УДК 004.89
doi:10.31799/1684-8853-2025-4-58-70
EDN: XJDOLW

Научные статьи
Articles

Метод определения уязвимостей программного кода на основе кластерного анализа и контекстной адаптации больших языковых моделей

Р. Н. Бакеев[а], *руководитель центра, orcid.org/0009-0008-3149-3836*
В. Н. Кузьмин[б], *доктор воен. наук, ведущий научный сотрудник, orcid.org/0000-0002-6411-4336, vka@mil.ru*
А. Б. Менисов[б], *доктор техн. наук, старший преподаватель, orcid.org/0000-0002-9955-2694*
Т. Р. Сабиров[б], *канд. техн. наук, старший преподаватель, orcid.org/0000-0002-6807-2954*
[а]*Фонд перспективных исследований, Бережковская наб., 22, стр. 3, Москва, 121059, РФ*
[б]*Военно-космическая академия им. А. Ф. Можайского, Ждановская наб., 13, Санкт-Петербург, 197198, РФ*

Введение: *обнаружение уязвимостей в исходном коде остается одной из приоритетных задач в области обеспечения кибербезопасности. Классические методы анализа кода часто не учитывают контекст исполнения и слабо масштабируются при росте объема программ. В условиях сложных архитектур и неполных аннотаций данных требуется контекстно-адаптированный подход, способный выявлять уязвимости на основе семантического и структурного анализа.* ***Цель:*** *разработать метод обнаружения уязвимостей в программном коде с использованием кластерного анализа и контекстной адаптации больших языковых моделей, способных учитывать не только синтаксис, но и семантическую структуру программ.* ***Методы:*** *разработанный подход сочетает кластеризацию кодовых сегментов с применением предварительно дообученных языковых моделей, адаптированных к программному коду. Для повышения результативности используется выделение признаков уязвимости, включающих как непосредственно уязвимый фрагмент, так и его контекст — управляющие конструкции, переменные, вызовы функций. Обучение и оценка проводились на размеченных открытых датасетах с использованием предобученных больших языковых моделей.* ***Результаты:*** *метод позволяет автоматически группировать фрагменты кода по структурному сходству, после чего осуществляется семантический анализ с помощью больших языковых моделей, способных распознавать шаблоны уязвимостей. Эксперименты показали, что включение контекстной информации существенно повышает эффективность определения уязвимостей в исходном коде. На датасетах BigVul и CVEfixes предложенный метод достиг точности до 78 % и полноты 82 %, что на 9–12 % выше по сравнению с существующими решениями. Метод демонстрирует устойчивость к синтаксическим вариациям и может быть использован для анализа ранее неразмеченного кода.* ***Практическая значимость:*** *метод применим в системах автоматического анализа исходного кода и способен значительно сократить затраты на ручной аудит, особенно в условиях анализа больших кодовых баз. Также он может использоваться в образовательных и исследовательских целях для анализа паттернов уязвимостей.* ***Обсуждение:*** *результаты подтверждают эффективность использования больших языковых моделей в задачах анализа безопасности программ. Перспективным направлением является расширение метода на другие языки программирования, а также исследование гибридных подходов с участием графовых нейросетей. Открытым остается вопрос аргументации решений модели и автоматического объяснения причин классификации сегмента как уязвимого.*

Ключевые слова *— информационная безопасность, обнаружение уязвимостей, большие языковые модели, кластерный анализ, машинное обучение.*

Для цитирования: Бакеев Р. Н., Кузьмин В. Н., Менисов А. Б., Сабиров Т. Р. Метод определения уязвимостей программного кода на основе кластерного анализа и контекстной адаптации больших языковых моделей. *Информационно-управляющие системы*, 2025, № 4, с. 58–70. doi:10.31799/1684-8853-2025-4-58-70, EDN: XJDOLW
For citation: Bakeev R. N., Kuzmin V. N., Menisov A. B., Sabirov T. R. Method for identifying software code vulnerabilities based on cluster analysis and contextual adaptation of large language models. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2025, no. 4, pp. 58–70 (In Russian). doi:10.31799/1684-8853-2025-4-58-70, EDN: XJDOLW

Введение

Уязвимости программного обеспечения требуют значительных усилий по их обнаружению и устранению [1], так как представляют существенные проблемы в информационной безопасности [2]. Они имеют критическое значение, поскольку не устраненные уязвимости могут потенциально привести к скомпрометированной информационной инфраструктуре и огромным экономическим потерям [3]. Точное и быстрое выявление, а самое главное, обоснованное определение уязвимостей имеет решающее значение для снижения рисков [4] и привлекает значительное внимание со стороны ученых, разработчиков информационных систем и специалистов информационной безопасности [5, 6].

Обнаружение уязвимостей программного кода критично на всех этапах разработки безопасного программного обеспечения и в основном базируется на положениях динамических и статических подходов [7]. Динамические методы трудозатратны и направлены на выявление уязвимостей путем выполнения кода и наблюдения

за выводом или внутренними состояниями программы [8]. При реализации статического подхода анализ кода проводят без его выполнения, используя анализ синтаксиса, потока данных и управления. Инструменты статического тестирования безопасности приложений популярны благодаря своей низкой стоимости, быстрой работе и способности находить ошибки без запуска программы. Однако эти методы имеют главный недостаток — высокую долю ложноположительных результатов.

Применение моделей машинного обучения преимущественно относится к статическому анализу, который в последние годы стал ключевым направлением исследований [9]. Помимо глубокого обучения, недавние достижения в области больших языковых моделей (БЯМ) значительно повлияли на обнаружение уязвимостей программного обеспечения. Продвинутые методы, такие как дополненный поиск [10], контекстное обучение на примерах [11] и тонкая настройка моделей машинного обучения [12], уже использовались для обнаружения уязвимостей [7]. Вводя цепочки инструкций (промтов), основанные на семантической структуре кода, БЯМ могут достичь более высокой результативности обнаружения и обладают способностью рассуждения об уязвимостях. Хотя был достигнут значительный прогресс в применении методов глубокого обучения для обнаружения уязвимостей [13–15], эффективное использование этих методов для выявления и объяснения уязвимостей остается сложной проблемой. В настоящее время лишь ограниченное количество исследований фокусируется на объяснительных возможностях моделей, основанных на глубоком обучении, в отношении уязвимостей [16].

Анализ научно-технических реализаций

Большие языковые модели в последние годы находят широкое применение в различных сферах экономики, включая автоматизацию бизнеса [17], анализ данных [18] и их безопасность [19]. В области информационной безопасности они используются для автоматизированного анализа угроз [20], распознавания вредоносных программ и обнаружения уязвимостей исходного кода [21]. Решения, основанные на БЯМ, продемонстрировали свою эффективность для обнаружения уязвимостей программного обеспечения [22, 23]. Применяемые БЯМ — это модели машинного обучения, включающие большое количество параметров и предварительно обученные на огромных объемах исходного кода, текста и других типах данных. В последнее время наблюдается значительное увеличение размера обучающих данных и параметров БЯМ [24]. Однако эффективность их применения в этих задачах зависит от способности адаптации к специфическим контекстам программирования и анализа безопасности. Даже современные БЯМ, такие как GPT-4 [25] и Code Llama [26], демонстрируют высокую точность при анализе программного кода, но требуют адаптации к конкретным доменам и типам уязвимостей.

В области развития БЯМ активно развиваются подходы, направленные на раскрытие внутренней логики вывода [27, 28] и объяснения классификационных решений в программном коде через контекстные зависимости переменных и вызовов [29].

