БЕЗОПАСНОСТЬ ЦИФРОВЫХ ТЕХНОЛОГИЙ. – 2026. – № 1 (120). – 9–28

МЕТОДЫ И СИСТЕМЫ ЗАЩИТЫ ИНФОРМАЦИИ, ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

УДК 004.032.26 DOI: 10.17212/2782-2230-2026-1-9-28

МНОГОАГЕНТНАЯ СИСТЕМА ГЕНЕРАЦИИ И ЭВОЛЮЦИИ ПРАВИЛ СТАТИЧЕСКОГО АНАЛИЗА CODEQL НА БАЗЕ LLM ДЛЯ C/C++*

А.Б. АРХИПОВА[1], Е.Д. АЗЕЕВА[2]

[1] *630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет, доцент кафедры защиты информации. E-mail: arhipova@corp.nstu.ru*

[2] *630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет, лаборант кафедры защиты информации. E-mail: evaazeeva@yandex.ru*

В работе рассматривается проблема обеспечения безопасности программных систем на C/C++, для которых статический анализ кода является одним из ключевых инструментов выявления уязвимостей, но остается дорогим в сопровождении из-за сложности разработки правил и taint-спецификаций. Анализируются существующие подходы: промышленные статические анализаторы (Svace, Irbis), IFDS-анализ, решения на основе ML/NN и гибридные схемы с участием больших языковых моделей (LLM), применяемых для фильтрации предупреждений, генерации кода и автоматического синтеза правил (QLPro, KNighter, CodePatchLLM). Показано, что прямое использование LLM в роли самостоятельного детектора уязвимостей дает высокую полноту, но сопровождается значительным числом ложных срабатываний и галлюцинаций. В качестве альтернативы предложен многоагентный модуль для SAST-инструмента CodeQL, ориентированный на C/C++, который использует LLM для генерации, валидации и эволюции правил на языке запросов QL в замкнутом контуре «генерация, проверка, доработка, фильтрация, ранжирование». Описана архитектура системы с агентом-генератором, валидатором, рефайнером, фильтром и ранкером, оркестрация которой осуществляется с помощью LangGraph и PostgreSQL. Валидация правил выполняется по CodeQL-базам Juliet, SARD и реальных проектов с расчетом метрик Precision, Recall, F1, FPR и Specificity и с учетом CVSS-баллов, что позволяет отбирать правила приемлемого качества и снижать нагрузку на разработчиков за счет автоматизированной эволюции базы правил.

Ключевые слова: статический анализ кода, безопасность программных систем, C/C++, SAST-инструменты, CodeQL, большие языковые модели, агентные LLM-системы, генерация правил статического анализа, CWE

* Статья получена 22 января 2026 г.

ВВЕДЕНИЕ

Рост масштаба программных систем на C и C++ приводит к тому, что одна уязвимость в исходном коде может приводить к массовым инцидентам кибербезопасности [1]. Традиционные практики тестирования и ручного аудита кода больше не подходят для крупных открытых и корпоративных репозиториев, содержащих миллионы строк кода.

Показательным примером последствий эксплуатации уязвимости в серверном приложении является инцидент с Equifax (Equifax Inc., США) в 2017 году. По данным компании и независимых исследований, атакующие использовали известную на тот момент уязвимость Apache Struts (Common Vulnerabilities and Exposures (CVE) 2017-5638), что привело к компрометации персональных данных около 143 млн пользователей, причиной стала несвоевременная установка обновления и отсутствие эффективных процедур выявления уязвимых компонентов веб-приложения. Такие инциденты демонстрируют, что эксплуатация, казалось бы, «локальной» ошибки в коде приводит к системным рискам, особенно когда затрагиваются персональные данные и критическая инфраструктура.

В российском правовом поле требования к обеспечению безопасности программных систем закреплены в ряде базовых федеральных законов и подзаконных актов. Федеральный закон № 187-ФЗ «О безопасности критической информационной инфраструктуры Российской Федерации» [2] устанавливает приоритет предотвращения компьютерных атак и обязанности субъектов критической информационной инфраструктуры КИИ по созданию и эксплуатации системы безопасности значимых объектов, включающей организационные и технические меры по предотвращению неправомерного доступа и недопущению нарушения функционирования таких объектов.

Согласно этому закону субъекты КИИ обязаны внедрять средства обнаружения, предупреждения и ликвидации последствий компьютерных атак и инцидентов, обеспечивать их корректную установку и эксплуатацию, а также выполнять требования уполномоченного органа по обеспечению безопасности КИИ. Эти формулировки прямо ориентируют организации на системный подход к обнаружению уязвимостей и дефектов в программном обеспечении, включая применение автоматизированных средств анализа кода.

Федеральный закон № 152-ФЗ «О персональных данных» в ст. 19 [3] обязывает оператора персональных данных принимать «необходимые правовые, организационные и технические меры» для защиты персональных данных от неправомерного или случайного доступа, уничтожения, изменения, блокирования, копирования и иных незаконных действий. Дополнительно в ряде редакций и разъяснений подчеркивается необходимость взаимодействия опе-

раторов с государственной системой обнаружения, предупреждения и ликвидации последствий компьютерных атак, в том числе при инцидентах, повлекших незаконную передачу персональных данных. Практические руководства по исполнению ФЗ-152 для операторов ПДн включают в перечень необходимых мер регулярное выявление уязвимостей, использование сертифицированных средств защиты и «регулярную проверку системы на отсутствие признаков взлома», что, по сути, предполагает внедрение процессов безопасной разработки и анализа кода.

