ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

УДК 04.032.26

**Отчет об исследовательском проекте на тему:**
**Применение графовых нейросетей для программного анализа**

**Выполнил:**

Студент группы БПМИ213, 3 курса
Бугаев Егор Петрович

**Принял руководитель проекта:**

Шайхелисламов Данил Салаватович
Старший преподаватель
Факультет компьютерных наук НИУ ВШЭ

Москва 2024

# Содержание

# Аннотация

Сейчас стоит проблема оценки качества кода, написанного большими языковыми моделями (LLM). Учитывая их быстрое развитие и все большее применение в промышленном программировании, важно хорошо понимать, насколько уязвим генерируемый ими код. Одной из целей этой работы является обнаружение уязвимостей кода, созданного LLM, в частности, с помощью использования графовых нейросетей.

Графовые нейронные сети хорошо позволяют находить закономерности в данных, хорошо представляемых в виде графа. В данной работе исследуется применение графовых нейронных сетей для анализа компьютерных программ, представленных в графовом виде. Используются различные виды графовых представлений, в частности, AST (Abstract Syntax Tree), Control Flow графы и другие.

В последнее время было получено множество результатов в сфере применения GNN для компьютерного анализа. В данной области происходит как создание новых моделей, Exphormer [12], так и подбор и анализ новых наборов данных, например, MalNet [3].

Целью данной работы является исследование применения GNN для обнаружения уязвимостей в сгенерированном коде. В частности, изучается модель Devign [15] и сравнивается с традиционным статическим анализатором Svace [5].

По результатам работы GNN позволяют находить уязвимости в сгенерированном коде, которые статический анализатор найти может не всегда. Svace обнаружил уязвимости в 12.5% программ на языке Java среди тех, которые компилируются, в то время как Devign смог найти уязвимости среди 16% программ, написанных на языке C. Кроме того, Devign достиг Accuracy в 92% на валидационной выборке в датасете [2].

# Ключевые слова

**Датасет** (англ. **Dataset**) - набор данных (в данной работе чаще всего - набор программ на разных языках программирования с их характеристикой).

**GitHub** - платформа (удаленный репозиторий), где храниться код и прочие данные проекта.

**Графовые нейросети** - нейронные сети, предназначенный для работы с данными в виде графов.

**Большие языковые модели** (англ. **LLM** - large language model) - языковые модели, состоящие из нейронной сети со множеством параметров. В контексте данной работы подразумеваются модели, специально дообученные на генерирование кода.

**LeetCode** - сайт с задачами на алгоритмическое программирование разного уровня сложности.

**Эмбеддинг** (англ. **Embedding**) - представление (обычно слов языка) в векторном виде. В данной работе подразумевает еще и перевод графовых данных в более привычный для нейросетей вид (то есть тензоры).

**Граф** - математический объект с четкой структурой. В данной работе применяется относительно представления программы как графа с вершинами, отвечающими за части программы в том или ином виде, и ребрами, отражающими некое взаимодействие между этими частями.

# 1 Введение

## 1.1 Описание предметной области

Задача исследования кода на наличие уязвимостей всегда остро стоит при разработке любых программ всех уровней сложности. Это отнимает большие ресурсы у команд разработки, тратит деньги компаний, а ошибки приводят к огромным потерям и большим возможностям для злоумышленников.

Особенно остро она встала сейчас, когда часть кода сгенерирована LLM, и необходимо вдвойне тщательно проверять его работоспособность. Поэтому на новый уровень актуальности выходит тема анализа программ с помощью автоматизированных инструментов. Классическими в этой сфере всегда считались статические анализаторы, к примеру, Svace [5].

Однако в последнее десятилетие с развитием нейросетей набирают обороты исслеедования их приложения к программному анализу. В частности, одним из наиболее интересных способов является применение графовых нейросетей. Были предприняты попытки анализа с помощью более классических глубоких моделей, которые взаимодействовали с кодом как с обычной последовательностью [16], но они не позволяют учитывать сложную настоящую структуру кода, где следующие строки критически зависят от предыдущих, и влияние намного запутаннее, чем в обычном коде.

В то же время оптимальным для анализа структурированных последовательностей (таких как код) видом являются графы. Дальше будут в больших подробностях описаны способы, как можно перевести программу в граф (например, AST деревья), но в целом они позволяют тонко настраивать связи между каждым смысловым блоком кода. Примером могут быть такие модели, как [12].

Не менее важным аспектом, чем разработка самих моделей, является создание больших и разноплановых датасетов. Хорошими примерами являются [3], где заранее собраны графы выполнения вирусных и безопасных приложений для системы Android, а так же датасеты с исходным кодом, такие как следующий набор программ на языках C и C++ [2]

## 1.2 Постановка задачи

Задачей данной работы является исследование методов для анализа кода на наличие уязвимостей в коде программ, сгенерированных нейросетью. Инструмент, который подсказывал бы разработчику, что код, сгенерированный нейросетью, содержит слабые места, помог бы избежать многочисленных взломов и ошибок в будущем, когда LLM еще плотнее войдут в жизнь промышленных разработчиков.

Для данной работы выбрана GNN с архитектурой Devign [15], которая зарекоммендовала себя как одна из метрик качества работы алгоритмов анализа данных. Подробнее про ее архитектуру будет в обзоре литературы.

