DOI: 10.15514/ISPRAS-2025-37(6)-5

Повышение точности статического анализа кода при помощи больших языковых моделей

Д.Д. Панов, ORCID: 0009-0003-5809-1187 <d.panov@ispras.ru>
Н.В. Шимчик, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
Д.А. Чибисов, ORCID: 0009-0002-1198-3148 <dchibisov@ispras.ru>
А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@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.

Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация. В данной работе описывается подход к проверке результатов статического анализа кода при помощи больших языковых моделей (LLM), выполняющий фильтрацию предупреждений с целью удаления ложных. Для составления запроса к LLM предложенный подход сохраняет информацию, собранную анализатором, такую как абстрактное синтаксическое дерево программы, таблицы символов, резюме типов и функций. Эта информация может как напрямую передаваться в запросе к модели, так и использоваться для более точного определения фрагментов кода, необходимых для проверки истинности предупреждения. Подход был реализован в SharpChecker – промышленном статическом анализаторе для языка C#. Его тестирование на реальном коде показало повышение точности результатов на величину до 10 процентных пунктов при сохранении высокой полноты (от 0,8 до 0,97) для чувствительных к контексту и путям межпроцедурных детекторов утечки ресурсов, разыменования null и целочисленного переполнения. Для детектора недостижимого кода применение информации из статического анализатора позволило повысить полноту на 11–27 процентных пунктов по сравнению с подходом, использующим в запросе только исходный код программы.

Ключевые слова: статический анализ кода; большие языковые модели LLM.

Для цитирования: Панов Д.Д., Шимчик Н.В., Чибисов Д.А., Белеванцев А.А., Игнатьев В.Н. Повышение точности статического анализа кода при помощи больших языковых моделей. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 83–100. DOI: 10.15514/ISPRAS–2025–37(6)–5.

Increasing Precision of Static Code Analysis Using Large Language Models

D.D. Panov, ORCID: 0009-0003-5809-1187 <d.panov@ispras.ru>
N.V. Shimchik, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
D.A. Chibisov, ORCID: 0009-0002-1198-3148 <dchibisov@ispras.ru>
A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@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.

Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. This paper describes an approach to verifying the results of static code analysis using large language models (LLMs), which filters warnings to eliminate false positives. To construct the prompt for LLM, the proposed approach retains information collected by the analyzer, such as abstract syntax trees of files, symbol tables, type and function summaries. This information can either be directly included in the prompt or used to accurately identify the code fragments required to verify the warning. The approach was implemented in SharpChecker – an industrial static analyzer for the C# language. Testing on real-world code demonstrated an improvement in result precision by up to 10 percentage points while maintaining high recall (0.8 to 0.97) for context-sensitive and interprocedural path-sensitive detectors of resource leaks, null dereferences, and integer overflows. In case of unreachable code detector, use of information from the static analyzer improved recall by 11–27 percentage points compared to an approach that only uses the program's source code in the prompt.

Keywords: static code analysis; large language models LLM.

For citation: Panov D.D., Shimchik N.V., Chibisov D.A., Belevantsev A.A., Ignatyev V.N. Increasing precision of static code analysis using large language models. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 83-100 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-5.

1. Введение

Статический анализ исходного кода является важным подходом к поиску ошибок и уязвимостей и позволяет искать ошибки в коде программы без ее запуска. Одной из проблем, возникающих в ходе применения статических анализаторов, является большое количество предупреждений, выдаваемых на больших проектах. Средняя плотность предупреждений промышленного статического анализатора составляет 10 сообщений на 1000 строк кода, что даже при высокой точности, достигающей 90% истинных предупреждений, приводит к сотням ложных срабатываний. Более сложные детекторы ошибок, как например, поиск утечек ресурсов, разыменований null могут выдавать до 40% ложных предупреждений. На их анализ тратятся ресурсы квалифицированных разработчиков. Поэтому актуальна задача автоматизации разметки или фильтрации ложных предупреждений. Ее основная сложность состоит в исключении только ложных предупреждений без потери истинных.

В современном промышленном анализаторе очень сложно добиться повышения точности без снижения полноты только за счет усовершенствования имеющихся алгоритмов и моделей программы. Поэтому для фильтрации ложных предупреждений можно использовать *верификацию* – их дополнительную проверку другими алгоритмами. Например, предупреждения нечувствительного к путям анализа помеченных данных можно проверять динамическим анализом или символьным выполнением [1], результаты чувствительного к путям анализа – методами машинного обучения [2].

В настоящее время одними из самых совершенных методов на основе машинного обучения является использование больших языковых моделей (Large Language Model, LLM). Тривиальным подходом их применения к задаче верификации является запрос к LLM на

основе предупреждения анализатора и небольшой окрестности кода (десятки строк) вокруг. Это дает неплохие результаты, но только для тех случаев, когда весь код, важный для оценки истинности предупреждения, попадает в запрос [3]. Это условие на реальных проектах выполняется в основном для детекторов, анализирующих абстрактное синтаксическое дерево (АСД), а для межпроцедурных случаев или ситуаций, требующих изучения нескольких фрагментов кода, не дает существенных результатов, поскольку его истинность зависит от контекста, находящегося за пределами текущей функции или даже файла. Поскольку размер контекстного окна в больших языковых моделях ограничен, а также качество ответа модели ухудшается с ростом размера запроса, важной задачей является поиск релевантной для понимания текущего предупреждения информации.

Применение больших языковых моделей для верификации в промышленном статическом анализаторе сопряжено с дополнительными ограничениями. Так, для защиты пользовательского кода, часто составляющего коммерческую тайну, от доступа третьих лиц исключена возможность использования больших моделей, предоставляемых в виде сервиса через интернет, как например, GPT-4 [4]. А для локального развертывания доступны лишь модели ограниченного размера с открытыми весами. Однако преимуществом использования таких моделей является возможность их дообучения на корпусе размеченных предупреждений анализатора. Кроме того, подходы, демонстрирующие высокие показатели на синтетических тестах с ошибками, никогда не достигают аналогичных показателей на реальном коде.

