УДК 004.056.53 Вестник УрФО № 2(60) / 2026, с. 47–58

Гладких К.И., Шабалин А.М., Захаров А.А.

DOI: 10.14529/secur260205

МЕТОДОЛОГИЯ ФОРМИРОВАНИЯ КОНТЕКСТА УЯЗВИМОСТЕЙ ДЛЯ ИХ ОБНАРУЖЕНИЯ ЯЗЫКОВЫМИ МОДЕЛЯМИ НА ОСНОВЕ ГРАФА СВОЙСТВ КОДА

В статье рассматривается задача формирования программного контекста для обнаружения уязвимостей исходного кода языковыми моделями. Ограниченность анализа изолированных функций приводит к потере межпроцедурных зависимостей, потоков данных и условий использования, влияющих на проявление уязвимости. Предложена методология, основанная на графе свойств кода, который объединяет структурные элементы программы, граф вызовов и зависимости данных. Методология включает синтаксический разбор Python-проектов, построение узлов функций, классов, переменных и блоков кода, формирование ребер вызовов и потоков данных, загрузку графа в Neo4j, связывание результатов статического анализа с узлами кода, извлечение локального подграфа вокруг потенциально уязвимого участка и ранжирование элементов контекста по графовой удаленности и повторяемости. Полученный контекст адаптируется к ограничению контекстного окна языковой модели. Апробация на 400 экземплярах из набора CVEFixes показала снижение доли ложноотрицательных срабатываний и рост коэффициента Фи для большинства протестированных моделей. Результаты подтверждают, что учет семантического программного контекста повышает устойчивость обнаружения уязвимостей. Работа соответствует пунктам 15 и 17 паспорта научной специальности 2.3.6.

Ключевые слова: *уязвимость, статический анализ, большая языковая модель, граф свойств кода, программный контекст, информационная безопасность.*

A METHODOLOGY FOR CONSTRUCTING VULNERABILITY CONTEXT FOR DETECTION BY LANGUAGE MODELS BASED ON CODE PROPERTY GRAPHS

The article addresses the problem of constructing program context for vulnerability detection in source code using language models. Function-level analysis often loses interprocedural dependencies, data flows, and usage conditions that determine whether a vulnerability can manifest. The proposed methodology is based on a Code Property Graph that combines program structure, call relations, and data-flow dependencies. The methodology includes parsing Python projects, constructing nodes for functions, classes, variables, and code blocks, creating call-graph and data-flow edges, loading the graph into Neo4j, linking static-analysis findings with code nodes, extracting a local subgraph around a potentially vulnerable fragment, and ranking context elements by graph distance and repetition frequency. The resulting context is adapted to the context-window limit of a language model. The evaluation on 400 CVEFixes-based instances showed a reduction in false-negative predictions and an increase in the Phi coefficient for most tested models. The results indicate that semantically grounded program context improves the robustness of language-model-based vulnerability detection.

Keywords: *vulnerability, static analysis, large language model, code property graph, program context, information security.*

Введение

Большие языковые модели активно применяются для анализа исходного кода [1,2] однако их результативность в задачах поиска уязвимостей зависит не только от архитектуры модели и промпта, но и от состава программного контекста. В распространенной постановке задача сводится к бинарной классификации отдельной функции или небольшого фрагмента. Такая постановка удобна для бенчмарков, но часто не отражает природу уязвимости: источник данных, цепочка вызовов, состояние объекта и условие выполнения могут находиться вне анализируемой функции [3–6].

Отсутствие релевантного контекста приводит к двум типам ошибок. Во-первых, модель может не обнаружить уязвимость, если необходимый источник данных или обработчик расположен в другой части проекта. Во-вторых, расширение контекста без приоритизации увеличивает шум, расходует контекстное окно и может ухудшать классификацию. Поэтому ключевой задачей является не простое добавление фрагментов кода, а выбор компактного множества элементов, непосредственно связанных с потенциально уязвимым участком [7–10].

В литературе выделяются три группы решений. Первая группа использует retrieval-augmented generation, при котором в запрос к модели добавляются сведения об известных уязвимостях, исправлениях или связанных фрагментах репозитория [8,9]. Вторая группа объединяет LLM со статическим анализом, программными срезами и графовыми представлениями кода [7,11–13]. Третья группа применяет агентные архитектуры, где мо-

дель последовательно вызывает инструменты анализа, уточняет гипотезы и формирует итоговый ответ [14,15]. Эти подходы полезны, но имеют ограничения: retrieval может не учитывать фактические зависимости внутри проекта, графовые конвейеры часто требуют ресурсоемкой обработки, а агентные системы зависят от стоимости, времени выполнения и недетерминированности модели.

Дополнительная методологическая проблема связана с качеством наборов данных. Многие датасеты формируются по коммитам исправления уязвимостей и содержат только измененные строки или функции. Это упрощает разметку, но снижает полноту программного контекста и может создавать статистические артефакты, по которым модель учится отличать исправленный код от уязвимого без анализа семантики программы [16–21].

Проблема, цель и задачи исследования

Проблема исследования состоит в том, что существующие способы подачи кода в языковую модель либо анализируют слишком локальные фрагменты, либо добавляют избыточный контекст без явной связи с проявлением уязвимости. В результате модель получает неполную или зашумленную информацию, а качество обнаружения остается нестабильным.

Цель исследования – разработать и экспериментально оценить методологию формирования компактного программного контекста потенциально уязвимых участков кода на основе графа свойств кода, обеспечивающую повышение качества обнаружения уязвимостей языковыми моделями по сравнению с анализом изолированных фрагментов.