Кроме того, целью работы являлось создание простого и легко настраиваемого датасета со сгенерированными LLM программами. Это позволило как оценивать качество поиска ошибок и уязвимостей, так и открыло доступ для других экспериментов в дальнейшем.

## 1.3 Структура работы

Работа делится на три существенные части:

- Предобучение и доработка нейросети Devign [15]
- Создание датасета с кодом, сгенерированным нейросетью
- Запуск сети Devign на новом датасете, сравнение с конкуретном

Первой частью работы было получение предобученной версии модели Devign. Так как в открытом доступе предобученной модели найдено не было, вместо этого была взята, доработана и обучена частично реализованная авторами статьи версия (ссылка есть в приложении). Для обучения этой GNN использовался датасет [2], на котором она смогла показать хорошее качество на валидационной выборке.

Второй частью работы было получение датасета с кодом, созданным нейросетью. Для этих целей был использовал сайт `LeetCode.com`, где содержатся задачи на алгоритмическое программирование разной степени сложности. Была реализована программа, которая умеет обращаться к сайту, получать текст задачи, генерировать код с помощью LLM и, опционально, отправлять результат кодогенерации обратно на сайт для получения статуса решения. Для кодогенерации была использована модель CodeLlama [9].

Наконец, третей частью работы было применение обученной нейросети к сгенерированному выше коду. Для проверки качества генерации и получения аналога для сравнения был использован статический анализатор [5]. Был произведен поиск количества неточностей с помощью обоих методов и затем ручной отсмотр выбранных функций с целью проверки качества работы модели.

# 2 Обзор литературы

## 2.1 Графовые представления программ

Программу можно представлять в графовом виде несколькими путями. Одним из основных является граф вызовов программы (call graph, function call graph), где вершинами являются исполняемые части программы, а ребра обозначают вызов одной функции из другой. Существуют различные методы построения этих графов [10].

Другим вариантом представления программы является AST-дерево (abstract syntax tree, абстрактное синтаксическое дерево), которое представляет собой оригинальный код программы, разобранный на древовидную структуру зависимостей. Получить такой вид представления легче всего, так как он не требует глубокого анализа программы или ее запуска.

Существует еще и Control Flow Graph (граф управления программой), описывающий все возможные пути исполнения программы, и Data Flow Graph, показывающие использование различных переменных в ходе работы программы.

Наконец, в работе [15] предлагается рассмотреть еще NCS (Natural Code Sequence), в котором соединяются вершины, отвечающие за соседние участки кода.

Преимуществом выбранной архитектуры нейросети является то, что она использует одновременно несколько графовых представлений программы, собирая их в мультиграф. В частности, в используемой нами реализации затрагиваются AST представления и CFG представления, а так же используется дополнительная информация, полученная напрямую из текстового представления программы. Подробнее про разные представления графов, описанные здесь, можно почитать в разделе про перевод в графовое представление статьи [15].

Для получения графовых представлений используются различные анализаторы, в данной работе, как и в [15] и [16], используется анализатор Joern [13]. Он выгоден сразу набором характеристик, в частности, удобен для использования на большом количестве программ, необходимом для создания датасета, а так же сравнительно простым API.

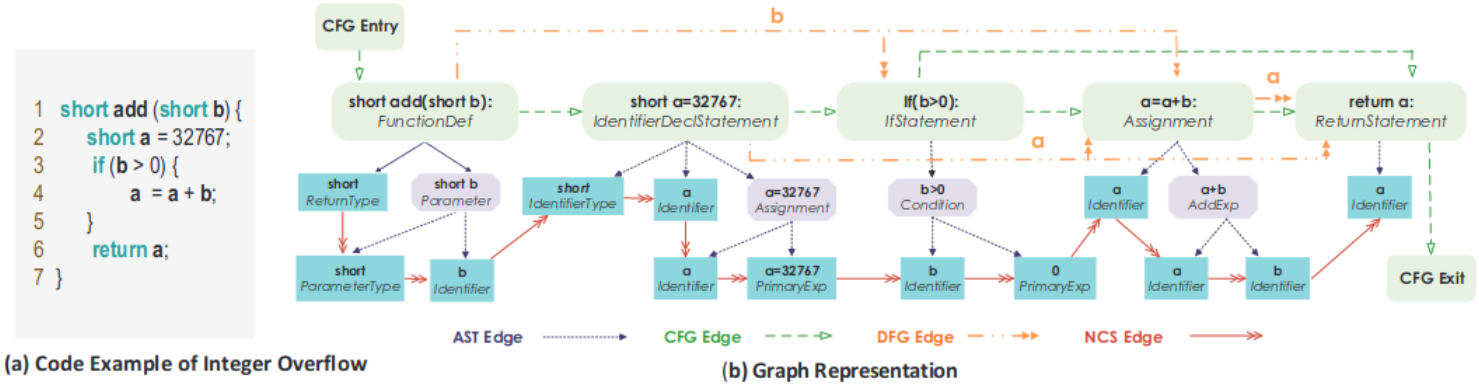


Рис. 2.1: Пример графового представления программы, рисунок из [15]

## 2.2 Датасеты для программного анализа

Благодаря росту количества работ, сфокусированных на программном анализе, стало появляться больше открытых наборов данных, нацеленных на обучение методов нахождения

уязвимостей. В основном такие наборы данных состоят из программ, представленных в одном из видов, целевой переменной - наличие или отсутствие уязвимости в каждой из программ, и дополнительных характерестик программ.