В данной работе предлагается метод верификации предупреждений промышленного статического анализатора Svace на основе специально собранной в процессе анализа информации. Собранная информация включает резюме методов и типов, содержащие их основные, важные для понимания ошибки свойства; АСД для извлечения минимальных осмысленных блоков кода, связанных с каждой точкой трассы предупреждения и другие данные. Кроме этого, для повышения точности и полноты фильтрации применяется дообучение на наборе из сотен истинных и ложных предупреждений на реальных проектах, собранных и размеченных в процессе разработки анализатора. Предложенный метод реализован для языка C# в анализаторе SharpChecker [5] (является частью Svace [6]). Реализация подхода для сложных детекторов ошибок разыменования null, утечки ресурсов и целочисленного переполнения, найденных чувствительным к путям межпроцедурным анализом в коде на языке C#, повышает их совокупную точность на 11 процентных пунктов до 86% при полноте 91% на реальных проектах. Для детектора недостижимого кода полнота фильтрации предупреждений не достигла 70%, но был заметен её прирост благодаря использованию информации, собранной статическим анализатором.

Структура данной статьи следующая. О существующих подходах к верификации результатов статического анализатора с помощью методов машинного обучения рассказывается в разделе 2. Общая схема предлагаемого в данной работе метода описывается в разделе 3. Сбор информации из статического анализатора описывается в разделе 4. Генерация запросов и интерпретация ответов модели описывается в разделе 5. Результаты тестирования полученного решения при анализе проектов с открытым исходным кодом проводится в разделе 6.

2. Обзор существующих решений

Существуют различные подходы к подтверждению результатов статического анализа, начиная с воспроизведения трассы предупреждения при помощи динамического анализа или фаззинга, заканчивая подходами на основе машинного обучения, которые предназначены для оценки вероятности истинности предупреждения.

Остановимся подробнее на подходах, использующих большие языковые модели.

В работе Li H. et al. [7] был представлен инструмент LLift, объединяющий статический анализатор UBITech и большие языковые модели. Схема работы инструмента состояла из трёх этапов:

1) нечувствительное к путям символьное выполнение, позволяющее определить множество возможных ошибок;
2) чувствительный к путям анализ, после которого часть ошибок подтверждается или отсеивается, а часть остаётся неклассифицированными;
3) для неклассифицированных потенциальных ошибок создаётся запрос к большой языковой модели (в качестве примера – GPT-4), включающий в себя собранную информацию об инициализаторах всех имеющих отношение к предупреждению переменных.

Данный подход смог продемонстрировать 50% точность при анализе ядра Linux без потерянных истинных срабатываний и показал пользу от использования вычисляемой анализатором информации в запросах к моделям.

В работе Mohajer M. et al. [8] описывается подход к использованию больших языковых моделей для проверки предупреждений типов «разыменование нулевого указателя» и «утечка ресурсов». Авторам удалось достичь повышения точности результатов статического анализатора Infer с 65% до 94% и с 54% до 63% соответственно, однако продемонстрированная полнота оказалась заметно ниже – 64% и 80% соответственно.

В работе Wen C. et al. [9] описывается инструмент LLM4SA, также решающий задачу проверки истинности предупреждений с помощью больших языковых моделей. Авторы предложили решать проблему извлечения релевантного контекста из исходного кода при помощи графа зависимостей программы, отображающего зависимости по данным и по управлению между инструкциями. Демонстрируемый модели фрагмент кода включает в себя не только ближайшую окрестность, но и те части исходного кода, от которых может зависеть код, в котором было выдано предупреждение. Также для повышения стабильности результатов авторы выполняли больше одного запроса к модели и усредняли результаты классификации.

Если для какого-то предупреждения не было заметного перевеса в сторону «истинности» или «ложности», оно помечалось как «неопределённое». Помимо GPT-3.5-Turbo, авторы продемонстрировали часть результатов с использованием языковой модели с открытыми весами Llama-2-70B и сделали вывод о том, что та смогла продемонстрировать сравнимые результаты. При тестировании на реальных проектах инструмент показал полноту в 93%, однако общую точность около 5%, что в первую очередь вызвано крайне низкой точностью выбранных авторами статических анализаторов.

В работе Li Z. et al. [10] описывается инструмент IRIS, в котором большие языковые модели используются для улучшения результатов статического анализа помеченных данных, проводимого инструментом CodeQL. Использование языковых моделей для пополнения списка используемых инструментом истоков, стоков и пропагаторов помеченных данных позволило более чем в два раза повысить количество обнаруженных уязвимостей. Для повышения точности они использовали запросы, в которых предоставляли модели контекст найденных предупреждений и просили дать текстовый комментарий для демонстрации пользователю, а также оценить, является ли предупреждение истинным. Использование GPT-4 позволило поднять нижнюю оценку точности с 10% до 15%, в то время как для других использованных в работе моделей повышение полноты анализа сопровождалось падением его точности. После ручной проверки выборки из 50 предупреждений авторы получили оценку точности, равную 54% при использовании GPT-4.

В работе Khare A. et al. [11] проводится общая оценка применимости 16 больших языковых моделей к поиску уязвимостей на наборе из 5000 тестов, частично состоящих из

искусственных примеров и частично основанных на коде реальных проектов. Хотя эта работа напрямую не связана с верификацией результатов статического анализа, авторы делают ряд полезных выводов, в частности: отдельные сравнительно небольшие модели (14 и 32 миллиарда параметров) смогли показать лучшие результаты на реальных проектах, чем значительно бóльшие модели, включая GPT-4. Также авторы отметили некоторые типы уязвимостей, для которых языковые модели показали лучший результат, чем статический анализатор CodeQL – в основном это предупреждения, для оценки которых не требуется знание глобального контекста или понимание сложных структур данных.

Таким образом, хотя использование больших языковых моделей для фильтрации предупреждений статического анализатора активно исследуется, среди рассмотренных нами работ не нашлось таких, которые бы продемонстрировали одновременно высокую точность и высокую полноту результатов. Кроме того, эти работы в первую очередь ориентировались на проприетарные языковые модели, которые отличаются от открытых большим количеством весов, но невозможностью их локального запуска на собственном оборудовании, необходимостью передачи демонстрируемого в запросе исходного кода третьим лицам, а также ограниченными возможностями по дообучению модели под свои нужды.

3. Схема разработанного метода

При ответе на запрос о верификации предупреждения LLM генерирует результат как на основе информации, представленной в запросе, так и использует закодированные в модели данные, например, информацию о популярных библиотечных функциях или даже исходный код анализируемого открытого проекта, если он был использован при обучении модели. Метод, предложенный в работе, основан на предположении, что любая информация, присутствующая в коде или доступная анализатору, которая может влиять на истинность предупреждения, должна быть включена в запрос к модели. В противном случае истинность предупреждения невозможно установить корректно даже аналитику. Таким образом, для каждого предупреждения требуется выделить необходимую для его проверки информацию:

1) фрагменты кода, влияющие на истинность срабатывания, например условия ветвлений;
2) функции, переменные, типы и их свойства, например, инициализация неизменяемых (readonly) переменных или аллокацию ресурсов.