Нормативная база стимулирует организации к внедрению процессов управления уязвимостями и качеством кода. Статический анализ исходного кода становится одним из ключевых элементов обеспечения безопасности программных систем.

В рамках статьи рассматривается исследование применения больших языковых моделей (LLM) для генерации и эволюции правил статического анализатора кода на C/C++ на базе CodeQL, а также прототип модуля AI-агентов, обеспечивающего фильтрацию, доработку и создание новых правил для инструмента статического анализа безопасности приложений (SAST-инструмента). Такой модуль ориентирован на использование базы CWE совместно с возможностями LLM, что потенциально позволяет строить более адаптивную систему безопасной разработки и анализа кода.

1. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ

В области обнаружения уязвимостей и генерации правил статического анализа уже сформировалось несколько классов решений, в которых большие языковые модели либо дополняют классические статические анализаторы, либо пытаются заменить их в роли самостоятельного инструмента анализа кода. Ниже приведен сравнительный обзор таких решений.

Таблица 1

Характеристики подходов

Подход / работа	Языки / домен	Роль LLM	Инструмент / основа
Многоуровневый статический анализ [4]	C, Java, большие промышленные проекты	Многослойные детекторы с разной глубиной анализа	Собственный анализатор Svace, многоуровневая архитектура

Продолжение табл. 1

Подход / работа	Языки / домен	Роль LLM	Инструмент / основа
Irbis – статический Interprocedural Finite Distributive Subset (IFDS) taint-анализатор [5]	C/C++ (Juliet, OpenSSL)	taint-анализ с настраиваемыми источниками	Собственный IFDS-движок; сигнатуры источников / стоков в JSON
Обзор машинного обучения / нейронных сетей (ML/NN) для поиска уязвимостей [6]	В основном веб-приложения, SQLi, XSS, отдельные CWE	ML/Deep Learning как самостоятельные классификаторы уязвимостей	Специализированные модели поверх абстрактного синтаксического дерева (AST)
LLM для фильтрации предупреждений статического анализа [7]	C (промышленный код)	LLM как фильтр / верификатор результатов статического анализатора	SharpChecker + несколько LLM-моделей
CodePatchLLM – LLM+Svace для доработки сгенерированного кода [8]	Java, Python (генерация кода по заданию)	LLM как генератор кода, LLM исправляет код по отчётам анализатора	Модель-агностичный фреймворк поверх нескольких LLM
QLPro – LLM+ CodeQL для генерации правил [1]	Java-проекты (CodeQL-анализ)	LLM классифицирует taint-спецификации и генерирует CodeQL-запросы	CodeQL + LLM с трехролевым механизмом Writer/ Execute/Repair
KNighter – LLM-синтез чекеров Clang Static Analyzer [9]	C (ядро Linux)	LLM синтезирует полноценные Clang Static Analyzer (CSA) чекеры по патчам	CSA + многоагентный пайплайн анализа патча, планирования и генерации чекера

Окончание табл. 1

Подход / работа	Языки / домен	Роль LLM	Инструмент / основа
LLM как самостоятельный детектор уязвимостей [10]	C/C++ (buffer overflow) и другие языки	LLM (GPT-3.5/4, CodeGen) напрямую классифицируют код как уязвимый/нет	Наборы CodeGadgets и CVEfixes; сравнение с Flawfinder, Checkmarx и VulDeePecker
De-Hallucinator – уменьшение галлюцинаций LLM в коде [11]	Python, JavaScript	LLM остается основным генератором, но основывается на реальные программные интерфейсы приложений (API)	Итеративное RAG-подобное (генерация с дополнением извлеченными данными) обогащение промптов API-фрагментами проекта

Таблица 2

Эффективность и проблемы подходов

Подход / работа	Результаты	Ограничения
Многоуровневый статический анализ [4]	Обеспечивает покрытие широкого набора дефектов при приемлемой производительности; практическое применение в реальных продуктах	Высокая стоимость разработки и сопровождения правил; слабая адаптация к специфике конкретного проекта без ручной донастройки
Irbis – статический IFDS taint-анализатор [5]	Стопроцентное покрытие taint-подмножества Juliet для реализованных CWE, полное подавление ложных срабатываний на этих тестах	Требуется ручное определение и сопровождение источников;
Обзор ML/NN для поиска уязвимостей [6]	Показано высокое гармоническое среднее между точностью и полнотой (F-меры) (до 0,9 для отдельных задач)	Требуют тщательно размеченных датасетов; слабая переносимость на другие типы уязвимостей и проекты

