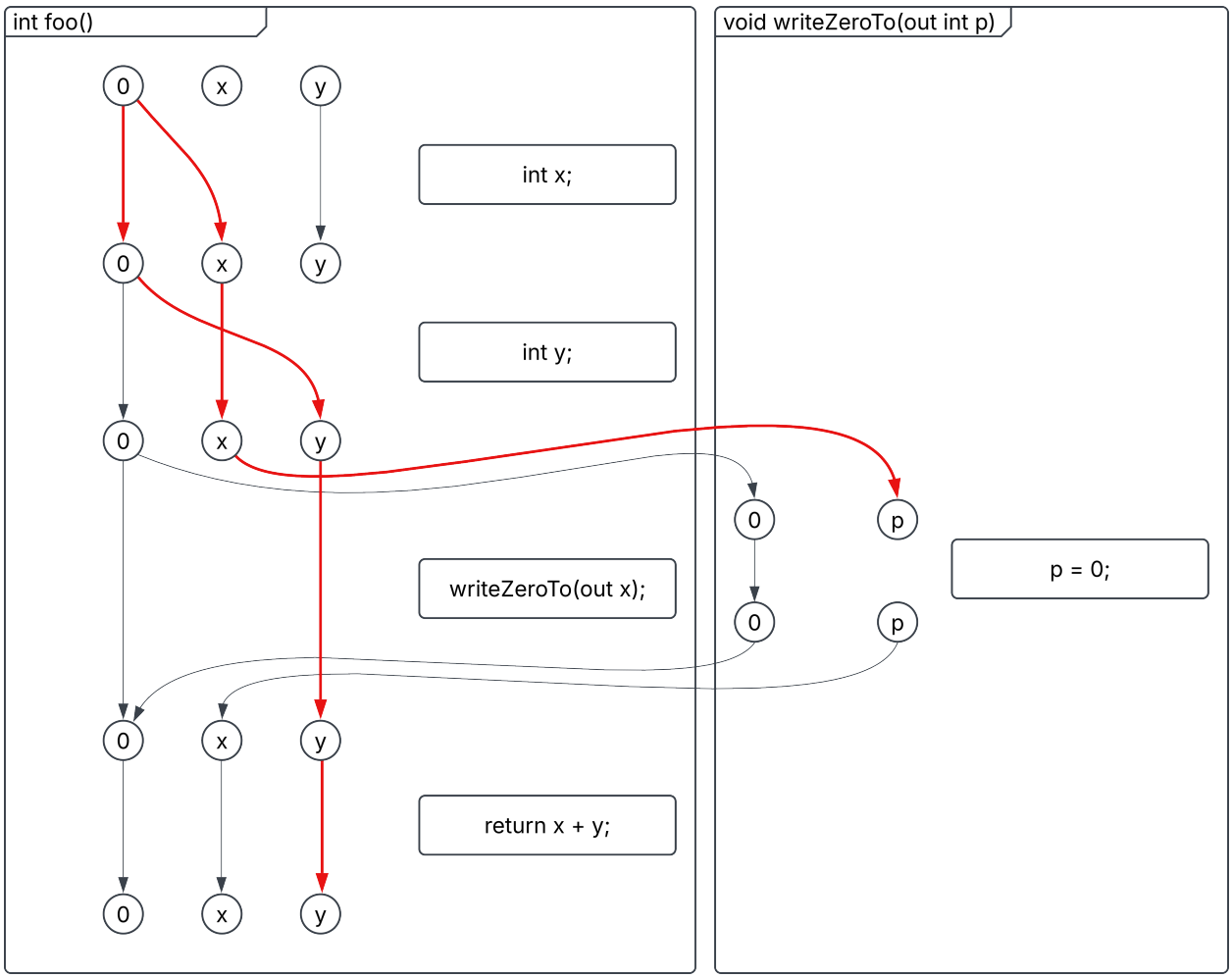


Рис. 2: Граф представленной программы

Вершинами данного графа являются факты для каждой инструкции рассматриваемой программы, а именно:

- нулевые факты,
- факты о локальных переменных,
- факты об аргументах.

Ребра данного графа определяются flow-функциями. Рассмотрим их подробнее для каждой инструкции.

1. `int x`.
    - Не влияет на достижимость последующих, поэтому в графе присутствует ребро `0` $\to$ `0`. Для последующих инструкций данное описание опущено для краткости изложения.

- Переменная объявлена без инициализации, поэтому в графе есть ребро `0` $\rightarrow$ `x`, означающее, что из достижимости инструкции следует неинициализированность переменной `x`.
- Не влияет на значение переменной `y`, поэтому в графе есть ребро `y` $\rightarrow$ `y` (если `y` была неинициализированна до инструкции, то останется неинициализированна после).

2. `int y` — аналогично предыдущей инструкции.
3. `call writeZeroTo(out x)`.
   - Не влияет на переменную `y`, поэтому `y` $\rightarrow$ `y`.
   - Переменная `x` передается в вызов в качестве аргумента, поэтому `x` $\rightarrow$ `p`. Ребра `x` $\rightarrow$ `x` нет, потому что передача происходит по ссылке, а значит в вызываемой функции значение `x` может измениться.
4. `p = 0`. Ребра `p` $\rightarrow$ `p` нет, потому что данная инструкция инициализирует переменную `p`.
5. `return from writeZeroTo(out x)`. Функция принимала параметр `p` по ссылке, а значит его изменения должны отразиться в месте вызова, поэтому `p` $\rightarrow$ `x`.
6. `return x + y` не влияет на значения локальных переменных, поэтому `x` $\rightarrow$ `x`, `y` $\rightarrow$ `y`.

