УДК 004.414.38:004.032.26
DOI

ПОДХОДЫ СТАТИЧЕСКОГО АНАЛИЗА КОДА С ИСПОЛЬЗОВАНИЕМ НЕЙРОННЫХ СЕТЕЙ

Удалова Ю.В.

ФГАОУ ВО «Сибирский федеральный университет», Красноярск, e-mail: uuuu82@inbox.ru

С быстрым развитием информационных технологий программное обеспечение играет важную роль в различных аспектах и сферах жизнедеятельности человека, в то же время потенциальные проблемы безопасности становятся все более серьезными и комплексными. С усложнением требований к программному коду, появлением новых рисков и угроз в цифровой сфере, традиционных методов анализа и контроля становится недостаточно. В связи с этим цель статьи заключается в исследовании возможности применения нейронных сетей для задач статического анализа кода и особенностей решения ими проблем, присущих классическим статическим анализаторам, а также в изучении практической реализации статического анализа кода с помощью графовой нейронной сети. Применены методы: системный анализ, моделирование, прогнозирование, численная оптимизация, теория вероятности, систематизация, обобщение. В процессе исследования детально описаны возможности нейронных сетей в статическом анализе кода. Также выделены сферы проверки кода, в которых нейронные сети могут найти свое широкое применение. Отдельное внимание уделено сравнению архитектур нейронных сетей для целей статического анализа, рассмотрены такие модели, как многослойный перцептрон, рекуррентная нейронная сеть, графовая нейронная сеть. Практические аспекты использования нейронного инструментария машинного обучения в программировании рассмотрены на примере поиска неправильного использования переменных в коде на базе графовой нейронной сети. Применение нейронных сетей для статистического анализа кода позволяет снизить количество ложноположительных результатов и повысить уровень истинно положительных обнаружений ошибок и неточностей в его написании.

Ключевые слова: нейронная сеть, анализ, ошибка, код, переменная, обучение, граф, вероятность

APPROACHES OF STATIC CODE ANALYSIS USING NEURAL NETWORKS

Udalova Yu.V.

Siberian Federal University, Krasnoyarsk, e-mail: uuuu82@inbox.ru

With the rapid development of information technology, software plays an important role in various aspects and spheres of human life, while potential security problems are becoming increasingly serious and complex. With the increasing complexity of software code requirements and the emergence of new risks and threats in the digital sphere, traditional methods of analysis and control are becoming insufficient. In this regard, *the aim* of this article was to consider the features of using neural networks in the process of static code analysis as a potential way to solve problems present in classical approaches, as well as to study the practical aspects of its implementation using the example of a graph neural network. *Materials and methods*: system analysis, modelling, forecasting, numerical optimisation, probability theory, systematisation, generalisation. The study describes in detail the capabilities of neural networks in static code analysis. It also highlights areas of code verification where neural networks can find wide application. Special attention is paid to comparing neural network architectures for static analysis, considering models such as multilayer perceptrons, recurrent neural networks, and graph neural networks. Practical aspects of using machine learning neural tools in programming are considered using the example of searching for incorrect use of variables in code based on a graph neural network. The use of neural networks for statistical code analysis reduces the number of false positives and increases the level of true positives in detecting errors and inaccuracies in code.

Keywords: neural network, analysis, error, code, variable, training, graph, probability

Введение

Статистический анализ программного кода – это обширная область исследования языков программирования, которая на протяжении десятилетий не теряет своей актуальности и практической значимости [1]. Целью анализа кода является определение свойств программы в отношении ее поведения. Статический анализ проверяет код на наличие ошибок или плохих практик без выполнения самого кода. Это означает, что код читается, возможно, компилируется и проверяется на наличие заранее предопределенных шаблонов, которые помечаются как потенциально проблемные. В некоторых случаях статический анализ также предлагает исправления, которые изменяют эти шаблоны, чтобы устранить предполагаемую проблему. Традиционно методы статического анализа призваны предоставлять формальные гарантии относительно некоторого свойства программы, например что выход функции всегда удовлетворяет некоторому условию или что программа всегда завершается. Для обеспечения таких гарантий традиционный анализ кода опирается на строгие математические методы, которые могут детерминированно и окончательно доказать

или опровергнуть формальное утверждение о поведении программы [2].

