МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет вычислительной техники

Кафедра «Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

РАСПОЗНАВАНИЕ ЗАШИФРОВАННЫХ СООБЩЕНИЙ С ПОМОЩЬЮ ГЛУБОКОГО ОБУЧЕНИЯ

Студент 08.06.2026 (подпись, дата) Шмелев Данила Валерьевич (ФИО полностью)

Руководитель 08.06.2026 (подпись, дата) М.А. Митрохин (фамилия, инициалы)

Нормоконтролёр 08.06.2026 (подпись, дата) А.С. Бычков (фамилия, инициалы)

Рецензент нач. УИТ (должность, место работы) (подпись, дата) Д.В. Севастьянов (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.2026 № 14)

Заведующий кафедрой (подпись) М.А. Митрохин (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 23.06.2026 № 24)

Секретарь ГЭК (подпись) Е.И. Гурин (фамилия, инициалы)

Пенза 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	Кафедра
вычислительной техники	«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____ М.А. Митрохин
«20» 09 2024 г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

1. Студент Шмелев Данила Валерьевич гр. 24ВВИм1 факультета ВТ

2.Тема работы Распознавание зашифрованных сообщений с помощью глубокого обучения
Тема утверждена приказом ПГУ № 920/о от "20" 09 2024 г.

3. Руководитель магистерской диссертации д.т.н., профессор Митрохин М.А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)
Провести исследование способности моделей глубокого обучения различать артефакты алгоритмов шифрования от случайного шума. Разработать программный комплекс для обнаружения зашифрованных данных с использованием технологий глубокого обучения и больших языковых моделей. Создать программную реализацию конвейера данных, который позволяет генерировать обучающие и тестовые выборки на основе выбранных алгоритмов шифрования и псевдослучайного шума. Спроектировать и обучить специализированную нейросетевую модель на базе архитектуры трансформера для бинарной посимвольной классификации последовательностей. Разработать модуль автоматизированного тестирования LLM с применением структурированных prompt-инструкций. Провести сравнительный анализ точности распознавания и вычислительной ресурсоемкости универсальных генеративных сетей и обученного классификатора.
5. Перечень подлежащих разработке вопросов
Требуется выполнить анализ предметной области в сфере обнаружения зашифрованных данных. Изучить математические свойства симметричных алгоритмов шифрования и ограничения классических статистических методов выявления скрытой структуры. В рамках исследования, определить возможности использования современных больших языковых моделей для бинарной классификации. Провести разработку унифицированной методологии тестирования LLM.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение предметной области	10%	20.09.24-15.11.2024	
2.	Формирование требований к наборам данных и методологии тестирования.	10%	16.11.2024–20.12.2024	
3.	Изучение научной литературы по нейросетевому криптоанализу и применению больших языковых моделей.	10%	21.12.2024–31.01.2025	
4.	Проектирование архитектуры программного конвейера генерации данных	15%	01.02.2025–15.03.2025	
5.	Разработка конвейера данных, реализация и обучение модели-классификатора	30%	16.03.2025–31.01.2026	
6.	Проведение вычислительных экспериментов: тестирование LLM и валидация реализованной модели.	10%	01.02.2026–28.02.2026	
7.	Сравнительный анализ метрик точности классификации, оценка структурных ошибок и вычислительной ресурсоемкости.	10%	01.03.2026–30.04.2026	
8.	Оформление пояснительной записки	5%	01.05.2026–20.05.2026	

Дата выдачи задания "20" 09 2024 г.

Руководитель магистерской диссертации _____ Митрохин М.А.
(подпись, дата)

Задание к исполнению принял студент _____ Шмелев Д.В.
(подпись, дата)

Работу к защите допустить
Декан факультета _____ 10.06.2026 Фионова Л.Р.

РЕФЕРАТ

Пояснительная записка содержит здесь до приложения 66 листов, 17 рисунков, 11 формул, 6 таблиц, 30 источников, 2 приложения.

КРИПТОАНАЛИЗ, ГЛУБОКОЕ ОБУЧЕНИЕ, БОЛЬШАЯ ЯЗЫКОВАЯ МОДЕЛЬ, LLM, ТРАНСФОРМЕР, БИНАРНАЯ КЛАССИФИКАЦИЯ, АЛГОРИТМЫ ШИФРОВАНИЯ

Целью работы является исследование способности моделей глубокого обучения распознавать зашифрованные сообщения. Для достижения поставленной цели необходимо разработать методику проведения экспериментов, сформировать обучающие и валидационные выборки данных, а также провести сравнительный анализ результатов.

В ходе выполнения магистерской работы на базе фреймворка PyTorch реализована программная реализация модели глубокого обучения. Созданы алгоритмы для генерации и обработки наборов данных, а также написаны скрипты, обеспечивающие процесс обучения и исполнения модели классификатора. Дополнительно разработаны инструменты для интеграции и тестирования больших языковых моделей.

В результате проведенного исследования обучена нейросетевая архитектура на базе энкодера трансформера. Успешно реализован полноценный конвейер подготовки данных, а также внедрен модуль автоматизированного тестирования LLM для оценки их аналитических возможностей.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 4

ВВЕДЕНИЕ..... 6

1 Обзор предметной области..... 9

1.1 Выбор алгоритмов шифрования 12

1.2 Применение нейронных сетей в криптоанализе..... 18

1.3 Выбор LLM..... 25

2 Методология исследования 28

2.1 Формирование наборов данных 28

2.2 Архитектура модели классификатора 34

2.3 Методика тестирования больших языковых моделей 40

3 Разработка и проведение экспериментов..... 44

3.1 Обучение модели..... 44

3.2 Оценка возможностей больших языковых моделей 54

3.3 Сравнительный анализ моделей 57

3.4 Оценка вычислительной эффективности и ресурсоемкости 59

3.5 Анализ результатов 61

ЗАКЛЮЧЕНИЕ 62

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 64

ПРИЛОЖЕНИЕ А – Листинг программы 67

ПРИЛОЖЕНИЕ Б – Презентация..... 99

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяются следующие сокращения и обозначения.

ИИ – Искусственный интеллект

AES (Advanced Encryption Standard) – Блочный алгоритм шифрования

CNN (Convolutional Neural Network) – Сверточная нейронная сеть

CPU (Central Processing Unit) – Центральный процессор

DPI (Deep Packet Inspection) – Технология глубокого анализа сетевых пакетов

FFN (FeedForward Network) – Полносвязная нейронная сеть прямого распространения

GELU (Gaussian Error Linear Unit) – Функция активации на основе гауссовской ошибки

GPU (Graphics Processing Unit) – Графический процессор

IC (Index of Coincidence) – Индекс совпадений

IDS (Intrusion Detection System) – Система обнаружения сетевых вторжений

IoT (Internet of Things) – Интернет вещей

IPS (Intrusion Prevention System) – Система предотвращения сетевых вторжений

IV (Initialization Vector) – Вектор инициализации

JSON (JavaScript Object Notation) – Словарь данных

KV-cache (Key-Value cache) – Кэш состояний ключей и значений в архитектуре генеративных моделей

LLM (Large Language Model) – Большая языковая модель

LSTM (Long Short-Term Memory) – Рекуррентная нейронная сеть долгой краткосрочной памяти

MHA (Multi-Head Attention) – Механизм многоголового самовнимания

MLP (Multilayer Perceptron) – Многослойный перцептрон

QKV (Query, Key, Value) – Механизм запросов, ключей и значений в архитектуре внимания

ReLU (Rectified Linear Unit) – Линейная функция активации

RNN (Recurrent Neural Network) – Рекуррентная нейронная сеть

SLM (Small Language Model) – Малая языковая модель

SPN (Substitution-Permutation Network) – Подстановочно-перестановочная сеть

SVM (Support Vector Machine) – Метод опорных векторов

VRAM (Video Random Access Memory) – Видеопамять графического процессора

YAML (YAML Ain't Markup Language) – Формат сериализации данных для конфигурационных файлов

ВВЕДЕНИЕ

Традиционные методы выявления зашифрованных данных опираются на статистические критерии: оценку энтропии, индекс совпадений или частотный анализ. Эти подходы работают для простых перестановочных шифров, но теряют эффективность при анализе текстов, обработанных более сложными алгоритмами шифрования. При использовании современных подходов зашифрованная последовательность символов становится практически неотличима от белого шума [4].

Развитие глубокого обучения и переход к механизмам внимания и моделям класса трансформер открыли дорогу для абсолютно новых инструментов криптоанализа. Нейросети способны находить скрытые нелинейные зависимости там, где стандартные статистические тесты не фиксируют никаких аномалий [5, 6]. В последние годы криптоанализ, основанный на моделях глубокого обучения, выделился в самостоятельное направление. Уже сегодня такие модели способны успешно классифицировать блочные шифры при условии целенаправленного обучения модели на соответствующих наборах данных [2].

Параллельно с этим крупные языковые модели демонстрируют поразительные результаты в задачах различного спектра применения, которые выходят за рамки простой генерации естественного языка. Однако их применение в криптографии изучено значительно меньше. Научные исследования показывают, что большие языковые модели пока способны лишь запоминать зашифрованные фрагменты на этапе предварительного обучения [1]. Открытым остается вопрос о том, могут ли современные LLM в режиме zero-shot, то есть без дообучения, отличать зашифрованный текст от случайного шума, и как их результативность соотносится с метриками более компактных нейронных сетей, которые были спроектированы исключительно под эту задачу. Сравнение универсальных тяжеловесных моделей с легковесными специализированными архитектурами определяет вектор и общую актуальность данного исследования.

Исходя из заявленной проблематики, объектом исследования выступают тексты на естественном языке, преобразованные алгоритмами шифрования различной сложности, а также псевдослучайные последовательности символов. Предметом исследования являются методы бинарной классификации таких последовательностей на базе архитектуры трансформер.

Главная цель диссертационной работы заключается в разработке методики и проведении сравнительной оценки эффективности распознавания зашифрованных сообщений с использованием больших языковых моделей и специализированной модели глубокого обучения.

Для достижения этой цели потребовалось решить комплекс взаимосвязанных задач. Во-первых, проанализировать существующие подходы к распознаванию зашифрованных данных и оценить применимость архитектуры трансформер в этом контексте. Во-вторых, сформировать несколько наборов данных под каждый выбранный алгоритм шифрования. В качестве исходной выборки будет выступать открытый словарь английских слов, сформированный MIT [3]. Каждый датасет должен содержать информацию об исходном слове, его зашифрованной версии и случайно сгенерированной последовательности символов. В-третьих, необходимо выбрать модели LLM, разработать методику тестирования больших языковых моделей, подобрать гиперпараметры тестирования и сформировать оптимальные prompt-инструкции для задачи бинарной классификации. Следующим шагом выступает проектирование, реализация и обучение собственной узкоспециализированной нейросетевой модели на базе архитектуры модели трансформер. После всего вышеперечисленного необходимо привести результаты экспериментов для сопоставления метрик качества и вычислительной сложности инференса исследуемых моделей.

Методологической базой исследования послужили фундаментальные принципы криптографии и методы машинного и глубокого обучения.

Практическая реализация собственного классификатора, его обучение и валидация выполнены на языке Python с использованием фреймворка PyTorch.

Научная новизна полученных результатов состоит в том, что в работе впервые проведено прямое эмпирическое сравнение языковых моделей малого и среднего размера со специализированным легковесным трансформером в задаче детекции симметричных шифров. В ходе работы определены эмпирические границы применимости LLM в криптоанализе: доказано, что модели способны выявлять паттерны шифрах подстановки, но при анализе более сложных блочных шифров их точность падает до уровня случайного угадывания. Кроме того, предложена архитектура компактного классификатора, которая обеспечивает высокую точность распознавания при минимальных затратах вычислительных ресурсов.

В практическом смысле эти результаты применимы в интеллектуальных системах сетевой безопасности (IDS/IPS) для первичной фильтрации трафика. Разработанный алгоритм способен отсеивать случайный шум от зашифрованных данных до их передачи на этап более сложного криптоанализа. Отказ от LLM в пользу предложенной архитектуры существенно снижает требования к аппаратному обеспечению систем безопасности при их локальном развертывании.

Сформулированные теоретические выводы и практические результаты формируют основные положения, выносимые на защиту. Первое положение закрепляет методику формирования наборов данных и извлечения признаков, которая обеспечивает корректное сравнение эффективности универсальных LLM и узкоспециализированных моделей в задаче бинарной классификации. Второе положение содержит вывод о наличии у современных LLM скрытой способности к криптоанализу простых шифров подстановки через базовые prompt-запросы. Третье положение утверждает, что разработанная легковесная нейросетевая архитектура превосходит языковые модели по соотношению точности классификации к вычислительным затратам.

1 Обзор предметной области

Развитие криптоанализа исторически основывалось только лишь на использовании статистических особенностей естественных языков. Любой незашифрованный текст характеризуется определённой избыточностью и набором закономерностей: неодинаковой частотой появления отдельных символов, довольно частыми сочетаниями отдельных букв, а также характерной структурой слов и предложений. Поэтому одной из ключевых задач при анализе перехваченного сообщения является определение того, содержит ли исследуемая последовательность скрытую структуру и связана ли она с каким-либо осмысленным текстом.

Одним из первых методов решения данной задачи стал частотный анализ. Его принцип основан на том, что в ряде классических шифров каждый символ исходного текста заменяется другим символом по фиксированному правилу. В результате частотное распределение символов в зашифрованном тексте сохраняет основные особенности распределения символов исходного языка, хотя сами символы оказываются смещены относительно исходного алфавита [7]. Однако этот метод быстро теряет эффективность при использовании шифров, в которых более одного алфавита символов или более современных стандартов в криптографии. Для формальной оценки «случайности» текста применяется математический аппарат теории информации, в первую очередь – это концепция энтропии.

Понятие информационной энтропии введено Клодом Шенноном в 1948 году как математическая мера неопределенности или количества информации, приходящейся на один символ входной последовательности [8]. Для дискретной случайной величины X, принимающей значения $x_1, x_2, \dots, x_n$ с вероятностями $p(x_i)$, энтропия Шеннона вычисляется по формуле:

$$H(X) = -\sum_{i=0}^{n} p(x_i)\, log_2\big(p(x_i)\big), \tag{1.1}$$

В контексте распознавания данных энтропия позволяет количественно оценить степень хаотичности исследуемого текста. Если рассматривать базовый латинский алфавит, состоящий из 26 символов, то теоретическая максимальная энтропия, то есть состояние полной неопределенности, где каждый символ встречается с одинаковой вероятностью, составляет $log_2(26) \approx 4.7$ бит на символ. При этом реальная энтропия осмысленного текста на английском языке находится в пределах от 1.0 до 1.5 бит на символ из-за высокой лингвистической избыточности и грамматических правил [9].

Если на вход поступила неизвестная ранее последовательность, расчет энтропии становится первым фильтром для дальнейшего анализа. Если вычисленное значение $H(X)$ близко к 4.7 битам, то данные либо сильно сжаты, либо обработаны алгоритмом с высокой криптографической стойкостью. В противном случае, это значит, что входная последовательность символов содержит скрытые структурные зависимости. Чаще всего такие зависимости характерны для простых схем шифрования, которые основаны на подстановках или перестановках [10].

Для более глубокого анализа полиалфавитных шифров используется индекс совпадений. До появления этого подхода, криптоаналитики, в основном, применяли метод Касиски предложенный в 1863 году. Его суть заключалась в поиске повторяющихся фрагментов текста, которые позволяли бы оценить длину используемого ключа [11]. Но эффективность этого метода существенно зависела от объёма сообщения и наличия в нём выраженных повторяющихся символов. В 1920 году Уильям Фридман предложил новый статистический подход, который позволяет проводить анализ зашифрованных сообщения без надобности поиска конкретных слов или фраз.

Индекс совпадений характеризует вероятность того, что два символа, случайно выбранные из текста, окажутся одинаковыми. Для текста длины N, в котором каждый символ i (из алфавита размерностью c) встречается f_i раз, индекс рассчитывается следующим выражением:

$$IC = \frac{\sum_{i=0}^{c} f_i(f_i - 1)}{N(N-1)}, \tag{1.2}$$

Одним из ключевых свойств индекса является его устойчивость к моноалфавитным заменам. Для обычного английского текста значение данного показателя составляет приблизительно 0.0667. Для случайной же последовательности символов английского алфавита оно близко к величине $1/26 \approx 0.0385$ [12]. Благодаря этому индекс совпадений позволяет не только выявлять признаки полиалфавитного шифрования, но и оценить параметры используемого ключа.

Частотный анализ, оценка энтропии и индекс совпадений составляют основу классических методов обнаружения зашифрованных данных. Их преимуществами являются вычислительная простота и низкие требования к вычислительным ресурсам. Но эффективность данных подходов напрямую зависит от наличия статистических закономерностей в анализируемой последовательности символов.

Для современных алгоритмов шифрования, указанные методы оказываются значительно менее результативными. Это связано с тем, что новые подходы используют последовательность нелинейных преобразований и операций перестановки, обеспечивающих выраженный лавинный эффект. Даже незначительное изменение входных данных приводит к существенному изменению зашифрованного текста. В результате статистические характеристики выходной последовательности становятся близкими к характеристикам случайного шума: распределение символов стремится к равномерному, индекс совпадений приближается к значениям случайной

последовательности, а энтропия достигает величин, близких к своему теоретическому максимуму. Такие алгоритмы шифрования классические статистические методы идентифицируют зашифрованный текст как истинно случайную последовательность. Именно поэтому для поиска скрытых нелинейных зависимостей необходимо перейти к алгоритмам машинного обучения и нейронным сетям.

1.1 Выбор алгоритмов шифрования

Для того чтобы объективно оценить то, насколько хорошо модели глубокого обучения справляются с нахождением признаков в зашифрованных данных, необходимо, в первую очередь, определиться с выбором алгоритмов шифрования. В рамках данного исследования выбраны три симметричных алгоритма шифрования: шифр Цезаря, шифр Виженера и стандарт AES. Выбор именно симметричных алгоритмов шифрования обусловлен тем, что асимметричные алгоритмы, требуют передачи сразу нескольких ключей, чтобы расшифровать сообщение. Это сильно усложняет задачу бинарной классификации и поэтому было принято решение отказаться от асимметричных шифров.

Начальным уровнем сложности выступает алгоритм Цезаря, который является одним из простейших алгоритмов моноалфавитной подстановки. Его принцип заключается в сдвиге каждого символа исходного сообщения на фиксированное число позиций вправо по алфавиту [13]. процесс шифрования для символа с индексом m и ключом сдвига k в алфавите из n символов описывается формулой:

$$c = (m + k)\ mod\ n, \tag{1.3}$$

Процедура дешифрования выполняется обратным сдвигом:

$$c = (m - k)\ mod\ n, \tag{1.4}$$

Несмотря на простоту реализации, шифр Цезаря представляет интерес в рамках данного исследования как базовый пример моноалфавитного шифрования. В нём каждому символу открытого текста всегда соответствует один и тот же символ шифротекста вследствие чего сохраняются основные статистические характеристики исходного языка, включая частотное распределение букв [14]. Благодаря этому зашифрованный текст продолжает сохранять структурные закономерности, которые потенциально могут быть выявлены как классическими методами криптоанализа, так и современными моделями машинного и глубокого обучения.

Использование шифра Цезаря позволяет оценить способность языковых моделей обнаруживать простейшие преобразования текста, при которых большая часть статистической информации сохраняется. Такой подход служит отправной точкой для дальнейшего анализа более сложных алгоритмов шифрования.

Следующим уровнем сложности в работе рассматривается шифр Виженера. Исторически он был разработан как развитие идей моноалфавитного шифрования и предназначался для устранения основных недостатков простых подстановочных шифров, прежде всего их уязвимости к частотному анализу. Это полиалфавитный шифр, представляющий собой последовательность нескольких шифров Цезаря с различными значениями сдвига. Сдвиг для каждой конкретной буквы определяется соответствующим символом ключевого слова [15]. Если длина ключа меньше длины сообщения, ключ циклически повторяется.