Для решения задачи IFDS необходимо найти путь от начала метода (т. е. нулевого факта до первой инструкции) до его конца. На рис. 2 пути от нулевого факта выделены красным цветом. До конца метода доходит только путь переменной `y`, что означает что функция возвращает неинициализированные данные.

### 2.1.3 Анализ помеченных данных

*Анализ помеченных данных* (*анализ «загрязненных» данных*, англ. *Taint-analysis*) [11, 12] — это специализированный вид статическо-

го анализа, который используется для отслеживания распространения «загрязнённых» данных в программе. Таковыми данными считаются поступающие из ненадежных источников, таких как пользовательский ввод, файлы или сетевые запросы. Основная цель данного анализа — выявить, могут ли такие данные достичь опасных точек программы (SQL-запросы, системные команды и др.).

В ходе такого анализа необходимо найти в программе *истоки* (англ. *sources*) — места программы, в которых могут возникнуть потенциально небезопасные данные, а также *стоки* (англ. *sinks*) — места, попадание подобных данных в которые считается недопустимым. После чего остается отследить путь небезопасных данных от истоков к стокам, например, с помощью анализа потока данных.

## 2.2 Инструменты статического анализа для платформы .NET

В данной главе рассматриваются существующие инструменты для статического анализа, поддерживающие языки программирования платформы .NET.

### 2.2.1 Инструмент CodeQL

CodeQL [4] — статический анализатор, разработанный GitHub, который преобразует исходный код в реляционную базу данных для семантического анализа. Данный инструмент поддерживает множество языков программирования, а также мультиязыковой анализ. Кроме того, имеет интеграцию с GitHub Actions и Visual Studio Code. Анализ данным инструментом происходит с помощью выполнения запросов на специальном языке к базе данных исходного кода. CodeQL/, предоставляет набор запросов для различных видов анализа, в том числе для taint-анализа. Однако он также поддерживает и пользовательские запросы. В сравнении с другими рассмотренными инструментами CodeQL демонстрирует большую точность анализа, однако требует для этого больших ресурсов.

### 2.2.2 Инструмент Semgrep

Semgrep [17] — это статический анализатор с открытым исходным кодом, основанный на шаблонно-ориентированном анализе. Инструмент преобразует исходный код в синтаксическое дерево, а затем проверяет его на соответствие набору правил. Semgrep имеет реестр правил, поддерживает множество языков программирования, а также пользовательские правила. Ключевая особенность данного инструмента — использование методов машинного обучения для подтверждения результатов работы анализатора.

## 2.3 Проект USVM

USVM — проект, направленный разработку инструментов автоматического анализа программ. Изначально он разрабатывался как инструмент анализа, использующий в своей основе технику символьного исполнения. Однако позже были добавлены другие методы исследования: статический анализ, фаззинг. Анализаторы проекта USVM поддерживают несколько языков программирования: Java, Python, TypeScript. Такая универсальность достигается за счет использования специального представления кода программы, а также реализации независимых от языка программирования аспектов анализа на основе этого представления. Таким образом для поддержки какого-то вида анализа для нового языка необходимо:

1. построить промежуточное представление исходного кода данного языка;

2. реализовать зависящие от языка программирования аспекты этого анализа.