Значительным достижением в обнаружении уязвимостей на основе БЯМ является эмуляция процессов экспертного рассуждения человека [30], т. е. БЯМ следует пошаговой схеме при анализе кода на наличие уязвимостей. Этот процесс гарантирует, что модели сначала генерируют цепочку рассуждений, прежде чем принимать решения о потенциальных уязвимостях. Дополнительно, используя контекстное обучение с несколькими примерами, реализуя возможности БЯМ, можно повысить результативность обнаружения уязвимостей, совершенствуя способность модели объяснять признаки для обнаружения, делая результаты более аргументированными и заслуживающими доверия. Пошаговое рассуждение также улучшает контекстное понимание фрагментов кода. Вместо того чтобы рассматривать код как изолированные сегменты, БЯМ способны анализировать взаимосвязи между различными компонентами и повышать контекст уязвимостей [31]. Это целостное понимание имеет решающее значение для обнаружения сложных уязвимостей, которые охватывают несколько функций или модулей.

Несмотря на значительные достижения в методологиях обнаружения уязвимостей, сохраняется ряд трудностей в обучении и оценивании этих систем.

1. Эффективность систем обнаружения уязвимостей, под которой в данной работе понимается совокупность таких метрик, как точность (precision), полнота (recall), F1-мера и уровень ложноположительных срабатываний (false positive rate), существенно зависит от характеристик используемых обучающих и тестовых наборов данных [32, 33].

Доказано, что даже высокоточные алгоритмы значительно теряют в производительности при обучении на плохо размеченных наборах данных [34]. Поэтому при разработке эффективных методов выявления уязвимостей необходим особый акцент на отбор, препроцессинг и валидацию

исходных датасетов. Эти проблемы с качеством данных обусловливают важность адекватного оценивания текущего состояния практики обнаружения уязвимостей и подчеркивают необходимость улучшать стратегии проверки наборов данных.

2. Многие открытые наборы данных в предметной области обнаружения уязвимостей имеют значительный дисбаланс классов, с гораздо большим количеством отрицательных примеров (безопасный код), чем положительных (уязвимый код). Устранение этого дисбаланса требует специализированных подходов для повышения эффективности обнаружения уязвимостей в программном коде от алгоритмических решений до дополнения наборов данных. Однако существуют и специализированные датасеты, включающие преимущественно или исключительно уязвимые прецеденты, например gitbug-java [35].

Поскольку разнообразие уязвимостей будет только возрастать, привлекательной и перспективной для постоянных исследований и разработок будет задача совершенствования методов обнаружения уязвимостей. Таким образом, необходимо сосредоточить усилия на устранении недостатков качества наборов данных, обработки дисбаланса классов и разработки гибридных подходов, которые объединяют сильные стороны различных методологий.

Этапы метода

В разработанном методе рассматриваются две задачи анализа уязвимостей программного обеспечения: 1) обнаружение уязвимости (обнаружение потенциальных уязвимостей в образцах кода без указания их типов); 2) идентификация уязвимости (имеет ли предоставленный образец кода определенный тип уязвимости). Эти задачи охватывают различные уровни анализа уязвимости, представляя собой ключевые шаги в защите от уязвимостей программного обеспечения. Следовательно, анализ кода должен гарантированно определять уязвимости и их типы.

Задача обнаружения уязвимостей формулируется как бинарная классификация. Пусть имеется множество фрагментов программного кода C. Целью является создание детектора $M{:}C \to V$ с выделением подмножества потенциально уязвимых фрагментов $V = \{0, 1\}$, где 1 обозначает уязвимый сегмент кода, а 0 обозначает безопасный сегмент.

Задача идентификации типа уязвимости направлена на определение того, имеет ли данный образец кода особенный тип уязвимости (например, CWE). Для этого в исходном коде Pr необходимо провести предварительную обработку кода и его отнесение к кластеру уязвимого. Учитывая, что исходный код Pr состоит из n фрагментов: $C = \{c_1, c_2, \ldots, c_n\}$, каждый фрагмент c_i, представленный как вектор (эмбенддинг) $x_i \in R$, должен быть нормализован $||x_i|| = 1$, для чего используется метрика сходства — косинусного, евклидового или Махалонобиса — в качестве метрики расстояния для других сегментов.

Таким образом, в основе предлагаемого разработанного метода лежит гипотеза о сходстве уязвимых фрагментов кода (рис. 1). Для ее проверки реализуется двухступенчатый подход, включающий кластерный анализ и использование БЯМ с контекстной адаптацией.

Целью кластеризации является определение n фрагментов исходного кода в кластеры путем минимизирования внутрикластерной дисперсии:

$$\arg\min \sum_{i=1}^{m} \sum_{c_j \in k_i} \left\| x_i - \mu_i \right\|^2, \qquad (1)$$

где μ_i — центроид кластера уязвимых фрагментов кода. Это позволяет группировать потенциально уязвимые участки, даже если они находятся в разных частях программы.

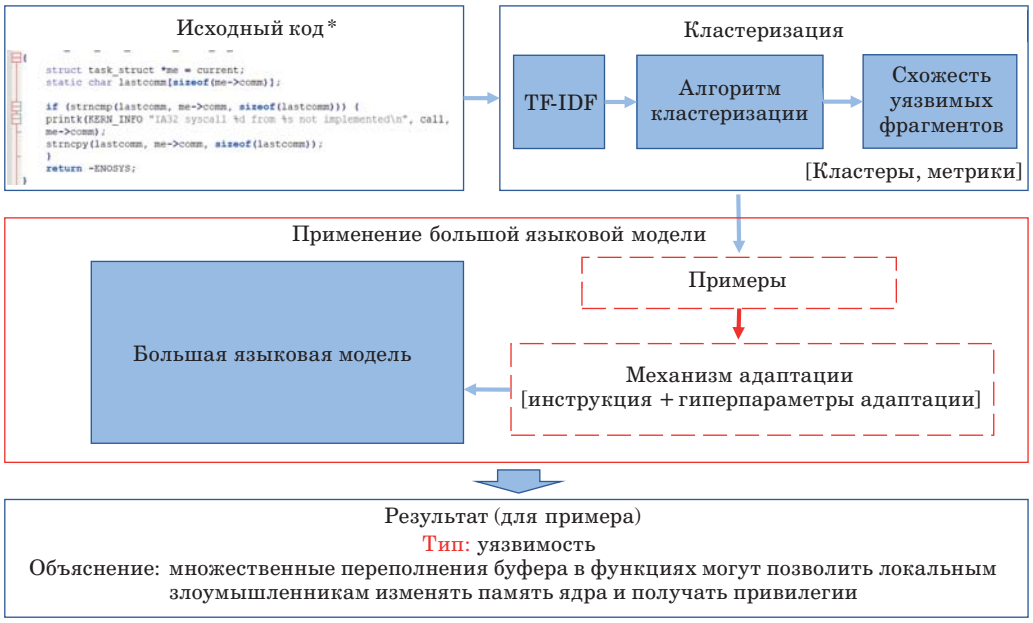
При определении принадлежности к кластеру каждому фрагменту кода присваивается значение метрики к ближайшему центроиду на основе наименьшего расстояния и обновляются центроиды соответствующим образом до тех пор, пока не произойдет схождение после T итераций.

Основная цель применения БЯМ для определения уязвимостей в программном коде — сформировать оптимальный набор центроидов кластеров уязвимых фрагментов кода и итеративно их корректировать для минимизации внутрикластерной дисперсии. Это связано со следующими особенностями выявления уязвимостей в программном коде:

1) хотя программа может включать в себя множество строк кода, только небольшая часть может быть подвержена уязвимостям. Следовательно, сосредоточение на этих критических фрагментах, составляющих семантику уязвимости, позволяет БЯМ эффективнее проводить анализ уязвимости;

2) большинство БЯМ основаны на трансформерах, которые, как известно, демонстрируют сниженную производительность при обработке/генерации обширного текста. Поэтому сокращение длины контекста позволяет улучшить сосредоточение на семантике уязвимости.