В математическом виде шифрование i-го символа текста (M_i) с использованием i-го символа ключа (K_i) выглядит так:

$$C_i = (M_i + K_i)\ mod\ n, \tag{1.5}$$

Шифр Виженера «размазывает» статистику текста. Одна и та же буква в исходном сообщении может быть зашифрована разными символами в зависимости от ее позиции и текущей буквы ключа [16]. Для нейросетевых моделей задача усложняется: необходимо выявить не статический сдвиг, а паттерн циклического применения ключа. Модель должна обладать достаточным окном внимания, чтобы связать символы, зашифрованные одной и той же буквой ключа на расстоянии, равном его длине.

Третьим и наиболее сложным алгоритмом в контексте эксперимента выступает AES. Он был принят Национальным институтом стандартов и технологий США (NIST) в 2001 году и сегодня является мировым стандартом симметричного блочного шифрования [17]. В отличие от потоковых шифров и исторических алгоритмов подстановки, AES оперирует не отдельными символами, а блоками данных фиксированного размера.

Процесс шифрования в AES строится на архитектуре подстановочно-перестановочной сети и состоит из нескольких раундов [18]. Каждый раунд алгоритма состоит из четырёх последовательно выполняемых преобразований: SubBytes, ShiftRows, MixColumns и AddRoundKey.

Первая стадия раунда является единственным нелинейным преобразованием в алгоритме, которое обеспечивает свойство «путаницы» по Шеннону. Как показано на рисунке 1.1, трансформация применяется независимо к каждому байту состояния $a_{i,j}$, заменяя его на новый байт $b_{i,j}$, с помощью фиксированной таблицы подстановок (S-box или S_{RD}).

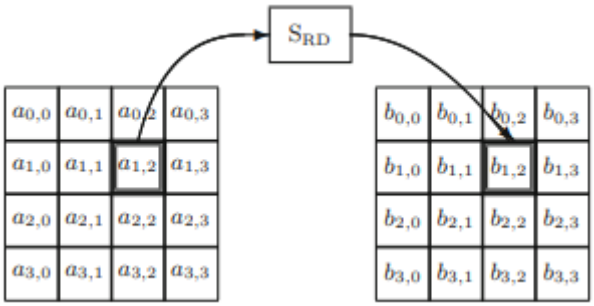


Рисунок 1.1 – Независимая замена каждого байта состояния

S-box вычисляется в два этапа: сначала находится мультипликативная инверсия байта в конечном поле Галуа $GF(2^8)$, а затем к результату применяется аффинное преобразование над полем $GF(2)$. Это делает шифр устойчивым к методам линейного и дифференциального криптоанализа, исключая возможность выявления алгебраических закономерностей.

Стадия ShiftRows отвечает за линейную диффузию на уровне строк. Как проиллюстрировано на рисунке 1.2, байты в каждой строке матрицы состояния циклически сдвигаются влево на различное количество позиций.

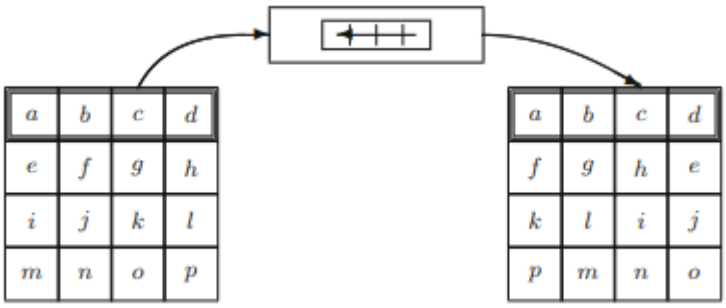


Рисунок 1.2 – циклический сдвиг строк матрицы

Благодаря этому преобразованию байты, находившиеся в одной колонке, распределяются по разным колонкам, что гарантирует распространение влияния одного бита на весь блок данных в последующих раундах.

Следующая трансформация MixColumns обеспечивает диффузию на уровне столбцов матрицы состояния. В этой операции каждый столбец, состоящий из 4 байт, рассматривается как многочлен над полем $GF(2^8)$, и умножается по модулю $x^4 + 1$ на фиксированный многочлен $c(x)$ (рисунок 1.3).

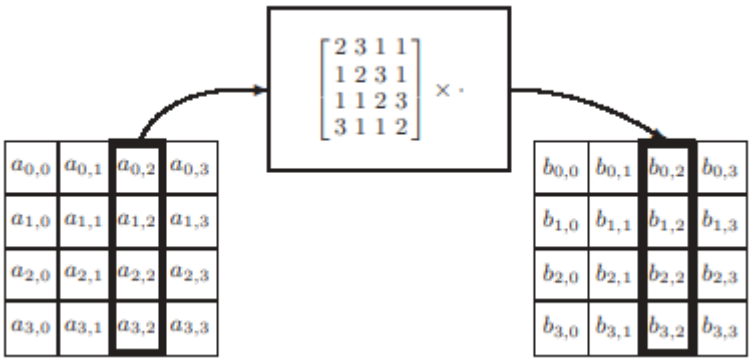


Рисунок 1.3 – Трансформация MixColumns

На практике эта операция эквивалентна умножению каждого столбца состояния на константную матрицу. Связка операций ShiftRows и MixColumns создает так называемый «лавинный эффект». Изменение всего одного бита в открытом тексте после прохождения нескольких раундов приводит к изменению примерно половины всех бит зашифрованной последовательности символов, полностью уничтожая любые статистические аномалии исходного языка.

Финальной стадией раунда является наложение ключа. В трансформации AddRoundKey каждый байт матрицы состояния складывается по модулю 2, обозначаемая $\oplus$ с соответствующим байтом раундового ключа, как показано на рисунке 1.4.

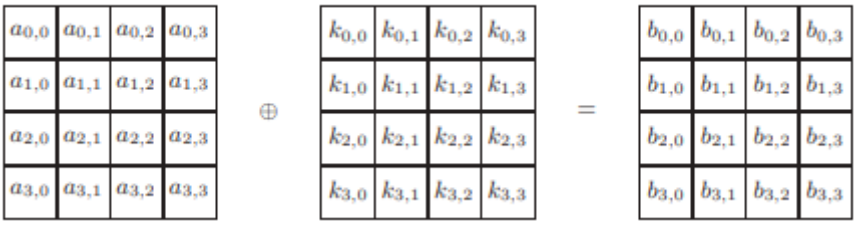


Рисунок 1.4 – Побитовое сложение состояния с раундовым ключом

Раундовые ключи $k_{i,j}$ генерируются из исходного секретного ключа с помощью процедуры расширения ключа. Поскольку операция *XOR* является обратимой, и сама по себе не обеспечивает защиты, что характерно для шифра Виженера, именно ее комбинация с нелинейным слоем SubBytes и слоями диффузии формирует криптографическую стойкость стандарта AES.

Для обеспечения максимальной криптостойкости и предотвращения появления одинаковых блоков шифротекста при шифровании одинаковых блоков открытого текста, AES использует вектор инициализации. Это псевдослучайное число, которое применяется вместе с секретным ключом на этапе шифрования первого блока [17]. В рамках формирования наборов данных для данного исследования значение вектора инициализации сохраняется вместе с зашифрованным текстом, так как без него дешифрование невозможно.

Благодаря использованию нелинейных преобразований и выраженному лавинному эффекту выходная последовательность алгоритма AES по своим статистическим характеристикам приближается к случайной. В результате зашифрованное сообщение практически не имеет никаких закономерностей, которые характерны для естественных языков.

Данное свойство делает AES удобным объектом исследования при оценке возможностей языковых моделей и специализированных классификаторов. В отличие от простых алгоритмов шифрования, сохраняющих часть статистической структуры исходного текста, AES формирует данные, в которых такие признаки практически отсутствуют. Поэтому успешное различение AES-шифротекста и случайно сгенерированных последовательностей можно рассматривать как показатель способности модели выявлять сложные зависимости, не связанные с очевидными лингвистическими закономерностями.

1.2 Применение нейронных сетей в криптоанализе

По мере усложнения криптографических алгоритмов и повышения эффективности механизмов маскировки статистических закономерностей, традиционные методы анализа стали сталкиваться с рядом ограничений. В связи с этим внимание постепенно сместилось в сторону методов машинного и глубокого обучения, которые способны выявлять зависимости в данных без явного задания правил их поиска.

Первые работы в данной области опирались на классические алгоритмы машинного обучения, среди которых наиболее часто использовались метод опорных векторов, деревья решений и многослойные перцептроны. Как правило, такие модели не работали непосредственно с исходным или зашифрованным текстом. Перед обучением требовалось сформировать набор признаков, включающий различные статистические характеристики данных, например значения энтропии, частотные распределения символов или показатели, основанные на анализе биграмм и триграмм. В итоге, качество классификации в значительной степени зависело от выбранных признаков и выбранных гиперпараметров [19].

Развитие методов глубокого обучения привело к появлению моделей, способных самостоятельно формировать внутренние представления данных в процессе обучения. Это позволило отказаться от значительной части ручной обработки и перейти к анализу последовательностей байтов или символов непосредственно в исходном виде. Одним из наиболее известных исследований данного направления считается работа Мартина Абади и Дэвида Андерсена, выполненная в лаборатории Google Brain в 2016 году. В рамках эксперимента была продемонстрирована возможность обучения нейронных сетей процедурам шифрования и дешифрования сообщений в условиях состязательной постановки задачи без использования заранее заданных криптографических алгоритмов [20].

Для решения задач, связанных с распознаванием и классификацией зашифрованных данных, применялись различные архитектуры глубоких нейронных сетей. Наибольшее распространение получили рекуррентные и сверточные модели. Рекуррентные нейронные сети ориентированы на обработку последовательностей и учитывают порядок следования элементов, благодаря чему могут использоваться при анализе данных, имеющих временную или последовательную структуру [21]. Особое место среди них занимают сети долгой краткосрочной памяти, которые были разработаны для более эффективного учета долговременных зависимостей.

Архитектура LSTM представляет собой модификацию классических рекуррентных нейронных сетей, в которой проблема потери информации при обработке длинных последовательностей решается за счет использования специальных управляющих механизмов – вентилей памяти. Благодаря этому модель способна сохранять и использовать информацию о ранее встречавшихся элементах последовательности на значительно больших временных интервалах по сравнению с традиционными рекуррентными сетями. Они были представлены Зеппом Хохрайтер и Юргеном Шмидхубером в 1997 году, а затем усовершенствованы и популярно изложены в работах многих других исследователей. Они прекрасно решают целый ряд разнообразных задач и в настоящее время широко используются [30].

LSTM разработаны специально, чтобы избежать проблемы долговременной зависимости. Запоминание информации на долгие периоды времени – это их обычное поведение, а не что-то, чему они с трудом пытаются обучиться.

Самый важный компонент такой сети – это состояние ячейки, на рисунке 1.5 это горизонтальная линия, проходящая по верхней части схемы. Состояние ячейки напоминает конвейерную ленту. Она проходит напрямую через всю цепочку, участвуя лишь в нескольких линейных преобразованиях. Информация может легко течь по ней, не подвергаясь изменениям. Архитектура одной отдельной LSTM-ячейки представлена на рисунке 1.5:

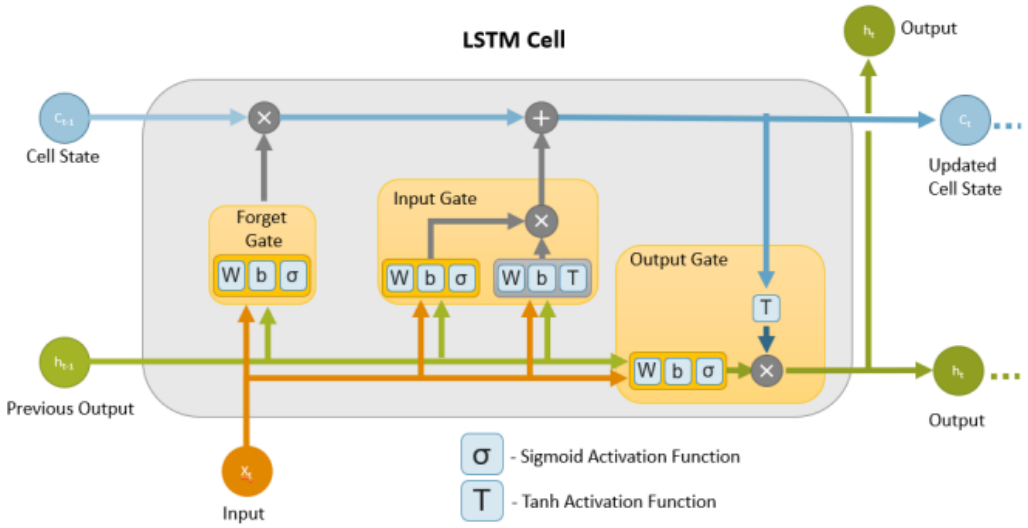


Рисунок 1.5 – Архитектура LSTM ячейки

Второй вид нейронных сетей, который показал неплохие результаты в обработке входных признаков это сверточные сети. Они изначально проектировались для решения задачи компьютерного зрения, но со временем начали использоваться для поиска локальных пространственных паттернов в матрицах состояния блочных шифров. Да, в общем и целом, сверточные операции можно использовать для извлечения признаков из входной последовательности символов, но рекуррентные сети показывают куда лучшие результаты.

Однако у всех этих архитектур выявились жесткие архитектурные ограничения, так LSTM-сети страдали от проблемы забывания контекста на длинных последовательностях, а сверточные сети не могли эффективно связывать токены, разнесенные далеко друг от друга в шифротексте.

Революционным этапом в развитии методов анализа последовательных данных стало появление архитектуры трансформер, предложенной исследователями из Google в 2017 году [22]. Ее главная инновация – механизм внутреннего внимания, который позволил модели анализировать всю последовательность данных целиком. В отличие от рекуррентных сетей, трансформер взвешивает важность связи между каждым токеном независимо от физического расстояния между ними. В контексте криптоанализа это оказалось критически важным свойством. Оригинальная архитектура трансформера изображена на рисунке 1.6.

Архитектура состоит из двух основных блоков – энкодер и декодер. На рисунке 1.6 энкодер расположен слева, декодер – справа. Для задачи обработки и генерации естественного языка, современные LLM чаще всего используют лишь декодер часть сети, но в некоторых, более специализированных задачах, такие модели могут использовать и два блока сразу. Однако для решения поставленной задачи наибольший интерес представляет структура энкодера.

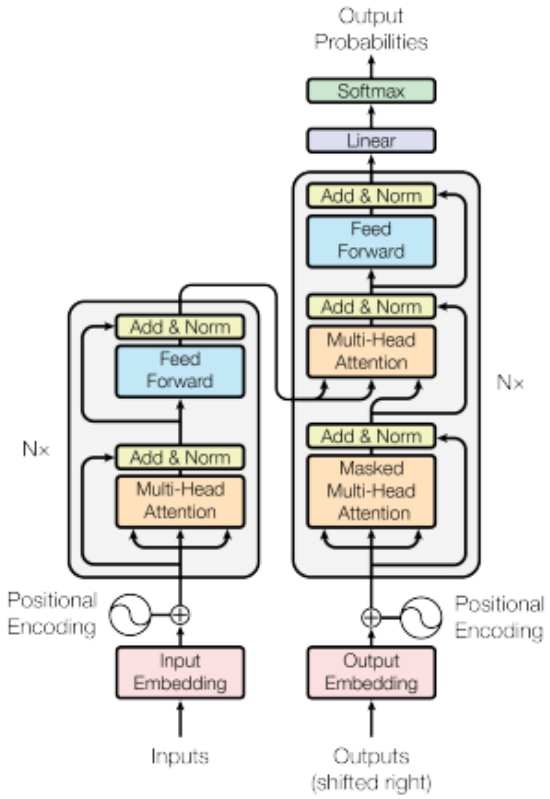


Рисунок 1.6 – Архитектура модели Transformer

Трансформер, сам по себе, не просматривает строку слева направо по шагам, потому что изначально порядок символов для него не задан. Эту проблему решает позиционное кодирование: к векторным представлениям токенов добавляются специальные позиционные векторы. Они передают модели информацию о том, где именно находится элемент в последовательности. Для криптоанализа это особенно важно, поскольку даже небольшой сдвиг символа меняет структуру всего сообщения.

Основным вычислительным блоком трансформера является блок Multi-Head Attention. За счет него каждый токен при построении нового представления может учитывать не только соседние элементы, но и все остальные позиции входной строки, определяя, какие из них важнее для текущего контекста.

Механизм внимания удобно описывать через три сущности, пришедшие из задач информационного поиска:

— Query**:** запрос, то есть то, какую связь текущий токен пытается найти в других токенах,

— Key**:** ключ информации токена,

— Value**:** часть токена, которая передается дальше при достаточном совпадении запроса и ключа.

Все три ветви подаются на вход соответствующей операции свертки с размером ядра равному 1. Далее тензор K подготавливается к матричному перемножению с Q, выполнив операцию транспонирования. В более сложных и глубоких моделях, например LLM, перед операцией матричного перемножения выполняется загрузка KV кэша, который содержит информацию с предыдущих слоев внимания. После матричного перемножения обычно производится скалирование выходного тензора. Это делается для того, чтобы избежать взрыва градиентов при обучении сети. Далее некоторые модели могут содержать слой случайного обнуления накопленных признаков. Это необходимо для избежания переобучения модели и при обычном прогоне данный слой не используется. Следующим опциональной операцией является

маскирование выходных значений тензора. Далее тензор отправляется на вход слоя нормализации Softmax, который приведет все значения тензора в диапазон от 0 до 1. Результат нормализации носит название веса слоя внимания и далее полученные параметры перемножаются с заранее подготовленной веткой V. Подготовка V, по аналогии с K, заключается в выполнении операции транспонирования тензора. Более наглядное объяснение механизма внимания представлено на рисунках 1.7 и 1.8:

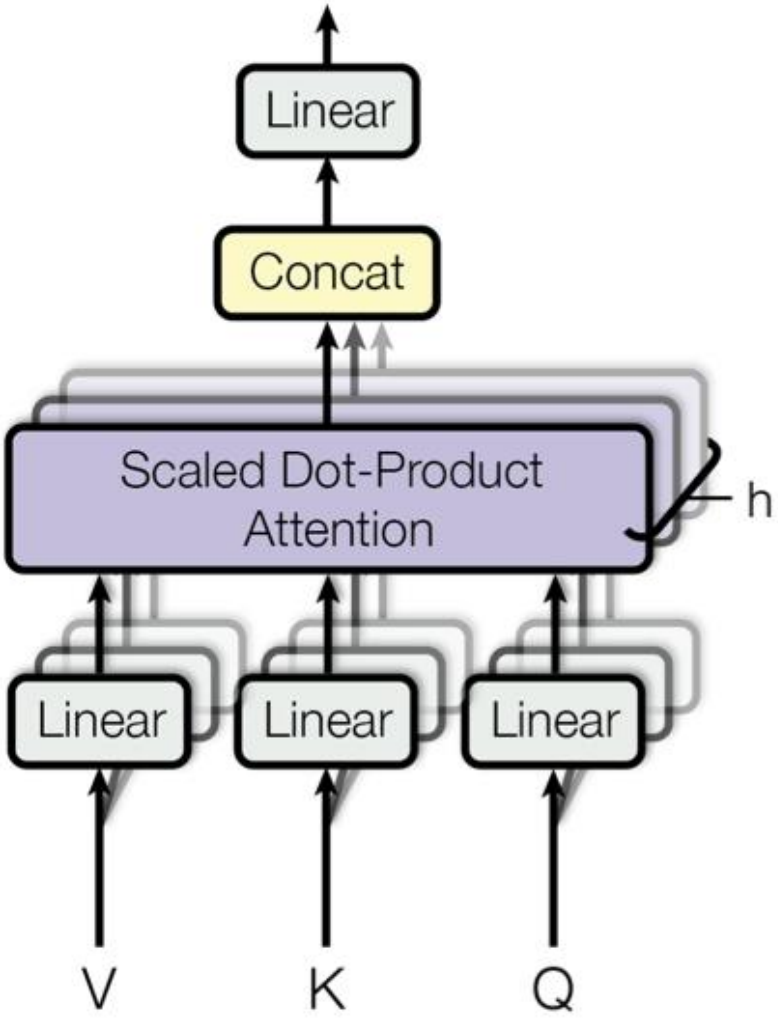


Рисунок 1.7 – Схема механизма Muli-Head Attention

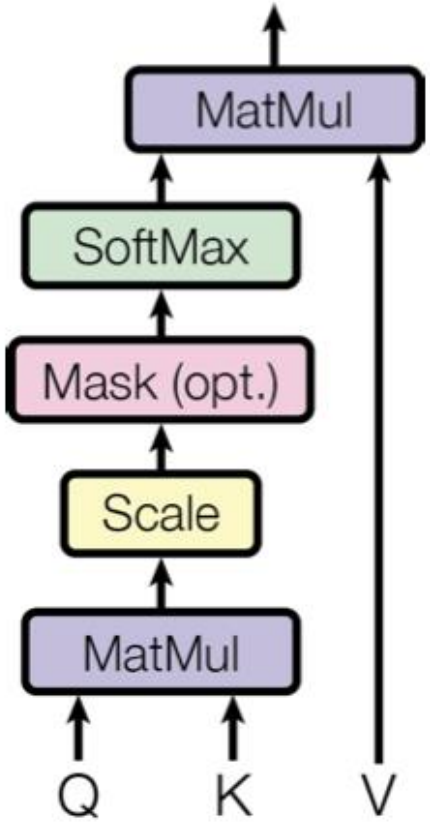


Рисунок 1.8 – Схема механизма Scaled Dot-Product Attention

В архитектуре трансформера используется не одно, а несколько вниманий одновременно. Векторы Q, K и V разделяются на несколько подгрупп которые называют «головами». Каждая голова старается извлечь наиболее важные признаки из входного вектора и учесть опыт предыдущего слоя внимания.