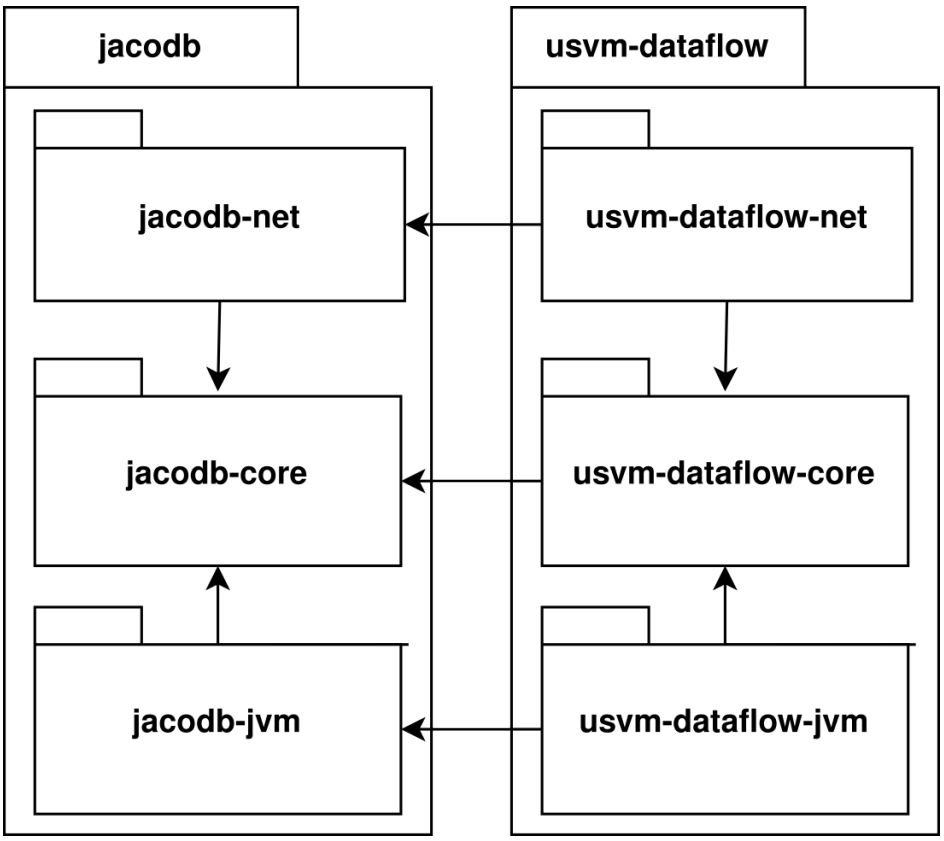


Рис. 3: Пакетная диаграмма статического анализа проекта USVM

На рис. 3 представлена пакетная диаграмма статического анализа в USVM. Она состоит из двух частей: `jacodb` реализует промежуточное представление кода программы, а `usvm-dataflow` — логику анализа потока данных. Рассмотрим их устройство подробнее.

- `jacodb-core` содержит абстрактное представление для кода любого языка программирования.
- `jacodb-net` и `jacodb-jvm` реализуют промежуточное представление кода программы для языков платформ, соответственно, .NET и JVM.
- `usvm-dataflow-core` реализует прямой taint-анализ с помощью модифицированного алгоритма IFDSна основе абстрактного промежуточного представления из `jacodb-core`. Данная реализация чувствительна к потоку и контексту вызова, а также содержит точки расширения, позволяющие внедрить реализации зависимых от языка программирования аспектов taint-анализа.

- `usvm-dataflow-net` и `usvm-dataflow-jvm` реализуют эти аспекты для языков платформ, соответственно, .NET и JVM.

В рамках данной работы был разработан `usvm-dataflow-net`, а также модифицирован `jacodb-net`.

# 3 Анализ потока данных программ на платформе .NET

В рамках данной работы были реализованы специфичные для языков программирования платформы .NET аспекты прямого taint-анализа на базе проекта USVM. Taint-анализ в данном проекте базируется на модифицированном алгоритме решения задач IFDS.

Рассмотрим устройство фактов, которыми оперирует разработанная система. Факт $\langle x.f, \tau \rangle$ состоит из базовой локации $x$, цепочки обращений $f$, и taint-марки $\tau$. Последняя может быть произвольной, потому что для анализа ее устройство не важно. Базовая локация может быть:

- объектом, метод которого исследуется;
- локальной переменной метода;
- аргументом метода;
- константой;
- статическим классом;
- результатом вызова метода.

Цепочка обращений логически представляет из себя последовательность различных доступов: к полю, к элементу (оператор $[*]$). Вместе базовая локация и цепочка обращений задают локацию taint-марки.

Важным аспектом разработанного анализатора являются flow-функции, которые, основываясь на семантике инструкции, манипулируют фактами. Разработанный анализ поддерживает следующие инструкции: присваивание, чтение из поля или элемента, запись в поле или элемент, вызов метода, возврат из метода. Таким образом flow-функция задается, как $[\![s]\!] : D \to 2^D$, где $D$ — множество фактов, а $s$ задается следующим

образом, а $x$, $y$ — локации.

$$
\begin{aligned}
s ::= {} & x \leftarrow y \\
\mid {} & x \leftarrow y.f \\
\mid {} & x.f \leftarrow y \\
\mid {} & x.m(y) \\
\mid {} & \text{return } x
\end{aligned}
$$

Flow-функции можно разделить на две группы: для последовательного исполнения и для межпроцедурного исполнения, — в зависимости от наличия вызова метода в них. Рассмотрим каждую группу отдельно.

$$
\begin{aligned}
[\![\text{return } x]\!](\langle z.g, \tau\rangle) &= \{\langle z.g, \tau\rangle, \langle \text{return}, \tau\rangle\}, \text{если } x = z \\
[\![x \leftarrow y]\!](\langle z.g, \tau\rangle) &= \begin{cases} \{\langle x.f, \tau\rangle, \langle y.f, \tau\rangle\}, \text{если } y = z \\ \emptyset, \text{если } x = z \\ \{\langle z.g, \tau\rangle\}, \text{иначе} \end{cases} \\
[\![x \leftarrow y.f]\!](\langle z.g, \tau\rangle) &= \begin{cases} \{\langle x.h, \tau\rangle, \langle y.g, \tau\rangle\}, \text{если } y = z, g = fh \\ \emptyset, \text{если } x = z \\ \{\langle z.g, \tau\rangle\}, \text{иначе} \end{cases} \\
[\![x.f \leftarrow y]\!](\langle z.g, \tau\rangle) &= \begin{cases} \{\langle x.fg, \tau\rangle, \langle y.g, \tau\rangle\}, \text{если } y = z \\ \emptyset, \text{если } x = z, g = fh \\ \{\langle z, \tau\rangle\}, \text{иначе} \end{cases}
\end{aligned}
$$

Рис. 4: Flow-функции для последовательного исполнения

На рис. 4 представлены flow-функции для последовательного исполнения. Возврат из метода (return $x$), применяемый к факту $\langle z.g, \tau\rangle$, помимо исходного факта, порождает новый факт с той же taint-маркой и базовой локацией «возвращаемое значение», если исходный факт был об $x$. Присваивание ($x \leftarrow y$), применяемое к факту $\langle z.g, \tau\rangle$, имеет три случая. Если исходный факт об $y$, то путь исходного факта можно продлить на $x$. Если исходный факт об $x$, то его путь прерывается, потому

что данные в $x$ будут перезаписаны после исполнения этой инструкции. Иначе же, инструкция не влияет на исходный факт. Следующие две инструкции наиболее интересны.