Для достижения цели были поставлены следующие задачи:

• проанализировать существующие подходы к формированию контекста для LLM-обнаружения уязвимостей и определить их ограничения;

• разработать способ построения графа свойств кода для Python-проектов на основе синтаксического разбора и межпроцедурных связей;

• предложить алгоритм извлечения и ранжирования контекстного подграфа по релевантности;

• реализовать сборку программного контекста с учетом ограничения контекстного окна модели;

• провести апробацию на данных CVEFixes и оценить статистическую значимость различий между базовым и контекстным подходами.

Методология формирования программного контекста

Объектом анализа является исходный код программного проекта на языке Python. Предметом анализа являются связи между потенциально уязвимым участком и другими элементами программы: вызовами функций, передачей аргументов, определениями переменных, принадлежностью элементов классам и блокам кода. Область исследования ограничена статическим анализом исходного кода и бинарной классификацией экземпляра как уязвимого или неуязвимого. Предполагается, что потенциально уязвимые участки могут быть локализованы либо по строкам исправления в CVE-записях, либо по находкам статических анализаторов.

Методология строится как последовательный конвейер. Сначала проект разбирается синтаксическим анализатором. Затем строится граф свойств кода. После этого результаты статического анализа связываются с узлами графа. Далее вокруг корневых узлов извлекается локальный подграф, его узлы ранжируются, а соответствующие фрагменты исходного кода добавляются в промпт до достижения лимита контекстного окна. Общая схема представлена на рис. 1.

Построение графа свойств кода

Граф свойств кода (Code Property Graph, CPG) представляет программу как граф, в котором синтаксические элементы, потоки управления и зависимости данных описываются в единой структуре [21]. В данной работе CPG адаптирован для Python-кода. Синтаксическая структура извлекается с помощью tree-sitter-python; затем из дерева формируются узлы и ребра, отражающие семантически значимые отношения между частями программы.

В граф включаются четыре типа узлов. FunctionNode описывает функцию или метод, содержит имя, файл и диапазон строк, а также связывается с параметрами. ClassNode описывает класс и его тело. VariableNode фиксирует параметры, присваивания и переменные текущей области видимости. CodeBlockNode представляет непрерывный блок императивных операторов уровня модуля и используется как вызывающий контекст для скриптового кода.

Ребра графа делятся на две основные груп-

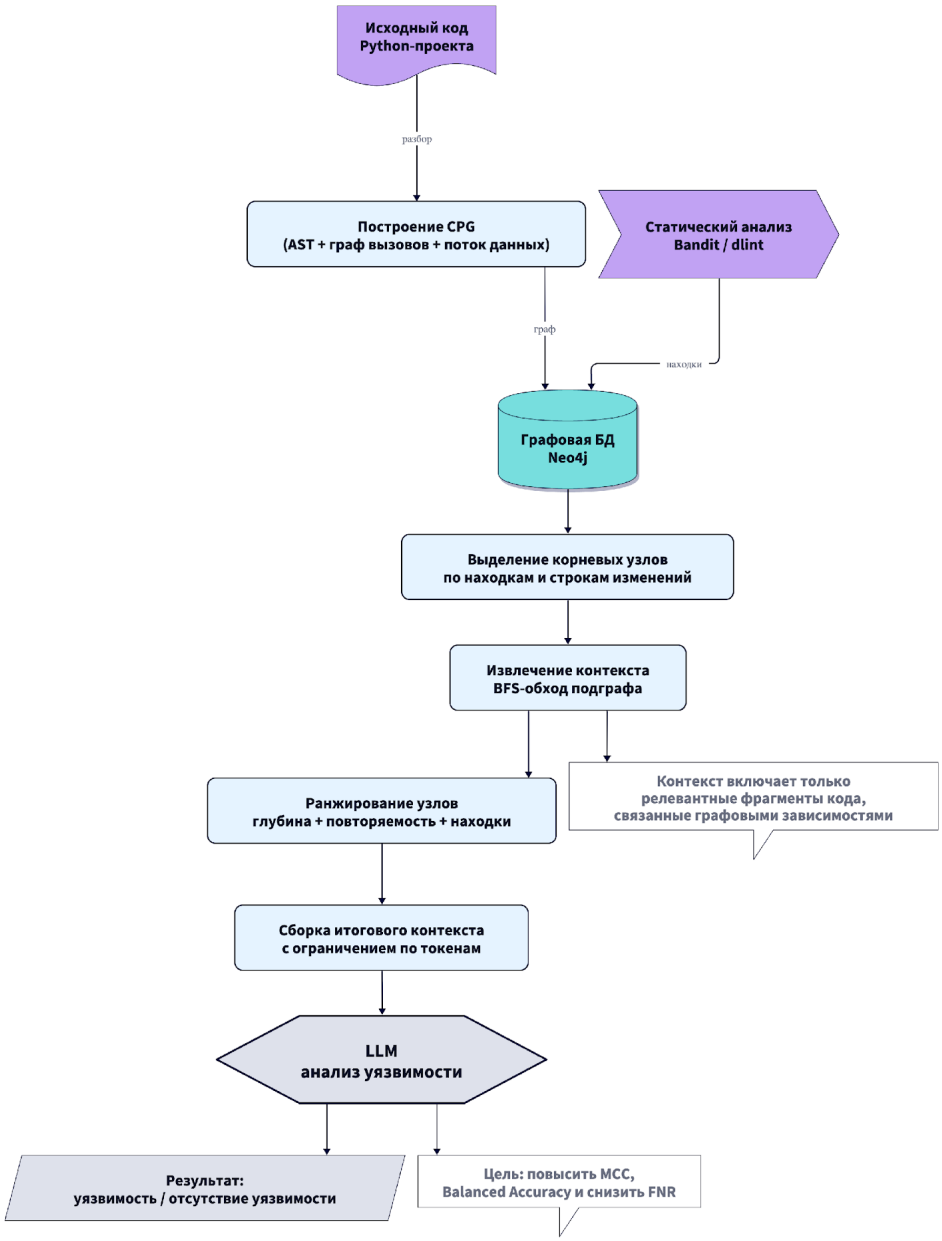

Рис. 1. Схема формирования программного контекста уязвимости

пы. Ребра CallGraphCalls и CallGraphCalledBy фиксируют вызовы функций и методов. Если вызываемый объект является идентификатором, он разрешается по цепочке областей видимости; если это атрибут, разрешается имя метода. Ребра DataFlowDefinedBy и DataFlowFlowsTo описывают определение значения и передачу данных в аргументы вызовов. Такая модель позволяет извлекать не только локальный фрагмент, но и окружающие его зависимости.