Использование БЯМ для увеличения признакового пространства уязвимости в программном коде заключается не только в использовании тестового образца программного кода, но и в фор-



*Пример для CVE_2004_1151.

■ ***Рис. 1.*** Общая схема метода определения уязвимостей программного кода на основе кластерного анализа и контекстной адаптации БЯМ

■ ***Fig. 1.*** General scheme of the method for determining vulnerabilities in program code based on cluster analysis and contextual adaptation of large language models

мировании набора из нескольких образцов уязвимостей кода, которые предоставляют примеры для БЯМ в целях выполнения контекстной адаптации обучения.

Каждый сегмент кода c_i анализируется в контексте окружающих утверждений $T_i = \{t_{i-1}, t_i, t_{i+1}, \ldots, t_{i+n}\}$. Это могут быть логически связанные строки, функции, блоки кода, комментарии и др., а совместный вход в языковую модель задается как $x_i = \text{concat}(t_{i-1}, t_i, t_{i+1}, \ldots, t_{i+n})$.

Языковая модель M преобразует вход в контекстно-зависимый эмбеддинг $h_i = M(x_i)$, а оценка вероятности уязвимости производится через функцию классификации:

$$p_i = \sigma\left(w^T h_i + b\right); \quad (2)$$

$$\hat{y}_i = \begin{cases} 1, & p_i \geq \tau \\ 0, & p_i < \tau \end{cases}, \quad (3)$$

где $p_i \in [0, 1]$; σ — сигмоида; w, b — параметры линейного слоя; порог τ определяет, считать ли фрагмент кода уязвимым.

Для каждого фрагмента кода, попавшего в кластер, применяется БЯМ, которая анализирует не только сам фрагмент, но и его окружение — контекстуальные утверждения, влияющие на семантику: $x_1 = (c_i, T_i)$. Это позволяет уточнить вероятность наличия уязвимости, основываясь не только на структуре, но и на смысле кода. В частности, каждый образец состоит из инструкции, примера кода и шагов рассуждения. Итоговая функция предсказания объединяет структурные признаки из кластеров C и семантический контекст $\hat{y}_i = F\left(V, h_i, x_i\right)$.

Для примера с рис. 1 рассмотрим следующий фрагмент исходного кода ядра Linux:

```
struct task_struct *me = current;
static char lastcomm[sizeof(me->comm)];
if (strncmp(lastcomm, me->comm, sizeof(lastcomm))) {
printk(KERN_INFO "IA32 syscall %d from %s not implemented\n", call, me->comm);
strncpy(lastcomm, me->comm, sizeof(lastcomm));
}
return -ENOSYS;
```

Метод обрабатывает данный код поэтапно.

1. Предварительный анализ кода, при котором извлекаются имена переменных, вызовы функций и операторы. Фрагмент представляется в виде векторного пространства, которое далее поддается кластерному анализу. Кластеризация позволяет определить, принадлежит ли данный фрагмент к подмножеству ранее размеченных уязвимых фрагментов. Данный код входит в кластер, характеризующийся следующими признаками:

— использование фиксированных буферов (*char lastcomm*[...]);

— копирование данных из структуры (*me->comm*);

— повторное использование буфера без инициализации между итерациями.

Эти признаки указывают на структурное сходство с известными уязвимостями, связанными с переполнением буфера.

2. Большая языковая модель получает кодовый фрагмент с расширенным контекстом как в пределах функции, так и вне ее (включая структуры, переменные и комментарии близких в кластере примеров кода) и делает вывод, что потенциальная уязвимость связана с тем, что *me->comm* может быть не *null-terminated* и использование *strncpy* без явной защиты ведет к риску записи за границы буфера *lastcomm*.

Экспериментальные исследования

Для проведения экспериментов было разработано программное обеспечение, реализующее предложенный метод обнаружения уязвимостей. Реализация выполнена на языке Python и развернута на выделенном сервере с поддержкой Docker-контейнеризации в целях обеспечения изоляции процессов и возможности масштабирования. Использовалась модульная архитектура, предусматривающая запуск БЯМ как микросервисов с REST API, что обеспечивает параллельную обработку нескольких запросов (обработку программных кодов) и многопользовательский доступ. Среднее время анализа одного проекта (до 150 строк кода) составляет от 12 до 15 с, включая этапы предобработки, кластерного анализа (кластеры в формате .pkl) и вывода БЯМ. Код решения и инструкция по воспроизведению экспериментов будут опубликованы после государственной регистрации интеллектуальной собственности.

Исходные данные. В исследовании на разных этапах использовались разные наборы данных. На этапе адаптации БЯМ применялся BigVul [36], а для проверки решений — CVEfixes [37].

BigVul — набор данных об уязвимостях кода C/C++ из проектов с открытым исходным кодом Github. Всего набор данных содержит 217 007 записей об изменениях кода (извлеченных из исправлений зафиксированных версий). Все записи разделены на уязвимые (если в функции были изменены строки с дефектами) и неуязвимые функции.

Второй набор данных CVEfixes создан на основе Национальной базы данных уязвимостей США. В первоначальном выпуске набор данных охватывает все опубликованные CVE по состоянию на 9 июня 2021 года. Набор данных организован как реляционная база и охватывает 5495 коммитов по исправлению уязвимостей в 1754 проектах с открытым исходным кодом для всего 5365 CVE в 180 различных типах CWE. Набор данных включает исходный код до и после исправления 18 249 файлов и 50 322 функций.

Предобработка исходного кода включает в себя конвертирование в векторное пространство с помощью статистической меры TF-IDF [38], которая отражает значимость уязвимого сегмента кода v_i относительно коллекции программ C:

$$TF - IDF\left(v_j,\, c_i,\, C\right) = TF\left(v_j,\, c_i\right) \cdot IDF\left(v_j,\, C\right); \quad (4)$$

$$TF\left(v_j,\, c_i\right) = \frac{\text{Количество уязвимостей } v_i \text{ в сегменте кода } c_i}{\text{Общее количество сегментов в исходном коде}}; \quad (5)$$

$$IDF\left(v_j,\, C\right) = \log\left(\frac{\text{Общее количество исходных кодов в наборе}}{\text{Количество сегментов, содержащих уязвимость } v_j}\right). \quad (6)$$

Алгоритм кластеризации данных, содержащих уязвимый код. Существует множество алгоритмов кластеризации. Они различаются в первую очередь тем, как они измеряют «сходство» или «близость», и с какими типами признаков они работают. Алгоритм k-средних [39] понятен и прост в применении в контексте выявления признаков уязвимостей программного кода. Кластеризация k-средних создает кластеры, помещая центроиды внутрь пространства признаков. Каждая точка в наборе данных назначается кластеру того центроида, к которому она ближе всего. k — это количество центроидов (т. е. кластеров).

Для набора данных BigVul, используемого для валидации, количество кластеров соответствует категории CWE, а определение близости сегментов кода начинается с расчета метрики до центроида всех CWE. Пример расчета по метрике Махалонобиса представлен в табл. 1 для сегмента кода

```
def vulnerable_function(user_input):
eval(user_input) # Потенциальная уязвимость инъекции кода
```

Используемые большие языковые модели. Табл. 2 содержит информацию о трех БЯМ, использованных в эксперименте.