Например, одним из больших датасетов для поиска уязвимостей является MalNet [3], где программы содержатся в графовом представлении (графы вызовов). Такой датасет тяжело использовать для обучения выбранной архитектуры, так как в нем графовые представления уже даны и их сложно сопоставить с исходной программой, а выбранная модель подразумевает генерацию графового представления в своем, специфическом виде.

Датасет с исходными кодами программ был представлен и в статье с самой архитектурой, выбранной для данной работы [15]. В нем собраны программные коды из разных источников, например, ядра операционной системы Linux и других open-source проектов на языке C. Под программами в данном датасете подразумеваются коды, извлеченные из конкретных коммитов в эти проекты. Эти кусочки кода были оценены экспертами в компьютерной безопастности на наличие уязвимостей для получения целевой переменной.

К сожалению, в работе с этим набором данных обнаружились проблемы, которые не получилось решить, про это подробнее стоит смотреть в части про обучение модели.

Вместо этого был использован схожий набор данных [2], где тоже представлены наборы программ на C и C++. В сравнении с набором данных выше, из которого в открытом доступе есть только часть, это датасет целиком в открытом доступе и содержит больше примеров программ, в которых уязвимостей обнаружено не было.

## 2.3 Графовые нейросети и альтернативные способы анализа

Классическим методом анализа кода программы на уязвимости являются статические анализаторы, которые по коду программы проводят различные эвристические и алгоритмические оценки для нахождения уязвимостей и ошибок. В качестве инструмента, который будет применяться к новым данным вместе с нашей моделью, в данной работе был выбран статический анализатор Svace [5]. Он был выбран как проверенный временем анализатор с хорошим качеством.

Сравнительно недавным подходом к анализу кода на уязвимости является применение графовых нейросетей. Анализируя представление программы в том или ином графовом виде, они лучше усваивают структуру программы и позволяются добиться отличных результатов.

Примером state-of-the-art графовой нейросети, использованной, в частности, для анализа уязвимостей, является Exphormer [12], сочетающий в себе черты классических графовых нейросетей и трансформеров.

В данной работе вместо него был выбран Devign [15], так как, в отличие от Exphormer, он легче применим к данным, подающимся в виде исходного кода программы (в частично реализованной версии Devign уже были серьезные наработки по использованию анализатора Joern для получения графового представления). В то время как Exphormer и многие другие модели ожидают на вход уже граф, Devign (и статья, и реализация) изначально подразумевают создание графового представления собственного формата из нескольких других представлений, что дает лучшую структуру и упрощает реализацию в текущей работе.

## 2.4 Модели для кодогенерации

Сейчас большинство LLM масштаба ChatGPT-3 или ChatGPT-4 уже могут генерировать достаточно качественный код и часто используются разработчиками в реальной работе. Однако с ними возникают сложности в получении платного API, поэтому вместо них для данной работы была выбрана бесплатная и выложенная в open-source модель CodeLlama [9].

Ее преимуществом является как полная открытость архитектуры, так и специальная заточенность на генерацию кода (модель была специально дольше дообучена на supervised примерах кодогенерации). Кроме того, модель представлена в нескольких параметрах, что позволило использовать сжатую версию (7b параметров) для данной работы, так как вычислительные ресурсы были ограничены.

# 3 Обучение и доработка Devign

## 3.1 Краткий обзор архитектуры

Devign в классической версии, описываемой в статье, предполагает принятие на вход графов в сразу нескольких форматах (AST, CFG, DFG, а так же дополнительной информации в вершинах, полученной из текста программы). В версии, примененной в данной работе, используются только форматы AST и CFG, формат DFG (ориентированный на поток данных в программе и переход информации между переменными) реализован лишь частично, т.к. сложнее получаем с помощью Joern. Тем не менее авторы статьи в комментариях к репозиторию показывают эксперименты, где указано, что модель обучается и в таком режиме.

Модель получает AST и CFG представление функции (сгенерированное Joern). Важно отметить, что CFG граф имеет тот же набор вершин, что и AST (либо его подмножество), что используется моделью, которая объединяет оба эти графа в мультиграф. Кроме того, учитывается дополнительная информация напрямую из текстового представления кода (предварительно проходит через токенизатор и переводится в векторные представления). Каждой вершине сопоставляется кусочек кода, который после обработки становится вектором признаков этой вершины.

Дальше применяются Gated Graph Recurrent Layers [7], который переводят наше исходное графовое представление с ребрами и фичами (числами) каждой вершины в векторное пространство, кратное размерности $|V|$ (то есть количеству вершин в графе). Иными словами, убираем из графа ребра и заменяем представление каждой вершины на новое, таким образом преобразуя данные в набор векторов. В их работе применяется итеративное вычисление новых представлений, где между итерациями происходят переподсчеты по соседним ребрам и применяется Gated Reccurent Unit из рекуррентных сетей [1].

После на полученном наборе векторов применяются одномерные свертки чередуясь с линейными слоями, набор укороченных с помощью сверток векторов разворачивается в одномерный вектор и с помощью линейной головы переводится в логит предсказания, к которому в конце применяют сигмоиду для получения вероятности того, что код содержит уязвимости.