Тривиальное вырезание фрагментов кода из нескольких строк до и после вставляет в запрос обрывки операторов, которые могут усложнить его понимание, поэтому требуется выделение структурных блоков. А для получения свойств функций, например, при каких условиях она может выбросить исключение, требуется нетривиальный анализ. К счастью, результаты такого анализа доступны во время работы статического анализатора и необходимо лишь их упрощение и экспорт в понятном для модели формате. Например, анализатор задает условия в терминах идентификаторов символьных значений, а модель сможет их воспринять лишь в терминах переменных, определенных в коде.

Во многих статических анализаторах предупреждение описывается не одной точкой в программе и текстовым описанием ошибки, а целой последовательностью точек, помогающих человеку лучше понять детали ошибки. Эта последовательность пар «точка в программе, текстовое описание» называется *трассой предупреждения*. Например, для ошибки «разыменование null» анализатор может показать условия, при которых переменная может стать null, а также путь выполнения, достигающий точки разыменования. Таким образом, наличие трассы упрощает поиск релевантных фрагментов кода. Однако для выявленных локаций в коде остается задача выделения фрагмента кода и встраивания сообщений трассы в запрос. Например, их можно вставлять в код в виде комментария в конце строки или строкой выше, или перечислять отдельно с указанием номеров строк.

Включение в запрос исходного кода всех связанных функций невозможно, т.к. они в свою очередь могут зависеть от других функций и типов. Поэтому необходимо исследование различных способов добавления их резюме в запрос.

После выполнения запросов каждый ответ модели должен быть классифицирован как истинное или ложное предупреждение, после чего в итоговом отчёте каждому исходному предупреждению присваивается соответствующий статус, а также опционально – указывается текст ответа модели. Интерпретация ответов модели также представляет исследовательский интерес, поскольку модель генерирует ответы в виде текста в свободной форме, а их классификация должна выполняться программно без участия человека – подробнее о способах решения этой проблемы рассказывается в разделе 5.

Таким образом, метод верификации состоит из трех рассмотренных далее этапов, представленных на рисунке 1: экспорт информации из анализатора; построение набора запросов к модели на основе предупреждений и сохраненной информации и интерпретация ответов модели. Построенная схема также используется и для дообучения на размеченных предупреждениях.

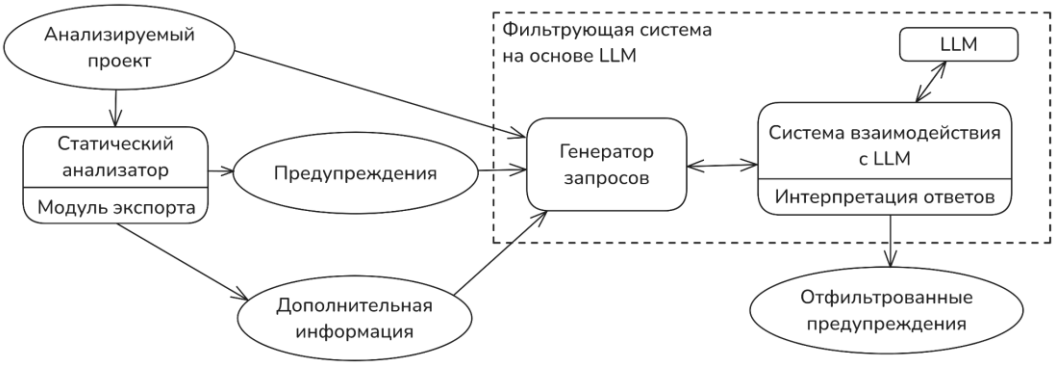

Рис. 1. Общая архитектура решения.
Fig. 1. General structure of the solution.

4. Сбор и сохранение информации о программе

В данном разделе рассматриваются структуры данных, которые получены из статического анализатора и содержат информацию для понимания контекста предупреждения. Для ее извлечения в анализаторе SharpChecker был реализован отдельный модуль, сохраняющий всю необходимую информацию в виде файлов в формате JSON, которые будут далее использованы при генерации запросов к LLM.

Предупреждение анализатора включает основную точку в коде, сообщение, *трассу*, состоящую из пар [точка в коде, сообщение], а также некоторую служебную информацию, как например полное имя метода, содержащего предупреждение. В анализаторе SharpChecker для хранения предупреждений используется XML-подобный формат Svace, подходящий для дальнейшего использования, поэтому ничего дополнительно сохранять не требуется.

Абстрактное синтаксическое дерево (АСД) применяется в компиляторах на этапе синтаксического анализа для описания иерархической структуры кода. Для получения АСД не обязательно встраиваться в анализатор, а возможно использование сторонних, мульти-языковых инструментов, как например, tree-sitter [12]. Однако они не всегда способны дать точный результат из-за возможных неточностей в грамматиках, отсутствия поддержки новых стандартов языка или отсутствия информации о содержимом подключаемых заголовочных файлов. Экспорт АСД непосредственно из компилятора или анализатора позволяет учитывать все особенности сборки: макроопределения, заголовочные и генерируемые файлы и т.д.

Существуют подходы [13] на основе машинного обучения, в которых АСД являются частью входной информации модели, однако большие языковые модели предназначены для работы с текстовой информацией, потому в данном подходе синтаксические деревья играют вспомогательную роль, предоставляя информацию о границах функций и отдельных блоков кода, для выделения окрестности кода – например, для многострочных операторов.

Статический анализатор SharpChecker основан на компиляторной инфраструктуре Roslyn [14] и использует ее представление АСД. Поскольку в настоящее время АСД используется только для выделения осмысленного законченного фрагмента кода для произвольной точки в программе, сохраняется лишь необходимая его часть: вершины объявлений пространств имён, типов, методов, полей, свойств, всех видов функций Перечисленные вершины, начиная с корня дерева, преобразуются в JSON формат с атрибутами, задающими участок кода, соответствующий вершине; тип вершины; имя объявляемой сущности и множество непосредственных потомков.