Инструкции для больших языковых моделей (Приложение) включают три варианта: ролевой запрос, последовательность запросов с контекстной адаптацией БЯМ и запрос на основе кластерного анализа и контекстной адаптации БЯМ.

■ ***Таблица 1.*** Пример определения близости сегментов кода к центроидам уязвимых сегментов кода
■ ***Table 1.*** Example of determining the proximity of code segments to the centroids of vulnerable code segments

№ п/п	ID CWE	Метрика схожести	№ п/п	ID CWE	Метрика схожести	№ п/п	ID CWE	Метрика схожести	№ п/п	ID CWE	Метрика схожести
1	119	0,0320	12	22	0,4360	23	369	0,2455	34	668	0,0000
2	125	0,0357	13	254	0,0487	24	399	0,0183	35	674	0,4075
3	134	0,4255	14	264	0,0354	25	400	0,2434	36	704	0,2191
4	16	0,0000	15	269	0,4345	26	404	0,4912	37	732	0,1006
5	17	0,0000	16	284	0,0999	27	415	0,1709	38	74	0,0000
6	172	0,0000	17	287	0,5000	28	416	0,0305	39	77	0,4112
7	189	0,0403	18	295	0,0000	29	476	0,0291	40	772	0,0996
8	19	0,1784	19	310	0,1067	30	502	0,0000	41	787	0,1013
9	190	0,0651	20	311	0,4628	31	59	0,2579	42	79	0,2453
10	20	0,0244	21	347	0,0000	32	611	0,4404	43	834	0,4382
11	200	0,0401	22	362	0,1044	33	617	0,2368	44	835	0,4531

■ ***Таблица 2.*** Используемые большие языковые модели
■ ***Table 2.*** Large language models used

Название и версия	Организация и год анонсирования	Размер (количество параметров)	Размер контекста (количество токенов)
Meta-Llama-3-8B-Instruct	Meta, 18 апреля 2024 г.	8 млрд	128 000
Mistral-7B-Instruct-v0.3	Mistral AI SAS, 22 мая 2024 г.	7 млрд	8000–128 000
SOLAR-10.7B-Instruct-v1.0	Upstage AI, 13 декабря 2023 г.	10,7 млрд	128 000

Метрики оценивания определения уязвимостей в исходном коде. Для оценки качества обнаружения уязвимостей в исходном коде применяются классические метрики из области машинного обучения и анализа данных. Основными являются точность, полнота и F-мера, вычисляемые из матрицы несоответствия (confusion matrix).

Матрица несоответствия представляет собой таблицу, в которой фиксируются результаты сравнения предсказаний решения с истинными метками. Для задачи бинарной классификации (уязвимый/неуязвимый код) матрица включает следующие элементы:

	Фактически уязвимый	Фактически неуязвимый
Предсказано уязвимый	True Positive (TP)	False Positive (FP)
Предсказано неуязвимый	False Negative (FN)	True Negative (TN)

где:

– TP (истинно положительное): правильно определенный уязвимый участок кода;

– FP (ложно положительное): ошибочно помеченный безопасный участок как уязвимый;

– FN (ложно отрицательное): не обнаружена существующая уязвимость;

– TN (истинно отрицательное): корректно определено, что участок безопасен.

Точность показывает долю правильно предсказанных уязвимых участков среди всех участков, классифицированных как уязвимые:

$$\text{Точность} = \frac{TP}{TP + FP}. \quad (7)$$

Высокая точность характеризует число ложных срабатываний.

Полнота показывает, какую долю всех реально уязвимых участков модель смогла обнаружить:

$$\text{Полнота} = \frac{TP}{TP + FN}. \quad (8)$$

Высокая полнота означает, что модель способна выявлять большинство уязвимостей, даже если при этом возникают ложные срабатывания.

F-мера является гармоническим средним между точностью и полнотой, обеспечивая баланс между этими двумя характеристиками:

$$F_1 = 2 \cdot \frac{\text{Точность} \cdot \text{Полнота}}{\text{Точность} + \text{Полнота}}. \quad (9)$$

F-мера особенно полезна в случаях, когда необходимо учитывать как количество обнаруженных уязвимостей, так и качество предсказаний.

Результаты экспериментов представлены в табл. 3 и на рис. 2.

■ ***Таблица 3.*** Результаты экспериментов
■ ***Table 3.*** Experimental results

Модель	Режим	Точность	Полнота	F1-мера
Meta-Llama-3-8B-Instruct	РЗ	0,621	0,687	0,652
	ПЗ	0,710	0,769	0,738
	КлА+КА	0,758	0,805	0,781
Mistral-7B-Instruct-v0.3	РЗ	0,604	0,670	0,634
	ПЗ	0,692	0,740	0,712
	КлА+КА	0,749	0,802	0,776
SOLAR-10.7B-Instruct-v1.0	РЗ	0,640	0,698	0,667
	ПЗ	0,734	0,770	0,750
	КлА+КА	0,783	0,826	0,804

РЗ — ролевой запрос; ПЗ — последовательность запросов; (КлА+КА) — запрос на основе кластерного анализа и контекстной адаптации БЯМ.

Обсуждение

Результаты экспериментов позволили сформулировать следующие закономерности:

— все БЯМ способны выявлять уязвимости в коде;

— добавление примеров в запросы улучшает результаты классификации;

— использование наиболее похожих примеров при кластерном анализе и запрос на дополнительный анализ кода повышают эффективность моделей.

Результаты экспериментов доказывают, что комбинация различных техник формирования запросов может повысить качество обнаружения уязвимостей в исходном коде до 80 % и более.

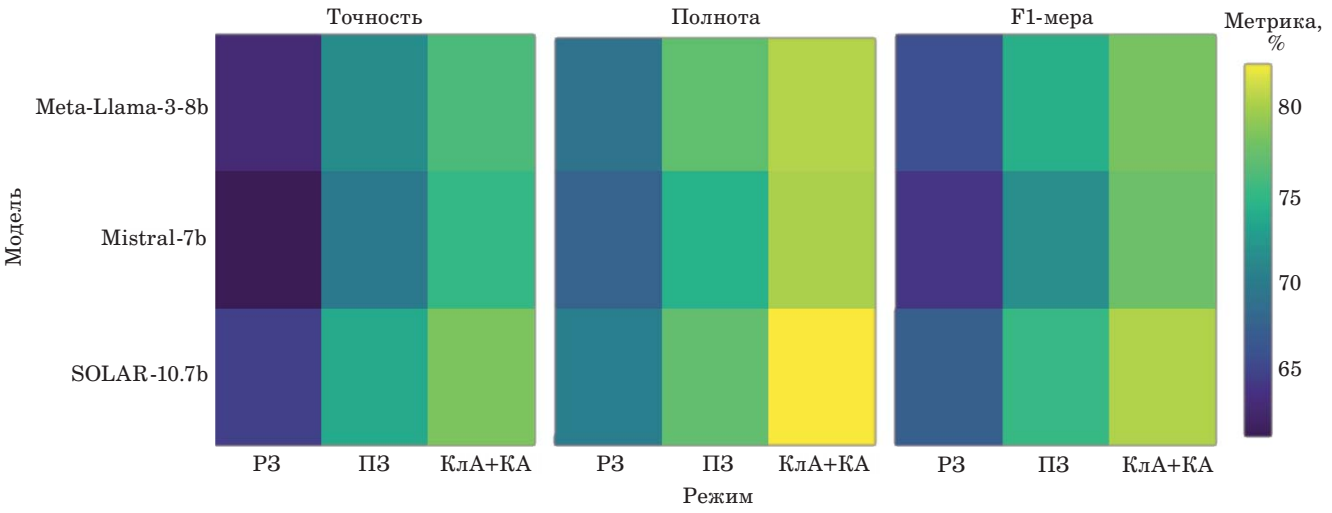
Сравнительный анализ результатов экспериментов позволяет сделать следующие выводы:

— влияние механизмов формирования запросов и использование релевантных примеров улучшают способность моделей выявлять уязвимости, предоставляя контекст и примеры для обучения;

— размер модели влияет на результат, так как более крупные модели, такие как SOLAR-10.7B, обладают большей способностью к обобщению и выявлению сложных уязвимостей, однако и требуют больше вычислительных ресурсов;

— для достижения высокой точности в задачах обнаружения уязвимостей может потребоваться дополнительная настройка моделей на специализированных наборах данных, а также эффективных механизмов учета признаков в этих данных.