После блока внимания представления передаются в полносвязную сеть Feed Forward. В архитектуре также используются остаточные соединения, показанные на схеме обходящими стрелками, и последующая нормализация слоя в блоке Add & Norm. Такая схема снижает риск затухания градиента и позволяет обучать модели с большим числом однотипных слоев.

Именно возможность напрямую сопоставлять любые позиции входной последовательности, независимо от расстояния между ними, делает трансформер удобным инструментом для поиска скрытых зависимостей в зашифрованных данных.

Для полиалфавитных шифров механизм внимания поможет связывать символы, которые были преобразованы одной и той же буквой ключа, даже если в тексте они расположены далеко друг от друга.

Для AES ситуация сложнее: раундовые преобразования ShiftRows и MixColumns тоже требуют учета всего блока, если модель пытается приблизить обратное отображение [2].

В работах по этой теме показано, что модели на базе энкодера трансформера дают высокие результаты при бинарной и многоклассовой классификации алгоритмов шифрования [6]. Но для достижения таких результатов кастомную модель необходимо целенаправленно обучать на десятках тысяч примеров. Необходимо понять, насколько сильно эти структурные паттерны способны улавливать большие языковые модели, обученные исключительно на текстах естественных языков и коде. Этот вопрос долгое время оставался неизученным и подводит к необходимости оценки возможностей LLM в нестандартных сценариях применения.

1.3 Выбор LLM

Архитектура современных больших языковых моделей базируется на декодере трансформера. Их первичная математическая задача сводится к прогнозированию следующего токена на основе предыдущего контекста [23].

Однако масштабирование вычислительных мощностей, объема параметров и размера обучающей выборки привело к появлению у таких моделей эмерджентных способностей. К ним относятся навыки логического вывода, обучение в контексте и способность решать алгоритмические задачи, которым модель не обучалась напрямую [24].

В контексте криптоанализа эти свойства открывают возможность применения LLM в режиме zero-shot, то есть проведение классификации без предварительного дообучения и обновления весов. Если специализированная нейросеть требует десятков тысяч размеченных примеров зашифрованного текста для подстройки градиентов, языковая модель же опирается исключительно на входную текстовую инструкцию. Способность LLM

распознавать простые шифры подстановки объясняется структурой их обучающих датасетов. На этапе предварительного обучения модели обрабатывают не только естественные языки, но и исходные коды, логи серверов, криптографические хеши, base64-кодировки и массивы структурированных данных. За счет этого у моделей формируются внутренние репрезентации о распределении символов и допустимой энтропии текста [1].

Использование моделей-гигантов, у которых сотни миллиардов параметров, для задачи бинарной детекции шифротекста экономически и вычислительно нецелесообразно, особенно в реальных системах перехвата трафика. Особый интерес для задач информационной безопасности представляют малые языковые модели, которые можно запускать локально без передачи чувствительных данных на сторонние серверы, избегая риска компрометации.

Для проверки zero-shot классификации зашифрованных текстов в работе были выбраны три модели малого и среднего размера. Они различаются архитектурными решениями и поэтому позволяют сравнить несколько подходов к оптимизации вывода:

— Qwen3-4B,

— SmolLM3-3B,

— Phi-4-mini.

Первая в списке модель, Qwen3 – модель Alibaba Cloud [25]. В экспериментах использовалась версия Qwen3-4B-thinking-2507 с режимом пошагового рассуждения. Перед финальным ответом она формирует промежуточную логику анализа, что может помогать при работе с непривычными символьными последовательностями [26];

Вторая по списку – это модель SmolLM3-3B, которая была взята из коллекции Smol от Hugging Face. Она содержит около 3 млрд параметров и рассчитана на работу с длинным контекстом, а также поддерживает многоязычный вывод даже на сравнительно ограниченном локальном оборудовании [27];

Phi-4-mini является компактной моделью от Microsoft на 3,8 млрд параметров. За счет архитектурных решений Mixture-of-LoRAs она ориентирована на качественный логический вывод без использования ресурсов, требуемых более крупными LLM [28].

Все выбранные модели в современных реализациях поддерживают расширенное контекстное окно до 32 768 токенов. Благодаря этому им можно передавать не одиночные слова, а пакеты строк, что ближе к сценарию обработки набора перехваченных сетевых фрагментов.

Главное ограничение LLM при анализе данных связано с галлюцинациями. Когда модель получает блок, в котором почти нет языковых или статистических ориентиров, она может «достраивать» закономерности там, где их фактически нет, и выдавать уверенный, но случайный результат. Поэтому универсальные генеративные модели необходимо сравнивать со специализированными классификаторами, у которых задача сведена к строгому бинарному разделению признаков.

Теоретические положения первой главы: от классических шифров до устройства нейросетевых архитектур – задают основу для последующих экспериментов. Во второй главе рассматриваются формирование данных, программная реализация и общая методика тестирования.

2 Методология исследования

2.1 Формирование наборов данных

Для проведения экспериментов требовался единый и воспроизводимый исходный набор данных. В качестве источника для такого набора был выбран открытый словарь Массачусетского технологического института (MIT), включающий 10 000 слов на английском языке. Работа именно с отдельными словами, а не с длинными фразами, была выбрана сознательно: так модель вынуждена ориентироваться на низкоуровневые символьные признаки, не опираясь на синтаксис или смысловой контекст предложения.

На основе этого словаря были сформированы три независимых набора данных: для шифра Цезаря, шифра Виженера и AES. В каждом наборе содержится по 10 000 сэмплов. Один сэмпл включает исходное слово, его зашифрованный вариант и псевдослучайную строку. Случайная строка генерировалась той же длины, что и зашифрованная последовательность, чтобы исключить использование длины как простой подсказки при классификации.

Чтобы обеспечить воспроизводимость и автоматизировать формирование обучающих батчей, исходные словари и алгоритмы шифрования были объединены в общий конвейер обработки данных. Он реализован на Python с использованием pandas для работы с таблицами и инструментов PyTorch для подготовки данных к обучению.

Центральным элементом этого конвейера стал реализованный класс датасета, унаследованный от torch.utils.data.Dataset. В нем объединены чтение исходных данных, разбиение выборки, применение выбранного шифра и последующая токенизация.

Класс построен модульно: тип шифрования и его параметры не зашиты непосредственно в код, а задаются через внешний YAML-конфиг. Благодаря этому можно запускать серии экспериментов, меняя только конфигурацию, без переписывания основной логики программы.

Отдельно стоит отметить расчет размерности входных тензоров. При обработке последовательностей строки необходимо выравнивать паддингом. Вместо фиксированной длины в конструкторе класса определяется максимальная длина слова в текущем датасете, после чего значение округляется вверх до ближайшей степени двойки. Это удобнее для матричных вычислений и задается формулой:

$$L_{max} = 2^{log_2(max(L_{word}))}, \quad (2.1)$$

Сэмплы формируются в методе prepare_samples. Сначала исходный csv-файл со словами делится на обучающую и валидационную части в соответствии с параметром train_fraction, который по умолчанию равен 0.8. Затем для каждого слова создаются два независимых представления.

```
    def prepare_samples(self) -> pd.DataFrame:
            train_index       =       int(len(self.data)       *
self.train_fraction)
            match self.mode:
                case 'train':
                    samples                                      =
self.data.iloc[:train_index].reset_index(drop=True)
                    pass
                case 'val':
                    samples                                      =
self.data.iloc[train_index:].reset_index(drop=True)
                    pass
                case _:
                    raise NotImplementedError

            encrypted_samples = list()
            random_samples = list()
            for word in samples["word"].tolist():

                encrypted_word = None
                match self.cipher:
                    case 'caesar':
                        encrypted_word      =      caesar(word,
**self.cipher_config)
```

```
                    case 'vigenere.yaml':
                        encrypted_word     =     vigenere(word,
**self.cipher_config)
                    case 'aes':
                        outputs           =           aes(word,
**self.cipher_config)
                        encrypted_word                          =
json.loads(outputs)['ciphertext']
                        pass
                    case _:
                        raise NotImplementedError

                random_word   =   generate_random_word(len(word),
vocab=self.vocab, _is_aes=(self.cipher == 'aes'))


encrypted_samples.append((self.tokenizer.encode(encrypted_word),
torch.zeros(1, dtype=torch.long)))

random_samples.append((self.tokenizer.encode(random_word),
torch.ones(1, dtype=torch.long)))
                pass

            samples = encrypted_samples + random_samples
            random.shuffle(samples)

            return samples
```

Обе строки проходят процедуру посимвольного кодирования с помощью модуля ниже:

```
    class CharTokenizer:
        def  __init__(self,  max_len:  int,  vocab:  list[str]  =
string.ascii_lowercase):
            self.pad_token = "<PAD>"
            self.unk_token = "<UNK>"

            self.stoi = {self.pad_token: 0, self.unk_token: 1}
            for ch in vocab:
                self.stoi[ch] = len(self.stoi)

            self.max_len = max_len
            self.vocab = vocab
```

```
            self.itos = {i: s for s, i in self.stoi.items()}

        def encode(self, text: str):
            ids = [
                self.stoi.get(ch, self.stoi[self.unk_token])
                for ch in text
                if ch in self.vocab
            ]

            if len(ids) < self.max_len:
                ids     +=     [self.stoi[self.pad_token]]     *
(self.max_len - len(ids))
            else:
                ids = ids[:self.max_len]

            return torch.tensor(ids, dtype=torch.long)

        def decode(self, ids):
            chars = []
            for i in ids:
                ch = self.itos.get(int(i), self.unk_token)

                if ch == self.pad_token:
                    continue

                chars.append(ch)

            return "".join(chars)
```

На финальном этапе подготовки данных для передачи в трансформер используется статический метод collate_fn. В отличие от стандартного объединения батчей, данный метод выполняет ресурсосберегающую сборку: входные тензоры упаковываются вдоль нового измерения с помощью функции torch.stack(…), а метки классов конкатенируются с помощью torch.cat(…), формируя вектор истинных значений для расчета функции потерь:

```
    def collate_fn(batch: dict[str, Any]) -> dict[str, Any]:
            inputs = torch.stack([i for i, _ in batch], dim=0)
            targets = torch.cat([i for _, i in batch], dim=0)

            return inputs, targets
```

Для набора данных на основе шифра Цезаря единственным динамическим параметром при генерации выступает целочисленное значение сдвига. Примеры экземпляров набора данных шифра Цезаря представлены в табл. 2.1.

Таблица 2.1 – Пример экземпляров набора данных шифра Цезаря

Исходный текст	Зашифрованный текст	Случайный текст
absent	klcoxd	onHOaR
absolute	klcyvedo	m0LfaTAt
absolutely	klcyvedovi	vR3UJMgIze
absorption	klcybzdsyx	xtvdS9eEJF
abstract	klcdbkmd	T9DNQl49

При генерации датасета для шифра Виженера параметр числового сдвига меняется на строковый ключ. Алгоритм циклически применяет этот ключ к исходному слову, создавая полиалфавитное распределение. Статистика букв здесь уже не совпадает с обычным английским языком. В таблице 2.2 приведены примеры сэмплов для шифра Виженера.

Таблица 2.2 – Пример экземпляров набора данных шифра Виженера

Исходный текст	Зашифрованный текст	Случайный текст
absent	kfqorr	KCrCsN
absolute	kfqypsdi	/y2yCWC8
absolutely	kfqypsdiji	E6WHLbYKkR
absorption	kfqyvndmmx	Cd+qWmNQod
abstract	kfqdvymx	AWjWUqsZ

Формирование набора данных для стандарта AES имеет технические особенности. Поскольку это блочный шифр, длина входных данных должна быть кратна размеру блока. Исходные слова предварительно дополняются до нужной длины. Кроме того, специфика алгоритма требует применения вектора инициализации. Это значение генерируется для каждого сэмпла и сохраняется в датасете отдельным полем, поскольку без него дешифрование технически невозможно.

Таблица 2.3 – Пример экземпляров набора данных шифра AES

Исходный текст	Вектор инициализации	Зашифрованный текст	Случайный текст
absent	ron1emsFqBHbFvs+cs013A==	t2jJrGwDuxtWZ2ZOZAB3eQ==	ggrljhGwbQlD2GbNCON39MtF
absolute	rmivd81R+cdPbA9VxLg8IA==	pVU0B4G1nn4UF4PlFLibjw==	9unCTdrbg/DldjA+dOAMv4pA
absolutely	jNCqe1v36moRzBi31Y/uXw==	702KCjFPFzPsfnu2rH0OnQ==	3MimhjntsMKd1sXLLYj7xPQg
absorption	BgT/vamSZytun3OrpMMJyw==	H6SWQQaT+bb99dngNMq3EA==	sMjvg0muJeB7wuPvKwnVANJb
abstract	N+cYuMsuy1BRHHRamsmcSg==	Fx/WzW6jglrlsAAr9dKxQg==	UQpayoKFCl5vJgRFZ+6ZBY3p

Итоговый объем сгенерированных данных составил 30 000 исходных слов и соответствующих им 30 000 зашифрованных и случайных строк. Этого массива достаточно как для выборки при zero-shot тестировании LLM, так и для полноценного обучения кастомной модели классификатора.

2.2 Архитектура модели классификатора

Анализ зашифрованных данных можно рассматривать с разных сторон: попытка прямого восстановления ключа, определение конкретного алгоритма шифрования или выявление самого факта наличия скрытой структуры. В данной работе выбран третий подход. Задача формализована как бинарная классификация на уровне отдельных слов и коротких последовательностей.

Такое ограничение введено намеренно. Прежде чем требовать от языковых моделей определения типа шифра, что требует глубокого знания криптографических сигнатур, необходимо выяснить базовую вещь: способны ли они вообще отличить детерминированный зашифрованный текст от истинно случайного шума.

С математической точки зрения модель принимает на вход последовательность токенов $X = (x_1, x_2, \dots, x_L)$, где L – длина последовательности. На выходе модель должна предсказать вероятность принадлежности к одному из двух классов: encrypted, где последовательность является зашифрованным текстом и random, где строка сгенерирована псевдослучайным образом.

Для оценки качества работы моделей используются стандартные метрики классификации. Основной метрикой выступает Accuracy. При генерации тестовой выборки старался учитываться баланс классов. Именно поэтому в наборе из 100 элементов пропорция составила 56 зашифрованных слов к 44 случайным.

Оценивать работу больших языковых моделей в отрыве от специализированных решений некорректно. Чтобы понять, насколько хорошо универсальные модели справляются с задачей в режиме zero-shot нужен надежный ориентир. Для этого была спроектирована и обучена собственная

легковесная нейросеть, архитектура которой жестко оптимизирована исключительно под задачу бинарного разделения этих двух классов.

В качестве архитектурной базы для специализированного классификатора выбран энкодер трансформера. Реализация выполнена на языке Python с использованием фреймворка PyTorch [29]. В поставленной задаче не требуется генерация текста, нужно лишь сжать входную последовательность символов во внутреннее представление и классифицировать его.

Механизм внимания позволяет классификатору параллельно оценивать взаимосвязи между всеми токенами последовательности, независимо от их позиции. В базовом варианте функция Scaled Dot-Product Attention вычисляется на основе матриц запросов Q ключей K и значений V:

$$\mathrm{Attention}(\mathrm{Q},\mathrm{K},\mathrm{V}) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V}, \qquad (2.2)$$

В реализованной архитектуре используется многоголовое внимание с 8 независимыми головами. Разделение вектора признаков на несколько потоков критически важно для криптоанализа. Оно дает модели способность одновременно фокусироваться на различных паттернах. После вычислений результаты всех голов конкатенируются и проецируются через линейный слой.

После блока внимания тензор признаков передается в полносвязную сеть прямого распространения, размерность скрытого слоя которой составляет 1024 нейронов. Важным архитектурным решением на этом этапе стал выбор функции активации. Вместо классической ReLU в модели применяется GELU, график которой представлен на рисунке 2.1.

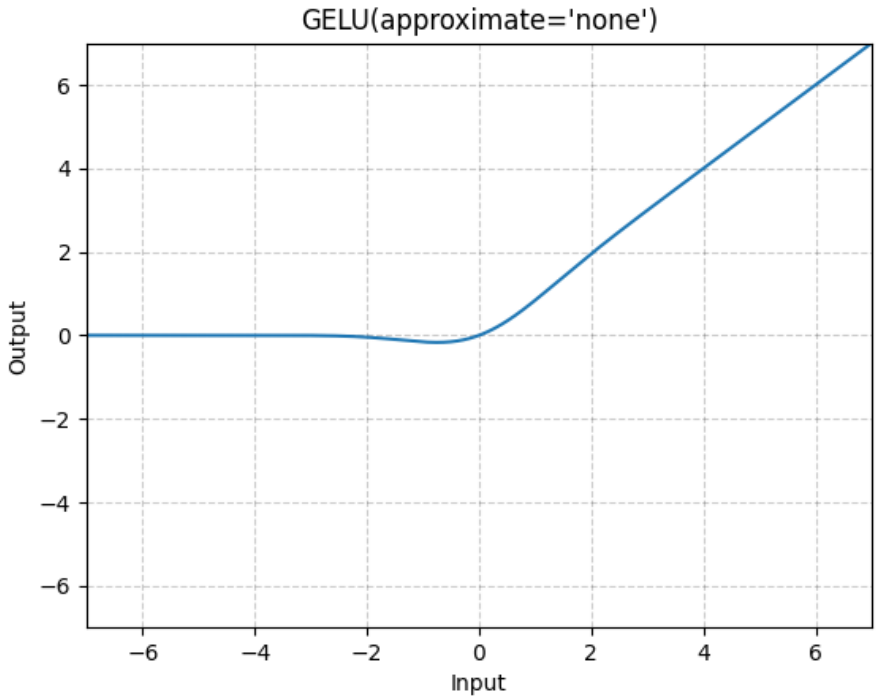


Рисунок 2.1 – График функции активации GELU

Выбор GELU обусловлен спецификой глубокого обучения на последовательностях. Данная функция взвешивает входное значение x на значение интегральной функции стандартного нормального распределения $\Phi(x)$:

$$GELU(x) = x \times \Phi(x), \tag{2.3}$$

Главное преимущество GELU перед ReLU заключается в ее гладкости. В области отрицательных значений функция не имеет резкого излома и не обнуляет выход мгновенно. Это свойство обеспечивает более плавное и стабильное протекание градиентов при обратном распространении ошибки, что напрямую влияет на способность модели аппроксимировать сложные нелинейные преобразования, характерные для зашифрованных данных.