Повышение объема кода и возрастание сложности программ и программных комплексов затрудняет процесс аналитического рассмотрения кода на предмет поиска ошибок, проблем, тупиков и неэффективных фрагментов программ. Классические статические анализаторы (например, SonarQube или Checkstyle), проверяющие синтаксические правила языков программирования и заданные правила и соглашения о необходимом построении программного кода, испытывают затруднения с определением сложных семантических ошибок, способных возникнуть даже в небольших программах, и паттернов уязвимостей, риски появления и сложность которых значительно возрастают в объемном программном коде. Другой недостаток, типичный для классических статических анализаторов, это высокий риск ложных предупреждений.

Альтернативой являются анализаторы, применяющие машинное обучение (например, CodeBERT или GraphCodeBERT), обучающиеся на больших объемах программного кода и комментариях к нему, с их помощью можно выявлять не только синтаксические, но и семантические ошибки и уязвимости, а также реализовывать дополнительные опции, например рефакторинг и перевод программного кода. Анализаторы, использующие генетические алгоритмы и машинное обучение, потенциально способны снизить риски появления ложноположительных срабатываний. Поэтому использование инструментов статического анализа на базе машинного обучения и внедрение вероятностных моделей поиска ошибок является перспективной областью научного знания.

Кроме указанных преимуществ применение нейронных сетей для статического анализа кода требует решения дополнительных подзадач и сталкивается со следующими проблемами. Для корректной работы статического анализатора с машинным обучением требуется объемный и качественный набор данных для обучения. Интерпретируемость решений, принимаемых моделями глубокого обучения, остается сложной задачей, что затрудняет анализ и отладку ошибок, обнаруженных ИИ [3]. Также отмечается высокая вычислительная стоимость обучения и работы таких моделей [4–6], что может быть неприемлемо для крупных кодовых баз. Другая важная проблема – это сложность обобщения моделей. Они могут плохо работать с новыми или редко встречающимися паттернами в коде, что приводит к ошибкам и ложным срабатываниям [7].

Поэтому углубление исследований в данном направлении составляет важную научно-практическую задачу, которая и предопределила выбор темы данной статьи.

Перспективы использования нейронных сетей для сбора лексических и синтаксических знаний на уровне предложений с дополнительным контекстом об ошибках и паттернах проектирования кода рассматривают в своих трудах М.В. Буйневич, К.Е. Израилов и др. [3], Sixuan Wang, Chen Huang [4], Jia Yang, Ou Ruan [5].

Над разработкой нейронной сети, которая может специально имитировать механизм исправлений существующих статических анализаторов, а также экстраполировать его на новые исправления, учитывая любое диагностическое сообщение, трудятся Д.А. Потапов, С.В. Корниенко [6], Chitti Babu Karakati, Sethukarasi Thirumaaran [7], И.В. Котенко, И.Б. Саенко, О.С. Лаута, А.С. Юрьев, М.С. Запруднов [8].

Возможности улучшения подхода к эффективной обработке сигналов тревоги статического анализа, который сочетает использование функций предупреждения, встраивание контекста, сгенерированного предварительно обученной моделью и различными архитектурами нейронных сетей, входят в круг научных интересов А.В. Козачка, Н.С. Ерохиной, Д.А. Николаева [9], Eren Bas, Erol Eğrioğlu [10], Peng Zeng, Guanjun Lin [11].

Однако, несмотря на многочисленные исследования и рекомендации, в данной предметной плоскости остается еще широкий спектр вопросов, которые до конца не решены. Разнообразные типы нейронных сетей и подходы к предварительной обработке данных влияют на работу и производительность модели, затрудняют решение задачи выбора максимально эффективной архитектуры. Подбор компонентов, включаемых в представление кода, для повышения качества результатов анализа и выявления ошибок и дефектов также является сложной интеллектуальной задачей.