Для выделения вклада работы на фоне уже существующих решений был проведен сравнитель-



■ ***Рис. 2.*** Тепловая карта результатов экспериментов
■ ***Fig. 2.*** Heat map of experimental results

ный анализ решений, таких как DeepWukong [40], VulBERTa [41] и GPT-анализаторы-ассистенты [42] (табл. 4).

Для детализации представим отличия предложенного подхода:

— от DeepWukong — вместо строгой привязки к графу потока управления применяется кластеризация кода по структурно-семантическому признаку, что снижает чувствительность к синтаксическим различиям и расширяет применимость к ранее неразмеченному коду, а использование БЯМ позволяет извлекать глобальный контекст, выходя за пределы локального анализа графа;

— от VulBERTa — не требует предварительной доразметки токенов и может применяться к неразмеченному коду, так как сначала происходит группировка схожих по структуре фрагментов, а затем контекстный анализ с помощью БЯМ;

— от GPT-ассистентов — предлагается интеграция контекста на уровне фрагмента кода, что позволяет учесть типовые зависимости (например, инициализацию переменных, вызов функций, структуру блоков).

Преимуществами предложенного метода являются отсутствие необходимости в токенизированной разметке, высокая устойчивость к синтаксическим вариациям и применимость к новому коду без дообучения. При этом указанные существующие решения служат важной основой для развития адаптивности и контекстной чувствительности определения уязвимостей в исходном коде.

■ ***Таблица 4.*** Сравнительный анализ аналогов
■ ***Table 4.*** Comparative analysis of analogues

Критерий	Решение			
	DeepWukong	VulBERTa	GPT-ассистент	Предложенный метод
Механизм	Графовая нейронная сеть	Трансформер-классификатор	Диалоговая БЯМ	Кластеризация + БЯМ
Учет структуры программы	+	–	–	+
Учет семантики	–	+	+	+
Требуется разметка	+	+	–	–
Поддержка контекста	–	–	Частично	+

Дополнительно следует отметить: нельзя с полной уверенностью утверждать, что части датасетов BigVul и CVEfixes не были применены для обучения используемых в эксперименте моделей. Это создает риск искусственного завышения показателей эффективности обнаружения уязвимостей в исходном коде.

Для устранения этих ограничений в дальнейшем планируется исследовать вопросы дообучения БЯМ на верифицированных, заведомо не пересекающихся подмножествах уязвимого и безопасного кода, или использовать темпорально отсеченные данные, обнаруженные после даты обучения модели, или проводить эксперименты на новых, ранее не встречавшихся в обучении уязвимостях из актуальных репозиториев кода.

Заключение

Предложенный метод демонстрирует потенциал предварительной кластеризации фрагментов кода и применения больших языковых моделей в области автоматического обнаружения уязвимостей в исходном коде. Такое сочетание позволяет определять как явно выраженные, так и скрытые шаблоны уязвимостей, особенно в случаях, где локальные признаки недостаточны для точной классификации.

Результаты экспериментов на общедоступных датасетах BigVul и CVEfixes подтверждают эффективность предложенного решения. Так, достигнута точность до 78 % и полнота 82 %, что на 9–12 % превосходит показатели ряда существующих подходов. Преимуществом метода является устойчивость к синтаксическим вариациям кода, а также способность анализировать ранее неразмеченные фрагменты без необходимости ручной верификации.

В процессе исследования также определена значительная зависимость качества выявления уязвимостей от используемой стратегии формирования входных представлений и степени учета контекстной информации. Это подтверждает необходимость дальнейшей адаптации моделей с учетом особенностей языка программирования, а также перспективность применения мультиязыковых решений для повышения универсальности метода.

Будущие направления исследований могут быть в области:

— дообучения БЯМ и использования дополненного поиска для выявления уязвимостей в исходном коде;

— развития специализированных токенизаторов, а также кастомизации эмбеддингов;

— создания специфичных эвристик в фазе кластерного анализа;

— интеграции информации о CWE в запросы для возможного сочетания БЯМ с методами статического анализа;

— использования более крупных и разнообразных датасетов и меньших моделей;

— формирования более продвинутых техник для запросов, таких как самосогласованность (Self-Consistency) [43] и самопознание (Self-Discover) [44];

— оценивания безопасности и функциональности предложенных моделью исправлений кода.

ПРИЛОЖЕНИЕ

Инструкции для больших языковых моделей

Вариант 1. Ролевой запрос

Инструкция: Вы — эксперт по безопасности программного обеспечения. Вам будет предоставлен программный код. Если он содержит какие-либо уязвимости безопасности CWE, напишите Vulnerable. Если код не содержит никаких уязвимостей, напишите Not Vulnerable. Отформатируйте свой ответ как объект JSON с “label” в качестве ключа для статуса уязвимости и “cwe” в качестве номера найденной уязвимости.

Шаблон вывода: Исходный код: ```{code}```, вывод: ”label”.

Вариант 2. Последовательность запросов

Инструкция: Вы — эксперт по безопасности программного обеспечения. Вам будет предоставлен программный код. Если он содержит какие-либо уязвимости безопасности CWE, напишите Vulnerable. Если код не содержит никаких уязвимостей, напишите Not Vulnerable. Отформатируйте свой ответ как объект JSON с “label” в качестве ключа для статуса уязвимости и “cwe” в качестве номера найденной уязвимости.

Продумайте ответ шаг за шагом и отвечайте только с помощью JSON.

Шаблон вывода:
Исходный код: ```{example_0}```
Вывод: {label_0}
Исходный код: ```{example_1}```
Вывод: {label_1}
Исходный код: ```{example_2}```
Вывод: {label_2}
Исходный код: ```{code}```
Конечный вывод: “label”.

Вариант 3. На основе кластерного анализа и контекстной адаптации больших языковых моделей

Вы — эксперт по безопасности программного обеспечения. Вам будет предоставлен программный код. Если он содержит какие-либо уязвимости безопасности CWE, напишите Vulnerable. Если код не содержит никаких уязвимостей, напишите Not Vulnerable.

Проверьте, соответствует ли предоставленная метка (Vulnerable / Not Vulnerable) фактическому состоянию кода.

Если код уязвим:

— Определите конкретные уязвимости в коде.

— Объясните, почему эти уязвимости опасны.

— Опишите потенциальные последствия эксплуатации этих уязвимостей.

Формат ответа:

— **Метка**: ‘{label}’.

— Если уязвим:

1. Опишите уязвимость как ‘{k_mean_example_code}’.

2. Объясните, почему это опасно, как ‘{k_mean_example_label}’.