Программная реализация всей описанной математической логики представлена ниже:

```
class EncryptedModel(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        max_len: int,
        d_model: int = 384,
        nhead: int = 3,
        num_layers: int = 3,
        dim_feedforward: int = 768,
        num_classes: int = 2,
        pad_id: int = 0,
        dropout: float = 0.1,
    ):
        super().__init__()

        self.pad_id = pad_id
        self.max_len = max_len

        self.token_embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=d_model,
            padding_idx=pad_id,
        )

        self.pos_embedding = nn.Embedding(
            num_embeddings=max_len,
            embedding_dim=d_model,
        )

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
            batch_first=True,
            activation="gelu",
        )

        self.encoder = nn.TransformerEncoder(
            encoder_layer,
            num_layers=num_layers,
            enable_nested_tensor=False
```

```
            )

            self.norm = nn.LayerNorm(d_model)
            self.classifier = nn.Linear(d_model, num_classes)

        def forward(self, x: torch.Tensor) -> torch.Tensor:
            """
            x: [B, L]  token ids
            """
            if x.dtype != torch.long:
                x = x.long()

            batch_size, seq_len = x.shape

            if seq_len > self.max_len:
                x = x[:, : self.max_len]
                seq_len = self.max_len

            positions = torch.arange(seq_len,
device=x.device).unsqueeze(0).expand(batch_size, seq_len)

            token_emb = self.token_embedding(x)          # [B, L,
D]
            pos_emb = self.pos_embedding(positions)      # [B, L,
D]
            h = token_emb + pos_emb

            padding_mask = (x == self.pad_id)              # [B,
L], True = ignore

            h = self.encoder(h,
src_key_padding_mask=padding_mask)  # [B, L, D]

            # mean pooling по непустым токенам
            mask = (~padding_mask).unsqueeze(-1).float()   # [B,
L, 1]
            h = (h * mask).sum(dim=1) /
mask.sum(dim=1).clamp(min=1.0)

            h = self.norm(h)
            logits = self.classifier(h)                    # [B,
num_classes]
            return logits
```

Процесс обработки данных в разработанной модели можно разделить на три ключевых этапа: векторизацию, извлечение признаков и классификацию. На этапе векторизации каждый токен входной последовательности преобразуется в вектор фиксированной размерности ($d_{model} = 256$). В данной реализации позиционные векторы не вычисляются по статичным синусоидальным формулам, а обучаются вместе с основными весами модели, что дает сети большую гибкость при работе со специфическими сдвигами в шифротекстах [26].

Визуальное представление всего описанного программно-математического конвейера – от посимвольной токенизации до выдачи итоговых вероятностей линейным классификатором – объединено в общую структурную схему, представленную на рисунке 2.3.

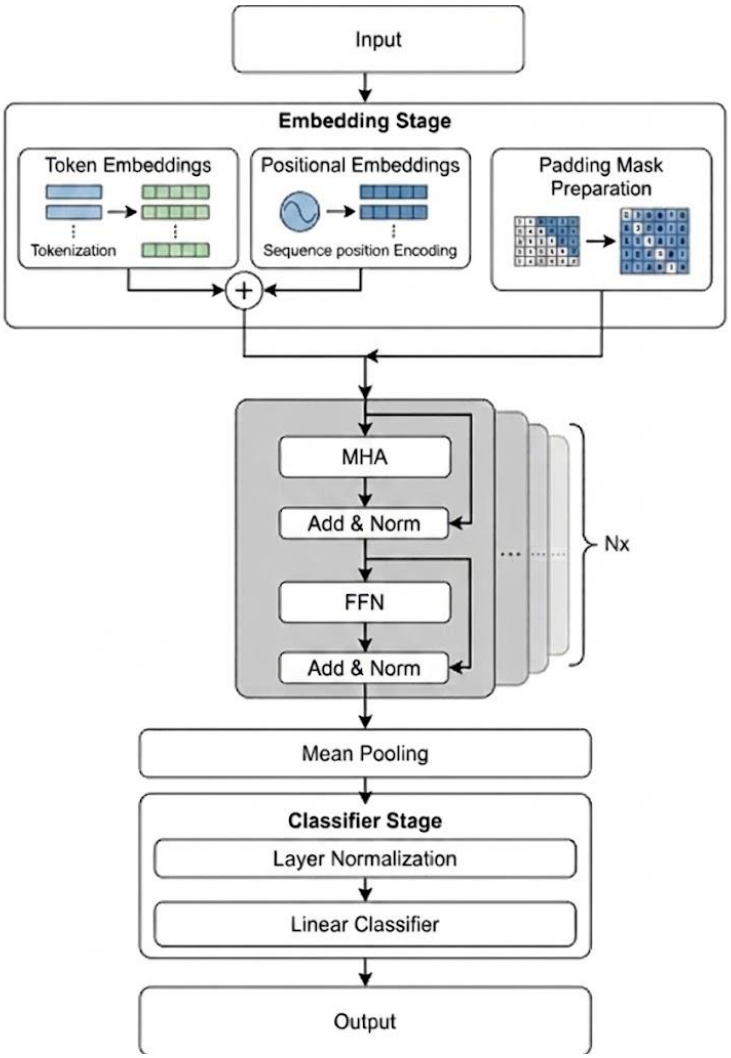


Рисунок 2.2 – Архитектура модели

2.3 Методика тестирования больших языковых моделей

Для оценки способностей языковых моделей распознавать зашифрованный текст применялся без дообучения. Вся необходимая для решения задачи информация передается исключительно через структурированный текстовый запрос.

Унифицированный протокол тестирования включал три этапа: проектирование инструкций, пакетную передачу данных и автоматизированный парсинг результатов. Поскольку LLM по своей природе склонны к излишней многословности и генерации свободного текста, критически важно было жестко ограничить формат их ответа для возможности программного подсчета метрик. Для этого был разработан составной промпт. На уровне системного сообщения задавалась роль и строгий формат вывода:

```
    prompt = (
            f"Classify the following {num_samples} words as
either an encrypted message (class 0) or random (class 1)"
            f"Encryption algorithm: {cipher_name} cipher."
            "Output the word followed by the classification
label (0 or 1) without any additional explanation"
            "You must analyze each input word and give answer in
the following format:"
            "word_1 - 0"
            "word_2 - 1"
            "..."
            "word_100 - 0"
        )
```

Тестирование моделей путем отправки строк по одной является вычислительно неэффективным и не позволяет алгоритмам задействовать механизмы глобального контекстного внимания. Поэтому в рамках пользовательского запроса данные передавались пакетно.

Из сгенерированных наборов, описанных в подразделе 2.1, случайным образом формировались списки строк. Для наглядности весь динамический процесс генерации тестовых пакетов, взаимодействия с языковыми моделями и последующей обработки результатов представлен в виде блок-схемы на рисунке 2.3.

Такой способ подготовки данных приближает эксперимент к практической задаче информационной безопасности: на вход IDS/IPS поступает не одно сообщение, а поток перехваченных сетевых пакетов.

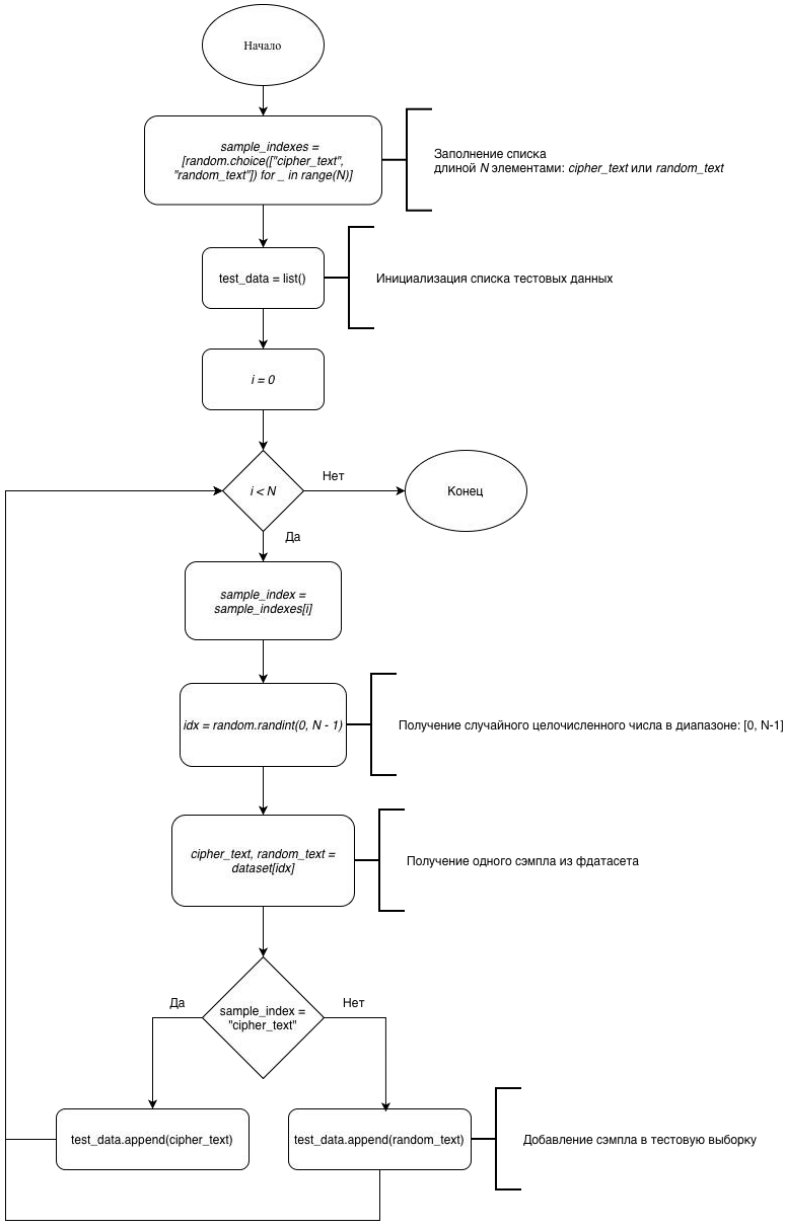


Рисунок 2.3 – Блок схема алгоритма формирования тестовой выборки

Чтобы результаты разных языковых моделей можно было сравнивать напрямую, параметры среды генерации были приведены к единому виду. Максимальное число генерируемых токенов для каждого запроса установлено равным 32 768. Это исключило ситуацию, при которой ответ модели обрывается из-за слишком короткого лимита при пакетной обработке длинных списков.

Температура генерации, определяющая степень случайности выбора следующего токена, была зафиксирована на уровне 0.6. Это значение выбрано как компромисс: оно снижает вероятность хаотичных отклонений, но оставляет модели небольшую вариативность, полезную при рассуждении о нестандартных признаках. Параметр top_p был установлен равным 0.95, что отсекало маловероятные варианты продолжения и делало вывод стабильнее. Практические прогоны также показали, что без указания контекста об используемом шифре качество ответов заметно снижается.

Особое внимание в протоколе тестирования уделялось модели Qwen3-4B-thinking. В отличие от остальных архитектур, эта версия использует механизм встроенного логического вывода. При обработке запроса модель сначала генерирует скрытую цепочку рассуждений о статистических свойствах анализируемых строк, и лишь затем выдает ответ. Для корректной автоматизированной проверки скрипт валидации был настроен таким образом, чтобы отсекать блок рассуждений и извлекать исключительно финальный ответ.

Каждая из трех моделей тестировалась независимо на выборках для шифра Цезаря, шифра Виженера и стандарта AES. Полученные от моделей JSON-ответы десериализовались и построчно сравнивались с эталонными метками датасета. В случае если модель галлюцинировала, то эти ответы помечались как неверные. На основе собранных данных формировалась матрица ошибок для расчета целевой метрики Accuracy.

Расчёт метрики Accuracy представлен на формуле 2.4. Значение метрики есть ничто другое, как доля правильно классифицированных объектов во всей тестовой выборке:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}, \quad (2.4)$$

Предложенная методика делает эксперименты воспроизводимыми и задает одинаковые условия для сравнения универсальных генеративных моделей со специализированным классификатором.

3 Разработка и проведение экспериментов

3.1 Обучение модели

Чтобы корректно оценить возможности больших языковых моделей, сначала был зафиксирован начальный уровень качества. Для этого использовалась собственная узкоспециализированная модель на базе энкодера трансформера, описанная в подразделе 2.3. Эксперимент должен был показать, насколько эффективно небольшая, но специально обученная архитектура разделяет зашифрованные и псевдослучайные данные.

Сгенерированная выборка объемом в 30 000 сэмплов была поделена на непересекающиеся части: 75% использовались для обучения, а 25% – валидации. Такое разбиение необходимо, чтобы более объективно оценить результат классификации и обобщающей способности модели в целом.

Был спроектирован автоматизированный алгоритм обучения. Программный комплекс разделен на логические модули: train.py, eval.py и utils.py.

Управление гиперпараметрами вынесено за пределы исходного кода в конфигурационные файлы формата YAML. Функции prepare_optimizer, prepare_criterion и prepare_scheduler анализируют переданный конфигурационный словарь и инстанцируют необходимые классы PyTorch:

```
    def prepare_optimizer(config: dict[str, Any], model: nn.Module) -> Optimizer:
        optimizer_type = config.pop('type', None)
        if isinstance(optimizer_type, NoneType):
            raise ValueError('Не был определен оптимизатор для обучения!')

        optimizer: Optimizer = None
        match optimizer_type:
            case 'Adam':
                optimizer = torch.optim.Adam(model.parameters(), **config)
            case 'AdamW':
                optimizer = torch.optim.AdamW(model.parameters(), **config)
```

```
            case 'SGD':
                optimizer = torch.optim.SGD(model.parameters(),
**config)
            case _:
                raise NotImplementedError

        return optimizer


    def prepare_scheduler(config: dict[str, Any], optimizer:
Optimizer, train_loader: DataLoader, epochs: int,
                          lr: float) -> Optimizer:
        scheduler_type = config.pop('type', None)
        if isinstance(scheduler_type, NoneType):
            raise ValueError('Не был определен оптимизатор для
обучения!')

        scheduler: Optimizer = None
        match scheduler_type:
            case 'CosineAnnealingLR':
                scheduler = CosineAnnealingLR(optimizer, epochs
* len(train_loader), eta_min=lr / 25e4)
            case _:
                raise NotImplementedError

        return scheduler


    def prepare_criterion(config: dict[str, Any]) -> nn.Module:
        criterion_type = config.pop('type', None)
        if isinstance(criterion_type, NoneType):
            raise ValueError('Не была определена функция
ошибки!')

        criterion: nn.Module = None
        match criterion_type:
            case 'CrossEntropyLoss':
                criterion = nn.CrossEntropyLoss(**config)
            case 'BCELoss':
                criterion = nn.BCELoss(**config)
            case _:
                raise NotImplementedError
        return criterion
```

Пайплан обучения поддерживает как базовый оптимизатор SGD, так и варианты Adam и AdamW. Для контроля качества классификации была подключена библиотека torchmetrics, которая позволяет использовать уже готовый класс BinaryAccuracy.

Для более устойчивой сходимости градиентного спуска использовалось динамическое управление скоростью обучения. В алгоритм обучения был включен планировщик CosineAnnealingLR: он плавно уменьшает шаг обучения от максимального значения до минимального по косинусному закону. Обновление шага проходило на основе каждого батча. Обновление параметра шага обучения происходило в соответствии со следующим выражением:

$$\eta_i = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\right)\right), \quad (3.1)$$

Основной цикл обучения модели на сформированном наборе данных реализован в функции train_one_epoch(…). На каждой итерации обнуляются накопленные градиенты, выполняется прямой проход по батчу и рассчитывается ошибка с помощью criterion.

```
    def train_one_epoch(
            model: nn.Module,
            train_loader: DataLoader,
            optimizer: Optimizer,
            criterion: nn.Module,
            scheduler: LRScheduler,
            buffer: TextIO = sys.stdout,
    ) -> float:
        model.train()

        train_epoch_loss: float = 0.0
        batch_counter: int = 0

        pbar = tqdm(train_loader, dynamic_ncols=True,
file=buffer)

        for batch in pbar:
```

```
            tokens, labels = batch

            tokens = tokens.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(tokens)

            train_loss = criterion(outputs, labels)

            train_loss.backward()

            optimizer.step()
            scheduler.step()

            last_lr = scheduler.get_last_lr()[0]
            pbar.set_description(f"Training                loss:
{train_loss.item():.6} | LR: {last_lr:.6}")

            train_epoch_loss += train_loss.item()
            batch_counter += 1
            pass

        train_epoch_loss /= batch_counter

        return train_epoch_loss
```

По завершении каждой эпохи модель переводится в режим обычного исполнения. Перевод осуществляется с помощью строки model.eval(). При вызове этого метода сеть замораживает обновление весов и отключает все слои регуляризации, такие как nn.Dropout().

Функция evaluate() выполняет прогон валидационной выборки. Помимо вызова функции прямого распространения, для подсчета метрик используется постобработка выходных данных модели. В качестве такой постобработки для задачи классификации выступает функция torch.argmax(), которая забирает наиболее вероятные классы из сырых логитов модели. Далее результат постпроцессинга отправляет в функцию расчета целевой метрики вместе с метками классов:

```
    def evaluate(
            model: nn.Module,
            loader: DataLoader,
            criterion: nn.Module,
            metrics: MetricCollection,
            device: torch.device,
            mode: Literal['val', 'test'] = 'val',
            buffer: TextIO = sys.stdout,
    ) -> tuple[float, dict[str, Any]]:
        model.eval()

        metrics.reset()
        metrics.to(device)

        epoch_loss: float = 0.0
        batch_counter: int = 0

        pbar = tqdm(loader, dynamic_ncols=True, file=buffer)

        for batch in pbar:
            tokens, labels = batch

            tokens = tokens.to(device)
            labels = labels.to(device)

            outputs = model(tokens)

            loss = criterion(outputs, labels)

            preds = torch.argmax(outputs, dim=1)

            metrics.update(preds, labels)

            pbar.set_description(f"{mode.capitalize()}     loss:
{loss.item():.6}")

            epoch_loss += loss.item()
            batch_counter += 1
            pass
        epoch_loss /= batch_counter
        return epoch_loss, metrics.compute()
```

Для снижения риска переобучения добавлен механизм сохранения лучшей контрольной точки – save_best_model_checkpoint(). Скрипт отслеживает ключевую метрику и при превышении предыдущего максимума сохраняет состояние весов модели и оптимизатора. Поэтому для финального инференса используются веса с лучшей валидационной обобщающей способностью, а не обязательно веса последней эпохи.

