УДК 004.891.3

РАЗРАБОТКА И ИССЛЕДОВАНИЕ СТАТИЧЕСКОГО АНАЛИЗАТОРА ИСХОДНОГО КОДА НА ОСНОВЕ ТРАНСФОРМЕРНОЙ МОДЕЛИ ГЛУБОКОГО ОБУЧЕНИЯ

Н.С. Корнев[1], Т.П. Рубцова[1]

[1]Самарский университет, г. Самара
e-mail: kornev2003nikita@gmail.com

Аннотация

Разработан статический анализатор кода на базе архитектуры Transformer на Java 22 без специализированных ML-фреймворков. Описаны механизмы внимания, оптимизатор Adam и backpropagation, реализованные с помощью apache.commons.math3. Модель обучена на сбалансированном датасете из 828 функций. Достигнута точность 0,91 в классификации 8 типов уязвимостей. Установлена оптимальность однослойной архитектуры для малых выборок. Прототип эффективно выявляет дефекты, недоступные традиционным методам анализа.

Ключевые слова: трансформер, статический анализ кода, Java, обратное распространение ошибки.

Введение

Автоматизация поиска уязвимостей в исходном коде является одной из ключевых задач обеспечения безопасности программного обеспечения, особенно в условиях быстрых циклов в командах backend-разработки. Традиционные методы статического анализа (SAST), основанные на правилах и эвристиках, часто страдают от большого количества ложноположительных срабатываний и неспособности обнаруживать сложные, контекстно-зависимые уязвимости [1].

Целью работы является разработка статического анализатора исходного кода, основанного на трансформерной модели глубокого обучения CodeBERT, реализация основана на классической статье [2].

1. Подготовка (парсинг) исходного кода и позиционное кодирование.

Для достижения поставленной цели необходимо решить следующие подзадачи: определить требования к разрабатываемому анализатору с учетом особенностей корпоративных Java-систем: разработать архитектуру программного решения на языке Java; реализовать прототип анализатора с использованием выбранной модели; обеспечить интеграцию анализатора в типичное веб-приложение; провести оценку эффективности и применимости разработанного решения. Процесс парсинга кода состоит из нескольких стадий.

Первая стадия – в классе «TrainingData.java» выделяются значимые части кода с помощью регулярных выражений: «"\s+|(?<=[(){};,.])|(?=[(){};,.])"» и «"[^a-zA-Za-яА-Я0-9@=+\\-*/|;.(){}\n]"». Благодаря ним все заглавные буквы преобразуются в строчный формат, а также скобки, точки с запятой, запятые, арифметические операции, специальные символы табуляции, переноса строки и точки сохраняются и обозначаются как отдельные

токены. Комментарии, обозначающиеся с помощью конструкций «/* … */» и «// … \n», не токенизируются.

На второй стадии в классе «Vocabulary.java» на основе всего обучающего корпуса формируется словарь, который сопоставляет каждому уникальному токену целочисленный индекс. В словарь также добавляются специальные токены: [PAD] (или нулевой токен) для выравнивания длины последовательностей, [CLS] для классификации при вычислении функции потерь, и [UNK] для токенов, не встречавшихся при обучении.

На третьей стадии в классе «EmbeddingTable.java» создается обучаемая матрица эмбеддингов размером V × D, где V — размер словаря, а D — размерность эмбеддинга. Каждый токен представляется в виде плотного D-мерного вектора.

На четвертой стадии случайно сгенерированная матрица в классе эмбеддингов передается в класс «PositionalEncoding.java», где каждый элемент приобретает значение функции синуса для четных элементов и значение косинуса, если элемент стоит под нечетным индексом в векторе из матрицы эмбеддингов. Расчет идет по формулам (1) и (2):

$$(pos, 2i) = sin(pos/10000^{((2i)/d_{model})}, \quad (1)$$

$$(pos, 2i+1) = cos(pos/10000^{((2i)/d_{model})}. \quad (2)$$

В текущей реализации при размерности эмбеддинга (embeddingDim) = 64 и максимальной длины последовательности (maxSeqLength) = 100 символов гарантирует, что при возведении 10000 в степень вектора будут уникальны, так как максимальное число элементов – 6400.

2. Расчет градиентов по цепному правилу и механизм многоголового внимания

Механизм многоголового внимания, реализованного в классе «MultiHeadAttention.java», позволяет модели взвешивать важность различных токенов в последовательности для формирования контекстно-зависимого представления каждого токена. Для входной матрицы X каждая голова i сначала вычисляет проекции матриц запроса (Q_i), ключа (K_i) и значения (V_i). Затем выход каждой головы вычисляется с помощью функции, описанной в формуле (3):

$$Attention(Q_i, K_i, V_i) = softmax\left((Q_i K_i^T)/sqrt(d_k)\right) * V_i, \quad (3)$$

где d_k – размерность векторов ключа.

Для задачи многоклассовой классификации используется перекрестная энтропия с Softmax. На forward проходе она вычисляет скалярную ошибку, а на backward – стартовый градиент, который инициирует весь процесс обратного распространения. Для расчета градиентов использовались формулы (4), (5) и (6):

$$\frac{\partial L}{\partial W} = x^T \frac{\partial L}{\partial y}, \quad (4)$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial x} W^T, \quad (5)$$

$$\frac{\partial L}{\partial b} = sum(\frac{\partial L}{\partial y}). \quad (6)$$

Для обновления весов на основе вычисленных градиентов реализован оптимизатор Adam [4]. Оптимизатор адаптирует скорость обучения для каждого параметра индивидуально, используя моменты первого и второго порядков градиентов.