Цель исследования – исследовать возможности применения нейронных сетей для задач статического анализа кода и особенности решения ими проблем, присущих классическим статическим анализаторам, а также в изучении практической реализации статического анализа кода с помощью графовой нейронной сети.

Материалы и методы исследования

Применены методы: системный анализ, моделирование, прогнозирование, численная оптимизация, теория вероятности, систематизация, обобщение. В процессе исследования проведено сравнение архитектур нейронных сетей для целей статического анализа, рассмотрены такие модели, как многослойный перцептрон, рекуррентная нейронная сеть, графовая нейронная сеть.

Результаты исследования и их обсуждение

Разработчики традиционно используют статический анализ для поиска гарантий: например, чтобы убедиться, что программа никогда не достигнет нежелательного состояния. Динамический анализ, в свою очередь, применяется для проверки конкретных аспектов выполнения программы, например, чтобы подтвердить, что определенные входные данные приводят к ожидаемым результатам. В отличие от этих методов, машинное обучение предлагает иной подход: оно помогает моделировать вероятности возникновения определенных событий, что может быть использовано для предсказания ошибок или уязвимостей до того, как они проявятся в коде. Развивающаяся область машинного обучения для кода наглядно продемонстрировала тот факт, что различные архитектуры нейронных сетей могут найти свое применение к исходному коду для решения ряда задач программной инженерии. Предпосылкой является то, что, хотя код имеет детерминированную, однозначную структуру, программисты в процессе его написания используют паттерны и неоднозначную информацию (например, комментарии, имена переменных), которые ценны для понимания его функциональности [12]. Именно этим явлением может воспользоваться анализ программ на базе нейронных сетей. В частности, машинное обучение может помочь анализу программ справиться с двумя общими источниками неоднозначности: скрытыми спецификациями и двойственными контекстами выполнения (например, из-за динамически загружаемого кода).

В табл. 1 систематизированы достоинства и недостатки нейронных сетей, таких как графовые, сверточные и трансформеры, чаще всего используемых для статистического анализа кода.

Последние достижения в области машинного обучения произвели революцию в представлении исходного кода, породив методы, которые могут быть сгруппированы в четыре категории: текстовые, лексические, синтаксические и семантические. Каждая группа обладает определенными характеристиками, а также имеет ряд достоинств и недостатков. Например, текстовые представления, такие как модели *n*-грамм, обученные на обширных корпоративных массивах, оказываются эффективными для предсказания лексем в различных областях [13].

Таблица 1

Описание возможностей нейронных сетей в статистическом анализе кода

Задача статического анализа	Тип метода	Преимущества	Недостатки
Автоматическое обнаружение уязвимостей	Обнаружение уязвимостей на основе сходства кода	На основе исходного кода обнаруживается множество типов клонов; на основе двоичного кода достигается более высокая точность обнаружения	Высокий процент ложноотрицательных результатов (исходный код); сложность анализа (двоичный код)
	Обнаружение уязвимостей на основе шаблонов кода	Нейронные сети позволяют обеспечить более высокий код	Недостаток информации о времени выполнения, низкое покрытие кода
Автоматическое исправление программ	Патчирование программ на основе грамматики	Анализ ошибок с помощью метода, основанного на токенах; простой метод, основанный на тексте, генерирует более качественные патчи	Плохая интерпретируемость патчей (токенов)
	Исправление программ на основе семантики	Метод достигает хорошего эффекта динамического ремонта и точно передает поведение программы	Ограниченные возможности исправления, высокая стоимость
Автоматическое прогнозирование дефектов	Предсказание дефектов в рамках проекта	Сплошная реализация, точное предсказание дефектных программных модулей	Некачественное расширение
	Прогнозирование дефектов по всем проектам	Эффективная интеграция ресурсов набора данных для лучшего продвижения новых практик разработки проектов	Чрезмерная детализация извлечения характеристик кода
	Прогнозирование дефектов точно в срок	Раннее выявление неисправных модулей и тонкий анализ, эффективно определяющий количество дефектов	Отсутствие большого количества обучающих данных для режима тренировок нейронной сети

Источник: составлено автором.