**Чтение из поля или элемента ($x \leftarrow y.f$).** Если исходный факт об $y$, то присваивание повлияет на $x$ только тогда, когда цепочка обращений факта начинается с доступов из цепочки обращений $f$. Если исходный факт об $x$, то его путь прерывается по аналогичной обычному присваиванию причине.

**Запись в поле или элемент ($x.f \leftarrow y$).** Если исходный факт об $y$, то его путь продлевается на $y$, а также и на $x$. Однако при этом доступы цепочки обращений факта добавляются в конец цепочки обращений $f$. Если исходный факт об $x$, то его путь прерывается тогда, когда цепочка обращений факта начинается с доступов цепочки $f$.

$$[\![x = source^{\tau}()]\!](\mathbf{0}) = \{\langle x, \tau\rangle\}$$

$$[\![x = c.m(y)]\!]_{call}(\langle z.g, \tau\rangle) = \begin{cases} \{\langle \text{this}.g, \tau\rangle\}, \text{если } c = z \\ \{\langle \text{arg}.g, \tau\rangle\}, \text{если } y = z \\ \{\langle \text{static}(A).g, \tau\rangle\}, \text{если } z = \text{static}(A) \\ \emptyset, \text{иначе} \end{cases}$$

$$[\![x = c.m(y)]\!]_{exit}(\langle z.g, \tau\rangle) = \begin{cases} \{\langle c.g, \tau\rangle\}, \text{если } z = \text{this} \\ \{\langle y.g, \tau\rangle\}, \text{если } z = \text{arg} \\ \{\langle x.g, \tau\rangle\}, \text{если } z = \text{return} \\ \{\langle \text{static}(A).g, \tau\rangle\}, \text{если } z = \text{static}(A) \\ \emptyset, \text{иначе} \end{cases}$$

Рис. 5: Flow-функции для межпроцедурного исполнения

На рис 5 представлены flow-функции для межпроцедурного исполнения. Первая из них отвечает за возникновение новых фактов из-за истоков.

**Вызов метода.** Если исходный факт об объекте, у которого вызывается метод, то путь факта продлевается на локацию «this». Если об аргументе, то на «arg». Пути фактов о статических классах продлеваются на себя же.

**Возврат из метода.** Если исходный факт о «this», то его путь продлевается на объект, метод которого был вызван. Если о «arg», то на соответствующий аргумент. Если «return», то на локацию, в которую происходит присваивание возвращаемого значения (если такое имеет место быть). Аналогично вызову метода пути фактов о статических классах продлеваются сами на себя.

# 4 Архитектура решения

Данная глава посвящена рассмотрению архитектуры разработанного решения. На рис. 6 представлена его компонентная диаграмма.

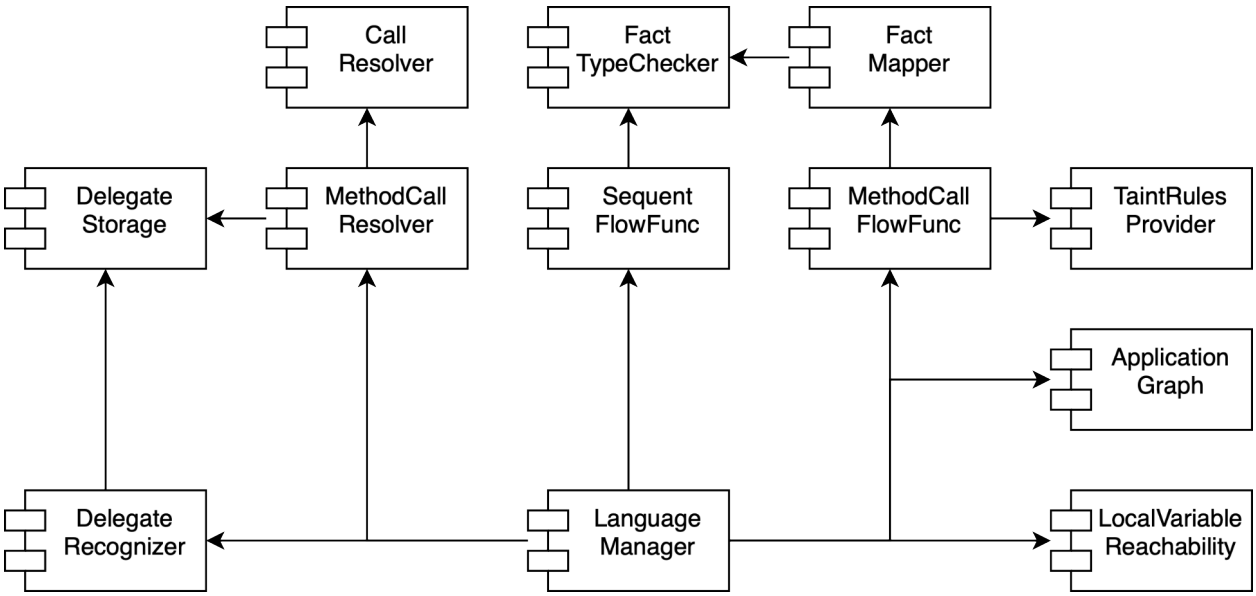


Рис. 6: Компонентная архитектура решения

Разработанное решение содержится в модуле `usvm-dataflow-net`. Рассмотрим его подробнее.