Окончание табл. 2

Подход / работа	Результаты	Ограничения
LLM для фильтрации предупреждений статического анализа [7]	Повышение точности при сохранении полноты 0,8…0,97 для утечек ресурсов, null-разыменований и переполнений	LLM не генерируют новые правила, зависят от качества информации, подготовленной анализатором; сохраняется стоимость LLM-инференса
CodePatchLLM – LLM+ Svace для доработки сгенерированного кода [8]	Среднее абсолютное снижение числа предупреждений Svace по Java-коду на 19,1 %	Фокус на качестве сгенерированного кода, а не на правилах; завязан на конкретный анализатор и его формат отчетов
QLPro – LLM+CodeQL для генерации правил [1]	Синтаксическая корректность QL-файлов до 90,9 % при коллаборации двух LLM; обнаружена 41 из 62 известных уязвимостей против 24 у официальных правил; найдено 6 новых 0-day	Ограничение на язык; сильная зависимость от качества извлеченных taint-спецификаций; не покрывает C/C++
KNighter – LLM-синтез чекеров Clang Static Analyzer [9]	Сгенерировано 39 валидных чекеров по 61 патчу; найдено 92 новых долгоживущих бага в ядре Linux, 77 подтверждено, 30 получили CVE	Требует исторических патчей; ориентация на конкретную инфраструктуру (CSA); сложность пайплайна и стоимости LLM-инференса
LLM как самостоятельный детектор уязвимостей [10]	После дообучения достигается высокая полнота (recall до 0,94 на code gadgets), LLM хорошо выучивают паттерны уязвимого кода	Очень высокий уровень ложных срабатываний – precision заметно ниже, чем у классических статических анализаторов; слабое понимание точного места и причины уязвимости
De-Hallucinator – уменьшение галлюцинаций LLM в коде [11]	Уменьшение галлюцинаций API, рост числа проходящих тестов; улучшение recall правильных API-вызовов на 24…61 %	Не решает задачу безопасного анализа напрямую, но критичен для корректной генерации кода и, следовательно, правил

Сопоставление существующих решений показывает, что классические анализаторы (Svace, Irbis) и IFDS-подходы остаются лидерами по точности и объяснимости, но сильно зависят от ручной разработки правил и taint-спецификаций. ML/NN-подходы и особенно LLM демонстрируют высокую способность к выявлению паттернов уязвимостей, но в отрыве от строгого анализа страдают от высокого уровня ложных срабатываний. Наиболее успешные LLM-решения в статическом анализе – это гибриды, где LLM либо фильтрует или обогащает результаты классического анализатора, либо автоматически генерирует и эволюционирует правила под контролем существующей инфраструктуры.

2. ВОЗМОЖНОСТИ LLM

Современные большие языковые модели, обученные на коде, уже демонстрируют возможности генерации и доработки правил для SAST-инструментов.

Во-первых, LLM хорошо справляются с выявлением и обобщением паттернов уязвимого кода даже без строгого символьного анализа. Тонкая настройка моделей типа Davinci и CodeGen на датасетах CodeGadgets и CVEfixes позволяет достичь высокой полноты при детектировании переполнений буфера и SQL-инъекций, причем модели уверенно распознают характерные комбинации библиотечных вызовов (strcpy, strncpy, memcpy), соединения строк в SQL-запросах и типичных шаблонов использования объектно-реляционных (ORM) библиотек [10]. Модели часто не распознают, где именно находится корень уязвимости, но узнают общий паттерн.

Во-вторых, LLM способны читать и использовать структурированную информацию от статического анализатора для более глубокого понимания предупреждений. Статический анализатор SharpChecker формирует для каждой проблемы расширенный контекст (AST-фрагменты, сводки по функциям, символические выражения), который затем передается LLM для классификации предупреждения как истинного или ложного [7]. Экспериментально показано, что такая схема позволяет увеличить F1-метрику детекторов утечек ресурсов, null-разыменований, целочисленных переполнений и недостижимого кода на 10–27 п.п. по сравнению с базовой конфигурацией SharpChecker без LLM. Это подтверждает, что модели умеют интерпретировать достаточно сложные структурированные представления анализа.

В-третьих, LLM демонстрируют способность генерировать код на специализированных предметно ориентированных языках (DSL), включая языки запросов статического анализа, при наличии корректного контекста и обратной связи по ошибкам. В QLPro эта способность используется для автомати-

ческой генерации CodeQL-файлов по заранее построенным парам «источник – сток»: LLM получает структуру API и выбранную пару, после чего синтезирует QL-запрос, который затем компилируется CodeQL-инструментом. За счет многошагового цикла «генерация → компиляция → исправление по сообщению об ошибке» удается достичь 90,9 % синтаксически корректных QL-файлов и существенно превзойти официальную библиотеку правил CodeQL по числу обнаруженных уязвимостей на датасете JavaTest [1].

Наконец, LLM могут работать как интерактивные исполнители корректировок по обратной связи статического анализатора, фактически закрывая контур «анализ → исправление → переанализ». В CodePatchLLM для каждого сгенерированного фрагмента Java/Python-кода запускается Svace; выявленные предупреждения транслируются в текстовые подсказки для LLM, которая вносит правки в код. Эксперименты показывают, что это позволяет в среднем уменьшить число предупреждений по Java-коду на 19,1 % без ухудшения точности [8].