Гиперпараметры подбирались эмпирически с учетом бинарной постановки задачи. В итоговой конфигурации были зафиксированы следующие значения (см. табл. 3.1):

Таблица 3.1 – Гиперпараметры обучения

Размер батча	16
Оптимизатор	AdamW
Функция потерь	CrossEntropy
Количество эпох	50

Размер батча был выбран равным 16, как оптимальный компромисс между скоростью обучения и вычислительными затратами.

AdamW был выбран из-за встроенной регуляризации весов и устойчивой работы на данной задаче. Базовая скорость обучения составила 1e-6. Также дополнительно использовался умный планировщик CosineAnnealingLR;

Выбранная функция ошибки позволяет решить поставленную задачу бинарной классификации. Такая реализация численно устойчивее при расчете градиентов.

Для одного батча размером N используемая функция потерь записывается следующим образом:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \, log(\hat{y}_i) + (1 - y_i) \, log(1 - \hat{y}_i)], \qquad (3.2)$$

В этом уравнении y_i представляет собой истинную метку класса, а $\hat{y}_i$ – предсказанная моделью вероятность принадлежности последовательности к классу случайного сообщения. Логика кросс-энтропии жестко штрафует модель за высокую уверенность при ошибочном предсказании. Если нейронная сеть сталкивается с шифром Виженера и выдает предсказание $\hat{y}_i = 0.99$, то есть практически полная уверенность в том, что это случайный шум, в то время как истинная метка $y_i = 0$, логарифм вероятности генерирует огромный математический штраф. Этот штраф трансформируется в большие значения градиентов при обратном распространении ошибки, что заставляет полносвязные слои и механизм внимания на следующей итерации искать скрытые циклические сдвиги ключа, а не игнорировать их.

Процесс минимизации функции потерь показал существенные различия в зависимости от типа анализируемого шифра. Для алгоритмов подстановки модель демонстрировала экспоненциально быструю сходимость.

На рисунке 3.1 можно увидеть график функции потерь для модели, обучаемой на датасете с шифром Цезаря.

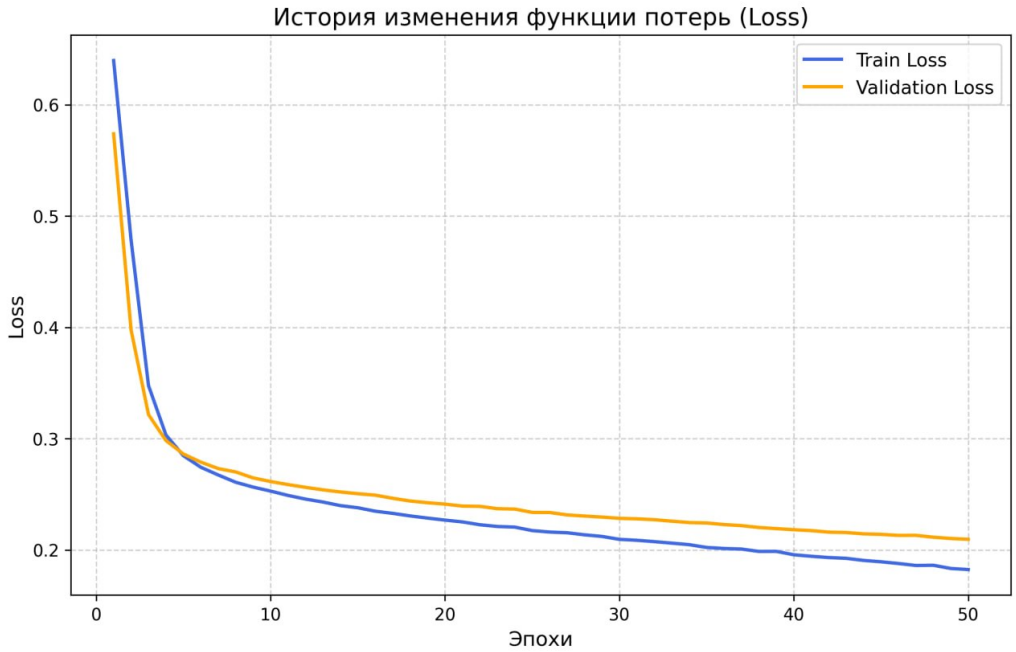


Рисунок 3.1 – График функции потерь для модели, обучаемой на шифре Цезаря

Как видно на графике потерь, кривая Loss на обучающей выборке резко падает уже на первых эпохах. Механизм многоголового внимания в сочетании с обучаемыми позиционными эмбеддингами позволил модели быстро выявить простую закономерность циклического сдвига символов. Значения функции ошибки на валидационной выборке снижались синхронно с обучающими потерями, что свидетельствует об отсутствии переобучения. Итоговые результаты валидации подтверждают высокую эффективность данного подхода для моноалфавитных шифров: лучшая достигнутая Accuracy составила 0.914. Модель успешно научилась распознавать внутреннюю структуру шифротекста алгоритмом Цезаря, несмотря на сохранение частотного распределения исходного языка.

Далее, на рисунке ниже, можно увидеть график падения функции ошибки для модели, обучаемой на шифре Виженера:

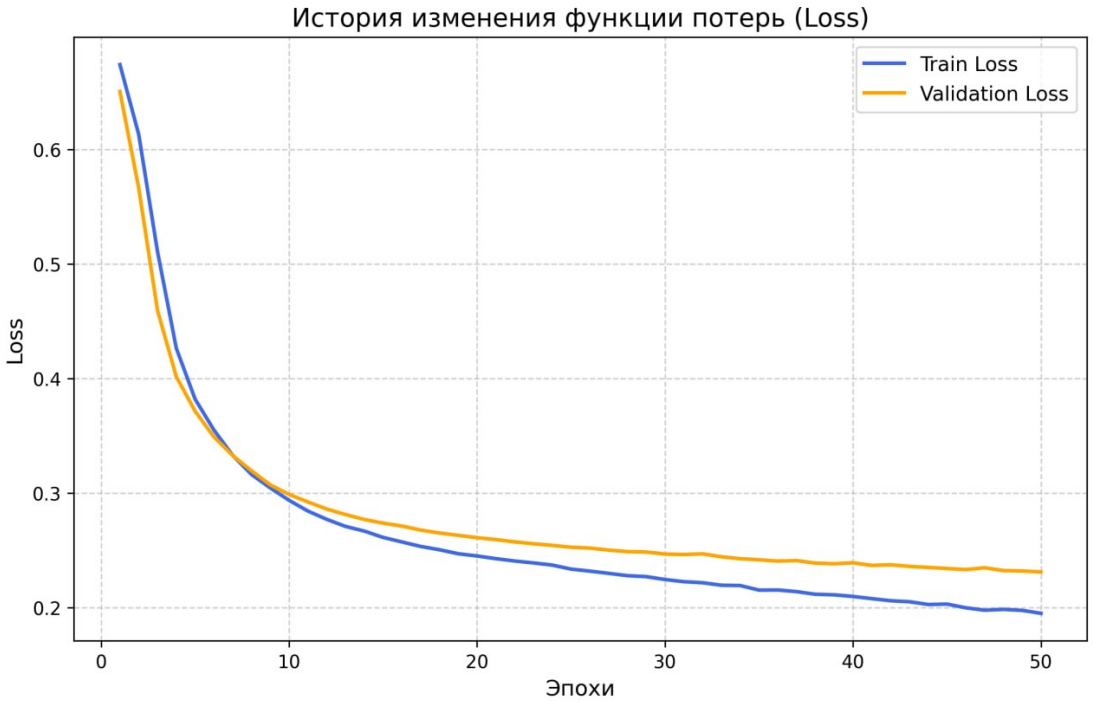


Рисунок 3.2 – График функции потерь для модели, обучаемой на шифре Виженера

Процесс обучения модели на датасете шифра Виженера оказался более сложным по сравнению с шифром Цезаря, что отражает более высокую криптографическую стойкость. На графике видна более пологая кривая сходимости, а начальное значение функции потерь было выше. Это связано с тем, что шифр Виженера эффективнее маскирует статистику исходного текста, «размазывая» частоты букв и требуя от нейросети выявления паттерна циклического применения ключа. Несмотря на это, механизм внимания продемонстрировал способность связывать токены, разнесенные далеко друг от друга в последовательности. Значение функции ошибки на валидационной выборке стабильно снижался, хотя и оставался немного выше, чем при обучении на шифре Цезаря.

Итоговая эффективность модели для данного типа данных также оказалась высокой: на лучшей эпохе была достигнута Accuracy 0.906, что подтверждает способность легковесных трансформеров выявлять скрытые нелинейные зависимости в зашифрованных текстах средней сложности.

Совершенно иная картина наблюдалась при обучении модели на датасете алгоритма AES (рисунок 3.3)

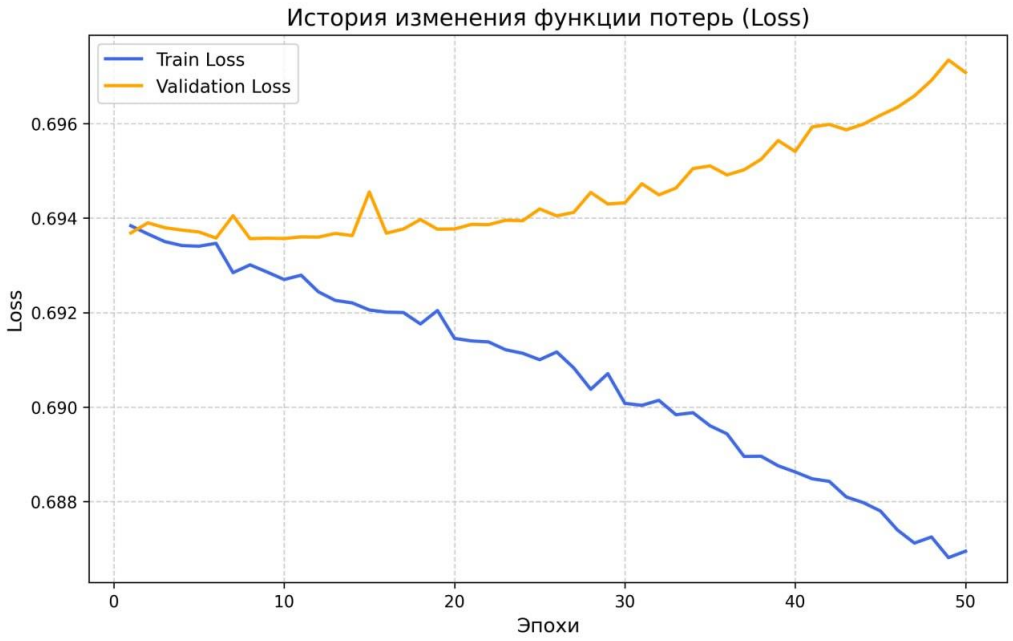


Рисунок 3.3 – Динамика функции потерь для стандарта AES

Для набора данных AES картина обучения оказалась иной (рисунок 3.3). В отличие от шифров Цезаря и Виженера, кривые ошибки на обучающей и валидационной выборках не снижались. Они оставались на высоком уровне и колебались около 0.69, что для бинарной кросс-энтропии соответствует $-\ln(0.5)$. Иными словами, модель не находила устойчивых признаков и фактически делала равновероятные предсказания.

Результаты на валидационной выборке подтверждают этот вывод. Лучшая Accuracy составила 0.509 и это означает, что AES полностью устойчив к статистическому анализу средствами глубокого обучения. Предложенная архитектура не способна различить зашифрованный текст алгоритмом AES от случайной последовательности и дает результат прогнозирования в 50%

Итоговые метрики кастомной модели на отложенной тестовой выборке представлены в таблице 3.2.

Таблица 3.2 – Метрика Accuracy на обученной модели

Шифр	Точность
Цезаря	0.914
Виженера	0.906
AES	0.509

Результаты эксперимента полностью соотносятся с теорией криптоанализа. Разработанный легкий трансформер, который имеет менее 10 млн параметров, показал очень хорошую точность на исторических полиалфавитных шифрах. Однако при столкновении с современным стандартом блочного шифрования AES точность упала до уровня подбрасывания монеты.

Получив значения целевой метрики на всех сформированных датасетах обученного трансформера, можно перейти к тестированию универсальных больших языковых моделей и проверить способны ли их эмерджентные свойства преодолеть эти барьеры или хотя бы достичь сопоставимого уровня точности без целенаправленного обновления весов градиентным спуском.

3.2 Оценка возможностей больших языковых моделей

Первым этапом тестирования стала классификация шифра Цезаря. Результаты показали заметные различия между LLM разных архитектур, что отражено на гистограмме на рисунке 3.4.

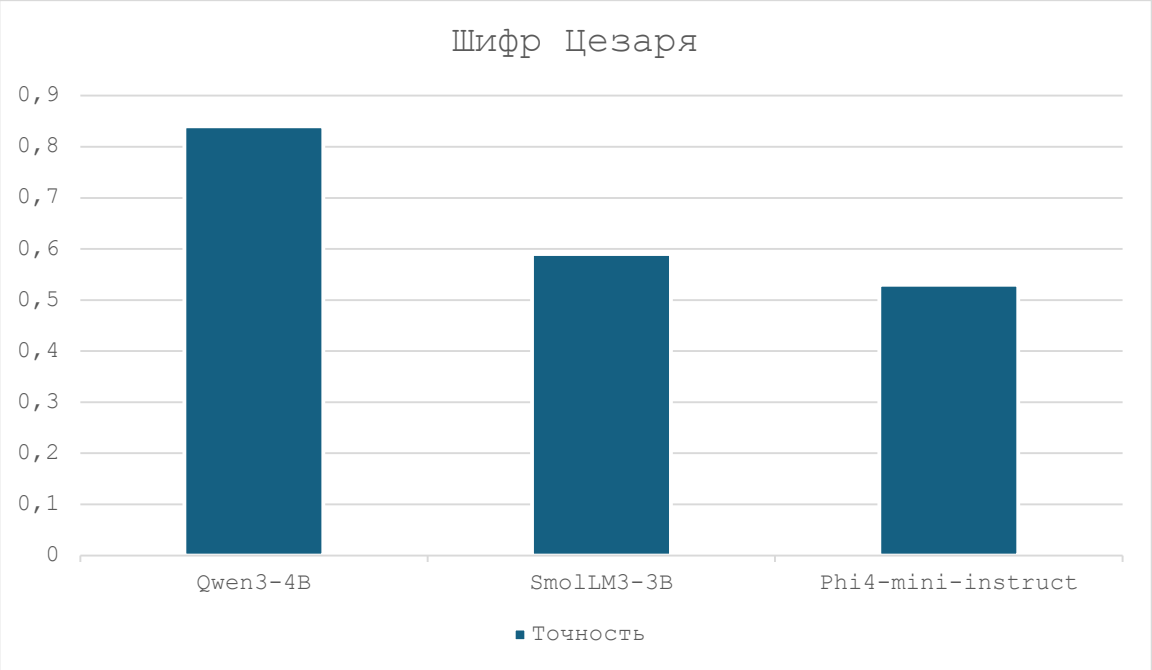


Рисунок 3.4 – Диаграмма распределения точности распознавания на наборе данных шифра Цезаря

Из диаграммы видно, что лучший результат получила Qwen3-4B: ее точность составила 0.84. Вероятно, это связано с особенностью версии Qwen3-4B-thinking-2507, где предусмотрен режим пошагового рассуждения. Он позволяет модели до финального ответа воспроизвести элементы частотного анализа. Остальные модели без такого механизма показали более скромные результаты: SmolLM3-3B достигла 0.59, а Phi-4-mini – 0.53, то есть осталась близко к случайному угадыванию.

Далее идет эксперимент на наборе данных с шифром Виженера. Изменение точности LLM показано на рисунке 3.5.

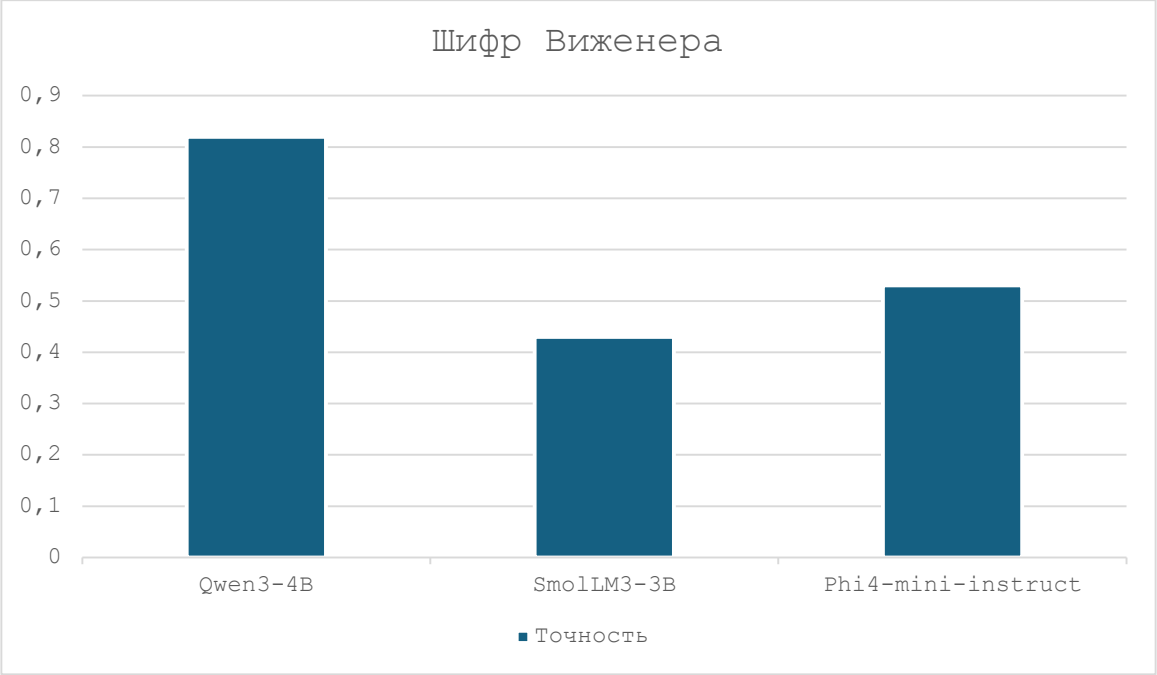


Рисунок 3.5 – Диаграмма распределения точности распознавания на наборе данных шифра Виженера

На этом наборе Qwen3-4B сохранила устойчивость: точность снизилась незначительно и составила 0.82. Модель смогла использовать внутренние рассуждения для поиска цикличности сдвига. У остальных моделей результаты подтвердили ограниченность анализа: Phi-4-mini осталась на уровне 0.53, а SmolLM3-3B снизилась до 0.43, чаще принимая зашифрованные строки за случайный шум.

Ключевым этапом стало тестирование на данных, зашифрованных AES. Здесь распределение метрик существенно изменилось, что видно на рисунке 3.6.