Подходы к применению нейронных сетей в статическом анализе кода
Источник: составлено автором

Представления на основе лексики обеспечивают возможность абстрагирования, в то время как представления на основе синтаксиса и семантики дают возможность достичь более высокого уровня абстракции, но требуют предварительной обработки, например, преобразования исходного кода в древовидную или графовую структуру [14]. Методы, основанные на синтаксисе и семантике, часто используют абстрактные синтаксические деревья (AST – abstract syntax trees), такие как ASTNN (abstract syntax tree neural network – нейронная сеть на основе абстрактного синтаксического дерева), которые учатся синтаксическим знаниям на основе более мелких поддеревьев. Методы, основанные на семантике, часто включают информацию о зависимостях кода, относящуюся к потоку данных и управления [15].

Систематизируя имеющиеся на сегодняшний день разработки, на рисунке представлена общая структура категоризации подходов к применению нейронных сетей в статическом анализе кода.

Как уже отмечалось ранее, для статического анализа кода с использованием нейронных сетей могут использоваться различных архитектуры и модели. В рамках проводимого исследования проведем сравнительный анализ трех наиболее популярных архитектур – многослойный перцептрон (MLP – Multi-layer perceptron), рекуррентная нейронная сеть (RNN – recurrent neural network) и графовая нейронная сеть (GNN – graph neural network).

MLP – это относительно простой алгоритм, который предполагает использование нескольких слоев узлов для составления прогноза на основе входных признаков. Однако данная архитектура имеет небольшую сферу применения в рамках проведения статического анализа кода. Это связано с ограниченными возможностями MLP по захвату всей информации, которая содержится во входных признаках, что затрудняет обучение модели и восприятие более сложных закономерностей, присутствующих в данных. В большинстве случаев на практике

MLP применяется в качестве базовой модели для установления эталона производительности при сравнении с последующими моделями [10].

Несмотря на то, что GNN часто ассоциируются с задачами компьютерного зрения, они также находят свое успешное применение в некоторых сферах статистического анализа кода. Например, к ним относится классификация предложений. Сеть рассматривает текстовые данные как одномерную последовательность контекстных вложений слов, где каждое слово представлено вектором в высокоразмерном пространстве. В работах Hongwei Tu, Yanqiang Han [15] GNN используются для применения сверточных операций к последовательности контекстных вложений с целью выявления локальных паттернов или особенностей в каждом входном предложении. Затем эти особенности проходят через слой объединения для уменьшения их размерности, после чего поступают в слой с полным подключением для классификации. Применяя сверточные операции к контекстным вложениям в коде, GNN могут научиться распознавать локальные паттерны или особенности в данных о сигналах предупреждений, что очень важно для отсеивания тех, которые являются ложными.

RNN являются эффективным инструментом, который позволяет точно и эффективно моделировать последовательные данные, это объясняется тем, что они способны поддерживать скрытое состояние, которое обновляется на каждом временном шаге. В качестве примера использования на практике RNN можно привести работу Chitti Babu Karakati, Sethukarasi Thirumaaran [7], в которой ученые предложили архитектуру двунаправленной рекуррентной нейронной сети (BiRNN – bidirectional recurrent neural network), включающей двунаправленный GRU-слой и полностью связанный слой для классификации. Двунаправленный GRU-слой состоит из двух наборов GRU-слоев: один обрабатывает входную последовательность слева направо, а другой – справа налево. Такая конфигурация позволяет модели учитывать контекстную информацию и эффективно моделировать входную последовательность в двунаправленном режиме. Модель BiRNN принимает последовательности контекстных сложений в качестве входных данных и направляет их через двунаправленный GRU-слой, который полностью включает в себя кодированное представление контекстной информации на выходе. Выходные данные слоя GRU в конечном итоге поступают в слой с полным подключением для классификации с целью распознавания предупреждений об ошибках в коде, которые могут быть приняты.