3. ПРОБЛЕМЫ СОВРЕМЕННОГО АНАЛИЗА

Статический анализ исходного кода давно стал одним из ключевых инструментов обеспечения безопасности и качества программ, особенно для проектов на C/C++, где ошибки управления памятью и работы с указателями напрямую приводят к уязвимостям уровня Heartbleed и аналогичным инцидентам [5]. Практический опыт крупных промышленных анализаторов показывает, что именно статический анализ дает возможность просматривать все потенциальные пути выполнения, в том числе редко выполняемые ветви и обработку ошибок, которые почти невозможно покрыть тестированием [4].

При этом домен C/C++ остается одним из самых сложных для SAST-инструментов из-за указателей, ручного управления памятью, сложных макросов и неочевидных соглашений библиотек. Уже на уровне базовых примеров из пакета Juliet Test Suite видно, что даже простые на вид паттерны: «чтение во внешний буфер», «формирование строки формата», «арифметика с размерами буфера» – порождают целый спектр CWE [5].

Несмотря на зрелость технологий статического анализа, и в научной, и в промышленной литературе регулярно подчеркиваются три системные проблемы: масштаб, точность и стоимость разработки правил.

Во-первых, масштаб современных проектов приводит к тому, что даже очень быстрые анализаторы вынуждены балансировать между глубиной анализа и приемлемым временем работы. Для поддержки больших систем (ядро, сложные платформы) приходится строить каскад из разных уровней проверки: от легких линтеров до тяжелых видов анализа с использованием

SMT-решателей; при этом общее время анализа измеряется десятками минут и часами, а ложноположительные срабатывания остаются значительными [4]. Добавление новых сложных правил почти всегда бьет по времени и объему памяти, поэтому любая эволюция набора правил – это дорогостоящий инженерный проект.

Во-вторых, высокая точность по одному классу уязвимостей обычно достигается путем повышения специфичности правил, которые плохо переносятся на другие проекты и стили кода [5].

В-третьих, разработка и сопровождение самих правил остаются трудоемкими и зависимыми от экспертов. Например, значительная часть инвестиций в Svace – это именно накопление и формализация знаний о многочисленных правилах из CWE, Computer Emergency Response Team (CERT), MISRA (Motor Industry Software Reliability Association, Великобритания) и других источников, включая адаптацию их под корпоративные стандарты и конкретные комплекты средств разработки (SDK) [4]. Все определения источников и стоков задаются в виде JSON-файлов, которые должны поддерживаться и расширяться экспертами по безопасности, чтобы инструмент оставался актуальным и находил новые типы уязвимостей [5].

На практике написание правила для SAST-инструмента – это не просто «поиск строки с небезопасной функцией». Даже для относительно простых сценариев необходимо следующее:

– сформулировать угрозу в терминах конкретного языка и библиотек (например, какие именно функции ввода считаются потенциальными источниками непроверенных данных в данном проекте) [5];

– описать разрешенные и запрещенные пути распространения данных, учитывая указатели, структуры, вызовы через обертки, макросы [5];

– минимизировать число ложных срабатываний с помощью дополнительных эвристик, часто завязанных на специфических для проекта соглашениях [4];

– формализовать все эти интуитивные соображения на языке запросов или внутреннем DSL анализатора [12].

При этом каждая новая библиотека, протокол или внутренний фреймворк фактически требует пересмотра части правил: появляются новые источники, стоки, особые случаи обработки ошибок, нестандартные форматы [6]. В результате крупные команды разработчиков SAST-инструментов тратят значительную долю времени не на улучшение самих анализов, а на инженерную работу по написанию, тестированию и поддержке правил [10].

4. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

Модуль AI-агентов, который автоматизирует генерацию, фильтрацию и доработку правил статического анализа для SAST-инструмента CodeQL в домене C/C++-кода, должен опираться на большие языковые модели (LLM), но быть встроен в уже существующую инфраструктуру статического анализа (CodeQL, тестовые наборы) и работать в замкнутом контуре «генерация/обновление правил, проверка, фильтрация и эволюция».

CodeQL сегодня является одним из наиболее распространенных промышленных инструментов статического анализа, в том числе в экосистемах открытого кода и корпоративных CI/CD-процессах [12]. Его ключевая особенность – представление кода в виде базы данных и возможность формулировать правила в терминах декларативных запросов, которые описывают паттерны уязвимостей на уровне абстрактных сущностей (функции, выражения, пути данных), а не конкретных текстовых шаблонов. Это делает CodeQL естественной платформой для автоматизации: правило – это обычный QL-файл, который можно синтезировать, анализировать, тестировать и развивать автоматически.

Практика промышленных анализаторов для C/C++ показывает, что этот стек нуждается в глубоком статическом анализе и предъявляет самые жесткие требования к качеству и полноте правил из-за указателей, ручного управления памятью и богатого набора небезопасных библиотечных вызовов [5].

Опыт существующих работ показывает, что прямое использование LLM в роли «одного умного компонента» для статического анализа приводит к высоким показателям полноты, но и к неприемлемо высокому уровню ложных срабатываний и галлюцинаций [10]. Эту проблему решают многошаговые агентные конвейеры, в которых разные роли и этапы четко разделены [1].

Дополнительным аргументом в пользу агентного подхода служат работы, направленные на уменьшение галлюцинаций и повышение надежности LLM в задачах работы с кодом. Модель LLM сначала генерирует черновой фрагмент кода, затем специальный компонент по этому фрагменту подбирает релевантные API из проекта и расширяет промпт, после чего модель повторно генерирует код; цикл может повторяться несколько раз, пока не будет достигнуто удовлетворяющее решение. Такой итеративный агентный подход существенно снижает количество неверных обращений к API [11].