Построение выполняется в несколько проходов. На уровне модуля сначала выделяются блоки императивного кода, затем предварительно связываются классы и функции, после чего выполняется полный рекурсивный обход синтаксического дерева. Для разрешения идентификаторов используется стек областей видимости, а для атрибуции вызовов – стек вызывающего контекста. При обработке каталога сначала формируется индекс экспортируемых имен по файлам, затем выполняется повторная сборка с учетом межмодульных импортов.

Практическая реализация разделена на три компонента. Класс CPGFileBuilder обрабатывает один исходный файл: считывает байты, декодирует текст в UTF-8, строит конкрет-

ное синтаксическое дерево с помощью tree_sitter.Parser и нормализует путь относительно корня проекта. CPGDirectoryBuilder распространяет обработку на дерево каталогов, формирует индекс экспортируемых имен и повторно строит граф с учетом межмодульных импортов. NodeProcessor выполняет рекурсивный обход дерева, поддерживая стек областей видимости и стек вызывающего контекста.

Текущая реализация имеет ограничения. Управляющие конструкции не выделяются в самостоятельный граф потока управления, а учитываются через кодовые узлы, в которые уже входит соответствующий синтаксический фрагмент. Разрешение вызовов методов выполняется по имени метода и не моделирует все случаи динамической диспетчеризации Python. Эти ограничения учитывались при интерпретации результатов.

Хранение графа и извлечение подграфа

Для хранения CPG используется графовая база Neo4j. Элементы исходного кода загружаются пакетно: каждая запись сопоставляется с узлом Code и дополнительной меткой, уточняющей его роль, например, Function, Class, Variable или Call. Для связей используется параметризованная загрузка Cypher-запросами. Применение MERGE обеспечивает идемпотентное создание узлов и снижает риск дублирования при повторной обработке проекта.

Контекст извлекается как локальный подграф вокруг множества корневых узлов. Корневым считается узел, связанный с потенциально уязвимым фрагментом. Запрос выполняет обход графа на глубину N по выбранным типам ребер и возвращает идентификатор корня, идентификатор найденного узла, путь к файлу, диапазон строк, имя, тип узла и длину пути. Длина пути используется как оценка графовой удаленности.

Пример контекстного подграфа, построенного для функции main при глубине 2, показан на рис. 2.

Определение потенциально уязвимых участков

Потенциально уязвимый участок кода определяется как фрагмент, который взаимодействует с внешними или частично доверенными источниками данных, ресурсами либо подсистемами: базами данных, файловой системой, сетевыми интерфейсами, пользовательским вводом или системными командами. Ошибка в таком фрагменте может нарушить конфиденциальность, целостность или доступность данных.

В экспериментальной реализации такие

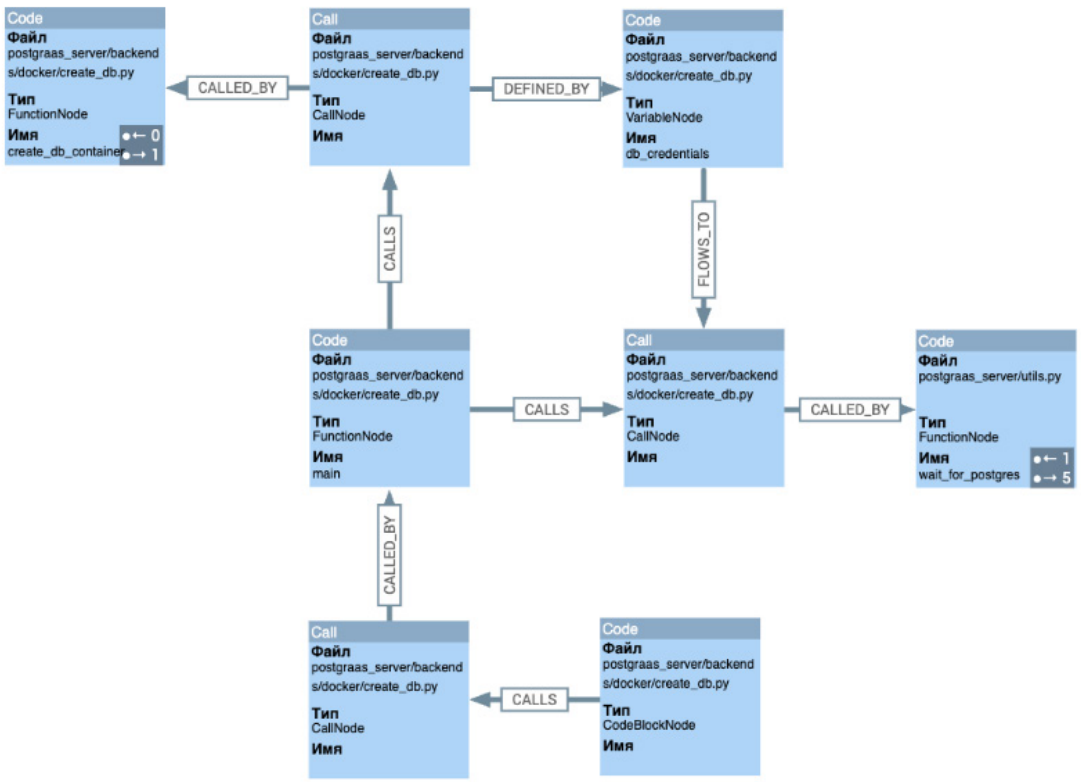

Рис. 2. Пример результата построения контекстного подграфа для функции main с глубиной 2

участки выявляются двумя способами. В режиме апробации на CVEFixes корневые узлы определяются по файлам и строкам, затронутым исправлением уязвимости. В режиме анализа проекта корневые узлы могут формироваться по находкам статических анализаторов Bandit и dlint. Каждая находка сохраняется в графе и связывается с соответствующим узлом ребром StaticAnalysisReports.