- `CallResolver` реализует логику нахождения подходящих реализаций виртуальных методов.
- `DelegateStorage` реализует хранение реализаций делегатов, а также поиск подходящего при вызове делегата.
- `DelegateRecognizer` позволяет распознавать инструкции создания делегатов в коде, находить оборачиваемый в делегат метод, а также сохранять его в `DelegateStorage`.
- `MethodCallResolver` содержит логику поиска подходящего метода при вызове. Он реализует функциональность определения типа вызова (виртуальный, делегат), а также использует `CallResolver` и `DelegateStorage` для поиска кода, который будет исполнен на самом деле.

- `FactTypeChecker` содержит логику проверки типов фактов.
- `FactMapper` реализует перемещение фактов с аргументов на параметры при вызове метода, а также с параметров на аргументы и с возвращаемого значения после вызова метода.
- `TaintRulesProvider` реализует представление конфигурации статического анализа.
- `SequentFlowFunc` реализует функции потока данных для инструкций, в которых нет вызова метода.
- `MethodCallFlowFunc` реализует функции потока данных для инструкций, в которых содержится вызов метода.
- `IlApplicationGraph` предоставляет доступ к графу потока управления исследуемого приложения.
- `LocalVariableReachability` содержит логику отбрасывания фактов о локальных переменных, если последние больше не используются в теле исследуемого метода.
- `IlLanguageManager` предоставляет интерфейс для использования разработанной системы в рамках независимой от языка программирования части анализа потока данных в проекте USVM.

Изначально анализ запускается на некотором наборе методов, указанном при конфигурации. Данные методы могут, например, представлять публичный API приложения. Далее в каждом из них анализатор начинает продлевать пути имеющихся фактов (изначально только нулевого), порождая новые факты с помощью flow-функций и конфигурации истоков, попутно отслеживая прохождение путей через стоки. Исследование заканчивается либо в связи с превышением ограничения по времени, либо, если на очередной итерации ни один путь не был продлен и ни один новый факт не был порожден. Рассмотрим подробнее продление пути конкретного факта на одну инструкцию.