3. Результаты экспериментального исследования

Для оценки эффективности реализованной модели был сформирован сбалансированный набор данных, включающий 828 функций Java, распределенных по 8 классам (в таблице 1). В качестве обучающей выборки использовались как примеры реального кода с известными уязвимостями, так и синтетически сгенерированные фрагменты.

Таблица 1 – Метрики качества классификации уязвимостей

Класс уязвимости	Precision	Recall	F1-score
Safe code	0,98	0,87	0,92
Resource Leak	0,72	0,89	0,8
Bad Exception Handling	0,87	0,99	0,92
XXE	0,93	0,86	0,89
NPE risk	1,0	1,0	1,0
RCE	0,97	0,69	0,81
SQL Injection	0,89	1,00	0,94
Unsafe API	0,99	0,94	0,97
Среднее значение	0,92	0,91	0,91

Для более эффективной оценки результатов на рисунке 1 представлена матрица ошибок.

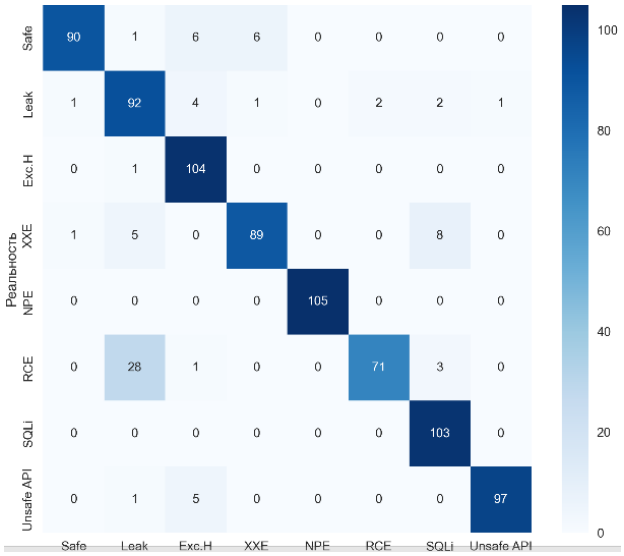

Рисунок 1 – Матрица ошибок предсказания меток

По данным результатам для однослойной модели и 828 примеров сбалансированного датасета, продемонстрированы высокие показатели выявления SQL-инъекций и нулевых значений (NullPointerException). Согласно матрице ошибок, наблюдается частичная интерференция между классами RCE и Resource Leak. Данный эффект объясняется схожестью векторизации функций, взаимодействующих с внешними объектами ввода-вывода и системными процессами. Экспериментально установлено, что для данного объема данных (до 1000 примеров) однослойная архитектура трансформера является оптимальной. Увеличение глубины до трех слоев приводило к затуханию градиентов и деградации точности до 0,12 – 0,15, модель может

использовать трехслойную архитектуру только в случае значительного увеличения датасета (его масштабирования).

Заключение

В работе предложен и реализован статический анализатор исходного кода, совмещающий классические методы статического анализа и продемонстрирована полная реализация этапов обучения модели трансформера на Java для задачи статического анализа безопасности кода по размеченному датасету [5]. Главным результатом является подтверждение возможности эффективного обучения глубоких моделей внимания «с нуля» без использования специализированных ML-фреймворков.

Разработанный прототип демонстрирует высокую точность (91%) в классификации 8 типов программных дефектов и уязвимостей. Доказано, что семантический анализ на основе механизмов Self-Attention превосходит классические методы в задачах, требующих учета контекста выполнения программы.

Список использованных источников

1. Empirical Study on Transformer-based Techniques for Software Engineering. / Y. Xiao, X. Zuo, L. Xue [et al.] // arXiv. – URL: https://arxiv.org/pdf/2310.00399 (дата обращения: 24.03.2026).

2. Attention is all you need / A. Vaswani, N. Shazeer, N. Parmar [et al.] // Adv. Neural Inf. Process. Syst. – 2017. – V. 30.

3. Ba, J. L. Layer normalization / J. L. Ba, J. R. Kiros, G. E. Hinton // arXiv:1607.06450 [cs.LG]. – 2016.

4. Kingma, D. P. Adam: A method for stochastic optimization / D. P. Kingma, J. Ba // arXiv:1412.6980 [cs.LG]. – 2014.

5. Transformer: repository / HyperKa. – URL: https://github.com/HyperKa/Transformer (дата обращения: 24.03.2026).

DEVELOPMENT AND RESEARCH OF A STATIC SOURCE CODE ANALYZER BASED ON A DEEP LEARNING TRANSFORMER MODEL

N.S. Kornev[1], T.P. Rubtsova[1]

[1]Samara University, Samara
e-mail: kornev2003nikita@gmail.com

Abstract

The paper considers the implementation of a Transformer model (CodeBERT architecture) in Java 22 without third-party ML frameworks. The mechanisms of multi-head attention, Adam optimizer, and backpropagation are described, implemented using standard libraries and apache.commons.math3. The model is trained on a balanced sample of 828 functions for vulnerability detection in source code. Experimental results showed a macro-average accuracy of 91%.

Keywords: transformer, static code analysis, Java, backpropagation.