Ранжирование и сбор контекста

Так как контекстное окно языковой модели ограничено, весь подграф не передается в модель. Узлы ранжируются по трем признакам: графовой глубине относительно корня, числу повторений в множестве подграфов и числу связанных находок статического анализа. В базовой реализации приоритет получает узел с меньшей глубиной, большей повторяемостью и большим числом находок.

После ранжирования исходный код узлов последовательно добавляется в результирующий контекст. Добавление прекращается, когда расчетное число токенов достигает заданного лимита. Оценка числа токенов выполняется библиотекой tiktoken для целевого типа языковой модели. Таким образом, методология не увеличивает контекст бесконтрольно, а использует граф как механизм отбора и упорядочивания программной информации.

Апробация методологии

Апробация выполнена на базе CVEFixes – набора данных, содержащего реальные CVE-записи, связанные репозитории и коммиты исправления уязвимостей [22]. Для каждой записи извлекалось состояние репозитория до исправления и после исправления. Затем для обоих состояний строился CPG, данные загружались в Neo4j, а по файлам и строкам исправления определялись корневые узлы.

Для получения контекста применялся обход графа в ширину на глубину 6. Из найденных подграфов извлекались уникальные узлы, подсчитывалась их повторяемость, затем узлы сортировались по возрастанию глубины, убыванию числа повторений и числу находок. Соответствующие фрагменты кода объединялись в экземпляр контекста до достижения лимита контекстного окна.

В итоговую выборку вошло 400 экземпляров исходного кода. Часть записей была исключена, так как результирующий код превышал допустимый размер контекстного окна или не мог быть корректно сопоставлен с узлами графа. Для сравнения использовались два варианта подачи данных: исходный фрагмент CVEFixes и контекст, сформированный предложенным методом ContextAssembler.

В экспериментах применялись модели Gemma3 4B, Llama 3.2 3B, Qwen3-4B thinking, Qwen3-8B thinking, Qwen3.5-4B и Qwen3.5-4B thinking. Температура была равна 0,3. Размер контекстного окна составлял 24 000 токенов для моделей с режимом рассуждения и 20 000 токенов для остальных моделей. Во всех запусках использовался одинаковый seed. Промпт к моделям представлен на рис. 3.

Оценка выполнялась по общей доле правильных классификаций (accuracy), полноте (recall), специфичности (specificity), доле ложноотрицательных срабатываний (FNR), сбалансированной точности и коэффициенту

```
You are a methodical security analyst who follows a systematic
approach to vulnerability detection.
Analysis Steps:
1. Read through the code to understand its purpose
2. Identify all input sources and data flows
3. Check for proper input validation and bounds checking
4. Look for memory management issues
5. Examine error handling and edge cases
6. Determine if vulnerabilities exist\n\nFinal Answer: After
your analysis, respond with \"VULNERABLE\" or \"SAFE\"
```

Рис. 3. Промпт к языковым моделям для анализа исходного кода на предмет уязвимостей

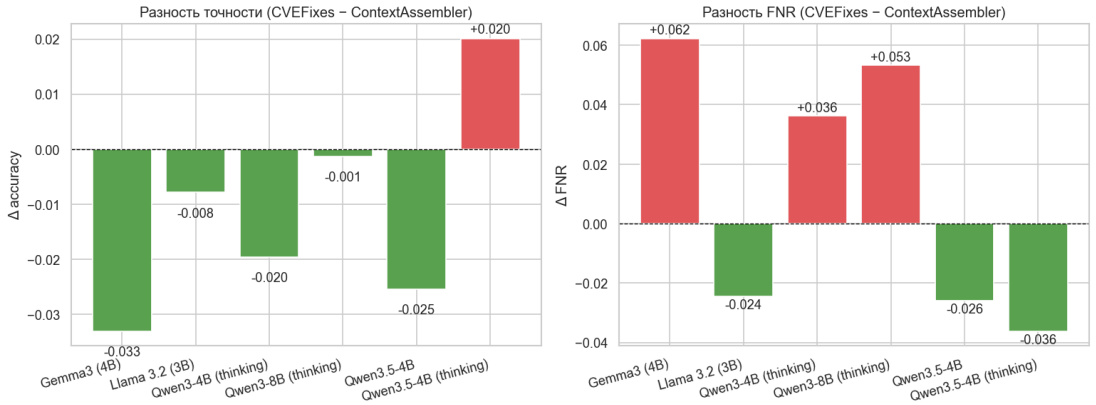

Рис. 4. Сравнение общей доли правильных классификаций и ложноотрицательных срабатываний моделей на разных наборах данных