Таблица 2

Характеристика архитектур нейронных сетей для статистического анализа кода

Задача статистического анализа	GNN	RNN	MLP
Обнаружение сходства кода	Итерация, отсев, скрытый_слой, скорость_градиента	Скрытый_слой, глубина сети, отсев, размер_пакета, итерация	Скрытый_слой, итерация, отсев, размер_слоя, скорость_обучения
Обнаружение шаблонов кода	Размер_фильтра, количество_фильтров, скрытый_слой	Отсев, размер партии, итерация, скорость обучения, размерность вектора	–
Грамматическое исправление программ	–	Скорость_обучения, размер_пакета, скрытый_слой, длина_лексики, скрытая_единица, итерация, размер_вставки, градиентный_оптимизатор	–
Семантическое исправление программ	–	Скрытая_единица, градиентный_оптимизатор, векторное_размерение, скорость_обучения, итерация, отсев, скрытое_размерение	–
Предсказание дефектов в рамках проекта	Число_фильтров, размер_фильтра, скрытый_узел, размер_партии, итерация, размер_вкрапления	Отсев, размерность_вектора, скрытый_слой, скрытый_узел, размер_пакета, итерация, скорость_обучения	Скрытый_слой, скрытый_узел, итерация, скорость_обучения, градиентный_оптимизатор, размер_пакета
Предсказание межпроектных дефектов	Размер_фильтра, количество_фильтров, скорость_обучения, размер_вектора	Отсев, скрытый_слой, скрытый_узел, размер_партии	Итерация, скрытый_слой, скрытый_узел

Источник: составлено автором.

Таблица 3

Сводка вершинных признаков, использованных для блоков графа потока управления

Название характеристики	Тип	Категория	Описание
Гистограмма фиксированных узлов	Гистограмма	Типы узлов	Гистограмма, отражающая распределение фиксированных узлов в блоке
Гистограмма плавающих узлов	Гистограмма	Тип узла	Гистограмма, отражающая распределение плавающих узлов в блоке
Расчетное количество циклов процессора	Числовой	Характеристика кода	Расчетное количество циклов процессора, необходимое для выполнения узлов в блоке
Расчетный размер сборки	Числовой	Характеристика кода	Расчетный размер узлов в блоке
Центральный процессор («дешевый граф»)	Числовой	Характеристика кода	Количество узлов в блоке, требующих 0 или 1 цикл процессора
Центральный процессор («дорогой граф»)	Числовой	Характеристика кода	Количество узлов в блоке, требующих 64 или более циклов процессора
Глубина разделения управления	Числовой	Поток управления	Количество блоков, содержащих cfs-узлы с преемником, который доминирует над данным блоком
Глубина циклов	Числовой	Поток управления	Количество вложенных циклов, охватывающих данный блок
Глубина доминатора	Числовой	Поток управления	Глубина блока в дереве доминатора

Источник: составлено автором.

В табл. 2 представлена сравнительная характеристика рассмотренных архитектур.

Рассмотрим практический пример применения нейронной сети для статического анализа кода. Учитывая преобладание графовых представлений кода, используем GNN для проведения статического анализа. GNN стали ценным инструментом для анализа программных кодов благодаря их гибкости в обучении на основе широкого спектра паттернов и простоте комбинирования с другими нейросетевыми компонентами. Получив представление графа программы, GNN вычисляют вкрапления сети для каждого узла, чтобы использовать их в последующих задачах.

На вход сети будет подаваться граф потока управления, что имеет ряд практических преимуществ для вывода профиля кода. Обозначим их более подробно.

1. Так как компиляторы, как правило, работают с графом потока управления, применение модели GNN и графа управления органично подходит для решения задач статического и динамического анализа кода.

2. Модели GNN свойственно минималистичное представление графа, эффективно абстрагирующее наиболее важную информацию о связях и поведении программы.

3. Компактное представление графа потока управления, сохраняющее при этом всю важную информацию, повышает эффективность обучения модели и обеспечивают быструю сходимость.