Таблица символов позволяет сопоставить идентификаторам в текущей области видимости соответствующие им объявления переменных, типов, функций. В анализаторе Svace таблица символов доступна как из компилятора (Roslyn для C#), так и из собранной для навигации по коду информации базы данных в формате DXR [15]. Преимущество DXR состоит в единообразном представлении для всех поддерживаемых анализатором языков программирования. Помимо поиска объявлений, в ней содержится информация из графа наследования, а также необходимые структуры для поиска всех использований символа и переопределений виртуальных методов.

Резюме типа содержит важные для анализа свойства типа данных. Примерами таких свойств являются имя, разновидность (примитивный тип, структура, класс, интерфейс и т.д.), родители, наследники, список полей класса, а также различные атрибуты типа.

Резюме типов можно подставлять в текст запроса вместо определений в форме кода. Они имеют унифицированный формат и обычно являются более компактными, чем определения соответствующих типов в исходном коде. Исходные данные для экспорта вычисляются анализатором SharpChecker и включают как общую информацию, например, является ли тип абстрактным, так и специальную, например, о том, что тип является ресурсом (реализует интерфейс IDisposable).

Резюме функций в статических анализаторах обычно используется для обеспечения масштабируемости анализа за счёт предварительного однократного вычисления ключевых свойств функции и их применения в точке вызова. Содержимое резюме функции зависит от поддерживаемых анализатором типов ошибок. Например, для детектора «разыменование null» существенной информацией является то, может ли функция вернуть значение null, а также то, при каких условиях в ней происходят разыменования аргументов.

Резюме экспортируется после анализа каждой функции и содержит имя и сигнатуру функции; ее тип (обычный метод, конструктор, аксессор, делегат и т. д.); возвращаемое значение; флаг, определяющий, может ли функция вернуть null; список измененных полей вместе с их новыми значениями; список выбрасываемых исключений с соответствующим условием. Помимо вышеперечисленных атрибутов, каждый детектор может помещать в резюме специализированные свойства функции, необходимые для поиска конкретного типа ошибок. К примеру, детектор ошибок типа разыменование значения null записывает в резюме массив условий возможного разыменования значений в данной функции; детектор ошибок типа утечка ресурсов записывает в резюме информацию о том, была ли вызвана функция закрытия ресурса, возвращает ли функция новый ресурс или уже существующий, какие ресурсы закрываются в функции и какие ресурсы сохраняются в объекте.

Использование анализатора в качестве источника данных позволяет сохранить в резюме предикаты (условия) различных событий, таких как выброс исключения, освобождение ресурса. Эти предикаты строятся при статическом символьном выполнении и представляют

собой формулы и выражения над символьными переменными. Для включения в запрос требуется их трансляция в формулу над переменными, объявленными в коде программы. Кроме того, полученные в результате символьного выполнения предикаты могут быть сложными формулами, которые все равно не будут корректно интерпретированы современными LLM, поэтому такие выражения либо заменяются на неизвестное условие, либо упрощаются. Пример трансляции внутреннего представления значений в читаемый вид приведен на рис. 2.

```
{
    "variable": "m3735",
    "condition": "[!pure3729 && !pure3732]"
}
{
    "variable": "_salt",
    "condition": "[!(string.IsNullOrEmpty(plainText)) and !(string.
       IsNullOrEmpty(sharedSecret))]"
}
```

Рис. 2. Пример преобразования символьных выражений в формулу над переменными, объявленными в коде.

Fig. 2. An example of transforming symbolic expressions into a formula over variables declared in the code.

При анализе в SharpChecker переменной _salt было сопоставлено символьное значение m3735, а условие ее разыменования построено в виде логической формулы над возвращаемыми значениями вызовов метода string.IsNullOrEmpty с различными аргументами.

Таким образом, использование резюме методов позволяет подать на вход модели в удобном формате важную для проверки истинности предупреждения информацию, использованную анализатором для его обнаружения. Достоверность такой информации высока, но не абсолютна, что может послужить причиной ошибки при верификации.

5. Генерация и интерпретация запроса

Модуль генерации в качестве входных данных принимает исходный код анализируемого проекта, результаты статического анализатора и базу данных с экспортированной информацией.

На первом этапе выполняется загрузка информации. Анализируемый проект может содержать сотни тысяч методов и десятки тысяч предупреждений, поэтому важно обеспечить эффективный менеджмент памяти, загружая только необходимые данные.

Генерация запроса выполняется независимо для каждого предупреждения. Сначала выделяется контекст в окрестности места предупреждения в коде. Контекст может задаваться количеством строк до и после, либо выбирается функция целиком. Соответствующие контексту строки помечаются.

Точки трассы представляются в виде комментариев в коде. При этом строки вокруг них также помечаются. В случае агентного подхода, возможна самостоятельная навигация модели по коду вдоль трассы предупреждения вместо предоставления ей фрагментов кода вокруг всех межпроцедурных точек трассы в одном запросе.

Далее формируется множество символов, которые используются внутри выделенных фрагментов кода. При помощи экспортированной таблицы символов в каждом участке кода происходит поиск всех идентификаторов и их свойств, таких как полное имя, вид (переменная, функция или тип), ссылка на определение и список ссылок на все использования. В запрос можно добавлять как определения переменных и типов в виде кода,

так и отдельные их свойства. Поведение определяется выбранной стратегией извлечения символов:

- без извлечения символов;
- с извлечением символов внутри участка кода вокруг предупреждения;
- с извлечением символов внутри участка кода не только вокруг предупреждения, но и вокруг точек трассы;
- с рекурсивным извлечением новых символов их определений уже добавленных.

При агентном подходе модели предоставлена возможность самостоятельно запрашивать информацию об идентификаторах, фигурирующих в исходном коде. Для каждой переменной из сформированного множества символов выделяются строки с её определением.

При использовании резюме функций и типов, формируется множество релевантных резюме. Оно определяется ранее построенным множеством релевантных символов или, если символы не используются, информацией о функциях, вызываемых из текущей. Основным форматом для представления резюме является JSON, как изображено на рис. 3. Также был опробован подход с его переводом в текстовое описание, но он не оказал существенного влияния на результаты модели.

```
Summary for Ordinary method ‘Pop‘:
{
    "Prototype": "NetMQ.NetMQFrame NetMQ.NetMQMessage.Pop()",
    "CanReturnNull": True,
    "ReturnValue": "this.m_frames.Object(0)",
    "ChangedVariables": [],
    "DereferenceConditions": [
        {"variable": "this.m_frames", "condition": "true"}
    ]
}
```

Рис. 3. Пример резюме для метода.
Fig. 3. Example of method summary.