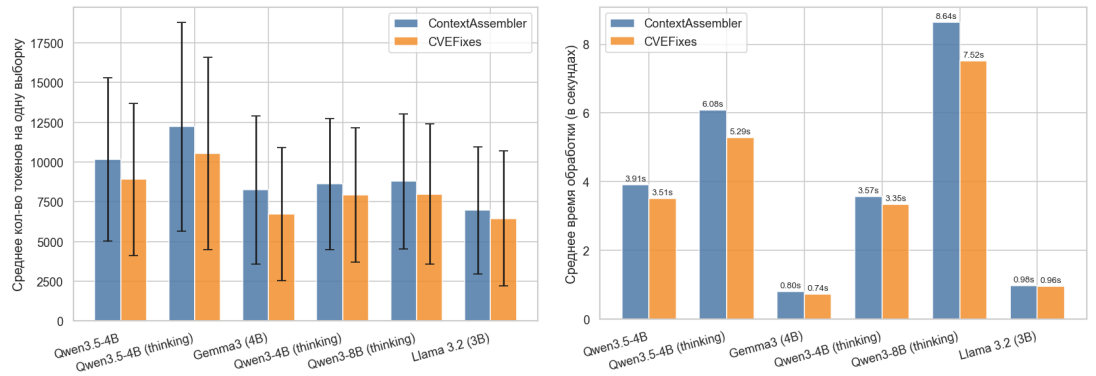

Рис. 5. Средние показатели потребления токенов и времени обработки одного экземпляра кода

корреляции Фи. Для проверки статистической значимости различий между двумя способами формирования контекста применялся тест Макнемара с поправкой на непрерывность. Предсказания агрегировались на уровне CVE-записей методом голосования большинством.

Результаты

На рис. 4 показано сравнение accuracy и доли ложноотрицательных срабатываний. Для большинства моделей ContextAssembler снижает FNR по сравнению с исходным представлением CVEFixes. Это особенно важно для задач безопасности, поскольку ложноотрицательный ответ означает пропуск уязвимости. При этом рост accuracy не является универсальным: для отдельных моделей наблюдаются небольшие отклонения, что указывает на зависимость эффекта от архитектуры и режима генерации.

На рис. 5 приведены средние показатели потребления токенов и времени обработки одного экземпляра. Увеличение числа токенов является ожидаемым, так как методология добавляет межпроцедурный контекст. Однако время обработки для большинства моделей изменилось умеренно; наиболее заметные отклонения наблюдались у семейства Qwen.

На рис. 6 показано сравнение коэффициента Фи и сбалансированной точности. Значения коэффициента Фи для ContextAssembler положительны во всех экспериментах, тогда как для базового CVEFixes в отдельных случаях они близки к нулю или отрицательны. Это означает, что контекстное представление уменьшает случайность классификации и повышает согласованность предсказаний с истинной разметкой.

Статистическая проверка показала, что для четырех из шести моделей различия между подходами значимы при $p < 0{,}01$. Для Gemma3 4B, Llama 3.2 3B, Qwen3-8B thinking и Qwen3.5-4B thinking ContextAssembler давал больше случаев, в которых контекстный подход был корректен, а базовый подход ошибался. Для Qwen3-4B thinking и Qwen3.5-4B статистическая значимость не достигнута. Результаты приведены в табл. 1.

Сводные метрики приведены в табл. 2. По

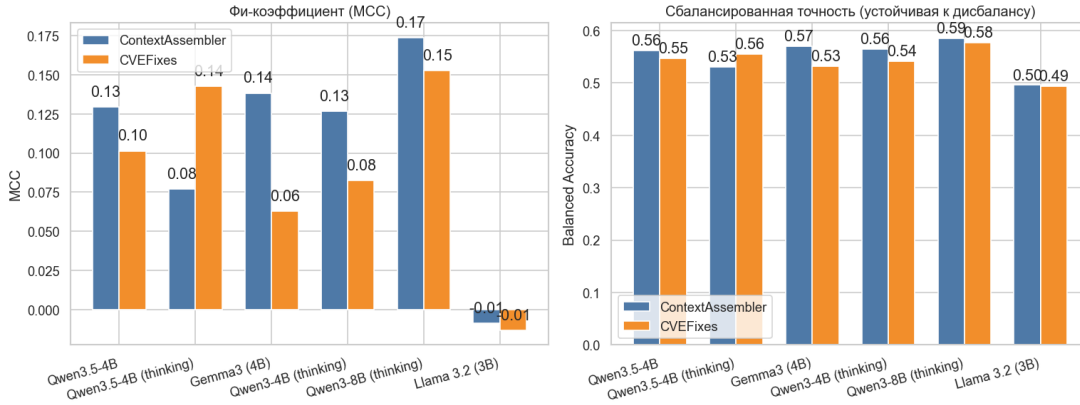

Рис. 6. Сравнение моделей по коэффициенту корреляции Фи и сбалансированной точности

Таблица 1

Оценка статистической значимости тестом Макнемара

Модель	Совпадение по CVE	Оба корректны	CA ✓ CVE ✗	CA ✗ CVE ✓	Оба некорректны	χ^2	p-value	Значимо
Gemma3 (4B)	120	20	37	16	47	7,547	0,0060	да
Llama 3.2 (3B)	131	12	41	18	60	8,203	0,0042	да
Qwen3-4B (thinking)	128	29	31	19	49	2,420	0,1198	нет
Qwen3-8B (thinking)	128	27	39	14	48	10,868	0,0010	да
Qwen3.5-4B	124	22	32	21	49	1,887	0,1696	нет
Qwen3.5-4B (thinking)	124	20	37	11	56	13,021	0,0003	да

сравнению с CVEFixes предложенный подход увеличивает коэффициент Фи для Gemma3 4B, Llama 3.2 3B, Qwen3-4B thinking, Qwen3-8B thinking и Qwen3.5-4B. Для Qwen3.5-4B thinking коэффициент Фи ниже, однако FNR также уменьшается, что показывает смещение модели в сторону более чувствительного обнаружения уязвимостей.

Дальнейшие направления для исследования

Полученные результаты показывают, что графовое формирование контекста особенно полезно для снижения ложноотрицательных срабатываний. Это соответствует предположению исследования: уязвимость часто определяется не только локальным кодом, но и связанными с ним источниками данных, вызовами и определениями. Одновременно результаты не позволяют утверждать, что методология всегда повышает accuracy. На отдельных моделях эффект выражается иначе, что требует дополнительного анализа промптов, архитектур и стратегии ранжирования.