Как показано в табл. 3, вершинные признаки графа потока управления делятся на три группы в зависимости от характеристик, которые они описывают.

Итак, статический анализ начинается с того, что сначала каждая сущность/узел v_i встраивается в векторное представление n_{vi}. Графы программ содержат богатую и разнообразную информацию в своих узлах, такую как значимые имена идентификаторов (например, max_len). Чтобы воспользоваться информацией, содержащейся в каждом узле с маркерами и символами, его строковое представление субтокенизируется (например, «max», «len»), и каждое исходное представление узла n_{vi} вычисляется путем объединения вкраплений субтокенов, то есть для узла v_i и объединения сумм входное представление узла вычисляется как

$$n_{vi} = \sum_{s \in Субтокен(v_i)} t_s,$$

где t_s – выученное вкрапление для подтокена s. Для синтаксических узлов их начальное состояние – вкрапление типа узла. Затем любая архитектура GNN, способная обрабатывать направленные неоднородные графы, может быть использована для вычисления вкраплений сети, то есть:

$$\left\{h_{v_i}\right\} = GNN\left(\mathcal{J}', \left\{n_{v_i}\right\}\right),$$

где GNN обычно имеет фиксированное число «слоев» (например, 8), $\mathcal{J}' = \left(\mathcal{V}, \mathcal{E} \cup \mathcal{E}_{inv}\right)$, а $\mathcal{E}_{inv}$ – множество обратных ребер $\mathcal{E}$, т.е. $\mathcal{E}_{inv} = \left\{\left(v_j, r^{-1}, v_i\right), \forall\left(v_i, r, v_j\right) \in \mathcal{E}\right\}$.

Вложения сети $\{h_{v_i}\}$ являются входными данными для нейронной сети, ориентированной на конкретную задачу.

Например, цель анализа неправильного использования переменных в коде состоит в том, чтобы (1) локализовать ошибку (если она существует), указав на неисправный узел, и (2) предложить исправление. Для этого предположим, что GNN вычислила вложения сети $\{h_{v_i}\}$ для всех узлов $v_i \in \mathcal{V}$ в программном графе G. Тогда пусть $\mathcal{V}_{vu} \subset \mathcal{V}$ – это множество узлов-токенов, которые ссылаются на использование переменных. Во-первых, модуль локализации нацелен на то, чтобы точно определить, какое использование переменной (если таковое имеется) является злоупотреблением переменной. Он может быть реализован в виде сети указателей на $\mathcal{V}_{vu} \cup \{\varnothing\}$, где $\varnothing$ обозначает событие «нет ошибки», с обучаемым вложением $h_{\varnothing}$. Затем, используя (обучаемую) проекцию u и *softmax*, можно вычислить распределение вероятностей по $\mathcal{V}_{vu}$ и специальное событие «отсутствие ошибки»:

$$p_{loc}(v_i) = \underset{v_j \in \mathcal{V}_{vu} \cup \{\varnothing\}}{softmax}\left(u^{\dot{O}} h_{v_i}\right).$$

Таким образом, GNN, обнаружив ошибку злоупотребления переменными в определенной строке, присвоит высокий p_{loc} узлу, соответствующему токену, который является местом расположения ошибки злоупотребления переменными.

Выводы

Резюмируя полученные результаты, сделаем следующие выводы:

1. Графовые нейронные сети имеют большой потенциал для повышения эффективности решения задач статического анализа кода. Графы потоков управления, применяемые при решении таких задач с помощью модели GNN, эффективно и компактно представляют программный код и лежат в основе работы компиляторов, что обеспечивает их эффективность и органичность использования.

2. Технологии искусственного интеллекта и методы машинного обучения, тренирующиеся на больших данных и сохраняющие исторические данные, способны понижать риск ложноположительных срабатываний, присущий классическим статическим анализаторам кода.