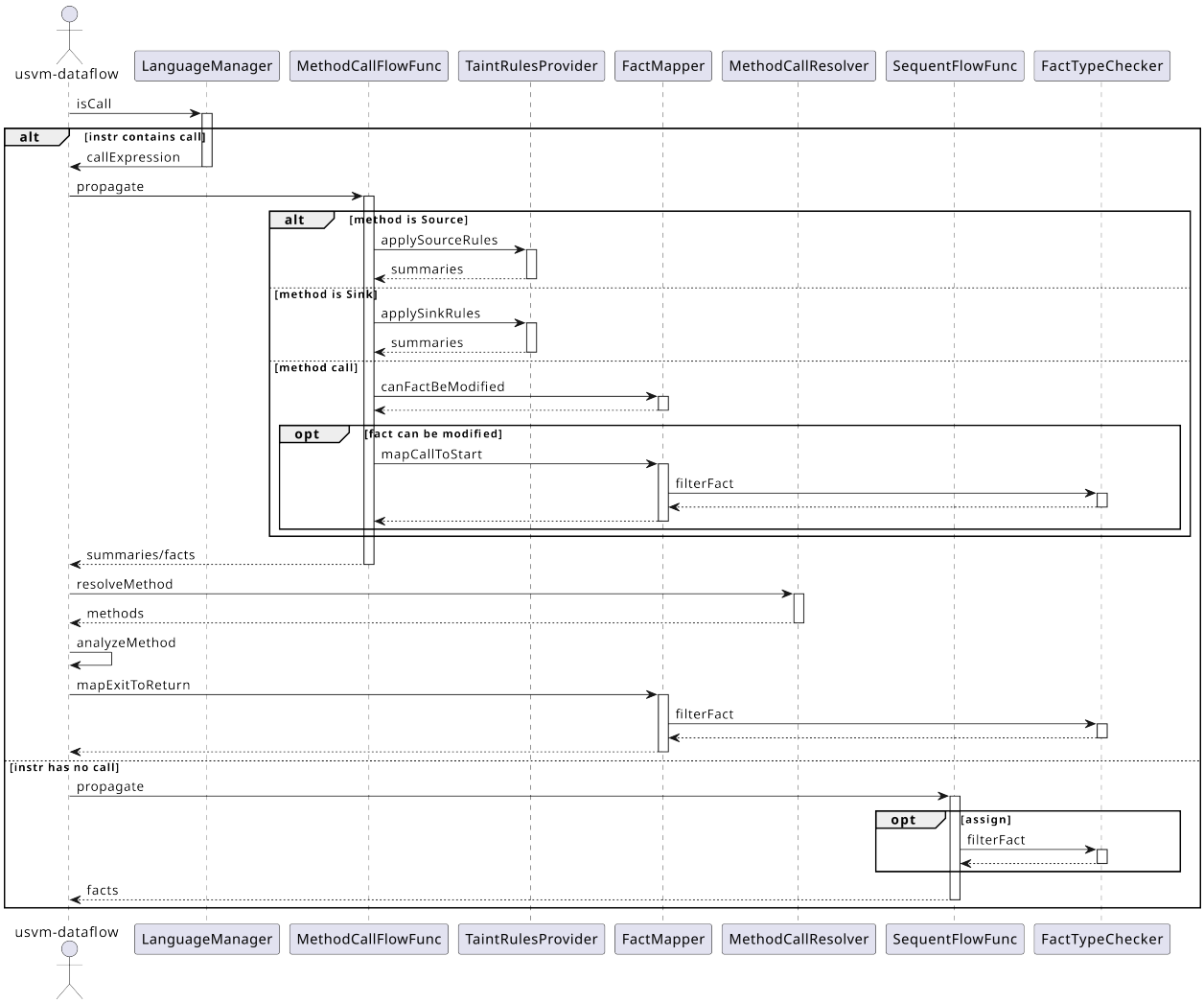


Рис. 7: Диаграмма взаимодействия `usvm-dataflow` и компонент системы

На рис. 7 изображено взаимодействие независящей от языка программирования компоненты анализа потока данных `usvm-dataflow` и разработанной системы на этапе исследования фиксированной инструкции и конкретного факта. Поведение системы можно разделить на два исключающих друг друга случая, в зависимости от наличия в рассматриваемой инструкции вызова метода. Рассмотрим случай, когда инструкция содержит его.

1. `usvm-dataflow` получает выражение, содержащее вызов метода.
2. `MethodCallFlowFunc` пытается продлить путь факта.
   (a) Применяются правила текущей конфигурации анализа пото-

ка данных. Если метод является стоком или истоком, то исследования метода не происходит, иначе исследование проводить нужно.

   (b) Проверяется, может ли текущей факт быть модифицирован в процессе исследования метода. Если нет, то путь такого факта не продлевается внутрь метода.

   (c) Путь факта продлевается на параметры метода, используя фильтр по типам факта и параметра.

3. Происходит подбор подходящих для данного вызова реализаций, которые затем анализируются.

4. После анализа реализации происходит продление путей фактов из параметров, а также из возвращаемого значения. Для этого тоже используется фильтры по типам.

В случае, когда инструкция не содержит вызов метода, проверяется, является ли она присваиванием. Если да, то факты из правого операнда продолжаются до левого с использованием фильтра по типам.

# 5 Детали реализации

В данной главе описаны особенности реализации разработанной системы в рамках проекта USVM. Реализация анализатора написана на языке Kotlin. Тестирование системы осуществлялось с использованием регрессионных тестов, что позволило убедиться в сохранении функциональности во время разработки. Инфраструктура для тестирования написана также на языке Kotlin, а сами тесты — на языке C#. Код реализации является приватным, в связи с чем не может быть приведен.

## 5.1 Передача аргументов по ссылке

Система типов .NET содержит *типы-значения* (англ. *value types*). При передаче переменных таких типов в качестве аргументов метода их содержимое копируется, что означает, что любые изменения соответствующих аргументов в теле метода никак не отражаются на них в месте вызова. Однако платформа .NET также позволяет передавать их по ссылке таким же образом, как и ссылочные типы. В данном случае тело метода может влиять на их значение после вызова.

С точки зрения используемого промежуточного представления кода при передаче по ссылке она должна быть предварительно помещена во временную локальную переменную, а затем созданная переменная передана в качестве аргумента. Однако при таком подходе факты, полученные в результате исследования метода, будут восприняты анализатором как факты о временной переменной, которая дальше в программе нигде не используется. Это связано с отсутствием в движке анализа псевдонимов из-за его большой вычислительной сложности. Чтобы реализовать рассматриваемую особенность .NET было решено нарушить правила построения промежуточного представления и совершать операцию взятия ссылки непосредственно в месте вызова. Таким образом, при сопоставлении фактов после анализа метода имеется возможность узнать, ссылка на какую переменную была передана в качестве аргумента и продлить путь факта на нее.

Важно заметить, что структуры являются типами-значениями и, в отличие от других подобных типов, могут иметь методы. Они могут изменять внутреннее состояние структуры, поэтому при их исследовании также необходимо учесть, что факты об этих изменениях должны распространяться за пределы метода. Это достигается за счет добавления в качестве первого аргумента ссылки на структуру, у которой он вызывается.

# 6 Экспериментальное исследование

Данная глава посвящена экспериментальному сравнению разработанной системы с инструментами CodeQL, Semgrep, а также анализу полученных результатов.

## 6.1 Описание исследования

Исследование проводилось на бенчмарке для статических анализаторов уязвимостей FlowBlot.NET. Код данного бенчмарка покрывает множество синтаксических конструкций языка C#, а также поделен на разделы, в зависимости от типов используемых истоков и стоков [7].