В контексте генерации и доработки правил для CodeQL это означает, что архитектура должна предусматривать:

– агента-генератора, который по результатам анализа формирует черновик правила на QL [7];

– агента-валидатора, запускающего CodeQL-анализ с новым правилом на тестовых наборах и собирающего статистику срабатываний;

– агента-рефайнера, который по отчетам и логам компиляции / запуска вносит правки в правило или помечает его как неперспективное [9].

Такое разбиение позволяет эксплуатировать сильные стороны LLM (понимание текстового и кодового контекста, генерация и переформулировка правил) при одновременном контроле их слабых сторон (нестабильность вывода, склонность к галлюцинациям, отсутствие строгого анализа данных) за счет четких интерфейсов между агентами и валидации результатов через существующую статическую инфраструктуру.

5. ПРОТОТИП РЕШЕНИЯ

5.1. ОБЩАЯ АРХИТЕКТУРА И РОЛИ АГЕНТОВ

Система строится как многоагентный конвейер поверх CodeQL для C/C++, где каждое правило проходит последовательные стадии: генерацию, валидацию, доработку, фильтрацию, ранжирование.

Таблица 3

Роли AI-агентов

Агент	Назначение	Вход	Выход	Основной критерий
Generator	Синтез черновика правила CodeQL	Описание CWE, примеры кода	QL-файл правила	Успешная компиляция QL
Validator	Оценка правила на тестовых датасетах	Правило, датасеты	True Positive, False Positive, False Negative, True Negative (TP, FP, FN, TN), метрики	Метрики ≥ заданных порогов
Refiner	Улучшение правила по результатам валидации	Правило + метрики	Обновленное правило	Снижение FP/FN по сравнению с прошлой версией
Filter	Отбор правил приемлемого качества	Правило + метрики	Решение: принять / отклонить	Precision и Recall

Окончание табл. 3

Агент	Назначение	Вход	Выход	Основной критерий
Ranker	Учет качества правила при оценке серьезности	Правило + метрики	Скорректированный приоритет	Согласованность стандарта для оценки степени критичности уязвимостей (CVSS) и надежности правила

Над агентами работает оркестратор, который управляет потоком.

Генератор получает описание уязвимости (например, CWE-идентификатор, текстовое описание, небольшой набор размеченных примеров кода) и формирует черновой QL-запрос.

Оркестратор проверяет синтаксис. Если правило не компилируется, генератору возвращается сообщение об ошибке и он пытается исправить запрос (несколько попыток).

Валидатор запускает правило на заранее подготовленных CodeQL-БД (Juliet, Software Assurance Reference Dataset (SARD), внутренние репозитории) и рассчитывает показатели TP/FP/FN/TN, а затем Precision, Recall, F-мера, доля ложноположительных срабатываний среди негативных примеров (FPR), Specificity.

Если метрики не достигают целевых значений, Refiner получает текст правила и отчет валидатора и уточняет запрос: добавляет дополнительные условия, сужает класс сущностей, исключает очевидные ложные случаи. Далее цикл Refiner + Validator повторяется до улучшения или до лимита итераций.

Фильтр отклоняет правила, которые создадут слишком много шума в разработке.

Ранкер учитывает и семантическую важность уязвимости (по CWE/CVSS), и качество правила. Для ненадежных правил понижается приоритет, даже если уязвимость критична.

Разработчик в итоге видит не просто «найденную уязвимость», а уязвимость, помеченную уровнем серьезности и оценкой надежности срабатывания правила.

5.2. МОДЕЛЬ ДАННЫХ И ФОРМАТЫ СООБЩЕНИЙ

Оркестратор не только управляет агентами, но и сохраняет версии правил, чтобы при ухудшении метрик была возможность вернуться к предыдущей версии. Сущности в БД (PostgreSQL):

– Rule (правило): id, name (строка, человекочитаемое имя), language (enum [c, cpp]), cwe_id (строка, например, "CWE-119"), ql_source (текст) (полный QL-код), status (enum [draft, validated, rejected, production]), created_at, updated_at;

– RuleVersion (версия правила): id, rule_id, version (int), ql_source (текст), generator_meta (JSON: модель, промпты, параметры), refiner_meta (JSON, что и почему изменено);

– Dataset (датасет): id, name, description, type (enum [juliet, sard, internal_repo]), codeql_db_path (путь к CodeQL-БД), ground_truth (ссылка на таблицу разметки уязвимостей);

– GroundTruth (разметка): id, dataset_id, file_path, line, function, cwe_id, type (enum [good, bad]);

– EvaluationRun (прогон валидации): id, rule_version_id, dataset_id, started_at, finished_at, status;

– Finding (срабатывание правила): id, evaluation_run_id, file_path, line, message, severity, matched_cwe (если правило помечает конкретный CWE), is_tp, is_fp, is_fn, is_tn (заполняется путем сопоставления с GroundTruth);

– Metrics (метрики качества): id, rule_version_id, dataset_id, tp, fp, fn, tn (int), precision, recall, f1, fpr, specificity (float), cvss_base, cvss_adjusted (float, базовый и скорректированный Ranker-ом балл).