Список литературы

1. Пантюхин М.А. Роль машинного обучения в оптимизации программного кода // Научный аспект. 2024. Т. 9. № 5. С. 1156–1161. EDN: OEWNVQ

2. Ломако А.Г., Исаев Н.Э., Менисов А.Б., Сабиров Т.Р. Подход к выявлению уязвимостей программного кода на основе адаптации с подкреплением предобученных моделей машинного обучения // Проблемы информационной безопасности. Компьютерные системы. 2025. № 1 (63). С. 83–96. DOI: 10.48612/jisp/7gnx-9z7f-fbrv.

3. Буйневич М.В., Израилов К.Е., Покусов В.В., Романов Н.Е. Способ вариативной классификации уязвимостей в программном коде. Ч. 2. Автоматизация на базе машинного обучения // Автоматизация в промышленности. 2022. № 4. С. 49–55. DOI: 10.25728/avtprom.2022.04.10.

4. Sixuan Wang, Chen Huang VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network // Software: Practice and Experience. 2023. Vol. 53, Is. 8. P. 1631–1658. DOI: 10.1002/spe.3205.

5. Jia Yang, Ou Ruan Tensor-based gated graph neural network for automatic vulnerability detection in source code // Software Testing, Verification and Reliability. 2023. Vol. 34, Is. 2. № e1867. DOI: 10.1002/stvr.1867.

6. Потапов Д.А., Корниенко С.В. Модель системы обнаружения дефектов программной среды на основе глубокого обучения с наиболее подходящими гиперпараметрами // Современная наука: актуальные проблемы теории и практики. Серия: Естественные и технические науки. 2024. № 6. С. 111–115. DOI: 10.37882/2223-2966.2024.06.33.

7. Chitti Babu Karakati, Sethukarasi Thirumaaran Software code refactoring based on deep neural network-based fitness function // Concurrency and Computation: Practice and Experience. 2023. Vol. 35, Is. 4. P. e7531. DOI: 10.1002/cpe.753.

8. Котенко И.В., Саенко И.Б., Лаута О.С., Юрьев А.С., Запруднов М.С. Протестное программное обеспечение: анализ и подход к обнаружению, основанный на машинном обучении // Вопросы кибербезопасности. 2024. № 6. С. 42–52. DOI: 10.21681/2311-3456-2024-6-42-52.

9. Козачок А.В., Ерохина Н.С., Николаев Д.А. Способ обнаружения программных дефектов в javascript-интерпретаторах методом фаззинг-тестирования // Вопросы кибербезопасности. 2024. № 2 (60). С. 74–80. DOI: 10.21681/2311-3456-2024-2-74-80.

10. Eren Bas, Erol Eğrioğlu A new recurrent pi-sigma artificial neural network inspired by exponential smoothing feedback mechanism // Journal of Forecasting. 2023. Vol. 42, Is. 4. P. 802–812. DOI: 10.1002/for.2919.

11. Peng Zeng, Guanjun Lin. Intelligent detection of vulnerable functions in software through neural embedding-based code analysis // International Journal of Network Management. 2023. Vol. 33, Is. 3. P. 50–54. DOI: 10.1002/nem.2198.

12. Changhui You, Hong Zheng. Tampering detection and localization base on sample guidance and individual camera device convolutional neural network features // Expert Systems. 2023. Vol. 40, Is. 1. P. 48–56. DOI: 10.1111/exsy.13102.

13. Романов Н.Е. Система поддержки интеллектуального программирования: машинное обучение feat. Быстрая разработка безопасных программ // Информатизация и связь. 2021. № 5. С. 7-16. DOI: 10.34219/2078-8320-2021-12-5-7-16.

14. Мяличева А.А. Анализ методов машинного обучения для прогнозирования дефектов в исходном коде // Труды Северо-Кавказского филиала Московского технического университета связи и информатики. 2024. № 2. С. 16-19. EDN: IVJCZF.

15. Hongwei Tu RotNet: A Rotationally Invariant Graph Neural Network for Quantum Mechanical Calculations // Small Methods. 2024. Vol. 8. Is. 1. P. e2300534. DOI: 10.1002/smtd.202300534.