Литература

1. **Jimmy F. N. U.** Cyber security vulnerabilities and remediation through cloud security tools. *Journal of Artificial Intelligence General Science (JAIGS)*, 2024, vol. 2, no. 1, pp. 129–171. doi:10.60087/jaigs.v2i1.102
2. **Fu M., Tantithamthavorn C., Le T., Kume Y., Nguyen V., Phung D., Grundy J.** AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, 2024, vol. 29, no. 1, p. 4. doi:10.1007/s10664-023-10346-3
3. **Jabbari M., Alibabaei A., Kavousi A., Rezvanjah M.** Vulnerability analysis of the tanks of an oil refinery to fire-induced dom-ino effects based on graph theory. *Iran Occupational Health*, 2021, vol. 18, no. 1, pp. 201–219. doi:10.52547/ioh.18.1.201
4. **Shiri Harzevili N., Boaye Belle A., Wang J., Wang S., Jiang Z. M. (J.), Nagappan N.** A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys*, 2024, vol. 57, no. 3, pp. 1–36. doi:10.1145/3699711
5. **Prümmer J., van Steen T., van den Berg B.** A systematic review of current cybersecurity training methods. *Computers & Security*, 2024, vol. 136, Article 103585. doi:10.1016/j.cose.2023.103585
6. **Camacho N. G.** The role of AI in cybersecurity: Addressing threats in the digital age. *Journal of Artificial Intelligence General Science (JAIGS)*, 2024, vol. 3, no. 1, pp. 143–154. doi:10.60087/jaigs.v3i1.75
7. **Guo Y., Patsakis C., Hu Q., Tang Q., Casino F.** Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection. *European Symposi-*

um on Research in Computer Security, 2024, pp. 271–289. doi:10.1007/978-3-031-70879-4_14

8. **Wang H., Tang Z., Tan S. H., Wang J., Liu Y., Fang H., Xia C., Wang Z.** Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639212
9. **Ferrag M. A., Battah A., Tihanyi N., Jain R., Maimut D., Alwahedi F., Lestable T., Thandi N. S., Mechri A., Debbah M., Cordeiro L. C.** Secure falcon: Are we there yet in automated software vulnerability detection with LLMs? *IEEE Transactions on Software Engineering*, 2025, pp. 1248–1265. doi:10.1109/TSE.2025.3548168
10. **Zhao S., Yang Y., Wang Z., He Z., Qiu L. K., Qiu L.** Retrieval augmented generation (rag) and beyond: A comprehensive survey on how to make your LLMs use external data more wisely. *arXiv preprint arXiv:2409.14924*, 2024. doi:10.48550/arXiv.2409.14924
11. **Patil R., Gudivada V.** A review of current trends, techniques, and challenges in large language models (LLMs). *Applied Sciences*, 2024, vol. 14, no. 5, Article 2074. doi:10.3390/app14052074
12. **Chen L., Varoquaux G.** What is the role of small models in the LLM era: A survey. *arXiv preprint arXiv:2409.06857*, 2024. doi:10.48550/arXiv.2409.06857
13. **Cao S., Sun X., Wu X., Lo D., Bo L., Li B., Liu W.** Coca: Improving and explaining graph neural network-based vulnerability detection systems. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639168
14. **Chu Z., Wan Y., Li Q., Wu Y., Zhang H., Sui Y., Xu G., Jin H.** Graph neural networks for vulnerability detection: A counterfactual explanation. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 389–401. doi:10.1145/3650212.365213
15. **Senanayake J. M. D., Kalutarage H., Petrovski A., Piras L., Al-Kadri V. O.** Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. *Journal of Information Security and Applications*, 2024, vol. 82, Article 103741. doi:10.1016/j.jisa.2024.103741
16. **Mahdavifar S., Saqib M., Fung B. C. M., Charland P., Walenstein A.** VulEXplaineR: XAI for vulnerability detection on assembly code. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2024, pp. 3–20. doi:10.1007/978-3-031-70378-2_1
17. **Shareef F.** RetailGPT: A fine-tuned LLM architecture for customer experience and sales optimization. *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, IEEE, 2024, pp. 1390–1394. doi:10.1109/ICSSAS64001.2024.10760685
18. **Berti A., Kourani H., Hafke H., Li C., Schuster D.** Evaluating large language models in process mining: Capabilities, benchmarks, and evaluation strategies. *International Conference on Business Process Modeling, Development and Support*, 2024, pp. 13–21. doi:10.1007/978-3-031-61007-3_2
19. **Ferrag M. A., Alwahedi F., Batah A., Cherif B.** Generative ai and large language models for cyber security: All insights you need. *Available at SSRN 4853709*, 2024. doi:10.48550/arXiv.2405.12750
20. **Hassanin M., Moustafa N.** A comprehensive overview of large language models (LLMs) for cyber defences: Opportunities and directions. *arXiv preprint arXiv:2405.14487*, 2024. doi:10.48550/arXiv.2405.14487
21. **Jelodar H., Bai S., Hamedi P., Mohammadian H., Razavi-Far R., Ghorbani A**. Large Language Model (LLM) for software security: Code analysis, malware analysis, reverse engineering. *arXiv preprint arXiv:2504.07137*, 2025. doi:10.48550/arXiv.2504.07137
22. **Li Z., Dutta S., Naik M.** LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024. doi:10.48550/arXiv.2405.17238
23. **Cao D., Jun W.** LLM-CloudSec: Large Language Model empowered automatic and deep vulnerability analysis for intelligent clouds. *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2024, pp. 1–6. doi:10.1109/INFOCOMWKSHPS61880.2024.10620804
24. **Wu J., Yang S., Zhan R., Yuan Y., Wong D. F., Chao L. S.** A survey on LLM-generated text detection: Necessity, methods, and future directions. *Computational Linguistics*, 2025, pp. 1–66. doi:10.1162/coli_a_00549
25. **Achiam J., Adler S., Agarwal S., Ahmad L., Akkaya I., Aleman F. L., Almeida D.** Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. doi:10.48550/arXiv.2303.08774
26. **Ersoy P., Erşahin M.** Benchmarking Llama 3 70B for code generation: A comprehensive evaluation. *Orclever Proceedings of Research and Development*, 2024, vol. 4, no. 1, pp. 52–58.
27. **Song Y.** Semantic attention and LLM-based layout guidance for text-to-image generation. *ICASSP 2025–2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2025, pp. 1–5. doi:10.1109/ICASSP49660.2025.10890155
28. **Manigrasso F., Schouten S., Morra L., Bloem P.** Probing LLMs for logical reasoning. *International Conference on Neural-Symbolic Learning and Reasoning*, 2024, pp. 257–278. doi:10.1007/978-3-031-71167-1_14
29. **Hashimoto W., Yasui M., Takeuchi K.** Basic investigation of code edit distance measurement by CodeBERT. *Conference: 2023 15th International Congress*

on Advanced Applied Informatics Winter (IIAI-AAI-Winter), IEEE, 2023, pp. 13–18. doi:10.1109/IIAI-AAI-Winter61682.2023.00012