Обмен данными между агентами реализуется через JSON-сообщения, описываемые Pydantic-моделями, например:

```
json
{
  "rule_id": "uuid",
  "rule_version": 3,
  "cwe_id": "CWE-119",
  "ql_source": "from ... select ...",
  "metrics": {
    "tp": 42,
    "fp": 10,
    "fn": 8,
    "tn": 1000,
    "precision": 0.808,
```

```
    "recall": 0.84,
    "f1": 0.824
  }
}
```

Оркестратор, построенный на LangGraph, использует эти структуры для явного управления состоянием: узлы графа соответствуют агентам, а рёбра – переходам состояний.

5.3. РАБОТА ВАЛИДАТОРА С CODEQL-БД

Validator отвечает за объективную оценку правила на репрезентативных наборах кода. Для каждого датасета заранее собирают CodeQL-БД, чтобы на этапе валидации ограничиться только запуском анализатора. Валидатор выполняет прогон правила по датасету. Результаты статического анализа программ (SARIF-результаты) сводятся в таблицу Finding. Для каждого Finding ищется запись в GroundTruth с совпадающим file_path и близкой позицией по строке (с учетом возможных смещений) и тем же cwe_id. Если найдена совпадающая уязвимость, то это TP, при отсутствии – FP. Для всех размеченных уязвимостей без соответствующих срабатываний правила формируются FN. Остальной код, где не размечено уязвимостей и нет срабатываний, дает TN.

5.4. МЕТРИКИ КАЧЕСТВА И ПОРОГИ

Метрики привязаны к рекомендациям NIST SARD (National Institute of Standards and Technology Software Assurance Reference Dataset, США), OWASP Benchmark (Open Worldwide Application Security Project Benchmark, международный проект, основан в США) и результатам работ по автоматической генерации правил (AutoGrep, RuleLLM), где целевые значения Precision лежат в диапазоне 70…85 %, а FPR не превышает 20 %.

Таблица 4

Метрики качества правила

Метрика	Формула	Цель	Интерпретация
Precision	TP / (TP + FP)	≥ 70 %	Доля верных предупреждений из всех найденных
Recall	TP / (TP + FN)	≥ 60 %	Доля найденных уязвимостей из всех реальных

Окончание табл. 4

Метрика	Формула	Цель	Интерпретация
F1-Score	2PR / (P + R)	≥ 0,70	Баланс между точностью и полнотой
FPR	FP / (FP + TN)	< 20 %	Доля ложных срабатываний
Specificity	TN / (TN + FP)	> 80 %	Способность «отсекать шум»

Использование разных типов датасетов (SARD, Juliet и реальные проекты) позволяет на синтетических наборах контролировать верхнюю границу достижимых значений, а на реальном коде оценивать практическую применимость правила, в том числе склонность к избыточным срабатываниям.

Ranker использует эти метрики совместно с базовым CVSS-баллом уязвимости, чтобы скорректировать приоритет: правила с низким Precision снижают итоговую оценку, чтобы разработчик видел их как менее надежный источник сигналов.

5.5. ТЕХНОЛОГИЧЕСКИЙ СТЕК

Таблица 5

Технологический стек прототипа

Компонент	Технология	Причина выбора
Граф агентов	LangGraph	Явное моделирование многократных переходов и состояний
Очередь задач	RabbitMQ	Асинхронная обработка долгих запусков CodeQL
БД	PostgreSQL + SQLAlchemy	Надежность, транзакции, JSON-поля для логов
Формат сообщений	JSON + Pydantic	Типобезопасность, валидация входных/выходных данных
Интеграция с LLM	LangChain + Mistral/Qwen	Унифицированный доступ к моделям

Окончание табл. 5

Компонент	Технология	Причина выбора
Контейнеризация	Docker Compose	Локальное тестирование и развертывание
Async-слой	Asyncio	Неблокирующий Input/Output, быстрый отклик оркестратора

ЗАКЛЮЧЕНИЕ

Проведенный обзор показывает, что классические статические анализаторы и IFDS-подходы остаются наиболее надежными средствами поиска уязвимостей, но требуют значительных затрат на ручную разработку и сопровождение правил, особенно в домене C/C++. Решения на основе ML/NN и LLM демонстрируют высокую способность к выявлению паттернов уязвимого кода, однако при использовании в качестве самостоятельных детекторов страдают от большого числа ложных срабатываний и недостаточной интерпретируемости. Наиболее перспективными оказываются гибридные схемы, в которых LLM работают в связке с существующей инфраструктурой статического анализа, выполняя роли генератора, фильтра и эволюционирующего редактора правил. Предложенный в работе многоагентный модуль для CodeQL реализует именно такой подход: LLM-агенты генерируют и уточняют QL-правила, а валидатор на базе CodeQL-БД Juliet, SARD и реальных проектов обеспечивает объективную оценку качества по набору метрик, привязанных к практическим рекомендациям. Использование оркестратора на LangGraph и хранилища метрик в PostgreSQL позволяет встроить модуль в существующие CI/CD-процессы и поддерживать итеративную эволюцию базы правил без постоянного участия экспертов. В дальнейшем развитие работы может быть связано с расширением набора поддерживаемых CWE, интеграцией дополнительных датасетов и исследованием стратегий автоматического выбора оптимальных порогов метрик для разных классов уязвимостей.