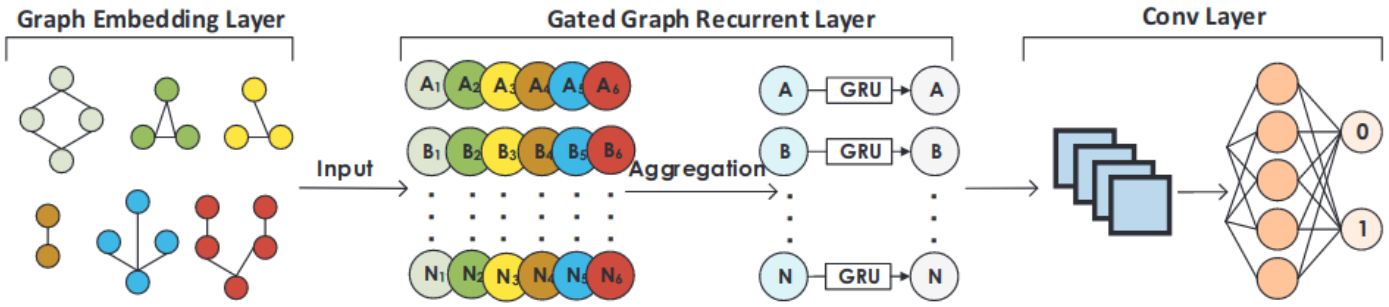


Рис. 3.1: Архитектура Devign, рисунок из [15]

## 3.2 Добавление нормализации в модель

В процессе обучения частичной реализации Devign от авторов (ссылка в приложении) была обнаружена проблема, связанная с плохим течением градиентов. С проблемой столкнулся не только я в свой работе, но и другие люди, что было видно по активному обсуждению в репозитории работы.

Проблема выражалась в отсутствии хоть какого-то прогресса в обучении модели, функция потерь (в данной реализации это бинарная кросс-энтропия с добавлением MAE между целевой переменной и нашей вероятностью, умноженной на маленькую, порядка $10^{-5}$ константу) застревала на отметке в 0.69 и никуда не двигалась (параметры обучения можно было менять, изменения были только при увеличении регуляризации, в данной работе это L2 регуляризация на веса модели, но это приводило просто к занулению весов модели и обучения все еще не было).

В итоге решением оказалось добавление одномерного слоя BatchNorm [4] (нормализации по батчу) перед финальной линейной головой. Применяем нормализацию по батчу прямо к распремленному в один вектор набору векторов после сверточных слове, под нормализацией подразумеваем следующую процедуру:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Здесь $\gamma, \beta$ - обучаемые параметры, E и Var собираем в процессе обучения и использования модели.

Использование такого слоя нормализации позволило преодолеть стагнацию функции потерь (предположительно улучшив протекание градиентов через сигмоиду) и стартовать обучение. В результате работы дальше планируется сделать pull-request в репозиторий [15], сейчас эта доработка ожидает комментариев от других участников дискуссии.

## 3.3 Обучение на другом датасете

После добавления слоя нормализации на датасете, предоставленном в работе [15] была обнаружена проблема серьезного переобучения. В работе были опубликованы в открытом доступе два датасета кода функций на C и C++ с отметкой о наличии или отсутствии уязвимости каждой функции, однако при попытках запустить обучение на каждом из наборов данных или их комбинаций модель не могла запомнить настоящие закономерности и либо функция потерь не уменьшалась (обучения не было), либо происходило полное переобучение, где функция потерь на валидационной выборке начинала расти с начала обучения, а функция потерь на обучающей выборке падала в ноль.

Использовались стандартные параметры обучения из [15]: оптимизатор Adam [6] с $LR = 10^{-4}$ и коэффициентом регуляризации $10^{-4}$, подробнее можно посмотреть в статье.

Снова оказалось, что с этой проблемой столкнулся не только я в своей работе, но и другие участники дискуссии в репозитории модели, где было предположено, что дело в самом наборе функций.

Поэтому были проанилизированы другие датасеты и выбран [2], включающий около 250000 функций, отмеченных как безопасные, и 12000, отмеченных как уязвимые. Это больше, чем в опубликованном в открытом доступе наборе данных из [15], где суммарно функций

было не более 100000. Источники функций схожие: в оригинальном датасете работы функции взяты из коммитов в open-source проекты, в новом наборе данных функции тоже взяты из коммитов в настоящие промышленные проекты.

Для балансировки классов были выбраны (случайно) 30000 функций без уязвимости и взяты все 11000 функций с уязвимостями.

Был написан код перевода этого датасета в формат, пригодный для обработки с помощью надстройки над Joern из репозитория Devign. Обработка этого датасета была одной из наиболее вычислительно затратных частей, т.к. пришлось обработать более 30000 функций, обработка происходила батчами по 50 штук на один запуск Joern (с большим количеством не справлялись вычислительные ресурсы).

На новом наборе данных получилось добиться отличных результатов, в частности получились результаты из Таблицы 3.1 и Графика 3.2, при этом обучение занимает около 30 минут на видеокарте NVIDIA A100. Здесь Accuracy получается больше Recall и Recall из-за дизбаланса классов, ведь негативных примеров примерно в 3 раза больше, чем положительных. Больше графиков можно найти в приложении.