Поля резюме можно фильтровать по именам полей, типам предупреждений и неизвестным предикатам.

Наконец, для формирования запроса требуется объединить релевантные предупреждению строки кода в один фрагмент для демонстрации модели. В простейшем случае берётся основной контекст в виде заданного количества строк выше и ниже предупреждения без выхода за границы текущей функции. К основному контексту могут быть добавлены помеченные строки вокруг точек трассы и с определениями переменных. Если используется АСД, то помимо этого осуществляется проход от всех вершин, в которых были помечены строки, к корню дерева: для листовых вершин соответствующие строки кода добавляются в итоговый фрагмент кода целиком, в то время как для внутренних вершин добавляется их заголовок (например, сигнатура функции или условие цикла) без тела оператора. Множество полученных строк объединяется в фрагменты кода с разбиением по файлам.

Помимо построения запроса исследовательской задачей является и его интерпретация. Для однозначной классификации ответов модели рассмотрено несколько подходов.

1) В запросе можно потребовать начать ответ с «Да» или «Нет», после чего проверять наличие этих слов в начале строки. В редких случаях модель игнорирует это требование. Также такой подход требует адаптации при работе с «рассуждающими» моделями, поскольку те сначала приводят последовательность размышлений и только потом дают окончательный ответ – в этом случае обработчику нужно знать ключевые слова, с которых начинается секция ответа. Во многих моделях для

упрощения последующей обработки это оформляется в виде одного или двух xml тегов, например: “<think>...</think> <answer>Ответ</answer>”.

2) Модели разрешается давать ответ в свободной форме, но после выполняется дополнительный запрос, который требует свести предыдущий ответ к простому «Да» или «Нет». Такой подход наиболее универсален, но влияет на общее время работы из-за выполнения дополнительных запросов.
3) Большинство современных систем взаимодействия с LLM поддерживает ограничение вывода модели при помощи JSON схем, при котором ответ выводится в виде корректного JSON объекта с заданным набором полей. Ключевым недостатком является то, что такой подход не подходит для рассуждающих моделей, так как не предоставляет пространства для размышлений перед окончательным ответом.
4) Некоторые системы поддерживают ограничение вывода модели при помощи контекстно-свободных грамматик. Они позволяют контролировать вывод модели с произвольной строгостью и могут заменить любой из перечисленных выше подходов, благодаря чему ответы становится легко классифицировать. Из возможных побочных эффектов – качество результатов может снижаться, если грамматика сильно ограничивает множество возможных ответов, а сам формат ответов существенно отличается от того, как модель ответила бы без применения грамматики.

В данной работе тестирование проводилось с использованием первого подхода. Отдельные эксперименты, проводимые другими методами интерпретации, показали, что использование информации анализатора сопоставимо влияет на результаты во всех случаях.

6. Тестирование

Тестирование подхода проводилось с использованием набора предупреждений статического анализатора SharpChecker, являющегося частью Svace. Они были получены 4 детекторами различных типов при анализе 15 проектов с открытым исходным кодом на языке C#, таких как Roslyn, Lucene.NET, OpenSimulator и других, использующихся для тестирования в ходе разработки инструмента, а потому имеющих достаточное количество размеченных вручную предупреждений.

Для тестирования были выбраны следующие большие языковые модели:

1) Phind/Phind-CodeLlama-34B-v2 – версия модели CodeLlama, специализирующаяся на решении задач, связанных с кодом, сокращённо будем называть её Phind;
2) Qwen/Qwen2.5-Coder-32B-Instruct – модель схожего размера, также специализирующаяся на работе с кодом;
3) deepseek-ai/DeepSeek-R1-Distill-Qwen-32B – «рассуждающая» модель, полученная путём переобучения модели Qwen2.5-32B на выводе модели DeepSeek-R1, сокращённо будем называть её R1-Qwen;
4) deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B – аналог предыдущей модели, но значительно меньшего размера, который можно применять даже без видеокарты;
5) deepseek-ai/DeepSeek-R1-Distill-Llama-70B – аналог предыдущей модели, но с 70 миллиардами параметров и использующий в качестве базовой модели Llama-3.3.

Для детерминизма результатов, все запуски выполнялись с температурой 0. Выполнение запросов осуществлялось с использованием библиотеки vLLM [16] на Nvidia A100 с 80 Гб видеопамяти. Для запуска 70B модели потребовалось использовать две такие видеокарты.

Основными метриками оценки результатов являются точность, полнота и F_1-мера. Точность показывает процент истинных срабатываний среди предупреждений, оставшихся после

фильтрации, и считается по формуле $P = \frac{TP}{TP+FP}$. Под полнотой R понимается отношение числа истинных срабатываний, оставшихся после фильтрации, к изначальному количеству истинных предупреждений, найденных статическим анализатором, и считается по формуле$R = \frac{TP}{TP+FN}$ – таким образом, полнота результатов до фильтрации принимается за единицу. F_1-мера считается по формуле $F_1 = 2\frac{P \cdot R}{P+R} = \frac{2TP}{2TP+FP+FN}$.

Текстовый запрос к модели имеет следующую структуру:

- указание на язык программирования;
- исходный код, выделенный для предупреждения описанным выше методом, обрамлённый символами ``;
- резюме типов и методов в формате json;
- описание типа предупреждения и его сообщения;
- вопрос об истинности предупреждения с указанием желаемого формата ответа и критериями для вынесения вердикта: ответ должен быть пошагово объяснён внутри тега “<think>”, окончательный ответ должен состоять из “Yes” или “No” в теге “<output>”, ответ “No” следует давать только в случае уверенности в ложности данного предупреждения.

В табл. 1, 2, 3 и 4 приведены подробные результаты фильтрации предупреждений типов «утечка ресурсов», «разыменование null», «недостижимый код» и «целочисленное переполнение» соответственно при использовании двух моделей: Phind-CodeLlama-34B-v2 и DeepSeek-R1-Distill-Qwen-32B. В этих таблицах:

1) столбец «АСД» показывает, использовались ли при построении запроса абстрактные синтаксические деревья для более точного определения границ функций и многострочных операторов;
2) столбец «Символы» показывает стратегию добавления в запрос строк кода, содержащих релевантных символы: без дополнительных строк, с добавлением кода в окрестностях точек трассы предупреждения и с рекурсивным добавлением символов, фигурирующих в определениях других символов;
3) столбец «Резюме» показывает, вставляются ли в запрос резюме функций и типов;
4) столбец «W_a; W_b» обозначает размер основного контекста – количество строк выше и ниже предупреждения, включаемых в запрос;
5) столбец «UC» (unclear) показывает количество запросов к модели, ответы на которые не удалось получить или классифицировать, а потому они не были отнесены ни к одной из четырёх категорий истинности/положительности предупреждения.