30. **Zhang B., Zhang X., Zhang J., Yu J., Luo S., Tang J.** CoT-based synthesizer: Enhancing LLM performance through answer synthesis. *arXiv preprint arXiv:2501.01668*, 2025. doi:10.48550/arXiv.2501.01668
31. **Nam D., Macvean A., Hellendoorn V., Vasilescu B., Myers B.** Using an llm to help with code understanding. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.363918
32. **Wang P., Yuan N., Li Y.** An integrated framework for data security using advanced machine learning classification and best practices. *Informatica*, 2025, vol. 49, no. 12, pp. 191–206. doi:10.31449/inf.v49i12.7838
33. **Croft R., Babar M. A., Kholoosi M. M.** Data quality for software vulnerability datasets. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 121–133. doi:10.1109/ICSE48619.2023.00022
34. **Jain A., Patel H., Nagalapatti L., Gupta N., Mehta S., Guttula Sh., Mujumdar Sh., Afzal Sh., Mittal R. Sh., Munigala V.** Overview and importance of data quality for machine learning tasks. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3561–3562. doi:10.1145/3394486.3406477
35. **Silva A., Saavedra N., Monperrus M.** Gitbug-Java: A reproducible benchmark of recent Java bugs. *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 118–122. doi:10.1145/3643991.3644884
36. **Fan J., Nguyen T. N.** AC/C++ code vulnerability dataset with code changes and CVE summaries. *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512. doi:10.1145/3379597.3387501
37. **Bhandari G., 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*, 2021, pp. 30–39. doi:10.1145/3475960.3475985
38. **Sharma G., Singh P.** Comparative study: Word2Vec Versus TF-IDF in software defect predictions. *International Conference on Data Science and Network Engineering*, Singapore, 2024, pp. 95–107. doi:10.1007/978-981-97-8336-6_8
39. **Yadav J., Sharma M.** A review of K-mean algorithm. *Int. Journal of Engineering Trends and Technology*, 2013, vol. 4, no. 7, pp. 2972–2976. doi:10.1016/j.ins.2022.11.139
40. **Cheng X., Wang H., Hua J., Xu G.** Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021, vol. 30, no. 3, pp. 1–33. doi:10.1145/3436877
41. **Hanif H., Maffeis S.** Vulberta: Simplified source code pre-training for vulnerability detection. *2022 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2022, pp. 1–8. doi:10.1109/IJCNN55064.2022.9892280
42. **Sun Y., Wu D., Xue Y., Liu H., Wang H., Xu Z., Xie X., Liu Y.** GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639117
43. **Ahmed T., Devanbu P.** Better patching using LLM prompting, via self-consistency. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 1742–1746. doi:10.1109/ASE56229.2023.00065
44. **Zhou P., Pujara J., Ren X., Chen X., Cheng H.-T., Le Q. V., Chi E. H., Zhou D., Mishra S., Zheng H. S.** Self-discover: Large language models self-compose reasoning structures. *Advances in Neural Information Processing Systems*, 2024, vol. 37, pp. 126032–126058.

UDC 004.89
doi:10.31799/1684-8853-2025-4-58-70
EDN: XJDOLW

Method for identifying software code vulnerabilities based on cluster analysis and contextual adaptation of large language models

R. N. Bakeev[a], Head of the Center, orcid.org/0009-0008-3149-3836
V. N. Kuzmin[b], Dr. Sc., Military, Leading Researcher, orcid.org/0000-0002-6411-4336, vka@mil.ru
A. B. Menisov[b], Dr. Sc., Tech., Senior Lecturer, orcid.org/0000-0002-9955-2694
T. R. Sabirov[b], PhD, Tech., Senior Lecturer, orcid.org/0000-0002-6807-2954
[a]Fund for Advanced Research, 22, bld. 3, Berezhkovskaya Emb., 121059, Moscow, Russian Federation
[b]A. F. Mozhaiskii Military Space Academy, 13, Zhdanovskaia Emb., 197198, Saint-Petersburg, Russian Federation

Introduction: Detecting vulnerabilities in source code remains one of the priority tasks in the field of cybersecurity. Classical code analysis methods often do not take into account the execution context and do not scale well with the growth of program volumes. In the context of complex architectures and incomplete data annotations, a context-adapted approach is required that can identify vulnerabilities

based on semantic and structural analysis. **Purpose:** To develop a method for detecting vulnerabilities in program code using cluster analysis and contextual adaptation of large language models capable of taking into account not only the syntax, but also the semantic structure of programs. **Methods:** The developed approach combines clustering of code segments with the use of pre-trained language models adapted to the program code. To improve the efficiency, vulnerability features are identified, including both the vulnerable fragment itself and its context — control structures, variables, function calls. We carry out training and evaluation on labeled open datasets using pre-trained large language models. **Results:** The method makes it possible to automatically group code fragments by structural similarity, after which semantic analysis is performed using large language models capable of recognizing vulnerability patterns. Experiments have shown that the inclusion of contextual information significantly increases the efficiency of identifying vulnerabilities in the source code. On the BigVul and CVEfixes datasets, the proposed method achieved accuracy of up to 78% and recall of 82%, which is 9–12% higher than existing solutions. The method demonstrates resistance to syntactic variations and can be used to analyze previously unlabeled code. **Practical relevance:** The method is applicable in automatic source code analysis systems and can significantly reduce the cost of manual auditing, especially when analyzing large code bases. It can also be used for educational and research purposes to analyze vulnerability patterns. **Discussion:** The results confirm the effectiveness of using large language models in software security analysis tasks. A promising direction is to extend the method to other programming languages, as well as to study hybrid approaches involving graph neural networks. The question of arguing the model's decisions and automatically explaining the reasons for classifying a segment as vulnerable remains open.

Keywords — information security, vulnerability detection, large language models, cluster analysis, machine learning.

For citation: Bakeev R. N., Kuzmin V. N., Menisov A. B., Sabirov T. R. Method for identifying software code vulnerabilities based on cluster analysis and contextual adaptation of large language models. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2025, no. 4, pp. 58–70 (In Russian). doi:10.31799/1684-8853-2025-4-58-70, EDN: XJDOLW