Таблица 3.1: Результаты обучения

| Метрики | Train | Validation |
|---|---|---|
| Precision | - | 0.88 |
| Recall | - | 0.86 |
| Accuracy | 0.92 | 0.93 |

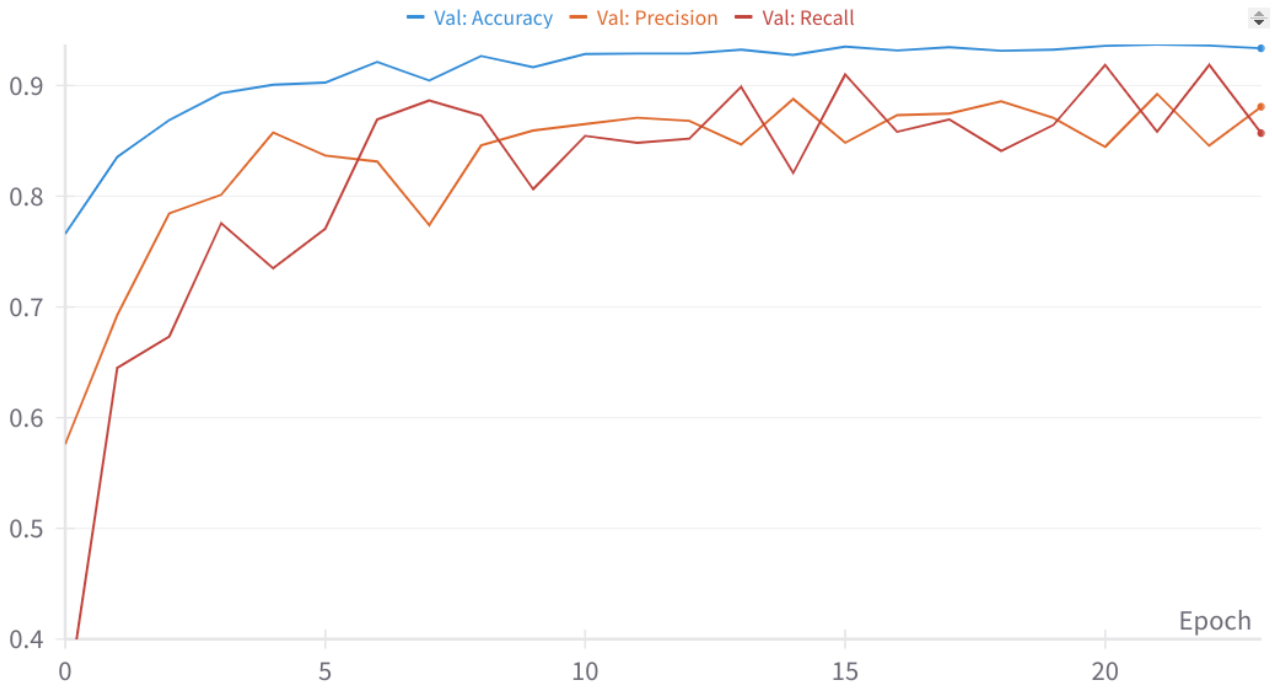


Рис. 3.2: Прогресс обучения на валидационной выборке

# 4 Создание датасета программ от LLM

## 4.1 Задачи с LeetCode

Для оценки уязвимостей в сгенерированном коде требовалось получить набор программ, сгенерированных LLM. Хотелось получить от набора максимальную адаптивность и не слишком большую сложность функций, так как оценка возможностей LLM в написании промышленного кода это пока отдельный вопрос, заслуживающий собственной работы.

Было принято решение генерировать код на основе алгоритмических задач, количество которых в открытом доступе крайне большое. За основу был взят набор задач с LeetCode, опубликованный на платформе HuggingFace.co (ссылка есть в приложении). В нем представлены задачи разной степени сложности.

Дополнительно с помощью найденного в открытом доступе интерфейса обращения к LeetCode из Python была добавлена возможность отправлять код в текстовом виде на сайт, чтобы посылка проверялось и можно было получить вердикт по коду. Рассматриваемые результаты посылки:

- Accepted - правильное решение
- Wrong Answer - решение выдает неправильный ответ на одном из тестов
- Runtime Error - решение падает с ошибкой во время исполнения на одном из тестов
- Compile Error - решение не компилируется

Кроме того, рассматриваемый датасет хорош тем, что задачи делятся на классы сложности Low, Medium и Hard, что позволяет регулировать сложность подаваемых в LLM запросов.

## 4.2 Кодогенерация

Для получения самих программ, сгенерированных LLM, была использована большая языковая модель CodeLlama [9]. Она была выбрана из-за своей открытой архитектуры и заточенности на генерацию кода по запросам на естественном языке (была специально дообучена в Supervised режиме на хороших примерах Запрос/Код), что и требуется для решения задач с LeetCode, подаваемых в текстовом виде.

Схема самой модели выходит за рамки данной работы, здесь она является исключительно представителем общего класса языковых LLM, способных писать код. Отдельным плюсом модели является то, что в открытом доступе есть предобученные версии разных размеров, в частности, для нашей работы использовалась версия с 7 миллиардами параметров, однако доступна опция переключения на 70 миллиардов при наличии требуемых для этого вычислительных ресурсов.

Для каждой задачи онлайн генерируется код с помощью модели (промпт можно посмотреть на Рисунке 4.1), дальше код подается в желаемый анализатор (в данной работе - в Svace или в Devign), и опционально отправляется на сайт LeetCode для проверки.

**Prompt:**

You are a helpful and honest code assistant expert in C. Please, provide all answers to programming questions in C. Write answer as a single function.

Given a string `s`, return *the longest palindromic substring* in `s`.