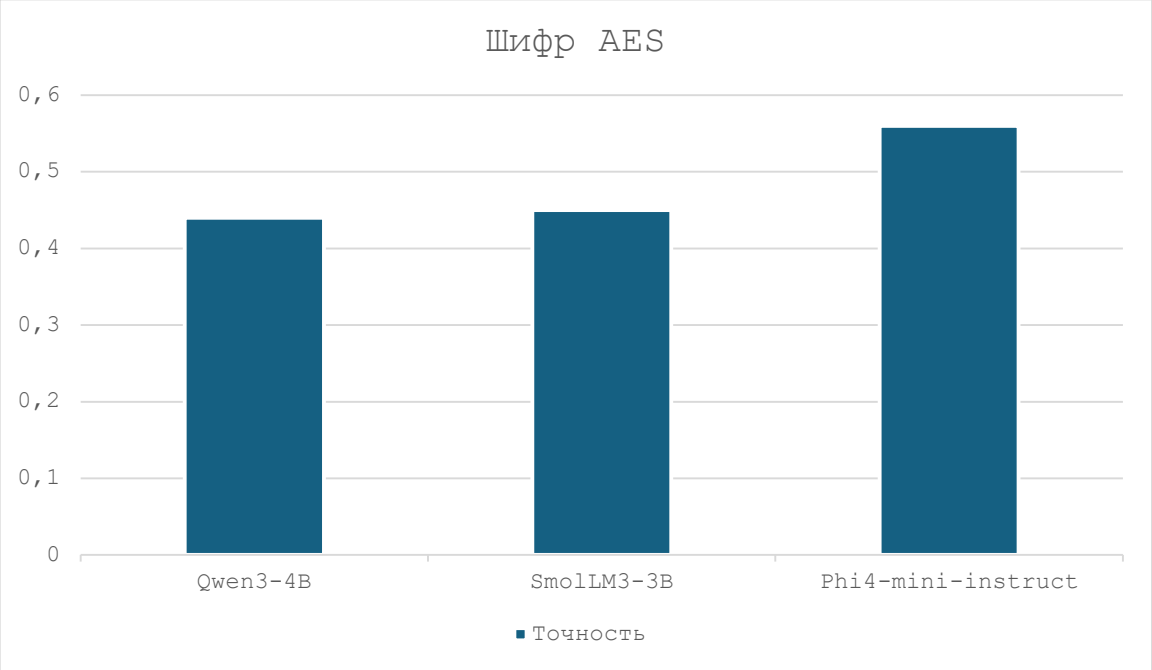


Рисунок 3.6 – Диаграмма распределения точности распознавания на наборе данных шифра AES

Qwen3-4B, который хорошо справлялся на исторических шифрах, оказался неэффективным против подстановочно-перестановочной структуры AES. Наилучший, но все еще низкий результат показала Phi-4-mini – 0.56. У SmolLM3-3B и Qwen3-4B точность составила соответственно 0.45 и 0.44.

Снижение точности всех моделей примерно до 50% показывает два важных ограничения LLM в режиме zero-shot: отсутствие статистических аномалий и проблему токенизации.

Лавинный эффект AES устраняет заметные структурные признаки, и зашифрованный блок становится практически неотличим от белого шума. В этих условиях даже рассуждающая Qwen3 строила неинформативные гипотезы и в итоге давала случайные ответы.

Большие языковые модели используют токенизаторы, обученные в основном на естественных языках. Высокоэнтропийные строки дробятся на редкие или нестабильные токены, поэтому модель теряет привычные

статистические опоры и хуже сопоставляет такие последовательности между собой.

Эксперимент показал, что универсальные языковые модели могут выполнять базовый анализ простых шифров только при наличии механизмов рассуждения и достаточно явных статистических следов. Для современных блочных алгоритмов, выбранный подход тестирования оказался недостаточным.

3.3 Сравнительный анализ моделей

Для сопоставления результатов все полученные метрики были сведены в единую таблицу. Таблица 3.3 показывает точность моделей на трех типах шифрования и позволяет сравнить как абсолютные значения, так и изменение качества при росте сложности задачи.

Таблица 3.3 – Сводные метрики точности в задаче детекции зашифрованных текстов

Модель	Шифр Цезаря	Шифр Виженера	AES
Своя реализация	0.914	0.906	0.509
Qwen3-4B	0.840	0.820	0.440
Phi-4-mini	0.530	0.530	0.560
SmolLM3-3B	0.590	0.43	0.450

Полученные метрики показывают четкую иерархию. Специализированная модель, обученная только на целевых признаках, ожидаемо превосходит LLM на шифрах Цезаря и Виженера. Это объясняется тем, что ее веса были напрямую адаптированы к задаче отделения шифротекста от случайной последовательности.

Разница в 7–9% связана с тем, что модели обрабатывают данные принципиально по-разному. LLM используют косвенные текстовые эвристики и пытаются интерпретировать вход как токенизированный текст. Специализированный трансформер работает с символьными представлениями и извлекает признаки, необходимые именно для бинарной классификации.

Главная научная ценность эксперимента заключается не в констатации превосходства узкоспециализированной модели над языковыми, а в разборе логических сбоев, которые допускают универсальные генеративные сети при анализе криптографии. Детальное изучение выявило две системные проблемы.

Модели Phi-4-mini и SmolLM3-3B регулярно классифицировали истинно случайные последовательности как зашифрованный текст. Получив системную инструкцию искать скрытые сообщения, сеть попадает в ловушку предвзятости подтверждения. Сталкиваясь со случайной строкой, токенизатор LLM разбивает ее на бессмысленные фрагменты. Обнаружив чередование заглавных букв или цифру, которая характерна для неформальных кодировок, генеративная сеть делает ложный вывод о наличии сложной структуры. Модель пытается найти паттерн любой ценой, что приводит к критическому падению точности, вплоть до 0.43 у SmolLM3-3B на датасете Виженера.

Показательный сбой дала модель Qwen3-4B при анализе датасета AES. Поскольку данные стандарта AES не читаются как текст, перед передачей в промпт они кодировались в формат base64.

Поверхностное распознавание форматов не заменяет математический анализ. Именно из-за таких ложных логических цепочек точность Qwen3 на стандарте AES обрушилась до 0.44.

Сравнительный анализ доказывает, что генеративные модели без глубокого дообучения пригодны лишь на роль анализаторов простых замен. При столкновении с современной криптографией они генерируют ложные срабатывания, пытаясь анализировать структурный шум. Для задач сетевой безопасности требуется разработка и внедрение узкоспециализированных легковесных архитектур.

3.4 Оценка вычислительной эффективности и ресурсоемкости

Высокая точность классификации не является единственным критерием пригодности алгоритма для практического использования. Модуль детекции зашифрованного текста, интегрируемый в системы предотвращения вторжений или анализаторы сетевого трафика, работает в условиях жестких временных и аппаратных ограничений. Сетевые пакеты поступают непрерывным потоком, и превышение допустимой задержки обработки неизбежно приводит к потере пакетов и деградации пропускной способности сети.

Особую актуальность приобретает требование к локальному исполнению моделей. Использование облачных API для анализа перехваченного трафика недопустимо в контексте информационной безопасности из-за рисков компрометации чувствительных данных. В связи с этим в рамках исследования была проведена оценка ресурсоемкости архитектур, результаты которой объясняют объективные ограничения при их развертывании на локальных вычислительных узлах.

Разработанная узкоспециализированная модель обладает минимальными аппаратными требованиями, что позволяет свободно запускать ее на стандартном потребительском оборудовании. Энкодер на 3 слоя с размерностью эмбеддинга 256 содержит менее 10 миллионов обучаемых параметров.

При формате float32 веса разработанной модели занимают около 30 Мегабайт в оперативной или видеопамяти. Профилирование локального запуска показало, что для обработки одного батча дополнительно требуется менее 100 Мегабайт временной памяти. Поэтому модель можно запускать не только на серверном GPU, но и на CPU или встроенном графическом ускорителе.

Ресурсоемкость больших языковых моделей, напротив, затрудняет их локальное применение в режиме реального времени. По архитектурным спецификациям видно, что в половинной точности одна только загрузка весов

требует примерно 6–8 Гигабайт видеопамяти. При этом основной расход возникает не столько из-за весов, сколько из-за динамической памяти.

Передача пакетных данных через крупный системный промпт и использование окна генерации до 32 768 токенов заметно увеличивают объем KV-кэша. В пиковых прогонах, описанных в подразделах 3.2 и 3.3, потребление VRAM превышало 12–14 Гигабайт. Для локального стенда это делает непрерывный запуск таких моделей практически невозможным без агрессивного квантования INT4, которое, в свою очередь, меняет веса и может дополнительно снижать точность распознавания сложных шифров.

Фундаментальное различие в скорости обработки обусловлено самой парадигмой работы нейросетей. Специализированный трансформер-энкодер работает по принципу единого прямого прохода. Модель считывает массив из N слов, вычисляет матричные умножения, применяет усреднение и мгновенно выдает тензор логитов для всего пакета. Вычислительная сложность этой операции на уровне токенов параллельна, $O(1)$ по времени генерации.

Время инференса на локальном процессоре или базовой видеокарте составляет миллисекунды, обеспечивая пропускную способность на уровне десятков тысяч слов в секунду.

Языковые модели работают по принципу генерации. Они вычисляют вероятности и генерируют ответ строго последовательно, токен за токеном. Сложность генерации составляет $O(N)$ где N – длина ответа. Худшую пропускную способность продемонстрировала Qwen3-4B. Механизм скрытых рассуждений, обеспечивший модели победу в распознавании шифров подстановки, требует огромных вычислительных затрат: на генерацию внутренних размышлений для одного пакета слов уходят десятки секунд. В масштабах гигабитных сетевых каналов такие задержки абсолютно неприемлемы.

3.5 Анализ результатов

Комплексный анализ экспериментальных данных позволяет сформулировать окончательный вывод исследования. Использование больших языковых моделей в режиме zero-shot для низкоуровневой фильтрации криптографического трафика нецелесообразно по трем причинам:

1) LLM не способны преодолеть лавинный эффект современных стандартов блочного шифрования, демонстрируя точность на уровне случайного угадывания $\approx 50\%$,

2) универсальные токенизаторы провоцируют ложноположительные срабатывания при анализе шума и форматов кодирования,

3) авторегрессионная природа и огромные требования к памяти исключают возможность их локального применения в системах реального времени.

В то же время, гипотеза об эффективности узконаправленного глубокого обучения полностью подтвердилась.

Разработанная легковесная архитектура на базе трансформера, работающая на символьном уровне, гарантирует сверхточную детекцию исторических шифров (свыше 90%), обрабатывает данные за миллисекунды и требует минимальных вычислительных мощностей.

ЗАКЛЮЧЕНИЕ

В ходе выполнения магистерской диссертации была решена актуальная научно-практическая задача разработки и сравнительного анализа методов обнаружения зашифрованных данных с применением технологий глубокого обучения.

На основе проведенного исследования сформулированы следующие основные выводы и результаты:

— Проведен системный анализ предметной области. Рассмотрены статистические свойства различных классов шифрования. От классических моноалфавитных и полиалфавитных подстановок до современных блочных алгоритмов. Обоснована неэффективность классических сигнатурных методов при работе с трафиком,

— Спроектирована и реализована модель классификатора. На базе фреймворка PyTorch разработана легковесная архитектура, которая использует 3 слоя многоголового внимания, обучаемые позиционные эмбеддинги и механизм усреднения признаков для бинарной классификации последовательностей на уровне символов,

— Разработана методология тестирования LLM. Сформирован унифицированный программный протокол, позволяющий оценивать эмерджентные способности современных генеративных сетей без предварительного дообучения,

— Проведена серия вычислительных экспериментов. Доказано, что кастомный трансформер обладает отличной обобщающей способностью при анализе шифре Цезаря и Виженера. Большие языковые модели показали приемлемый результат на простых шифрах, но на более сложном алгоритме шифрования результат прогнозирования близок к случайному,

— Выявлены ограничения нейросетевого криптоанализа. Экспериментально подтверждено, что ни специализированный трансформер, ни миллиардные LLM не способны в режиме классификации преодолеть статистическую защиту стандарта AES, демонстрируя точность распознавания на уровне случайного шум,

— Обоснована архитектурная эффективность. Анализ ресурсоемкости доказал, что генерация и высокое потребление VRAM исключают применение LLM для потокового анализа трафика. Кастомная модель требует менее 300 МБ памяти и обеспечивает пропускную способность порядка десятков тысяч слов в секунду.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Christodoulou P., Zizzo G., Maffeis S. MED: Exploring LLM Memorization of Encrypted Data // arXiv preprint. – 2024.
2. Yuan K. [и др.]. A Block Cipher Recognition Scheme Based on Deep Learning Transformer Algorithm // IEEE Access. – 2024.
3. Massachusetts Institute of Technology. Wordlist-10000 [Электронный ресурс]. URL: https://www.mit.edu/~ecprice/wordlist.10000 (дата обращения: 19.02.2026).
4. Kim S. [и др.]. Comprehensive Neural Cryptanalysis on Block Ciphers Using Different Encryption Methods // Mathematics. – 2024. – Т. 12. – № 13. – С. 1936.
5. Al-Qurishi M. A. A. [и др.]. Deep Learning-Based Cryptanalysis of a Simplified AES Cipher // IEEE Access. – 2022.
6. Zhang Y. [и др.]. Lightweight block cipher algorithm identification scheme based on transformer // Journal of Information Security and Applications. – 2026.
7. Sinkov A. Elementary Cryptanalysis: A Mathematical Approach. – Mathematical Association of America, 1966. – 222 с.
8. Shannon C. E. Communication Theory of Secrecy Systems // Bell System Technical Journal. – 1949. – Т. 28, № 4. – С. 656-715.
9. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. – 2nd ed. – John Wiley & Sons, 1996. – 758 с.
10. Stamp M. Information Security: Principles and Practice. – 2nd ed. – Wiley, 2011. – 448 с.
11. Kasiski F. W. Die Geheimschriften und die Dechiffrir-Kunst. – E. S. Mittler und Sohn, 1863. – 95 с.
12. Friedman W. F. The Index of Coincidence and Its Applications in Cryptography. – Riverbank Laboratories, 1922. – 87 с.
13. Смарт Н. Криптография. – М.: Техносфера, 2005. – 528 с.

14. Баричев С. Г., Гончаров В. В., Серов Р. Е. Основы современной криптографии. – М.: Горячая линия - Телеком, 2002. – 175 с.

15. Stallings W. Cryptography and Network Security: Principles and Practice. – 7th ed. – Pearson, 2017. – 768 с.

16. Рябко Б. Я., Фионов А. Н. Основы современной криптографии для специалистов в информационных технологиях. – М.: Научный мир, 2004. – 173 с.

17. National Institute of Standards and Technology. Advanced Encryption Standard (AES) // FIPS PUB 197. – 2001.

18. Daemen J., Rijmen V. The Design of Rijndael: AES - The Advanced Encryption Standard. – Springer-Verlag, 2002. – 238 с.

19. Nandi A. [и др.]. Machine Learning for Cryptography: A Review // IEEE Access. – 2019. – Т. 7. – С. 143110-143126.

20. Abadi M., Andersen D. G. Learning to Protect Communications with Adversarial Neural Cryptography // International Conference on Learning Representations (ICLR). – 2017.

21. Soelistyo C. [и др.]. Deep Learning for Cryptography and Cryptanalysis: A Review // Cryptography. – 2021. – Т. 5. – № 3. – С. 21.

22. Vaswani A. [и др.]. Attention is All You Need // Advances in Neural Information Processing Systems (NeurIPS). – 2017. – Т. 30.

23. Zhao W. X. [и др.]. A Survey of Large Language Models // arXiv preprint. – 2023.

24. Wei J. [и др.]. Emergent Abilities of Large Language Models // Transactions on Machine Learning Research. – 2022.

25. Qwen 3 Alibaba – Обзор, тесты и сравнение с GPT-4o и Claude 3.7 Sonnet [Электронный ресурс]. (дата обращения: 11.03.2026).

26. Yang A. [и др.]. Qwen3 Technical Report. – 2025.

27. HuggingFaceTB. SmolLM3-3B [Электронный ресурс]. URL: https://huggingface.co/HuggingFaceTB/SmolLM3-3B (дата обращения: 12.03.2026).

28. Microsoft [и др.]. Phi-4-Mini Technical Report: Compact yet Powerful Multimodal Language Models via Mixture-of-LoRAs. – 2025.

29. Paszke A. [и др.]. PyTorch: An Imperative Style, High-Performance Deep Learning Library // Advances in Neural Information Processing Systems (NeurIPS). – 2019. – Т. 32.

30. Habr. LSTM [Электронный ресурс]. URL: https://habr.com/ru/companies/wunderfund/articles/331310/ (дата обращения 07.02.2026)

ПРИЛОЖЕНИЕ А

(обязательное)

Распознавание зашифрованных сообщений с помощью глубокого обучения

Листинг программы

Весь код представлен в репозитории GitHub: https://github.com/dshmelov/MW

Файл dataset.py

```
import json
import math
import random
import string
from typing import Any, Literal

import pandas as pd
import torch
import yaml
from torch.utils.data import Dataset

from ciphers.aes import encrypt as aes
from ciphers.caesar import encrypt as caesar
from ciphers.vigenere import encrypt as vigenere
from datasets.utils import CharTokenizer, generate_random_word, generate_aes_vocab

__all__ = ['EncryptedDataset']


class EncryptedDataset(Dataset):
    def __init__(
            self,
            root: str,
            cipher_config_path: str,
            mode: Literal['train', 'val'] = 'train',
            train_fraction: float = 0.8,
            frac: float = 1.0,
            seed: int = 42
    ):
        super(EncryptedDataset, self).__init__()

        random.seed(seed)

        self.root = root
        self.mode = mode
        self.train_fraction = train_fraction

        self.cipher_config = yaml.load(open(cipher_config_path), Loader=yaml.FullLoader)
        self.cipher = self.cipher_config.pop('type', 'caesar')
```

```
            self.data    =    pd.read_csv(root,    names=['word'],
na_filter=False).sample(frac=frac,
random_state=seed).reset_index(
                drop=True)

            # берем ближайшее значение степени двойки
            self.max_len                =                pow(2,
math.ceil(math.log2(self.data["word"].str.len().max())))
            self.vocab = generate_aes_vocab() if self.cipher ==
'aes' else string.ascii_lowercase
            self.tokenizer = CharTokenizer(max_len=self.max_len,
vocab=self.vocab)

            # формируем сэмплы данных
            self.samples = self.prepare_samples()
            pass

        def __len__(self):
            return len(self.samples)

        def __getitem__(self, item):
            inputs, targets = self.samples[item]
            return inputs, targets

        @staticmethod
        def collate_fn(batch: dict[str, Any]) -> dict[str, Any]:
            inputs = torch.stack([i for i, _ in batch], dim=0)
            targets = torch.cat([i for _, i in batch], dim=0)

            return inputs, targets

        def prepare_samples(self) -> pd.DataFrame:
            train_index       =       int(len(self.data)       *
self.train_fraction)
            match self.mode:
                case 'train':
                    samples                                      =
self.data.iloc[:train_index].reset_index(drop=True)
                    pass
                case 'val':
                    samples                                      =
self.data.iloc[train_index:].reset_index(drop=True)
                    pass
                case _:
                    raise NotImplementedError
```

```
            encrypted_samples = list()
            random_samples = list()
            for word in samples["word"].tolist():

                encrypted_word = None
                match self.cipher:
                    case 'caesar':
                        encrypted_word      =      caesar(word,
**self.cipher_config)
                    case 'vigenere.yaml':
                        encrypted_word     =     vigenere(word,
**self.cipher_config)
                    case 'aes':
                        outputs           =           aes(word,
**self.cipher_config)
                        encrypted_word                          =
json.loads(outputs)['ciphertext']
                        pass
                    case _:
                        raise NotImplementedError

                random_word  =  generate_random_word(len(word),
vocab=self.vocab, _is_aes=(self.cipher == 'aes'))


encrypted_samples.append((self.tokenizer.encode(encrypted_word),
torch.zeros(1, dtype=torch.long)))

random_samples.append((self.tokenizer.encode(random_word),
torch.ones(1, dtype=torch.long)))
                pass

            samples = encrypted_samples + random_samples
            random.shuffle(samples)

            return samples
```