Основное ограничение работы связано с упрощенным ранжированием контекста. В текущей версии используются глубина, повторяемость и число находок, однако эти признаки не учитывают тип уязвимости, направление потока данных, доверенность источника и наличие санитизации. Второе ограничение состоит в неполной модели потока управления для Python. Третье ограничение связано с размером выборки: 400 экземпляров достаточно для первичной апробации, но не заменяет крупномасштабную оценку на разных языках программирования и типах проектов.

Дальнейшее развитие методологии пред-

Таблица 2

Полные результаты бенчмарка

Набор данных	Модель	Количество	Accuracy	Recall	Specificity	FNR	Фи
Context-Assembler	Gemma3 (4B)	394	0,551	0,667	0,472	0,333	0,138
	Llama 3.2 (3B)	400	0,455	0,700	0,292	0,300	–0,009
	Qwen3-4B (thinking)	400	0,565	0,562	0,567	0,438	0,127
	Qwen3-8B (thinking)	400	0,555	0,738	0,433	0,262	0,174
	Qwen3.5-4B	400	0,528	0,738	0,388	0,262	0,130
	Qwen3.5-4B (thinking)	400	0,465	0,856	0,204	0,144	0,077
CVEFixes	Gemma3 (4B)	394	0,518	0,604	0,459	0,396	0,063
	Llama 3.2 (3B)	400	0,447	0,724	0,264	0,276	–0,013
	Qwen3-4B (thinking)	400	0,545	0,526	0,558	0,474	0,082
	Qwen3-8B (thinking)	400	0,554	0,684	0,469	0,316	0,153
	Qwen3.5-4B	400	0,502	0,763	0,331	0,237	0,101
	Qwen3.5-4B (thinking)	400	0,485	0,892	0,218	0,108	0,143

полагает включение taint-анализа, учет типов CWE и CVE-метаданных, более строгую модель межпроцедурного потока данных, а также адаптацию промптов под конкретные классы уязвимостей. Отдельным направлением является оценка устойчивости метода на репозиторном уровне, где необходимо учитывать зависимости между файлами, внешними библиотеками и конфигурацией проекта.

Заключение

В статье предложена методология формирования программного контекста потенциально уязвимых участков кода на основе графа свойств кода. Методология объединяет синтаксическое представление программы, граф вызовов, потоки данных и результаты статического анализа, после чего извлекает локальный подграф вокруг корневого узла и преобразует его в компактный текстовый контекст для языковой модели.

Экспериментальная апробация на данных CVEFixes показала, что контекст, сформированный с помощью графа свойств кода, снижает долю ложноотрицательных срабатываний и повышает коэффициент Фи для большинства протестированных моделей. Статистический анализ тестом Макнемара подтвердил значимый сдвиг предсказаний для четырех из шести моделей. Следовательно, учет межпроцедурного и семантического контекста является существенным условием повышения надежности LLM-систем, применяемых для анализа безопасности исходного кода.

Литература

1. Мельников, А. В. Автоматизация оценки вредоносности скриптов на основе больших языковых моделей при расследовании компьютерных инцидентов / А. В. Мельников, И. Н. Ярышкин // Вестник УрФО. Безопасность в информационной сфере. – 2025. – № 4(58). – С. 36-43. – DOI 10.14529/secur250404. – EDN RWTFCR.

2. Нейронные сети для обеспечения безопасности веб-приложений / В. А. Частикова, А. А. Тесленко, М. К. Алиев, И. С. Игнатенко // Вестник УрФО. Безопасность в информационной сфере. – 2025. – № 2(56). – С. 65-73. – DOI 10.14529/secur250207. – EDN UYYZGA.

3. Risse, N. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection / N. Risse, J. Liu, M. Bohme. – Текст: непосредственный // ArXiv. – 2024. – № abs/2408.12986. – URL: https://arxiv.org/abs/2408.12986 (дата обращения: 01.04.2026).

4. Ding, Y. Vulnerability Detection with Code Language Models: How Far Are We? / Y. Ding, Y. Fu, O. Ibrahim [et al.]. – Текст: непосредственный // Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). – 2025. – С. 1729–1741. – DOI: 10.1109/ICSE55347.2025.00038.

5. Zhang, Y. Context and Multi-Features-Based Vulnerability Detection: A Vulnerability Detection Frame Based on Context Slicing and Multi-Features / Y. Zhang, Y. Hu, X. Chen. – Текст: непосредственный // Sensors. – 2024. – Т. 24, № 5. – Ст. 1351. – DOI: 10.3390/s24051351.

6. Li, Z. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities / Z. Li, S. Dutta, M. Naik. – Текст: непосредственный // ArXiv. – 2024. – № abs/2405.17238. – URL: https://arxiv.org/abs/2405.17238 (дата обращения: 02.04.2026).

7. Du, X. Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG / X. Du, G. Zheng, K. Wang [et al.]. – Текст: непосредственный // ArXiv. – 2024. – № abs/2406.11147. – URL: https://arxiv.org/abs/2406.11147 (дата обращения: 07.04.2026).

8. Wen, X.-C. VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection / X.-C. Wen, X. Wang, Y. Chen [et al.]. – Текст: непосредственный // ArXiv. – 2024. – № abs/2404.15596. – URL: https://arxiv.org/abs/2404.15596 (дата обращения: 07.04.2026).

9. Li, Y. Beyond Function-Level Analysis: Context-Aware Reasoning for Inter-Procedural Vulnerability Detection / Y. Li, T. Zhang, J. Shi [et al.]. – Текст: непосредственный // ArXiv. – 2026. – № abs/2602.06751. – URL: https://arxiv.org/abs/2602.06751 (дата обращения: 10.04.2026).