Отметим, что для детектора недостижимого кода не удалось достичь приемлемой полноты фильтрации, из-за чего F_1 мера оказалась значительно ниже исходного показателя. Тем не менее, полнота результатов при применении предложенного подхода оказалась на 11–27 процентных пунктов выше, чем без использования информации, предоставляемой статическим анализатором.

Для детектора целочисленного переполнения при использовании рассуждающей модели на любой конфигурации запроса точность фильтрации повысилась на 16–20 процентных пунктов (до 0,73–0,77) относительно базовой точности статического анализатора при полноте около 0,81. При этом использование информации, предоставляемой статическим анализатором, не показало значительного прироста к точности или полноте по сравнению с подходом, что может быть связано с отсутствием в резюме информации, важной для понимания предупреждений для данного детектора, а также тем, что при определении

релевантных фрагментов кода в них не включаются использования задействованных переменных.

Табл. 1. Результаты для детектора утечки ресурсов.
Table 1. Results for resource leak checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				374	0	54	0	0	0,87	1	0,93
Phind	-	-	-	300; 2	364	2	52	10	0	0,88	0,97	0,92
	-	-	+	300; 2	260	7	46	114	1	0,85	0,7	0,76
	+	-	-	2; 2	340	5	49	34	0	0,87	0,91	0,89
	+	по трассе	-	2; 2	330	7	47	44	0	0,88	0,88	0,88
	+	по трассе	+	2; 2	362	4	50	12	0	0,88	0,97	0,92
	+	рекурсивно	+	0; 0	366	5	49	8	0	**0,88**	**0,98**	0,93
R1-Qwen	-	-	-	300; 2	333	17	37	41	0	0,9	0,89	0,9
	-	-	+	300; 2	362	9	45	12	0	0,89	0,97	0,93
	+	-	-	2; 2	342	18	36	32	0	0,9	0,91	0,91
	+	по трассе	-	2; 2	340	17	37	34	0	0,9	0,91	0,91
	+	по трассе	+	2; 2	360	20	34	13	1	**0,91**	**0,97**	**0,94**
	+	рекурсивно	+	0; 0	358	10	43	16	1	0,89	0,96	0,92

Табл. 2. Результаты для детектора разыменования null.
Table 2. Results for null dereference checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				331	0	115	0	0	0,74	1	0,85
Phind	-	-	-	300; 2	325	3	112	6	0	0,74	0,98	0,85
	-	-	+	300; 2	262	42	73	68	1	**0,78**	0,79	0,79
	+	-	-	2; 2	311	9	106	20	0	0,75	0,94	0,83
	+	по трассе	-	2; 2	306	14	101	25	0	0,75	0,92	0,83
	+	по трассе	+	2; 2	218	57	57	112	2	**0,79**	0,66	0,72
	+	рекурсивно	+	40; 2	263	28	86	68	1	0,75	0,79	0,77
R1-Qwen	-	-	-	300; 2	266	75	39	64	2	0,87	0,81	0,84
	-	-	+	300; 2	256	50	57	55	28	0,82	0,82	0,82
	+	-	-	2; 2	268	69	45	63	1	0,86	0,81	0,83
	+	по трассе	-	2; 2	280	76	39	46	5	**0,88**	**0,86**	**0,87**
	+	по трассе	+	2; 2	295	61	54	35	1	**0,85**	**0,89**	**0,87**
	+	рекурсивно	+	40; 2	284	64	49	44	5	0,85	0,87	0,86

Табл. 3. Результаты для детектора недостижимого кода.
Table 3. Results for unreachable code checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				445	0	184	0	0	0,71	1	0,83
Phind	-	-	-	300; 2	185	109	75	260	0	0,71	0,42	0,52
	-	-	+	300; 2	280	57	126	163	3	0,69	0,64	0,66
	+	-	-	2; 2	263	44	137	182	3	0,66	0,59	0,62
	+	по трассе	-	2; 2	272	57	123	172	5	0,69	0,61	0,65
	+	по трассе	+	2; 2	307	53	128	138	3	0,71	**0,69**	0,7
R1-Qwen	-	-	-	300; 2	217	104	63	210	35	0,78	0,51	0,61
	-	-	+	300; 2	206	86	74	225	38	0,74	0,48	0,58
	+	-	-	2; 2	234	98	75	201	21	0,76	0,54	0,63
	+	по трассе	-	2; 2	250	96	72	182	29	0,78	0,58	0,66
	+	по трассе	+	2; 2	266	77	91	165	30	0,75	**0,62**	0,68

Табл. 4. Результаты для детектора целочисленного переполнения.
Table 4. Results for integer overflow checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				152	0	114	0	0	0,57	1	0,73
Phind	-	-	-	300; 2	96	48	66	56	0	0,59	0,63	0,61
	-	-	+	300; 2	98	51	63	54	0	0,61	0,64	0,63
	+	-	-	2; 2	99	54	58	51	4	0,63	0,66	0,64
	+	по трассе	-	2; 2	80	67	46	70	3	0,63	0,53	0,58
	+	по трассе	+	2; 2	82	78	34	68	4	**0,7**1	0,55	0,62
	+	рекурсивно	+	20; 2	82	72	42	67	3	0,66	0,55	0,6
R1-Qwen	-	-	-	300; 2	122	71	40	29	4	0,75	0,81	0,78
	-	-	+	300; 2	120	72	40	31	3	0,75	0,79	0,77
	+	-	-	2; 2	122	68	44	28	4	0,73	0,81	0,77
	+	по трассе	-	2; 2	120	74	39	30	3	0,75	0,8	0,78
	+	по трассе	+	2; 2	121	77	37	28	3	**0,77**	**0,81**	0,79
	+	рекурсивно	+	20; 2	121	69	45	27	4	0,73	0,82	0,77

Для детектора разыменования значения null лучшие результаты фильтрации были получены рассуждающей моделью с использованием АСД и резюме, а также с добавлением кода вдоль трассы предупреждения. Использование такой конфигурации позволило поднять точность на 11 процентных пунктов до 0,85 при полноте 0,89.

Для детектора утечки ресурсов при аналогичных параметрах удалось повысить точность на 4 процентных пункта до 0,91 при полноте 0,97.