Файл dataset/utils.py

```
import json
import math
import random
import string
from typing import Any, Literal

import pandas as pd
```

```
    import torch
    import yaml
    from torch.utils.data import Dataset

    from ciphers.aes import encrypt as aes
    from ciphers.caesar import encrypt as caesar
    from ciphers.vigenere import encrypt as vigenere
    from datasets.utils import CharTokenizer,
generate_random_word, generate_aes_vocab

    __all__ = ['EncryptedDataset']


    class EncryptedDataset(Dataset):
        def __init__(
                self,
                root: str,
                cipher_config_path: str,
                mode: Literal['train', 'val'] = 'train',
                train_fraction: float = 0.8,
                frac: float = 1.0,
                seed: int = 42
        ):
            super(EncryptedDataset, self).__init__()

            random.seed(seed)

            self.root = root
            self.mode = mode
            self.train_fraction = train_fraction

            self.cipher_config =
yaml.load(open(cipher_config_path), Loader=yaml.FullLoader)
            self.cipher = self.cipher_config.pop('type',
'caesar')

            self.data = pd.read_csv(root, names=['word'],
na_filter=False).sample(frac=frac,
random_state=seed).reset_index(
                drop=True)

            # берем ближайшее значение степени двойки
            self.max_len = pow(2,
math.ceil(math.log2(self.data["word"].str.len().max())))
            self.vocab = generate_aes_vocab() if self.cipher ==
'aes' else string.ascii_lowercase
```

```
            self.tokenizer = CharTokenizer(max_len=self.max_len,
vocab=self.vocab)

            # формируем сэмплы данных
            self.samples = self.prepare_samples()
            pass

        def __len__(self):
            return len(self.samples)

        def __getitem__(self, item):
            inputs, targets = self.samples[item]
            return inputs, targets

        @staticmethod
        def collate_fn(batch: dict[str, Any]) -> dict[str, Any]:
            inputs = torch.stack([i for i, _ in batch], dim=0)
            targets = torch.cat([i for _, i in batch], dim=0)

            return inputs, targets

        def prepare_samples(self) -> pd.DataFrame:
            train_index       =       int(len(self.data)       *
self.train_fraction)
            match self.mode:
                case 'train':
                    samples                                      =
self.data.iloc[:train_index].reset_index(drop=True)
                    pass
                case 'val':
                    samples                                      =
self.data.iloc[train_index:].reset_index(drop=True)
                    pass
                case _:
                    raise NotImplementedError

            encrypted_samples = list()
            random_samples = list()
            for word in samples["word"].tolist():

                encrypted_word = None
                match self.cipher:
                    case 'caesar':
                        encrypted_word      =      caesar(word,
**self.cipher_config)
                    case 'vigenere.yaml':
```

```
                        encrypted_word = vigenere(word,
**self.cipher_config)
                    case 'aes':
                        outputs = aes(word,
**self.cipher_config)
                        encrypted_word =
json.loads(outputs)['ciphertext']
                        pass
                    case _:
                        raise NotImplementedError

                random_word = generate_random_word(len(word),
vocab=self.vocab, _is_aes=(self.cipher == 'aes'))


encrypted_samples.append((self.tokenizer.encode(encrypted_word),
torch.zeros(1, dtype=torch.long)))

random_samples.append((self.tokenizer.encode(random_word),
torch.ones(1, dtype=torch.long)))
                pass

            samples = encrypted_samples + random_samples
            random.shuffle(samples)

            return samples
```

Файл caesar.py

```
    __all__ = ['encrypt', 'decrypt']


    def encrypt(text: str, shift: int = 10) -> str:
        """
        Функция шифрования Цезаря
(https://www.geeksforgeeks.org/ethical-hacking/caesar-cipher-in-
cryptography/)
        :param text: исходная последовательность символов
        :param shift: значение сдвига
        :return: зашифрованное сообщение
        """
        result = ""

        for i in range(len(text)):
            char = text[i]
```

```
            if char.isupper():
                result += chr((ord(char) + shift - 65) % 26 + 65)
            else:
                result += chr((ord(char) + shift - 97) % 26 + 97)

        return result


    def decrypt(text: str, shift: int) -> str:
        """
        Функция дешифрования Цезаря
(https://www.geeksforgeeks.org/ethical-hacking/caesar-cipher-in-
cryptography/)
        :param text: зашифрованный текст
        :param shift: значение сдвига
        :return: исходная последовательность символов
        """
        return encrypt(text, (26 - shift))
```

Файл vigenere.py

```
    __all__ = ['encrypt', 'decrypt']


    def generate_key(msg: str, key: str) -> str | list[str]:
        """
        Функция генерации ключа для шифра Виженера
(https://www.geeksforgeeks.org/dsa/vigenere-cipher/)
        :param msg: входная последовательность символов
        :param key: исходный ключ в виде строки
        :return: новый ключ для шифра Виженера
        """
        key = list(key)
        if len(msg) == len(key):
            return key
        else:
            for i in range(len(msg) - len(key)):
                key.append(key[i % len(key)])
        return "".join(key)


    def encrypt(msg: str, key: str) -> str:
        """
        Функция шифрования Виженера
(https://www.geeksforgeeks.org/dsa/vigenere-cipher/)
        :param msg: исходная последовательность символов
        :param key: ключ шифрования
```

```
        :return: зашифрованное сообщение
        """
        encrypted_text = []
        key = generate_key(msg, key)
        for i in range(len(msg)):
            char = msg[i]
            if char.isupper():
                encrypted_char = chr((ord(char) + ord(key[i]) -
2 * ord('A')) % 26 + ord('A'))
            elif char.islower():
                encrypted_char = chr((ord(char) + ord(key[i]) -
2 * ord('a')) % 26 + ord('a'))
            else:
                encrypted_char = char
            encrypted_text.append(encrypted_char)
        return "".join(encrypted_text)


    def decrypt(msg: str, key: str) -> str:
        """
        Функция                дешифрования                Виженера
(https://www.geeksforgeeks.org/dsa/vigenere-cipher/)
        :param msg: зашифрованное сообщение
        :param key: ключ шифрования
        :return: исходная последовательность символов
        """
        decrypted_text = []
        key = generate_key(msg, key)
        for i in range(len(msg)):
            char = msg[i]
            if char.isupper():
                decrypted_char = chr((ord(char) - ord(key[i]) +
26) % 26 + ord('A'))
            elif char.islower():
                decrypted_char = chr((ord(char) - ord(key[i]) +
26) % 26 + ord('a'))
            else:
                decrypted_char = char
            decrypted_text.append(decrypted_char)
        return "".join(decrypted_text)
```

Файл aes.py

```
    import json
    from base64 import b64encode, b64decode

    from Crypto.Cipher import AES
    from Crypto.Util.Padding import pad, unpad
```

```
    __all__ = ['generate_key', 'encrypt', 'decrypt']


    def generate_key(key: str) -> bytes:
        """
        Функция генерации ключа для шифра AES
(https://pycryptodome.readthedocs.io/en/latest/src/cipher/classi
c.html)
        :param key: исходный ключ в виде строки
        :return: новый ключ для шифра AES
        """
        return pad(key.encode('utf-8'), AES.block_size)


    def encrypt(data: str, key: str) -> str:
        """
        Функция шифрования AES
(https://pycryptodome.readthedocs.io/en/latest/src/cipher/classi
c.html)
        :param data: входная последовательность
        :param key: ключ шифрования
        :return: json-объект в виде строки
        """

        key = generate_key(key)

        # cipher = AES.new(key, AES.MODE_CBC)
        cipher = AES.new(key, AES.MODE_ECB)

        ct_bytes = cipher.encrypt(pad(data.encode('utf-8'),
AES.block_size))

        # iv = b64encode(cipher.iv).decode('utf-8')
        iv = b64encode(ct_bytes).decode('utf-8')
        ct = b64encode(ct_bytes).decode('utf-8')

        result = json.dumps({'iv': iv, 'ciphertext': ct})

        return result


    def decrypt(json_input: str, key: str) -> str | None:
        """
```

```
        Функция дешифрования AES
(https://pycryptodome.readthedocs.io/en/latest/src/cipher/classi
c.html)
        :param json_input: json-объект в виде строки
        :param key: ключ шифрования
        :return: исходная последовательность символов
        """
        key = generate_key(key)

        pt = None

        try:
            b64 = json.loads(json_input)
            iv = b64decode(b64['iv'])
            ct = b64decode(b64['ciphertext'])
            cipher = AES.new(key, AES.MODE_CBC, iv)
            pt          =          unpad(cipher.decrypt(ct),
AES.block_size).decode('utf-8')
        except (ValueError, KeyError):
            print('Ошибка при дешифровании')

        return pt
```

Файл model.py

```
import torch
import torch.nn as nn

__all__ = ["EncryptedModel"]

class EncryptedModel(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        max_len: int,
        d_model: int = 384,
        nhead: int = 3,
        num_layers: int = 3,
        dim_feedforward: int = 768,
        num_classes: int = 2,
        pad_id: int = 0,
        dropout: float = 0.1,
    ):
        super().__init__()

        self.pad_id = pad_id
```

```
            self.max_len = max_len

            self.token_embedding = nn.Embedding(
                num_embeddings=vocab_size,
                embedding_dim=d_model,
                padding_idx=pad_id,
            )

            self.pos_embedding = nn.Embedding(
                num_embeddings=max_len,
                embedding_dim=d_model,
            )

            encoder_layer = nn.TransformerEncoderLayer(
                d_model=d_model,
                nhead=nhead,
                dim_feedforward=dim_feedforward,
                dropout=dropout,
                batch_first=True,
                activation="gelu",
            )

            self.encoder = nn.TransformerEncoder(
                encoder_layer,
                num_layers=num_layers,
                enable_nested_tensor=False
            )

            self.norm = nn.LayerNorm(d_model)
            self.classifier = nn.Linear(d_model, num_classes)

        def forward(self, x: torch.Tensor) -> torch.Tensor:
            """
            x: [B, L]  token ids
            """
            if x.dtype != torch.long:
                x = x.long()

            batch_size, seq_len = x.shape

            if seq_len > self.max_len:
                x = x[:, : self.max_len]
                seq_len = self.max_len

            positions          =          torch.arange(seq_len,
device=x.device).unsqueeze(0).expand(batch_size, seq_len)
```

```
            token_emb = self.token_embedding(x)          # [B, L,
D]
            pos_emb = self.pos_embedding(positions)      # [B, L,
D]
            h = token_emb + pos_emb

            padding_mask = (x == self.pad_id)              # [B,
L], True = ignore

            h                  =                  self.encoder(h,
src_key_padding_mask=padding_mask)  # [B, L, D]

            # mean pooling по непустым токенам
            mask = (~padding_mask).unsqueeze(-1).float()   # [B,
L, 1]
            h      =      (h      *      mask).sum(dim=1)      /
mask.sum(dim=1).clamp(min=1.0)

            h = self.norm(h)
            logits = self.classifier(h)                    # [B,
num_classes]
            return logits
```

Файл train.py

```
import argparse
import os
import random
import sys
from types import NoneType
from typing import Any, TextIO

import torch
import torch.nn as nn
import yaml
from dotenv import load_dotenv
from torch.optim import Optimizer
from torch.optim.lr_scheduler import LRScheduler
from torch.utils.data import DataLoader
from torchmetrics import MetricCollection
from tqdm.auto import tqdm

from datasets.dataset import EncryptedDataset
from eval import evaluate
from models.model import EncryptedModel
```

```
    from utils import *


    def _init_args_() -> dict[str, Any]:
        parser = argparse.ArgumentParser()

        parser.add_argument('-cfg',  '--config_path',  type=str,
required=True, help='путь до конфига обучения модели')

        args = parser.parse_args()

        return vars(args)


    def train_one_epoch(
            model: nn.Module,
            train_loader: DataLoader,
            optimizer: Optimizer,
            criterion: nn.Module,
            scheduler: LRScheduler,
            buffer: TextIO = sys.stdout,
    ) -> float:
        model.train()

        train_epoch_loss: float = 0.0
        batch_counter: int = 0

        pbar     =     tqdm(train_loader,     dynamic_ncols=True,
file=buffer)

        for batch in pbar:
            tokens, labels = batch

            tokens = tokens.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(tokens)

            train_loss = criterion(outputs, labels)

            train_loss.backward()

            optimizer.step()
            scheduler.step()
```

```
            last_lr = scheduler.get_last_lr()[0]
            pbar.set_description(f"Training                loss:
{train_loss.item():.6} | LR: {last_lr:.6}")

            train_epoch_loss += train_loss.item()
            batch_counter += 1
            pass

        train_epoch_loss /= batch_counter

        return train_epoch_loss


    def train(
            num_epochs: int,
            model: nn.Module,
            train_loader: DataLoader,
            val_loader: DataLoader,
            optimizer: Optimizer,
            criterion: nn.Module,
            scheduler: LRScheduler,
            metrics: MetricCollection,
            key_metric: str = 'Accuracy',
            checkpoint_dir: str = './checkpoints/',
            buffer: TextIO = sys.stdout,
    ) -> None:
        pbar = tqdm(disable=True, file=buffer)

        t_key_metric: float = -1.0

        for epoch in range(num_epochs):
            train_loss = train_one_epoch(
                model=model,
                train_loader=train_loader,
                optimizer=optimizer,
                criterion=criterion,
                scheduler=scheduler,
                buffer=buffer,
            )
            pbar.write(f"Epoch   {epoch   +   1}/{num_epochs}   -
Training loss: {train_loss:.6}")

            val_loss, metric_values = evaluate(
                model=model,
                loader=val_loader,
```

```
                criterion=criterion,
                metrics=metrics,
                device=device,
                mode='val'
            )

            pbar.write(f"Epoch {epoch + 1}/{num_epochs} -
Validation loss: {train_loss:.6}")

            log_metrics(metric_values, pbar=pbar, buffer=buffer)

            save_best_model_checkpoint(
                model=model,
                optimizer=optimizer,
                loss=train_loss,
                metric_values=metric_values,
                key_metric=key_metric,
                key_metric_value=t_key_metric,
                checkpoint_dir=checkpoint_dir,
                epoch=epoch,
            )


    def main() -> None:
        args = _init_args_()

        config_path = args.pop('config_path', None)
        if isinstance(config_path, NoneType):
            raise ValueError('Не был передан конфигурационный
файл!')
        config = yaml.load(open(config_path, 'r'),
Loader=yaml.FullLoader)

        num_epochs = config.pop('num_epochs', 10)

        dataset_config = config.pop('dataset', None)
        if isinstance(dataset_config, NoneType):
            raise ValueError('Не было найдено настройки набора
данных!')

        train_dataset = EncryptedDataset(mode='train',
**dataset_config)
        val_dataset = EncryptedDataset(mode='val',
**dataset_config)

        loader_config = config.pop('loader', None)
```

```
        if isinstance(loader_config, NoneType):
            raise   ValueError('Не   было   найдено   настройки
загрузчика данных!')

        train_loader     =     DataLoader(dataset=train_dataset,
shuffle=True,                collate_fn=train_dataset.collate_fn,
**loader_config)
        val_loader       =       DataLoader(dataset=val_dataset,
shuffle=False,                collate_fn=val_dataset.collate_fn,
**loader_config)

        model_config = config.pop('model', None)
        if isinstance(model_config, NoneType):
            raise ValueError('Не было найдено настройки модели!')

        vocab_size = len(train_dataset.tokenizer.stoi)
        max_len = train_dataset.max_len
        model      =      EncryptedModel(vocab_size=vocab_size,
max_len=max_len, **model_config).to(device)

        optimizer_config = config.pop('optimizer', None)
        if isinstance(optimizer_config, NoneType):
            raise   ValueError('Не   было   найдено   настройки
оптимизатора!')
        optimizer = prepare_optimizer(optimizer_config, model)

        scheduler_config = config.pop('scheduler', None)
        if isinstance(scheduler_config, NoneType):
            raise   ValueError('Не   было   найдено   настройки
планировщика скорости обучения!')
        scheduler = prepare_scheduler(
            scheduler_config,
            optimizer=optimizer,
            train_loader=train_loader,
            epochs=num_epochs,
            lr=optimizer.param_groups[0]['lr']
        )

        criterion_config = config.pop('criterion', None)
        if isinstance(criterion_config, NoneType):
            raise ValueError('Не было найдено настройки функции
ошибки!')
        criterion                                              =
prepare_criterion(criterion_config).to(device)
```

```
        metrics_config = config.pop('metrics',
[{'BinaryAccuracy': dict()}])
        metrics = prepare_metrics(metrics_config)

        key_metric = config.pop('key_metric', 'Accuracy')
        if key_metric not in metrics.keys():
            raise ValueError('Ключевая метрика не была найдена в
списке метрик!')

        checkpoint_dir = config.pop('checkpoint_dir',
'./checkpoints/')

        train(
            num_epochs=num_epochs,
            model=model,
            train_loader=train_loader,
            val_loader=val_loader,
            optimizer=optimizer,
            criterion=criterion,
            scheduler=scheduler,
            metrics=metrics,
            key_metric=key_metric,
            checkpoint_dir=checkpoint_dir
        )

        pass


    if __name__ == '__main__':
        load_dotenv()

        device = torch.device('mps' if
torch.backends.mps.is_available() else 'cpu')

        torch.manual_seed(int(os.environ['SEED']))
        random.seed((os.environ['SEED']))

        main()
```

Файл eval.py

```
    import sys
    from typing import Literal, Any, TextIO

    import torch
    import torch.nn as nn
```

```
    from torch.utils.data import DataLoader
    from torchmetrics import MetricCollection
    from tqdm.auto import tqdm


    def evaluate(
            model: nn.Module,
            loader: DataLoader,
            criterion: nn.Module,
            metrics: MetricCollection,
            device: torch.device,
            mode: Literal['val', 'test'] = 'val',
            buffer: TextIO = sys.stdout,
    ) -> tuple[float, dict[str, Any]]:
        model.eval()

        metrics.reset()
        metrics.to(device)

        epoch_loss: float = 0.0
        batch_counter: int = 0

        pbar = tqdm(loader, dynamic_ncols=True, file=buffer)

        for batch in pbar:
            tokens, labels = batch

            tokens = tokens.to(device)
            labels = labels.to(device)

            outputs = model(tokens)

            loss = criterion(outputs, labels)

            preds = torch.argmax(outputs, dim=1)

            metrics.update(preds, labels)

            pbar.set_description(f"{mode.capitalize()}     loss:
{loss.item():.6}")

            epoch_loss += loss.item()
            batch_counter += 1

            pass
```

```
        epoch_loss /= batch_counter

        return epoch_loss, metrics.compute()


    def main() -> None:
        pass


    if __name__ == '__main__':
        main()
```