**Example 1:**

**Input:** s = "babad "

**Output:** "bab "

**Explanation:** "aba " is also a valid answer

**Example 2:**

**Input:** s = "cbbd "

**Output:** "bb "

**Constraints:** `1 <= s.length <= 1000` . `s` consist of only digits and English letters.

Рис. 4.1: Пример промпта для CodeLlama

В промпте можно просить сгенерировать код на различных языках, в том числе C, Java, Python и другие. В данной работе использовалась генерация на языках Java и C. Модель просим сгенерировать код в формате одной функции, так как именно такие файлы мы отдавали в Devign в процессе обучения, соответственно, для дальнейшей оценки качества хотим генерировать в таком же формате.

Для примера приведем статистику качества генерации кода (относительно статуса посылок на LeetCode) на языке Java. Видим, что большое количество посылок получает ошибку компиляции (т.е. модели тяжело создавать с первой попытки хороший код для задачи), при этом приблизительно каждая десятая генерация проходит все тесты успешно. Подробные результаты в Таблице 4.1

Таблица 4.1: Результаты генерации кода моделью на языке Java

| Статус посылки | Java |
|---|---|
| Accepted | 12% |
| Wrong Answer | 22% |
| Runtime Error | 3% |
| Compile Error | 62% |

# 5 Применение Devign к новым данным

## 5.1 Применение Svace

Для начала целью было проверить, есть ли в генерируемом коде в целом уязвимости (так как алгоритмические задачи требует специфичных решений, требовалось отдельно посмотреть на качество сгенерированного для них кода). Для этого использовали статический анализатор Svace, который позволяет находить в рабочем (компилируемом) коде ошибки и выдавать их в виде Warning (предупреждений).

Использование Svace с языком C, на котором обучали Devign, было затруднительно, так как он требовал создание как исполнительного, так и заголовочного файла для анализа. Если подавать такие требования в промпт модели, то падало качество генерируемого кода, так как модели тяжело было концентрироваться на большом количестве условий. Генерация же заголовочного файла по исполнительному руками, с помощью эвристик, затруднительна из-за большой вариативности кода (используются библиотеки, иногда нейросеть перед функцией генерирует структуры, которые использует дальше).

Отсюда было принято решение использование языка Java. Аналогично тестированию с помощью посылок на сайт LeetCode, генерировали по промпту код решения, сохраняли в файл, и дальше запускали анализ этого файла с помощью Svace. Важно отметить, что запускали анализ только на файлах, которые удалось скомпилировать (другие анализатор отказывается принимать и выдает просто ошибки компиляции).

Результаты для кода, сгенерированного на Java по задачам с LeetCode можно найти в Таблице 5.1. Здесь под Svace Error подразумевается наличие комментариев от Svace (учитываем любые комментарии как наличие уязвимости).

Таблица 5.1: Результаты запуска Svace

| Статус | Java |
|---|---|
| Svace Error | 12.5% |
| Compile Error | 62.3% |

Получаем, что даже для компилируемых решений достаточно существенный процент содержит недочеты. Встреченные недочеты включали возможность переполнения, возможный выход за границы массива, неосторожное обращение с указателями и другое. Отсюда имеет смысл запуск Devign на нашем датасете, так как там есть ошибки, которые будем находить.

## 5.2 Применение Devign

Для анализа с помощью Devign генерировали функции на языке C, так как сам Devign обучен на анализ кода на этом языке. Проблема в обучении его программам на языке Java (чтобы получить результаты, аналогичные результатам Svace) заключалась в настройке парсера Joern, с помощью которого создаем графовые представления программ. Его версия для анализа программ на Java появилась позже и имеет меньше функционала, что затрудняло использование репозитория с кодом Devign.

Узким местом так же являлось то, что при обучении Devign получал исключительно одиночные функции, а при генерации кода по промпту иногда модель (несмотря на явное указание создать одну функцию) добавляла предварительно одну структуру или диркетивы include для подключения библиотек. Анализ вручную графовых представлений, полученных из таких программ, показал, что это не является существенной проблемой и основная структура графа остается неизменна, то есть обученный Devign может обрабатывать и такие функции.

Результаты запуска Devign для анализа функций, сгенерированных с помощью CodeLlama по задачам LeetCode, можно посмотреть в Таблице 5.2. Здесь считаем Vulnerable функции, которые имеют итоговую вероятность по модели больше 0.5.

Таблица 5.2: Результаты анализа Devign

| Статус | C (percent) | C (absolute count) |
|---|---|---|
| Devign: Vulnerable | 16% | 56 |
| Devign: Safe | 84% | 288 |

Видим, что процент найденных уязвимостей близок к проценту, полученному Svace, хотя прямые выводы делать сложно из-за разных языков программирования и разных условий. Здесь, в отличие от Svace, мы передаем все функции, в том числе те, которые может быть Svace отклонил бы как ошибки компиляции. Про дополнительные идеи для сравнения будет в дальнейшей работе.

Ручной анализ результатов показал, что модель часто обращала внимание на код с указателями и указывала на уязвимость функций, принимающих указатель на объект и не проверяющих его валидность. Добавление проверки на нулевой указатель, по-видимому, уменьшает вероятность уязвимости по мнению модели, что соответствует действительности и может помочь пользователям.

Кроме того, модель достаточно часто повышала вероятность уязвимости при обращении к массиву по аргументу, передаваемому в функцию, так как это может вести к выходу за границы массива.