СПИСОК ЛИТЕРАТУРЫ

1. QLPro: Automated code vulnerability discovery via LLM and static code analysis integration / J. Hu, X. Jin, Y. Zeng, Y. Liu, Y. Li, D. Du, K. Xie, H. Zhu // arXiv preprint. – 2025. – arXiv:2506.23644v1.

2. О безопасности критической информационной инфраструктуры Российской Федерации: Федеральный закон от 14.06.2023 № 187-ФЗ // Собрание законодательства Российской Федерации. – 2023. – № 25. – Ст. 4–5, 8, 10.

3. О персональных данных: Федеральный закон от 27.07.2006 № 152-ФЗ // Собрание законодательства Российской Федерации. – 2006. – № 31. – Ст. 19.

4. *Белеванцев А.А.* Многоуровневый статический анализ исходного кода программ для обеспечения качества программ // Программирование. – 2017. – № 6. – С. 3–25.

5. *Шимчик Н.В., Игнатьев В.Н., Белеванцев А.А.* Irbis: статический анализатор помеченных данных для поиска уязвимостей в программах на C/C++ // Труды Института системного программирования РАН. – 2022. – Т. 34, вып. 6. – С. 51–66. – DOI: 10.15514/ISPRAS-2022-34(6)-4.

6. *Леонов Н.В., Буйневич М.В.* Машинное обучение vs поиск уязвимостей в программном обеспечении: анализ применимости и синтез концептуальной системы // Труды учебных заведений связи. – 2023. – Т. 9, № 6. – С. 83–94. – DOI: 10.31854/1813-324X-2023-9-6-83-94.

7. Повышение точности статического анализа кода при помощи больших языковых моделей / Д.Д. Панов, Н.В. Шимчик, Д.А. Чибисов, А.А. Белеванцев, В.Н. Игнатьев // Труды Института системного программирования РАН. – 2025. – Т. 37, вып. 6, ч. 1. – С. 83–100. – DOI: 10.15514/ISPRAS-2025-37(6)-5.

8. *Шайхелисламов Д.С., Дробышевский М.Д., Белеванцев А.А.* Интерактивная генерация кода на основе LLM: эмпирическая оценка // Труды Института системного программирования РАН. – 2025. – Т. 37, вып. 5. – С. 123–130. – DOI: 10.15514/ISPRAS-2025-37(5)-9.

9. KNighter: Transforming static analysis with LLM-synthesized checkers / C. Yang, Z. Zhao, Z. Xie, H. Li, L. Zhang // Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles. – ACM, 2025. – P. 1–15. – DOI: 10.1145/3731569.3764827.

10. Software vulnerability detection using large language models / M.D. Purba, A. Ghosh, B.J. Radford, B. Chu // IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW). – IEEE, 2023. – P. 112–119. – DOI: 10.1109/ISSREW60843.2023.00058.

11. *Eghbali A., Pradel M.* De-hallucinator: Mitigating LLM hallucinations in code generation tasks via iterative grounding // arXiv preprint. – 2024. – arXiv:2401.01701v3.

12. QL: Object-oriented queries on relational data / P. Avgustinov, O. Moor, M.P. Jones, M. Schäfer // 30th European Conference on Object-Oriented Programming (ECOOP 2016). – Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. – P. 2:1–2:25.

Архипова Анастасия Борисовна, доцент кафедры защиты информации Новосибирского государственного технического университета. Основное направление научных исследований – математическое моделирование в информационной безопасности, оценка качества социально значимой деятельности. E-mail: arhipova@corp.nstu.ru

Азеева Ева Дмитриевна, лаборант кафедры защиты информации Новосибирского государственного технического университета. E-mail: evaazeeva@yandex.ru

DOI: 10.17212/2782-2230-2026-1-9-28

LLM-based multi-agent system for generation and evolution of CodeQL static analysis rules for C/C++*

A.B. Arkhipova[1], E.D. Azeeva[2]

[1] *Novosibirsk State Technical University, 20 Karl Marx Prospekt, Novosibirsk, 630073, Russian Federation, Associate Professor of the Department of Information Security. E-mail: arhipova@corp.nstu.ru*

[2] *Novosibirsk State Technical University, 20 Karl Marx Prospekt, Novosibirsk, 630073, Russian Federation, laboratory assistant of the Department of Information Security. E-mail: evaazeeva@yandex.ru*

This paper addresses the problem of ensuring security for C/C++ software systems, where static code analysis serves as a key tool for vulnerability detection but remains costly to maintain due to the complexity of developing rules and taint specifications. Existing approaches are analyzed: industrial static analyzers (Svace, Irbis), IFDS analysis, ML/NN-based solutions, and hybrid schemes involving large language models (LLMs) used for warning filtering, code generation, and automatic rule synthesis (QLPro, KNighter, CodePatchLLM). It is shown that direct use of LLMs as standalone vulnerability detectors yields high recall but is accompanied by a significant number of false positives and hallucinations. As an alternative, a multi-agent module for the SAST tool CodeQL, tailored for C/C++, is proposed. It leverages LLMs for the generation, validation, and evolution of rules in the QL query language within a closed loop of "generation, verification, refinement, filtering, ranking". The system architecture is described, featuring generator, validator, refiner, filter, and ranker agents, whose orchestration is implemented using LangGraph and PostgreSQL. Rule validation is performed against CodeQL databases Juliet, SARD, and real-world projects, with calculation of metrics including Precision, Recall, F1, FPR, and Specificity, and consideration of CVSS scores. This enables the selection of rules of acceptable quality and reduces the burden on developers through automated evolution of the rule base.