В целом можно заметить, что за исключением детектора целочисленного переполнения, для рассуждающей модели использование резюме и извлечения символов вдоль трассы помогало поднять полноту на 8-11 процентных пунктов по сравнению с подходом, использующим только исходный код.

В табл. 5 приводятся результаты сравнения пяти моделей, перечисленных в начале раздела, на объединённом наборе данных, включающем все три типа предупреждений без детектора недостижимого кода. Параметры генерации запросов для каждого типа предупреждения соответствуют последней строке табл. 1, 2 и 4. Можно отметить, что из опробованных вариантов наилучших результатов удалось достичь при помощи рассуждающих моделей – при этом модели с 32 и 70 миллиардами параметров продемонстрировали схожие показатели F_1-меры при росте точности на 10 и 12 процентных пунктов соответственно.

Табл. 5. Сравнение результатов моделей различных типов и размеров.
Table 5. Comparison of result with models of different kinds and sizes.

Модель	Млрд. параметров	TP	TN	FP	FN	UC	P	R	F_1
До фильтрации результатов		857	0	283	0	0	0,75	1	0,86
DeepSeek-R1-Distill-Qwen-32B	32	763	143	137	87	10	0,85	**0,9**	0,87
DeepSeek-R1-Distill-Llama-70B	70	734	171	109	115	11	**0,87**	0,86	0,87
Phind-CodeLlama-34B-v2	34	711	105	177	143	4	0,8	0,83	0,82
DeepSeek-R1-Distill-Qwen-1.5B	1.5	661	79	199	170	31	0,77	0,8	0,78
Qwen2.5-Coder-32B-Instruct	32	540	201	82	314	3	0,87	0,63	0,73

Этап генерации запросов на основе исходного кода и сохранённых данных является общим для всех моделей и занимает несколько минут (в среднем менее секунды на 1 запрос). Время обработки запросов LLM зависит от размера модели, а также от количества токенов как в самом запросе, так и в ответе модели. Используемая библиотека vLLM позволяет объединять запросы в группы, выполняя их обработку параллельно при наличии достаточных ресурсов видеокарты. На этом наборе данных средний размер запроса составляет порядка тысячи токенов, меньшая из моделей завершает их обработку за 10 минут (2 запроса в

секунду), моделям среднего размера требуется около часа (3 секунды на один запрос), а для модели с 70 миллиардами параметров время обработки запросов увеличивается примерно до 2,5 часов (7 секунд на один запрос), а также требуется дополнительная видеокарта для работы.

Ответы моделей были просмотрены вручную, из них были выделены основные причины, по которым происходит некорректная классификация предупреждений:

1) Ошибочные рассуждения модели. Модель может делать некорректные выводы и игнорировать факты, очевидные из предоставленного контекста. LLM могут делать неверные предположения о свойствах переменных, путать пути выполнения, ошибочно интерпретировать информацию из резюме и галлюцинировать.
2) Нехватка или некорректность предоставленного контекста. Модель может неверно классифицировать предупреждения, если в контексте не содержатся важные для их понимания определения или использования переменных, резюме методов или типов. Реже на ответ модели влияют упрощённые предикаты и некорректная информация из резюме.
3) Различия в определении дефекта. В спорных ситуациях ответы модели и эксперта о том, нужно ли выдавать рассматриваемое предупреждение, могут расходиться из-за различий в критериях истинности предупреждений, которые могут отличаться в разных моделях и разных статических анализаторах. Для преодоления этой проблемы можно помещать в запрос критерии истинности предупреждения в виде набора инструкций.
4) Игнорирование моделью явно прописанных требований к ответу.
5) Ошибки эксперта при ручной разметке предупреждений анализатора.

Также в рамках данной работы было опробовано обучение низкорангового адаптера LoRA (Low-Rank Adaptation) для модели Phind на подмножестве предупреждений типа «утечка ресурсов». Разбиение набора предупреждений на обучающую и тестовую выборку производилось попроектно, чтобы избежать использования в тестировании примеров, схожих с теми, которые использовались при обучении. Обучающая выборка состоит из 278 примеров, а тестовая – из 150. Хотя такой размер выборки считается небольшим, мы посчитали его пригодным для дообучения, поскольку данная LoRA предназначена для проверки одного конкретного типа ошибки с фиксированными шаблонами вопроса и ответа. Для более сложных задач потребовалось бы увеличить размер набора данных, что представляет собой проблему из-за необходимости ручной разметки предупреждений анализатора на реальных проектах.

Запросы для обучающего набора составлялись по шаблону с абстрактными синтаксическими деревьями, добавлением символов по трассе и резюме функций. В качестве эталонного ответа использовалось только слово “Yes” или “No” без промежуточных рассуждений – формулировка запроса была изменена соответствующим образом. Обучение выполнялось с константой альфа, равной 128 и рангом адаптера, равным 64, продолжалось от 4 до 8 эпох, в зависимости от строки таблицы, и заняло порядка 100 минут для 8 эпох обучения. Результаты оценки приведены в табл. 6. Базовая точность анализатора отличается от указанной в табл. 1 из-за того, что оценка происходит только на тестовой выборке. Результаты модели без обучения, с которой происходит сравнение, также соответствуют запросу, требующему односложный ответ.

После дообучения наилучшим результатом для модели Phind стал прирост точности на 3 процентных пункта до 0,81 при полноте 0,94, что превосходит результаты этой же модели без использования LoRA.

Дообучение рассуждающих моделей представляет бóльшую сложность, поскольку для каждого предупреждения в обучающей выборке нужно указать не только сам ответ, но и

цепочку рассуждений, к нему приводящую. Обучающие примеры таких рассуждений можно получать при помощи моделей с бóльшим количеством весов (в том числе проприетарных) и отбирать из них те примеры, в которых модель дала правильный ответ. Чтобы повысить количество пригодных примеров, можно в изначальном запросе подсказывать правильный ответ и просить рассуждать так, будто модель догадалась до него самостоятельно. Более технически сложным вариантом является обучение с подкреплением, для которого не нужны эталонные примеры рассуждений, а оценивается именно итоговый ответ модели.

Табл. 6. Результаты для детектора утечки ресурсов с использованием LoRA.
Table 6 Results for resource leak checker with use of LoRA.