Для представления результатов работы инструментов, а также разметки уязвимостей в бенчмарке использовался формат Sarif. Данный формат позволяет указать участок кода, а также наличие или отсутствие уязвимостей на нем. Для классификации потенциальных уязвимостей использовались номера базы данных *Common Weakness Enumeration (CWE)*. Уязвимости в данной базе данных имеют иерархическую структуру [3], поэтому определенная инструментом потенциальная уязвимость признавалась верной, если определенный участок кода находится внутри размеченного, а определенная уязвимость является наследником размеченной.

На работу инструмента на одном разделе бенчмарка было установлено ограничение по времени в 30 секунд. Для сравнения результатов работы инструментов использовались следующие метрики: recall, false alarm (false positive rate), precision, f1 score.

Исследование проводилось на компьютере со следующими характеристиками.

- Процессор — Apple M1.
- Объем оперативной памяти — 16GB.
- Операционная система — macOS Sonoma 14.4.1.

## 6.2 Результаты исследования

В таблице 1 представлены результаты сравнения разработанной системы с CodeQL и Semgrep. В таблице используются обозначения, перечисленные ниже.

- Source — тип истока раздела, в соответствии с информацией из бенчмарка.
- Sink — тип стока раздела, в соответствии с информацией из бенчмарка.
- CWE — номер эксплуатируемой в разделе уязвимости.
- USVM— результаты разработанной системы.
- CQ— результаты инструмента CodeQL.
- SG— результаты инструмента Semgrep.

| Source | Sink | cwe | Recall | | | False alarm | | | Precision | | | F1 score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | USVM | CQ | SG | USVM | CQ | SG | USVM | CQ | SG | USVM | CQ | SG |
| Environment | ExternalCtrl | 15 | 0,06 | 0,49 | 0,0 | 0,05 | 0,5 | 0,0 | 0,78 | 0,76 | 0,0 | 0,1 | 0,6 | 0,0 |
| SqlDatabase | SqlInjection | 89 | 0,14 | 0,0 | 0,5 | 0,1 | 0,0 | 0,5 | 0,81 | 0,0 | 0,77 | 0,23 | 0,0 | 0,6 |
| NetClient | CmdInjection | 78 | 0,43 | 0,0 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 0,0 | 0,0 | 0,59 | 0,0 | 0,0 |
| SqlDatabase | ExternalCtrl | 15 | 0,18 | 0,49 | 0,0 | 0,08 | 0,5 | 0,0 | 0,89 | 0,76 | 0,0 | 0,31 | 0,6 | 0,0 |
| NetClient | ShorthandCmd | 426 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| File | CmdInjection | 78 | 0,43 | 0,0 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 0,0 | 0,0 | 0,59 | 0,0 | 0,0 |
| File | ShorthandCmd | 426 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| NetClient | SqlInjection | 89 | 0,18 | 0,0 | 0,5 | 0,1 | 0,0 | 0,5 | 0,85 | 0,0 | 0,77 | 0,3 | 0,0 | 0,6 |
| TcpConnection | ExternalCtrl | 15 | 0,07 | 0,49 | 0,0 | 0,08 | 0,5 | 0,0 | 0,75 | 0,76 | 0,0 | 0,13 | 0,6 | 0,0 |
| File | SqlInjection | 89 | 0,14 | 0,0 | 0,5 | 0,1 | 0,0 | 0,5 | 0,81 | 0,0 | 0,77 | 0,23 | 0,0 | 0,6 |
| File | ExternalCtrl | 15 | 0,18 | 0,49 | 0,0 | 0,08 | 0,5 | 0,0 | 0,89 | 0,76 | 0,0 | 0,31 | 0,6 | 0,0 |
| TcpConnection | ShorthandCmd | 426 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| Environment | ShorthandCmd | 426 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| SqlDatabase | CmdInjection | 78 | 0,18 | 0,0 | 0,0 | 0,1 | 0,0 | 0,0 | 0,85 | 0,0 | 0,0 | 0,3 | 0,0 | 0,0 |
| SqlDatabase | ShorthandCmd | 426 | 0,18 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,85 | 1,0 | 0,0 | 0,3 | 0,06 | 0,0 |
| NetClient | ExternalCtrl | 15 | 0,19 | 0,49 | 0,0 | 0,1 | 0,5 | 0,0 | 0,86 | 0,76 | 0,0 | 0,32 | 0,6 | 0,0 |
| Environment | PathTraversal | 23 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| ReadLine | ExternalCtrl | 15 | 0,06 | 0,49 | 0,0 | 0,08 | 0,5 | 0,0 | 0,7 | 0,76 | 0,0 | 0,1 | 0,6 | 0,0 |
| NetClient | PathTraversal | 23 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| SqlDatabase | PathTraversal | 23 | 0,07 | 0,03 | 0,0 | 0,08 | 0,0 | 0,0 | 0,73 | 1,0 | 0,0 | 0,12 | 0,06 | 0,0 |
| ReadLine | ShorthandCmd | 426 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| TcpConnection | CmdInjection | 78 | 0,43 | 0,35 | 0,0 | 0,1 | 0,13 | 0,0 | 0,93 | 0,9 | 0,0 | 0,59 | 0,5 | 0,0 |
| ReadLine | SqlInjection | 89 | 0,14 | 0,0 | 0,5 | 0,1 | 0,0 | 0,5 | 0,81 | 0,0 | 0,77 | 0,23 | 0,0 | 0,6 |
| ReadLine | PathTraversal | 23 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |
| Environment | SqlInjection | 89 | 0,14 | 0,0 | 0,5 | 0,1 | 0,0 | 0,5 | 0,81 | 0,0 | 0,77 | 0,23 | 0,0 | 0,6 |
| Environment | CmdInjection | 78 | 0,43 | 0,0 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 0,0 | 0,0 | 0,59 | 0,0 | 0,0 |
| TcpConnection | SqlInjection | 89 | 0,18 | 0,35 | 0,5 | 0,1 | 0,13 | 0,5 | 0,85 | 0,9 | 0,77 | 0,3 | 0,5 | 0,6 |
| ReadLine | CmdInjection | 78 | 0,43 | 0,0 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 0,0 | 0,0 | 0,59 | 0,0 | 0,0 |
| TcpConnection | PathTraversal | 23 | 0,43 | 0,3 | 0,0 | 0,1 | 0,1 | 0,0 | 0,93 | 0,9 | 0,0 | 0,59 | 0,45 | 0,0 |
| File | PathTraversal | 23 | 0,43 | 0,03 | 0,0 | 0,1 | 0,0 | 0,0 | 0,93 | 1,0 | 0,0 | 0,59 | 0,06 | 0,0 |