10. Lekssays, A. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models / A. Lekssays, H. Mouhcine, K. Tran [et al.]. – Текст: непосредственный // ArXiv. – 2025. – № abs/2507.16585. – URL: https://arxiv.org/abs/2507.16585 (дата обращения: 10.04.2026).

11. Wang, C. Boosting Static Resource Leak Detection via LLM-Based Resource-Oriented Intention Inference / C. Wang, J. Liu, X. Peng, Y. Liu, Y. Lou. – Текст: непосредственный // ArXiv. – 2023. – № abs/2311.04448. – URL: https://arxiv.org/abs/2311.04448 (дата обращения: 10.04.2026).

12. Chapman, P. J. Interleaving Static Analysis and LLM Prompting / P. J. Chapman, C. Rubio-Gonzalez, A. V. Thakur. – Текст: непосредственный // Proceedings of the 13th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis (SOAP). – New York: ACM, 2024. – С. 9–17. – DOI: 10.1145/3652588.3663317.

13. Yildiz, A. Benchmarking LLMs and LLM-Based Agents in Practical Vulnerability Detection for Code Repositories / A. Yildiz, S. G. Teo, Y. Lou [et al.]. – Текст: непосредственный // ArXiv. – 2025. – № abs/2503.03586. – URL: https://arxiv.org/abs/2503.03586 (дата обращения: 14.04.2026).

14. Guo, J. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing / J. Guo, C. Wang, X. Xu, Z. Su, X. Zhang. – Текст: непосредственный // Proceedings of the 42nd International Conference on Machine Learning. – PMLR, 2025. – Т. 267. – С. 21083–21100.

15. Li, Y. CleanVul: Automatic Function-Level Vulnerability Detection in Code Commits Using LLM Heuristics / Y. Li, T. Zhang, R. Widyasari [et al.]. – Текст: непосредственный // ArXiv. – 2024. – № abs/2411.17274. – URL: https://arxiv.org/abs/2411.17274 (дата обращения: 17.04.2026).

16. Ahmed, M. B. U. SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection / M. B. U. Ahmed, N. S. Harzevili, J. Shin, H. V. Pham, S. Wang. – Текст: непосредственный // ArXiv. – 2025. – № abs/2505.19828. – URL: https://arxiv.org/abs/2505.19828 (дата обращения: 17.04.2026).

17. Li, Yue. Everything You Wanted to Know About LLM-Based Vulnerability Detection But Were Afraid to Ask / Yue Li, Xiao Li, H. Wu [et al.]. – Текст: непосредственный // ArXiv. – 2025. – № abs/2504.13474. – URL: https://arxiv.org/abs/2504.13474 (дата обращения: 16.04.2026).

18. Zhang, X. GitPatchDB: A Large-Scale GitHub Commit Databank for Vulnerability Patch Analysis / X. Zhang, P. He, X. Jin, Y. Kao, S. Chen. – Текст: непосредственный // OpenReview. – 2025. – URL: https://openreview.net/forum?id=tX24AtTHyg (дата обращения: 16.04.2026).

19. Wang, X. ReposVul: A Repository-Level High-Quality Vulnerability Dataset / X. Wang, R. Hu, C. Gao [et al.]. – Текст : непосредственный // ArXiv. – 2024. – № abs/2401.13169. – URL: https://arxiv.org/abs/2401.13169 (дата обращения: 17.04.2026).

20. Guo, Y. A Comprehensive Analysis on Software Vulnerability Detection Datasets: Trends, Challenges, and Road Ahead / Y. Guo, S. Bettaieb, F. Casino. – Текст : непосредственный // International Journal of Information Security. – 2024. – Т. 23, № 5. – С. 3311–3327. – DOI: 10.1007/s10207-024-00888-y.

21. Yamaguchi, F. Modeling and Discovering Vulnerabilities with Code Property Graphs / F. Yamaguchi, N. Golde, D. Arp, K. Rieck. – Текст: непосредственный // 2014 IEEE Symposium on Security and Privacy. – San Jose: IEEE, 2014. – С. 590–604. – DOI: 10.1109/SP.2014.44.

22. Bhandari, G. P. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software / G. P. Bhandari, A. Naseer, L. Moonen. – Текст: непосредственный // Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2021). – New York: ACM, 2021. – Ст. 8. – 10 с. – DOI: 10.1145/3475960.3475985.

References

1. Mel'nikov, A. V. Avtomatizaciya ocenki vredonosnosti skriptov na osnove bol'shih yazykovyh modelej pri rassledovanii komp'yuternyh inci-dentov / A. V. Mel'nikov, I. N. YAryshkin // Vestnik UrFO. Bezopasnost' v informacionnoj sfere. – 2025. – № 4(58). – S. 36-43. – DOI 10.14529/secur250404. – EDN RWTFCR.

2. Nejronnye seti dlya obespecheniya bezopasnosti veb-prilozhenij / V. A. CHastikova, A. A. Teslenko, M. K. Aliev, I. S. Ignatenko // Vestnik UrFO. Bezopasnost' v informacionnoj sfere. – 2025. – № 2(56). – S. 65–73. – DOI 10.14529/secur250207. – EDN UYYZGA.