Модель	АСД	Символы	Резюме	W_a; W_b	Кол-во эпох	TP	TN	FP	FN	P	R	F_1
До фильтрации результатов						116	0	34	0	0,77	1	0,87
Phind	+	по трассе	+	2; 2	-	106	4	30	10	0,78	0,91	0,84
Phind + LoRA					4	116	2	32	0	0,78	1	0,88
					6	109	8	26	7	**0,81**	**0,94**	0,87
					8	105	10	24	11	0,81	0,91	0,86

7. Заключение

В статье описан метод повышения точности результатов статического анализа за счёт фильтрации предупреждений большой языковой моделью с использованием вспомогательной информации, извлечённой из анализатора, который был реализован в рамках статического анализатора SharpChecker. Для этого в статическом анализаторе был разработан модуль для сбора и сохранения необходимой информации, а также модуль генерации запросов к LLM и интерпретации ответов модели на основе собранной информации.

Метод был протестирован на размеченных вручную предупреждениях статического анализатора SharpChecker четырёх типов при помощи двух LLM с открытыми весами. Было показано, что предложенный подход позволяет повысить точность детектора ошибок целочисленного переполнения с 57% до 77% (прирост в 20%) при полноте 80%, детектора разыменований null – до 85% (прирост в 11%) при полноте 89%, детектора утечек ресурсов – до 91% (прирост в 4%) при полноте 97%. Использование вспомогательной информации, собранной статическим анализатором, повышает полноту фильтрации предупреждений как минимум на 8%, а для детектора недостижимого кода – до 27%.

Было проведено сравнение пяти моделей разных типов («рассуждающие» и обычные) и размеров (от 1,5 миллиардов параметров до 70) на наборе данных, объединяющем 3 типа предупреждений.

Кроме того, было опробовано дообучение LLM при помощи низкорангового адаптера LoRA, которое позволило повысить точность и полноту фильтрации ошибок типа «утечка ресурсов» на 3% (до 81% и 94%, соответственно).

Список литературы / References

[1]. Gerasimov A. Y. Directed dynamic symbolic execution for static analysis warnings confirmation. Programming and Computer Software, vol. 44, 2018, pp. 316-323. DOI: 10.1134/S036176881805002X

[2]. Tsiazhkorob U. V., Ignatyev V. N. Classification of Static Analyzer Warnings using Machine Learning Methods. Ivannikov Memorial Workshop (IVMEM), IEEE, 2024, pp. 69-74. DOI: 10.1109/IVMEM63006.2024.10659704.

[3]. Ignatyev V. N., Shimchik N. V., Panov D. D., Mitrofanov A. A. Large language models in source code static analysis. Ivannikov Memorial Workshop (IVMEM), IEEE, 2024, pp. 28-35. DOI: 10.1109/IVMEM63006.2024.10659715.

[4]. GPT-4 | OpenAI, available at: https://openai.com/index/gpt-4/, accessed 14.05.2025.

[5]. Koshelev V. K., Ignatiev V. N., Borzilov A. I., Belevantsev A. A. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, 2017, pp. 268-276. DOI: 10.1134/S0361768817040041.

[6]. Ivannikov V. P., Belevantsev A. A., Borodin A. E., Ignatiev V. N., Zhurikhin D. M., Avetisyan A. I. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, 2014, pp. 265-275. DOI: 10.1134/S0361768814050041.

[7]. Li H., Hao Y., Zhai Y., Qian Z. Enhancing static analysis for practical bug detection: An llm-integrated approach. Proceedings of the ACM on Programming Languages, vol. 8, No. OOPSLA1, 2024, pp. 474-499. DOI: 10.1145/3649828.

[8]. Mohajer M. M., Aleithan R., Harzevili N. S., Wei M., Belle A. B., Pham H. V., Wang S. Effectiveness of ChatGPT for static analysis: How far are we? Proceedings of the 1st ACM International Conference on AI-Powered Software, 2024, pp. 151-160. DOI: 10.1145/3664646.3664777.

[9]. Wen C., Cai Y., Zhang B., Su J., Xu Z., Liu D., Qin S., Ming Z., Cong, T. Automatically inspecting thousands of static bug warnings with large language model: How far are we? ACM Transactions on Knowledge Discovery from Data, vol. 18, No. 7, 2024, pp. 1-34. DOI: 10.1145/3653718.

[10]. Li Z., Dutta S., Naik M. IRIS: llm-assisted static analysis for detecting security vulnerabilities. arXiv preprint arXiv:2405.17238, 2024.

[11]. Khare A., Dutta S., Li Z., Solko-Breslin A., Alur R., Naik M. Understanding the effectiveness of large language models in detecting security vulnerabilities. 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2025, pp. 103-114. DOI: 10.1109/ICST62969.2025.10988968.

[12]. Introduction – Tree-sitter, available at: https://tree-sitter.github.io/tree-sitter/, accessed 14.05.2025.

[13]. Mou L., Li G., Zhang L., Wang T., Jin Z. Convolutional neural networks over tree structures for programming language processing. Proceedings of the AAAI conference on artificial intelligence, vol. 30, No. 1, 2016. DOI: 2016.10.1609/aaai.v30i1.10139.

[14]. GitHub – The Roslyn .NET compiler, available at: https://github.com/dotnet/roslyn, accessed 14.05.2025.

[15]. GitHub – mozilla/dxr, available at: https://github.com/mozilla/dxr, accessed 14.05.2025.

[16]. vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for LLMs, available at: https://github.com/vllm-project/vllm, accessed 14.05.2025.

Информация об авторах / Information about authors

Данила Дмитриевич ПАНОВ – старший лаборант ИСП РАН, студент ВМК МГУ. Его научные интересы включают статический анализ программного обеспечения и большие языковые модели.

Danila Dmitrievich PANOV – senior laboratory assistant at ISP RAS, student at CMC faculty of Lomonosov Moscow State University. His research interests include static analysis of programs and large language models.

Никита Владимирович ШИМЧИК – кандидат технических наук, младший научный сотрудник ИСП РАН. Его научные интересы включают статический анализ программного обеспечения, большие языковые модели.

Nikita Vladimirovich SHIMCHIK – Cand. Sci. (Tech.), researcher at ISP RAS. His research interests include static analysis of programs, large language models.

Дмитрий Александрович ЧИБИСОВ – аспирант ИСП РАН, Его научные интересы включают статический анализ программного обеспечения, большие языковые модели.

Dmitrii Aleksandrovich CHIBISOV – postgraduate student at ISP RAS. His research interests include static program analysis and large language models.

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

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., corresponding Member RAS, leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.

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.