В целом, уязвимости, найденные моделью, обычно легко объяснить при ручном осмотре функции. Посмотрев на результаты посылок кода на Java можно предположить, что модель не видит многие ошибки кода и обращает внимание именно на уязвимости (потому что ожидаемый процент Compilation Error кратно больше 12.5%), так как в обучающем наборе были как раз уязвимости, а не ошибки компиляции (обучающий набор собирали по реальным коммитам, где большинство ошибок компиляции были поправлены еще до создания коммита самими разработчиками).

# 6 Заключение

## 6.1 Выводы

Использование LLM для генерации кода все больше и больше будет входить в рутину каждого программиста. Поэтому критически важно, чтобы этот код был надежный и содержал как можно меньше уязвимостей, ставящих под угрозу проекты. Это тем более важно, что при использовании LLM для генерации программист пишет код не сам, а значит имеет меньше возможностей обнаружить ошибку.

Анализ с помощью Svace и отправка решений на LeetCode показывает, что код, создаваемый LLM, далек от совершенства. Кроме ошибок компиляции, которые легко выявить при запуске программы, он может и будет содержать скрытые уязвимости, которые могут проявиться при использовании кода позже. Инструменты, которые позволяли бы автоматизировать проверку качества созданного кода, помогли бы программистам сохранить преимущества использования LLM, такие как сокращение времени разработки, при этом защитив разработчиков от уязвимостей, вносимых LLM.

Графовые нейросети, хорошо улавливающие структуру программного кода, позволяют анализировать код на наличие уязвимостей. Развитие области в последние годы дало большое количество моделей и датасетов, пригодных для этой задачи. Пример использования Devign на датасете уязвимостей [2] показывает, что модели хорошо работают не только на тех данных, которые были использованы в их оригинальной статье, но и показывают обобщающую способность на других данных.

Достигнутые в данной работе на датасете [2] цифры (93% Accuracy на валидационной выборке) показывают пригодность Devign для анализа коммитов обычного кода разработчиков, что уже мотивирует пробовать встраивать графовые нейросети и Devign в частности в стандартные процессы тестирования.

Однако сами по себе графовые нейросети для анализа уязвимостей все еще находятся на достаточно ранней стадии. Перед использованием на новом датасете потребовалось немало инженерных усилий для успешного запуска процесса обучения. Код, написанный для данной работы, может упростить подобные переиспользования для разработчиков в дальнейшем.

Получены результаты применения графовых нейросетей к сгенерированному коду, способность находить ошибки сохраняется, однако перед твердыми утверждениями в их работоспособности для этой задачи стоит провести дополнительные исследования. Различия в результатах работы Svace и Devign показывают, что более детальное сравнение статических анализаторов и графовых нейросетей в анализе кода на уязвимости может позволить создать новые инструменты, которые показывают результаты надежнее каждой отдельной архитектуры.

Вспомогательные инструменты, написанные в данной работе, могут помочь дальнейшим исследованиям в области анализа качества работы графовых нейросетей на генерируемом коде, и представить новый бенчмарк для анализа уязвимостей.

## 6.2 Идеи для дальнейшей работы

В ходе работы над проектом получилось найти сразу несколько идей для дальнейшего развития.

Первостепенно интересной работой было бы более тщательное и строгое сравниние результатов Devign и Svace. Необходимо доделать инженерные инструменты, возможно, найти другой парсер программ в графовое представление или настроить Joern на работу с Java, чтобы получить возможность сравнивать их на одном языке программирования. Это позволило бы сделать более твердые утверждения об отличиях этих подходов и сравнить, какие уязвимости какие подходы могут находить лучше. Полученные данные возможно получится использовать для дообучения GNN. Однако на данный момент, как сказано в [15], использование результатов статических анализаторов для обучения GNN скорее приводит к ухудшению их качества из-за большого количетсва ложноположительных результатов статических анализаторов.

Хорошей идеей кажется и возможность дообучения LLM с помощью фидбека от графовых нейросетей, так как они показывают отличную Accuracy на знакомых им датасетах. Возможно, отдавая сгенерированный код в Devign или схожую модель для получения фидбека, и используя Reinforcement Learning для дообучения LLM [11], [8], можно получить версии языковых моделей, более внимательные к уязвимостям в генерируемом ими коде. Однако есть опасения, что такое фидбек либо будет недостаточно качественным, либо будет приводить к потере обобщающей способности у LLM.

Наконец, кажется интересной возможность добавления в диалог с моделью фидбека по ее коду. Особенно это интересно в связи с наличием графовых нейросетей, которые могут детально находить строки кода, похожие на уязвимости [14]. При подозрении в уязвимости можно просить LLM перегенироровать код, указав строки, где GNN подозревает наличие уязвимости, и получить более надежный код.

Таким образом, использование GNN как само по себе, так и в особенности в кодогенерации может помочь быстро находить часть уязвимостей и помогать разработчикам находить ошибки сразу, концентрируя их внимания на тех местах, где уязвимости подозреваются особенно сильно.