3. Risse N., Liu J., Bohme M. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. arXiv:2408.12986, 2024. Available at: https://arxiv.org/abs/2408.12986

4. Ding Y., Fu Y., Ibrahim O., Sitawarin C., Chen X., Alomair B., Wagner D., Ray B., Chen Y. Vulnerability Detection with Code Language Models: How Far Are We? Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). 2025, pp. 1729–1741. DOI: 10.1109/ICSE55347.2025.00038

5. Zhang Y., Hu Y., Chen X. Context and Multi-Features-Based Vulnerability Detection: A Vulnerability Detection Frame Based on Context Slicing and Multi-Features. Sensors, 2024, vol. 24, no. 5, article 1351. DOI: 10.3390/s24051351

6. Li Z., Dutta S., Naik M. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv:2405.17238, 2024. Available at: https://arxiv.org/abs/2405.17238

7. Du X., Zheng G., Wang K., Feng J., Deng W., Liu M., Chen B., Peng X., Ma T., Lou Y. Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG. arXiv:2406.11147, 2024. Available at: https://arxiv.org/abs/2406.11147

8. Wen X.-C., Wang X., Chen Y., Hu R., Lo D., Gao C. VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection. arXiv:2404.15596, 2024. Available at: https://arxiv.org/abs/2404.15596

9. Li Y., Zhang T., Shi J., Yang C., He J., Zhou X., Jiang J., Huang H., Leow W.B., Yin Y., Ouh E.L., Shar L.K., Lo D. Beyond Function-Level Analysis: Context-Aware Reasoning for Inter-Procedural Vulnerability Detection. arXiv:2602.06751, 2026. Available at: https://arxiv.org/abs/2602.06751

10. Lekssays A., Mouhcine H., Tran K., Yu T., Khalil I. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. arXiv:2507.16585, 2025. Available at: https://arxiv.org/abs/2507.16585

11. Wang C., Liu J., Peng X., Liu Y., Lou Y. Boosting Static Resource Leak Detection via LLM-Based Resource-Oriented Intention Inference. arXiv:2311.04448, 2023. Available at: https://arxiv.org/abs/2311.04448

12. Chapman P.J., Rubio-Gonzalez C., Thakur A.V. Interleaving Static Analysis and LLM Prompting. Proceedings of the 13th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis (SOAP). New York, ACM, 2024, pp. 9–17. DOI: 10.1145/3652588.3663317

13. Yildiz A., Teo S.G., Lou Y., Feng Y., Wang C., Divakaran D.M. Benchmarking LLMs and LLM-Based Agents in Practical Vulnerability Detection for Code Repositories. arXiv:2503.03586, 2025. Available at: https://arxiv.org/abs/2503.03586

14. Guo J., Wang C., Xu X., Su Z., Zhang X. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. Proceedings of the 42nd International Conference on Machine Learning. PMLR, 2025, vol. 267, pp. 21083–21100

15. Li Y., Zhang T., Widyasari R., Tun Y.N., Nguyen H.H., Bui T., Irsan I.C., Cheng Y., Lan X., Ang H.W., Liauw F., Weyssow M., Kang H.J., Ouh E.L., Shar L.K., Lo D. CleanVul: Automatic Function-Level Vulnerability Detection in Code Commits Using LLM Heuristics. arXiv:2411.17274, 2024. Available at: https://arxiv.org/abs/2411.17274

16. Ahmed M.B.U., Harzevili N.S., Shin J., Pham H.V., Wang S. SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection. arXiv:2505.19828, 2025. Available at: https://arxiv.org/abs/2505.19828

17. Yue Li, Xiao Li, Wu H., Xu M., Zhang Y., Cheng X., Xu F., Zhong S. Everything You Wanted to Know About LLM-Based Vulnerability Detection But Were Afraid to Ask. arXiv:2504.13474, 2025. Available at: https://arxiv.org/abs/2504.13474

18. Zhang X., He P., Jin X., Kao Y., Chen S. GitPatchDB: A Large-Scale GitHub Commit Databank for Vulnerability Patch Analysis. OpenReview, 2025. Available at: https://openreview.net/forum?id=tX24AtTHyg

19. Wang X., Hu R., Gao C., Wen X.-C., Chen Y., Liao Q. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. arXiv:2401.13169, 2024. Available at: https://arxiv.org/abs/2401.13169

20. Guo Y., Bettaieb S., Casino F. A Comprehensive Analysis on Software Vulnerability Detection Datasets: Trends, Challenges, and Road Ahead. International Journal of Information Security, 2024, vol. 23, no. 5, pp. 3311–3327. DOI: 10.1007/s10207-024-00888-y

21. Yamaguchi F., Golde N., Arp D., Rieck K. Modeling and Discovering Vulnerabilities with Code Property Graphs. 2014 IEEE Symposium on Security and Privacy. San Jose, IEEE, 2014, pp. 590–604. DOI: 10.1109/SP.2014.44

22. Bhandari G.P., Naseer A., Moonen L. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2021). New York, ACM, 2021, article 8, 10 p. DOI: 10.1145/3475960.3475985

ГЛАДКИХ Кирилл Игоревич, аспирант, Школа компьютерных наук, Тюменский государственный университет. 625003. г. Тюмень, ул. Володарского, д. 6. E-mail: boombarah@gmail.com.

ШАБАЛИН Андрей Михайлович, кандидат педагогических наук, доцент, доцент академического департамента, Школа компьютерных наук, Тюменский государственный университет. 625003. г. Тюмень, ул. Володарского, д. 6. E-mail: sham.omsk@gmail.com.

ЗАХАРОВ Александр Анатольевич, доктор технических наук, профессор, Школа компьютерных наук, Тюменский государственный университет. 625003. г. Тюмень, ул. Володарского, д. 6. E-mail: a.a.zakharov@utmn.ru.

GLADKIKH Kirill Igorevich, post-graduate student, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarsky St., 6. E-mail: boombarah@gmail.com.

SHABALIN Andrey Mihailovich, Candidate of Pedagogical Sciences, Associate Professor, Associate Professor of the Academic Department, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarsky St., 6. E-mail: sham.omsk@gmail.com.

ZAKHAROV Alexander Anatolievich, Doctor of Technical Sciences, Professor, School of Computer Science, Tyumen State University. 625003, Tyumen, Volodarsky St., 6. E-mail: a.a.zakharov@utmn.ru.