References

1. Jimmy F. N. U. Cyber security vulnerabilities and remediation through cloud security tools. *Journal of Artificial Intelligence General Science (JAIGS)*, 2024, vol. 2, no. 1, pp. 129–171. doi:10.60087/jaigs.v2i1.102
2. Fu M., Tantithamthavorn C., Le T., Kume Y., Nguyen V., Phung D., Grundy J. AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, 2024, vol. 29, no. 1, p. 4. doi:10.1007/s10664-023-10346-3
3. Jabbari M., Alibabaei A., Kavousi A., Rezvanjah M. Vulnerability analysis of the tanks of an oil refinery to fire-induced dom-ino effects based on graph theory. *Iran Occupational Health*, 2021, vol. 18, no. 1, pp. 201–219. doi:10.52547/ioh.18.1.201
4. Shiri Harzevili N., Boaye Belle A., Wang J., Wang S., Jiang Z. M. (J.), Nagappan N. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys*, 2024, vol. 57, no. 3, pp. 1–36. doi:10.1145/3699711
5. Prümmer J., van Steen T., van den Berg B. A systematic review of current cybersecurity training methods. *Computers & Security*, 2024, vol. 136, Article 103585. doi:10.1016/j.cose.2023.103585
6. Camacho N. G. The role of AI in cybersecurity: Addressing threats in the digital age. *Journal of Artificial Intelligence General Science (JAIGS)*, 2024, vol. 3, no. 1, pp. 143–154. doi:10.60087/jaigs.v3i1.75
7. Guo Y., Patsakis C., Hu Q., Tang Q., Casino F. Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection. *European Symposium on Research in Computer Security*, 2024, pp. 271–289. doi:10.1007/978-3-031-70879-4_14
8. Wang H., Tang Z., Tan S. H., Wang J., Liu Y., Fang H., Xia C., Wang Z. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639212
9. Ferrag M. A., Battah A., Tihanyi N., Jain R., Maimut D., Alwahedi F., Lestable T., Thandi N. S., Mechri A., Debbah M., Cordeiro L. C. Secure falcon: Are we there yet in automated software vulnerability detection with LLMs? *IEEE Transactions on Software Engineering*, 2025, pp. 1248–1265. doi:10.1109/TSE.2025.3548168
10. Zhao S., Yang Y., Wang Z., He Z., Qiu L. K., Qiu L. Retrieval augmented generation (rag) and beyond: A comprehensive survey on how to make your LLMs use external data more wisely. *arXiv preprint arXiv:2409.14924*, 2024. doi:10.48550/arXiv.2409.14924
11. Patil R., Gudivada V. A review of current trends, techniques, and challenges in large language models (LLMs). *Applied Sciences*, 2024, vol. 14, no. 5, Article 2074. doi:10.3390/app14052074
12. Chen L., Varoquaux G. What is the role of small models in the LLM era: A survey. *arXiv preprint arXiv:2409.06857*, 2024. doi:10.48550/arXiv.2409.06857
13. Cao S., Sun X., Wu X., Lo D., Bo L., Li B., Liu W. Coca: Improving and explaining graph neural network-based vulnerability detection systems. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639168
14. Chu Z., Wan Y., Li Q., Wu Y., Zhang H., Sui Y., Xu G., Jin H. Graph neural networks for vulnerability detection: A counterfactual explanation. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 389–401. doi:10.1145/3650212.3652139
15. Senanayake J. M. D., Kalutarage H., Petrovski A., Piras L., Al-Kadri V. O. Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. *Journal of Information Security and Applications*, 2024, vol. 82, Article 103741. doi:10.1016/j.jisa.2024.103741
16. Mahdavifar S., Saqib M., Fung B. C. M., Charland P., Walenstein A. VulEXplaineR: XAI for vulnerability detection on assembly code. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2024, pp. 3–20. doi:10.1007/978-3-031-70378-2_1
17. Shareef F. RetailGPT: A fine-tuned LLM architecture for customer experience and sales optimization. *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, IEEE, 2024, pp. 1390–1394. doi:10.1109/ICSSAS64001.2024.10760685
18. Berti A., Kourani H., Hafke H., Li C., Schuster D. Evaluating large language models in process mining: Capabilities, benchmarks, and evaluation strategies. *International Conference on Business Process Modeling, Development and Support*, 2024, pp. 13–21. doi:10.1007/978-3-031-61007-3_2
19. Ferrag M. A., Alwahedi F., Batah A., Cherif B. Generative ai and large language models for cyber security: All insights you need. *Available at SSRN 4853709*, 2024. doi:10.48550/arXiv.2405.12750
20. Hassanin M., Moustafa N. A comprehensive overview of large language models (LLMs) for cyber defences: Opportunities and directions. *arXiv preprint arXiv:2405.14487*, 2024. doi:10.48550/arXiv.2405.14487
21. Jelodar H., Bai S., Hamedi P., Mohammadian H., Razavi-Far R., Ghorbani A. Large Language Model (LLM) for software security: Code analysis, malware analysis, reverse engineering. *arXiv preprint arXiv:2504.07137*, 2025. doi:10.48550/arXiv.2504.07137
22. Li Z., Dutta S., Naik M. LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024. doi:10.48550/arXiv.2405.17238
23. Cao D., Jun W. LLM-CloudSec: Large Language Model empowered automatic and deep vulnerability analysis for intelligent clouds. *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2024, pp. 1–6. doi:10.1109/INFOCOMWKSHPS61880.2024.10620804
24. Wu J., Yang S., Zhan R., Yuan Y., Wong D. F., Chao L. S. A survey on LLM-generated text detection: Necessity, methods, and future directions. *Computational Linguistics*, 2025, pp. 1–66. doi:10.1162/coli_a_00549
25. Achiam J., Adler S., Agarwal S., Ahmad L., Akkaya I., Aleman F. L., Almeida D. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. doi:10.48550/arXiv.2303.08774

26. Ersoy P., Erşahin M. Benchmarking Llama 3 70B for code generation: A comprehensive evaluation. *Orclever Proceedings of Research and Development*, 2024, vol. 4, no. 1, pp. 52–58.
27. Song Y. Semantic attention and LLM-based layout guidance for text-to-image generation. *ICASSP 2025–2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2025, pp. 1–5. doi:10.1109/ICASSP49660.2025.10890155
28. Manigrasso F., Schouten S., Morra L., Bloem P. Probing LLMs for logical reasoning. *International Conference on Neural-Symbolic Learning and Reasoning*, 2024, pp. 257–278. doi:10.1007/978-3-031-71167-1_14
29. Hashimoto W., Yasui M., Takeuchi K. Basic investigation of code edit distance measurement by CodeBERT. *Conference: 2023 15th International Congress on Advanced Applied Informatics Winter (IIAI-AAI-Winter)*, IEEE, 2023, pp. 13–18. doi:10.1109/IIAI-AAI-Winter61682.2023.00012
30. Zhang B., Zhang X., Zhang J., Yu J., Luo S., Tang J. CoT-based synthesizer: Enhancing LLM performance through answer synthesis. *arXiv preprint arXiv:2501.01668*, 2025. doi:10.48550/arXiv.2501.01668
31. Nam D., Macvean A., Hellendoorn V., Vasilescu B., Myers B. Using an llm to help with code understanding. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.363918
32. Wang P., Yuan N., Li Y. An integrated framework for data security using advanced machine learning classification and best practices. *Informatica*, 2025, vol. 49, no. 12, pp. 191–206. doi:10.31449/inf.v49i12.7838
33. Croft R., Babar M. A., Kholoosi M. M. Data quality for software vulnerability datasets. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 121–133. doi:10.1109/ICSE48619.2023.00022
34. Jain A., Patel H., Nagalapatti L., Gupta N., Mehta S., Guttula Sh., Mujumdar Sh., Afzal Sh., Mittal R. Sh., Munigala V. Overview and importance of data quality for machine learning tasks. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3561–3562. doi:10.1145/3394486.3406477
35. Silva A., Saavedra N., Monperrus M. Gitbug-Java: A reproducible benchmark of recent Java bugs. *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 118–122. doi:10.1145/3643991.3644884
36. Fan J., Nguyen T. N. AC/C++ code vulnerability dataset with code changes and CVE summaries. *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512. doi:10.1145/3379597.3387501
37. Bhandari G., 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*, 2021, pp. 30–39. doi:10.1145/3475960.3475985
38. Sharma G., Singh P. Comparative study: Word2Vec Versus TF-IDF in software defect predictions. *International Conference on Data Science and Network Engineering*, Singapore, 2024, pp. 95–107. doi:10.1007/978-981-97-8336-6_8
39. Yadav J., Sharma M. A review of K-mean algorithm. *Int. Journal of Engineering Trends and Technology*, 2013, vol. 4, no. 7, pp. 2972–2976. doi:10.1016/j.ins.2022.11.139
40. Cheng X., Wang H., Hua J., Xu G. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021, vol. 30, no. 3, pp. 1–33. doi:10.1145/3436877
41. Hanif H., Maffeis S. Vulberta: Simplified source code pre-training for vulnerability detection. *2022 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2022, pp. 1–8. doi:10.1109/IJCNN55064.2022.9892280
42. Sun Y., Wu D., Xue Y., Liu H., Wang H., Xu Z., Xie X., Liu Y. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi:10.1145/3597503.3639117
43. Ahmed T., Devanbu P. Better patching using LLM prompting, via self-consistency. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 1742–1746. doi:10.1109/ASE56229.2023.00065
44. Zhou P., Pujara J., Ren X., Chen X., Cheng H.-T., Le Q. V., Chi E. H., Zhou D., Mishra S., Zheng H. S. Self-discover: Large language models self-compose reasoning structures. *Advances in Neural Information Processing Systems*, 2024, vol. 37, pp. 126032–126058.