Таблица 1: Результаты исследования

Результаты проведенного исследования показывают, что разработанная система на большей части разделов превосходит CodeQL и

Semgrep. Semgrep нашел больше уязвимостей на разделах с SQL-инъекциями (CWE 89), однако на них он имеет достаточно большой false alarm. Аналогичная ситуация и с CodeQL: на разделах с типом стока «ExternalCtrl» было найдено больше уязвимостей, чем с помощью разработанной системы, однако false alarm гораздо выше, что снижает практическую значимость данных инструментов.

# Заключение

В ходе работы были получены следующие результаты.

- Проведен обзор анализа потока данных, включающий в себя анализ помеченных данных и задачи IFDS, а также обзор инструментов статического анализа для платформы .NET: CodeQL, Semgrep.
- Разработана архитектура системы статического анализа, учитывающая такие особенности платформы .NET, как структуры, передача аргументов value-типа по ссылке, виртуальные вызовы, делегаты.
- Спроектированная система была реализована в инфраструктуре проекта USVM.
- Проведено сравнительное исследование реализованной системы с CodeQL и Semgrep, показавшее, что разработанная система демонстрирует более высокие метрики качества.

# Список литературы


[1] Bodden Eric. Inter-procedural data-flow analysis with ifds/ide and soot // Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. — 2012. — P. 3–8.

[2] C# language specification. — 2024. — URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/readme (дата обращения: 2025-01-10).

[3] CWE Website. — 2025. — URL: https://cwe.mitre.org (дата обращения: 2025-02-1).

[4] CodeQL documentation. — 2025. — URL: https://codeql.github.com/docs (дата обращения: 2025-01-10).

[5] Ecma TC39. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005.

[6] Emanuelsson Pär, Nilsson Ulf. A comparative study of industrial static analysis tools // Electronic notes in theoretical computer science. — 2008. — Vol. 217. — P. 5–21.

[7] FlowBlot.NET GitHub repository. — 2025. — URL: https://github.com/CodeThreat/FlowBlot.NET (дата обращения: 2025-04-25).

[8] Fosdick Lloyd D, Osterweil Leon J. Data flow analysis in software reliability // ACM Computing Surveys (CSUR). — 1976. — Vol. 8, no. 3. — P. 305–330.

[9] Gosain Anjana, Sharma Ganga. Static analysis: A survey of techniques and tools // Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014 / Springer. — 2015. — P. 581–591.

[10] Khedker Uday, Sanyal Amitabha, Sathe Bageshri. Data flow analysis: theory and practice. — CRC Press, 2017.

[11] Marashdih Abdalla Wasef, Zaaba Zarul Fitri, Suwais Khaled. An enhanced static taint analysis approach to detect input validation vulnerability // Journal of King Saud University-Computer and Information Sciences. — 2023. — Vol. 35, no. 2. — P. 682–701.

[12] Maskur Achmad Fahrurrozi, Asnar Yudistira Dwi Wardhana. Static code analysis tools with the taint analysis method for detecting web application vulnerability // 2019 International Conference on Data and Software Engineering (ICoDSE) / IEEE. — 2019. — P. 1–6.

[13] Naeem Nomair A, Lhoták Ondřej, Rodriguez Jonathan. Practical extensions to the IFDS algorithm // International Conference on Compiler Construction / Springer. — 2010. — P. 124–144.

[14] Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis / Dongjie He, Haofeng Li, Lei Wang et al. // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) / IEEE. — 2019. — P. 267–279.

[15] Reps Thomas, Horwitz Susan, Sagiv Mooly. Precise interprocedural dataflow analysis via graph reachability // Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1995. — P. 49–61.

[16] Security testing: A survey / Michael Felderer, Matthias Büchler, Martin Johns et al. // Advances in Computers. — Elsevier, 2016. — Vol. 101. — P. 1–51.

[17] Semgrep documentation. — 2025. — URL: https://semgrep.dev/docs/semgrep-code/overview (дата обращения: 2025-01-10).

[18] USVM GitHub repository. — 2025. — URL: https://github.com/UnitTestBot/usvm (дата обращения: 2025-01-15).