Keywords: static code analysis, software system security, C/C++, SAST tools, CodeQL, large language models, agent-based LLM systems, static analysis rule generation, CWE

* *Received 22 January 2026.*

REFERENCES

1. Hu J., Jin X., Zeng Y., Liu Y., Li Y., Du D., Xie K., Zhu H. QLPro: Automated code vulnerability discovery via LLM and static code analysis integration. *arXiv preprint*. 2025. arXiv:2506.23644v1.

2. O bezopasnosti kriticheskoi informatsionnoi infrastruktury Rossiiskoi Federatsii [On the security of the critical information infrastructure of the Russian Federation]. Federal Law of June 14, 2023 No. 187-FZ. *Sobranie zakonodatel'stva Rossiiskoi Federatsii = Collection of the legislation of the Russian Federation*, 2023, no. 25, art. 4–5, 8, 10.

3. O personal'nykh dannykh [On personal data]. Federal Law of July 27, 2006 No. 152-FZ. *Sobranie zakonodatel'stva Rossiiskoi Federatsii = Collection of the legislation of the Russian Federation*, 2006, no. 31, art. 19.

4. Belevantsev A.A. Mnogourovnevyi staticheskii analiz iskhodnogo koda programm dlya obespecheniya kachestva programm [Multilevel static analysis for improving program quality]. *Programmirovanie = Programming and Computer Software*, 2017, no. 6, pp. 3–25. (In Russian).

5. Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Irbis: staticheskii analizator pomechennykh dannykh dlya poiska uyazvimostei v programmakh na C/C++ [Irbis: static taint analyzer for vulnerabilities detection in C/C++]. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the RAS*, 2022, vol. 34, iss. 6, pp. 51–66. DOI: 10.15514/ISPRAS-2022-34(6)-4.

6. Leonov N.V., Buinevich M.V. Mashinnoe obuchenie vs poisk uyazvimostei v programmnom obespechenii: analiz primenimosti i sintez kontseptual'noi sistemy [Machine learning vs vulnerability detection in software: applicability analysis and conceptual system synthesis]. *Trudy uchebnykh zavedenii svyazi = Proceedings of Telecommunication Universities*, 2023, vol. 9, no. 6, pp. 83–94. DOI: 10.31854/1813-324X-2023-9-6-83-94.

7. Panov D.D., Shimchik N.V., Chibisov D.A., Belevantsev A.A., Ignatyev V.N. Povyshenie tochnosti staticheskogo analiza koda pri pomoshchi bol'shikh yazykovykh modelei [Improving static code analysis accuracy using large language models]. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the RAS*, 2025, vol. 37, iss. 6, pt. 1, pp. 83–100. DOI: 10.15514/ISPRAS-2025-37(6)-5.

8. Shaikhelislamov D.S., Drobyshevskii M.D., Belevantsev A.A. Interaktivnaya generatsiya koda na osnove LLM: empiricheskaya otsenka [Interactive code generation based on LLM: empirical evaluation]. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the RAS*, 2025, vol. 37, iss. 5, pp. 123–130. DOI: 10.15514/ISPRAS-2025-37(5)-9.

9. Yang C., Zhao Z., Xie Z., Li H., Zhang L. KNighter: Transforming static analysis with LLM-synthesized checkers. *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, 2025, pp. 1–15. DOI: 10.1145/3731569.3764827.

10. Purba M.D., Ghosh A., Radford B.J., Chu B. Software vulnerability detection using large language models. *IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2023, pp. 112–119. DOI: 10.1109/ISSREW60843.2023.00058.

11. Eghbali A., Pradel M. De-hallucinator: Mitigating LLM hallucinations in code generation tasks via iterative grounding. *arXiv preprint*. 2024. arXiv:2401.01701v3.

12. Avgustinov P., Moor O., Jones MP., Schäfer M. QL: Object-oriented queries on relational data. *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:25.

Для цитирования:

Архипова А.Б., Азеева Е.Д. Многоагентная система генерации и эволюции правил статического анализа CodeQL на базе LLM для C/C++ // Безопасность цифровых технологий. – 2026. – № 1 (120). – С. 9–28. – DOI: 10.17212/2782-2230-2026-1-9-28.

For citation:

Arkhipova A.B., Azeeva E.D. Mnogoagentnaya sistema generatsii i evolyutsii pravil staticheskogo analiza CodeQL na baze LLM dlya S/S++ [LLM-based multi-agent system for generation and evolution of CodeQL static analysis rules for C/C++]. *Bezopasnost' tsifrovykh tekhnologii = Digital Technology Security*, 2026, no. 1 (120), pp. 9–28. DOI: 10.17212/2782-2230-2026-1-9-28.