# 7 Приложение

- Датасет с задачами LeetCode:
  https://huggingface.co/datasets/greengerong/leetcode
- Репозиторий с реализаций Devign от авторов оригинальной статьи:
  https://github.com/epicosy/devign
- Репозиторий с кодом, написанным для данной работы:
  https://github.com/epbugaev/gnn_in_program_analysis
- График роста Accuracy на обучающей выборке для Devign с нормализацией на датасете [2]:

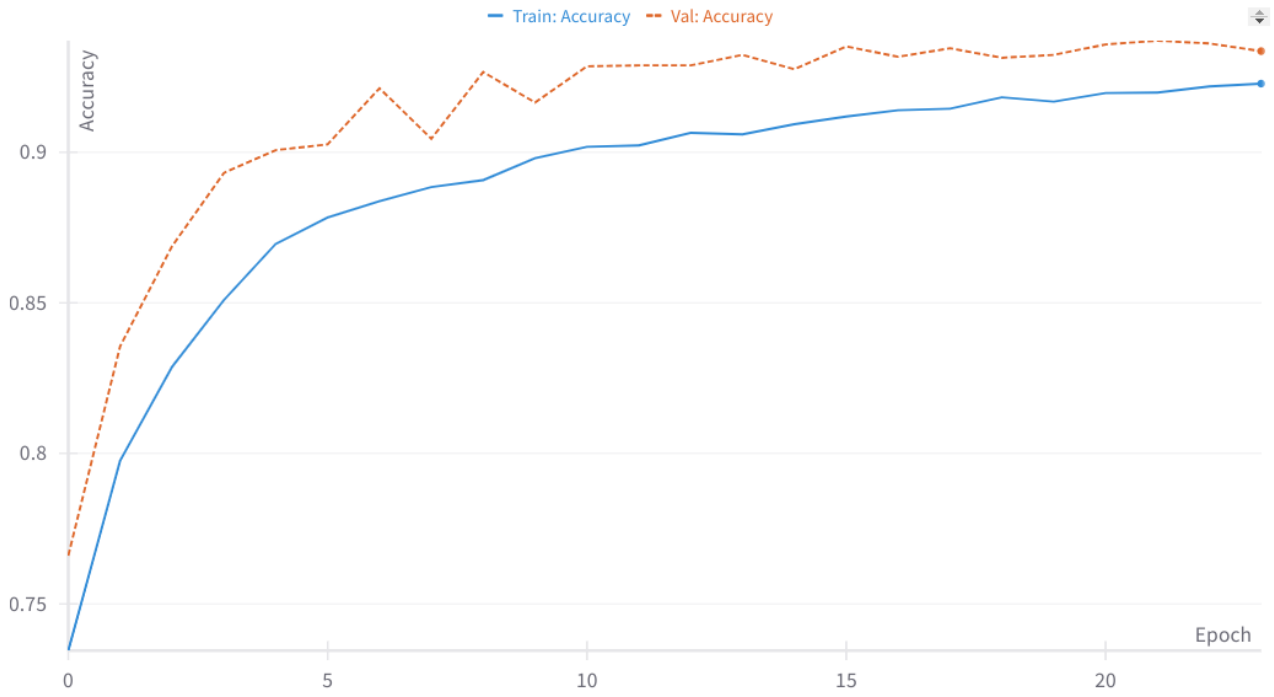


Рис. 7.1: Прогресс обучения на обучающей и валидационной выборке

# Список литературы


[1] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk и Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[2] Jiahao Fan, Yi Li, Shaohua Wang и Tien N. Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". В: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, с. 508—512. ISBN: 9781450375177. DOI: 10.1145/3379597.3387501. URL: https://doi.org/10.1145/3379597.3387501.

[3] Scott Freitas, Rahul Duggal и Duen Horng Chau. *MalNet: A Large-Scale Image Database of Malicious Software*. 2022. arXiv: 2102.01072 [cs.CR].

[4] Sergey Ioffe и Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[5] V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin и A. I. Avetisyan. "Static analyzer Svace for finding defects in a source program code". В: *Programming and Computer Software* 40.5 (сент. 2014), с. 265—275. ISSN: 1608-3261. DOI: 10.1134/s0361768814050041.

[6] Diederik P. Kingma и Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[7] Yujia Li, Daniel Tarlow, Marc Brockschmidt и Richard Zemel. *Gated Graph Sequence Neural Networks*. 2017. arXiv: 1511.05493 [cs.LG].

[8] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning и Chelsea Finn. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2023. arXiv: 2305.18290 [cs.LG].

[9] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom и Gabriel Synnaeve. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL].

[10] B.G. Ryder. "Constructing the Call Graph of a Program". В: *IEEE Transactions on Software Engineering* SE-5.3 (1979), с. 216—226. DOI: 10.1109/TSE.1979.234183.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford и Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[12] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J Sutherland и Ali Kemal Sinop. "Exphormer: Sparse transformers for graphs". В: *International Conference on Machine Learning*. 2023. arXiv: 2303.06147.

[13] Fabian Yamaguchi, Nico Golde, Daniel Arp и Konrad Rieck. "Modeling and discovering vulnerabilities with code property graphs". В: *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, май 2014.

[14] Nan Yin, Li Shen, Mengzhu Wang, Long Lan, Zeyu Ma, Chong Chen, Xian-Sheng Hua и Xiao Luo. *CoCo: A Coupled Contrastive Framework for Unsupervised Domain Adaptive Graph Classification*. 2023. arXiv: 2306.04979 [cs.LG].

[15] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du и Yang Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. 2019. arXiv: 1909.03496 [cs.SE].

[16] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li и Hai Jin. "VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection". B: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2021), с. 2224—2236. DOI: 10.1109/TDSC.2019.2942930.