Файл references/utils.py

```
    import os
    import sys
    from types import NoneType
    from typing import Any, TextIO

    import torch
    import torch.nn as nn
    from torch.optim import Optimizer
    from torch.optim.lr_scheduler import CosineAnnealingLR
    from torch.utils.data import DataLoader
    from torchmetrics import MetricCollection
    from torchmetrics.classification import BinaryAccuracy
    from tqdm.auto import tqdm

    __all__ = [
        'prepare_optimizer',
        'prepare_scheduler',
        'prepare_criterion',
        'prepare_metrics',
        'log_metrics',
        'save_model_checkpoint',
        'save_best_model_checkpoint'
    ]


    def prepare_optimizer(config: dict[str, Any], model:
nn.Module) -> Optimizer:
        optimizer_type = config.pop('type', None)
        if isinstance(optimizer_type, NoneType):
            raise ValueError('Не был определен оптимизатор для
обучения!')

        optimizer: Optimizer = None
```

```
        match optimizer_type:
            case 'Adam':
                optimizer = torch.optim.Adam(model.parameters(),
**config)
            case 'AdamW':
                optimizer                                          =
torch.optim.AdamW(model.parameters(), **config)
            case 'SGD':
                optimizer  =  torch.optim.SGD(model.parameters(),
**config)
            case _:
                raise NotImplementedError

        return optimizer


    def  prepare_scheduler(config:  dict[str,  Any],  optimizer:
Optimizer, train_loader: DataLoader, epochs: int,
                           lr: float) -> Optimizer:
        scheduler_type = config.pop('type', None)
        if isinstance(scheduler_type, NoneType):
            raise ValueError('Не был определен оптимизатор для
обучения!')

        scheduler: Optimizer = None
        match scheduler_type:
            case 'CosineAnnealingLR':
                scheduler = CosineAnnealingLR(optimizer, epochs
* len(train_loader), eta_min=lr / 25e4)
            case _:
                raise NotImplementedError

        return scheduler


    def prepare_criterion(config: dict[str, Any]) -> nn.Module:
        criterion_type = config.pop('type', None)
        if isinstance(criterion_type, NoneType):
            raise   ValueError('Не   была   определена   функция
ошибки!')

        criterion: nn.Module = None
        match criterion_type:
            case 'CrossEntropyLoss':
                criterion = nn.CrossEntropyLoss(**config)
            case 'BCELoss':
```

```
                criterion = nn.BCELoss(**config)
            case _:
                raise NotImplementedError

        return criterion


    def prepare_metrics(config: list[dict[str, Any]]) ->
MetricCollection:
        metrics = list()
        metric = None
        for cfg in config:
            for k, v in cfg.items():
                match k:
                    case 'BinaryAccuracy':
                        metric = BinaryAccuracy(**v)
                    case _:
                        raise NotImplementedError

            metrics.append(metric)

        return MetricCollection(metrics)


    def log_metrics(metric_values: dict[str, Any], pbar: tqdm =
None, buffer: TextIO = sys.stdout) -> None:
        if isinstance(pbar, NoneType):
            pbar = tqdm(disable=True, file=buffer)
            pass
        for k, v in metric_values.items():
            match k:
                case 'BinaryAccuracy':
                    pbar.write(f'Acc@1 = {v.item():.6}')
                case _:
                    raise NotImplementedError
        pass


    def save_model_checkpoint(
            model: nn.Module,
            optimizer: Optimizer,
            loss: float,
            metric_values: dict[str, Any],
            epoch: int,
            checkpoint_path: str,
    ) -> None:
```

```
        checkpoint = dict({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            'metrics': metric_values,
        })

        torch.save(checkpoint, checkpoint_path)
        pass


    def save_best_model_checkpoint(
            model: nn.Module,
            optimizer: Optimizer,
            loss: float,
            metric_values: dict[str, Any],
            key_metric: str,
            key_metric_value: float,
            epoch: int,
            checkpoint_dir: str,
    ) -> None:
        for k, v in metric_values.items():
            if k == key_metric:
                if v.item() > key_metric_value:
                    save_model_checkpoint(
                        model=model,
                        optimizer=optimizer,
                        loss=loss,
                        metric_values=metric_values,
                        epoch=epoch,

checkpoint_path=os.path.join(checkpoint_dir, 'model-best.pth')
                    )
```

Файл models/Phi_4_mini_instruct.py

```
"""
https://huggingface.co/microsoft/Phi-4-mini-instruct
"""
import torch
from transformers import pipeline

from .base import LLM

__all__ = ["Phi_4_mini_instruct"]


class Phi_4_mini_instruct(LLM):
    def __init__(self, model_name: str = 'microsoft/Phi-4-mini-instruct', device: str = 'cuda',
                 cache_dir: str = './.cache'):
        super(Phi_4_mini_instruct, self).__init__(model_name, device, cache_dir)

    @torch.no_grad()
    def inference(
            self,
            prompt: str,
            max_new_tokens: int = 32768,
    ) -> str:
        pipe = pipeline(task='text-generation', model=self.model, tokenizer=self.tokenizer)

        messages = [{
            "role": "user",
            "content": prompt
        }]

        generation_args = {
            "max_new_tokens": max_new_tokens,
            "return_full_text": False,
            "temperature": 0.6,
            "top_p": 0.95,
            "do_sample": False,
        }

        output = pipe(messages, **generation_args)

        return output[0]['generated_text']
```

Файл models/Qwen3_4B_Thinking_2507.py

```
"""
https://huggingface.co/Qwen/Qwen3-4B-Thinking-2507
"""
import torch

from .base import LLM

__all__ = ["Qwen3_4B_Thinking_2507"]


class Qwen3_4B_Thinking_2507(LLM):
    def __init__(self, model_name: str = 'Qwen/Qwen3-4B-Thinking-2507', device: str = 'cuda',
                 cache_dir: str = './.cache'):
        super(Qwen3_4B_Thinking_2507, self).__init__(model_name, device, cache_dir)
        pass

    @torch.no_grad()
    def inference(
            self,
            prompt: str,
            max_new_tokens: int = 32768,
    ) -> str:
        messages = [
            {"role": "user", "content": prompt}
        ]
        text = self.tokenizer.apply_chat_template(
            messages,
            tokenize=False,
            add_generation_prompt=True,
        )
        model_inputs = self.tokenizer([text], return_tensors="pt").to(self.device)

        generated_ids = self.model.generate(
            **model_inputs,
            max_new_tokens=max_new_tokens,
            temperature=0.6,
            top_p=0.95,
        )
        output_ids = generated_ids[0][len(model_inputs.input_ids[0]):].tolist()

        try:
```

```
                index = len(output_ids) - output_ids[::-
1].index(151668)
            except ValueError:
                index = 0

            content = self.tokenizer.decode(output_ids[index:],
skip_special_tokens=True).strip("\n")

            return content
```

Файл models/SmolLM3_3B.py

```
    """
    https://huggingface.co/HuggingFaceTB/SmolLM3-3B
    """
    import torch

    from .base import LLM

    __all__ = ["SmolLM3_3B"]


    class SmolLM3_3B(LLM):
        def __init__(self, model_name: str =
'HuggingFaceTB/SmolLM3-3B', device: str = 'cuda', cache_dir: str
= './.cache'):
            super(SmolLM3_3B, self).__init__(model_name, device,
cache_dir)
            pass

        @torch.no_grad()
        def inference(
                self,
                prompt: str,
                max_new_tokens: int = 32768,
        ) -> str:
            messages = [{
                "role": "user",
                "content": prompt
            }]

            text = self.tokenizer.apply_chat_template(
                messages,
                tokenize=False,
                add_generation_prompt=True,
                enable_thinking=False,
```

```
            )
            inputs           =           self.tokenizer([text],
return_tensors="pt").to(self.device)

            generated_ids     =     self.model.generate(**inputs,
max_new_tokens=max_new_tokens,
                                                  temperature=0.6,
                                                  top_p=0.95)

            output_ids                                          =
generated_ids[0][len(inputs["input_ids"][0]):]

            return             self.tokenizer.decode(output_ids,
skip_special_tokens=True)
```

Файл models/base.py

```
    import torch
    from transformers import AutoTokenizer, AutoModelForCausalLM

    __all__ = ["LLM"]
    class LLM:
        def __init__(self, model_name: str, device: str = 'cuda',
cache_dir: str = './.cache'):
            self.model_name = model_name
            self.device = device
            self.cache_dir = cache_dir

            self.tokenizer                                      =
AutoTokenizer.from_pretrained(self.model_name)
            self.model = AutoModelForCausalLM.from_pretrained(
                self.model_name,
                cache_dir=self.cache_dir,
            ).to(self.device)
            pass
        @torch.no_grad()
        def inference(
                self,
                prompt: str,
                max_new_tokens: int = 32768,
        ) -> str:
            raise  NotImplementedError('Дочерний  класс  обязан
определить этот метод')
```

Файл llm/utils.py

```
    import random
    from typing import Any, Callable
```

```
    import pandas as pd

    from datasets import *
    from models import *

    __all__ = ['prepare_dataset', 'prepare_model',
'prepare_prompt']


    def prepare_dataset(dataset_path: str) -> Any:
        dataset_name = dataset_path.split('/')[-2]
        cipher_name = dataset_name.split('_')[0]

        if cipher_name.lower().startswith('caesar'):
            shift = int(dataset_name.split('_')[-1])
            dataset = CaesarDataset(root=dataset_path,
shift=shift)
            pass
        elif cipher_name.lower().startswith('vigenere'):
            key = dataset_name.split('_')[-1]
            dataset = VigenereDataset(root=dataset_path,
key=key)
            pass
        elif cipher_name.lower().startswith('aes'):
            key = dataset_name.split('_')[-1]
            dataset = AESDataset(root=dataset_path, key=key)
            pass
        else:
            raise Exception('Неподдерживаемый алгоритм
шифрования')

        return dataset


    def prepare_model(model_name: str) -> LLM | None:
        if model_name.lower().startswith('phi'):
            model =
Phi_4_mini_instruct(model_name='/raid/USERS/shmelyov/repos/MW/.c
ache/microsoft--Phi-4-mini-instruct',
                                    device='cuda:0')
        elif model_name.lower().startswith('qwen'):
            model = Qwen3_4B_Thinking_2507(

model_name='/raid/USERS/shmelyov/repos/MW/.cache/models--Qwen--
```

```
Qwen3-4B-Thinking-
2507/snapshots/768f209d9ea81521153ed38c47d515654e938aea',
                device='cuda:0')
        elif model_name.lower().startswith('smol'):
            model = SmolLM3_3B(

model_name='/raid/USERS/shmelyov/repos/MW/.cache/models--
HuggingFaceTB--SmolLM3-
3B/snapshots/a07cc9a04f16550a088caea529712d1d335b0ac1',
                device='cuda:0')
        else:
            raise      Exception(f'Неподдерживаемая      модель:
{model_name}')
        return model


    def   prepare_prompt(dataset:   Callable,   prompt:   str,
num_samples: int) -> str:
        sample_indexes                                          =
[random.choice(dataset.data.columns[1:])      for      _      in
range(num_samples)]

        num_cipher_text   =   len([sample   for   sample   in
sample_indexes if sample == 'cipher_text'])

        prompt += '\n```\n'
        for sample_index in sample_indexes:

            i = random.randint(0, len(dataset) - 1)

            if isinstance(dataset, AESDataset):
                src_text, input_vector, cipher_text, random_text
= dataset[i]
            else:
                src_text, cipher_text, random_text = dataset[i]

            if sample_index.startswith('cipher'):
                prompt += f'{cipher_text}\n'
            elif sample_index.startswith('random'):
                prompt += f'{random_text}\n'

            pass

        prompt += '```'

        return prompt
```

```
    def calculate_accuracy(dataset: Callable, outputs:
pd.DataFrame, ) -> float:
        tp = 0
        sample_indexes = [random.choice(dataset.data.columns[-
2:]) for _ in range(len(outputs))]

        for j, output in outputs.iterrows():
            sample_index = sample_indexes[j]

            i = random.randint(0, len(dataset) - 1)

            if isinstance(dataset, AESDataset):
                src_text, input_vector, cipher_text, random_text
= dataset[i]
            else:
                src_text, cipher_text, random_text = dataset[i]

            if sample_index.startswith('cipher'):
                assert cipher_text == output['text']
                if output['class'] == 0:
                    tp += 1
            elif sample_index.startswith('random'):
                assert random_text == output['text']
                if output['class'] == 1:
                    tp += 1

        acc = tp / len(outputs)
        return acc

    if __name__ == '__main__':
        random.seed(42)
        outputs                                                =
pd.read_csv('/Users/dan/Desktop/repos/psu/MW/experiments/exp_aes
_phi4_mini_instruct_4b/output.txt', sep=' - ', names=['text',
'class'])
        dataset                                                =
prepare_dataset(dataset_path='/Users/dan/Desktop/repos/psu/MW/da
ta/aes_9999_key/data.csv')
        acc = calculate_accuracy(dataset, outputs)

        print(f'Accuracy: {acc}')
```

Файл llm/main.py

```
    import argparse
    import os
```

```
    import random
    from typing import Any

    from models import *
    from utils import *


    def _init_args_() -> dict[str, Any]:
        parser = argparse.ArgumentParser()

        parser.add_argument('-d',     '--dataset',     type=str,
required=True, help='dataset path')
        parser.add_argument('-m',      '--model',      type=str,
required=True, help='model name')
        parser.add_argument('-n',   '--num_samples',   type=int,
default=100, help='number of samples')

        args = parser.parse_args()

        return vars(args)


    def run_exp(
            model: LLM,
            prompt: str,
            exp_dir: str,
            cipher_name: str,
            model_name: str,
    ) -> None:
        os.makedirs(exp_dir, exist_ok=True)

        output         =         model.inference(prompt=prompt,
max_new_tokens=32768)

        exp_local_dir          =          os.path.join(exp_dir,
f'exp_{cipher_name}_{model_name}')
        os.makedirs(exp_local_dir, exist_ok=True)
        with   open(os.path.join(exp_local_dir,   'output.txt'),
'w') as fp:
            fp.write(output)


    def main() -> None:
        args = _init_args_()

        dataset_path = args['dataset']
```

```
        model_name = args['model']
        num_samples = args['num_samples']

        dataset = prepare_dataset(dataset_path)
        # model = prepare_model(model_name)

        cipher_name = dataset_path.split('/')[-2].split('_')[0]

        prompt = (
            f"Classify the following {num_samples} words as
either an encrypted message (class 0) or random (class 1)"
            f"Encryption algorithm: {cipher_name} cipher."
            "Output the word followed by the classification label
(0 or 1) without any additional explanation"
            "You must analyze each input word and give answer in
the following format:"
            "word_1 - 0"
            "word_2 - 1"
            "..."
            "word_100 - 0"
        )

        import torchvision.transforms as T
        T.ColorJitter

        prompt = prepare_prompt(dataset, prompt, num_samples)

        exp_dir = './experiments/'
        run_exp(model=model, prompt=prompt, exp_dir=exp_dir,
                model_name=model_name, cipher_name=cipher_name)


    if __name__ == '__main__':
        random.seed(42)
        main()
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Распознавание зашифрованных сообщений с помощью глубокого обучения
Презентация

РАСПОЗНАВАНИЕ ЗАШИФРОВАННЫХ СООБЩЕНИЙ С ПОМОЩЬЮ ГЛУБОКОГО ОБУЧЕНИЯ

Студент: **Шмелев Д.В.**
Научный руководитель: **д.т.н., профессор Митрохин М.А.**

Пенза, 2026

Цель и задачи исследования

Цель

Исследование способности моделей глубокого обучения различать артефакты шифрования и случайный шум.

Задачи

- Анализ предметной области;
- Разработка методики тестирования;
- Формирование обучающей и валидационной выборки данных;
- Выбор алгоритмов шифрования;
- Выбор LLM;
- Обучение модели классификатора;
- Сравнительный анализ результатов тестирования.

Стек технологий

PyTorch

pandas

NumPy

NVIDIA CUDA

Выбор алгоритмов шифрования

Алгоритм	Тип	Сложность признаков	Стойкость
Шифр Цезаря	Моноалфавитный	Статистика сохраняется	Низкая
Шифр Виженера	Полиалфавитный	Циклические паттерны	Средняя
AES	Блочный	Лавинный эффект	Высокая

Формирование наборов данных

Общий объем: 30,000 сэмплов

Выбор LLM

Конфигурация среды тестирования:

- **Режим:** Zero-Shot
- **Контекстное окно:** 32 768
- **Значение temperature:** 0.6
- **Значение top_p:** 0.95

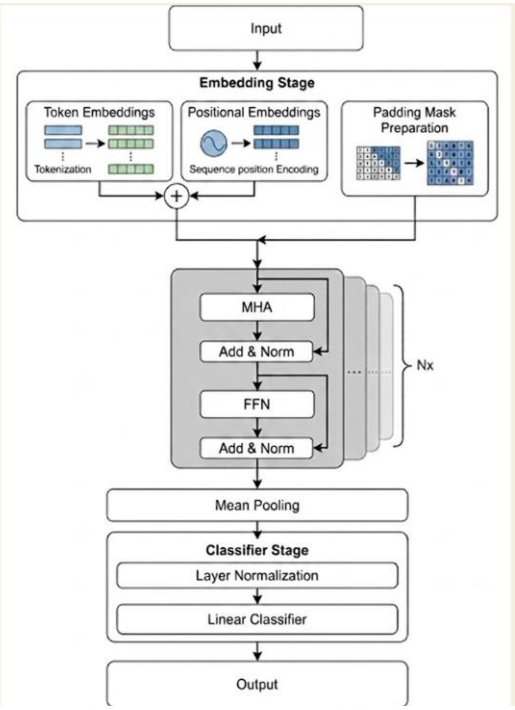
Qwen3

Phi-4

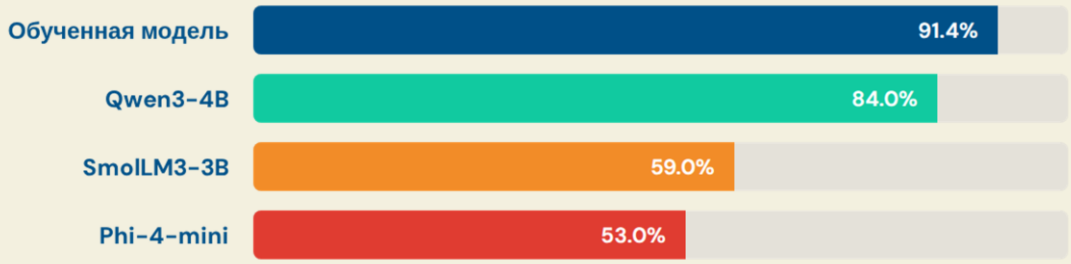
Архитектура модели

Спецификации классификатора:

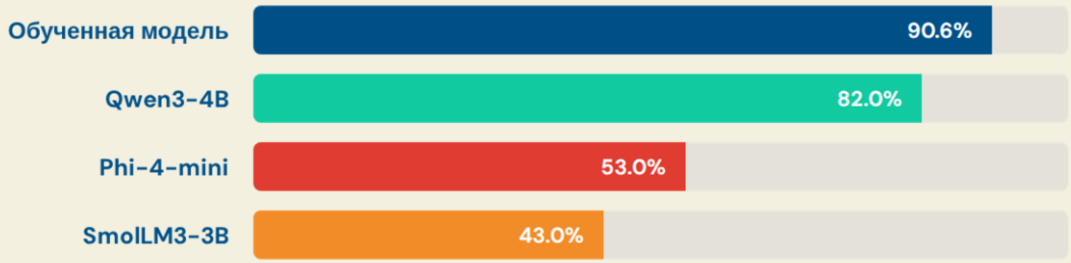
- **Архитектура:** Энкодер трансформера.
- **Параметры:** ~10 млн обучаемых параметров.
- **Количество голов внимания:** 8.
- **Активация:** GELU.



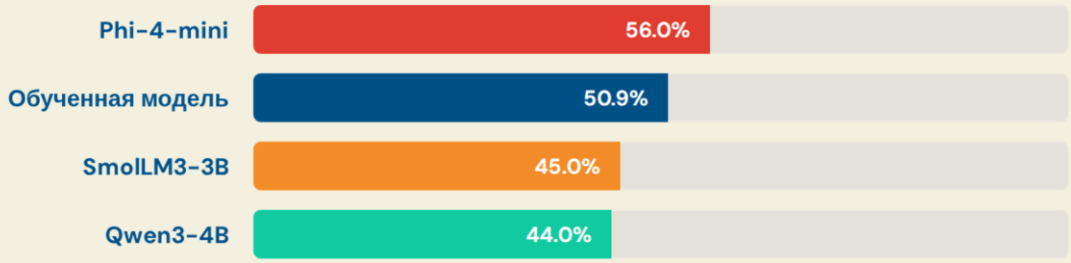
Точность классификации (Цезарь)



Точность классификации (Виженер)



Точность классификации (AES)



Результаты исследования

- **Разработан эффективный классификатор.** Обученная модель демонстрирует высокую точность на исторических шифрах, требуя минимальных аппаратных ресурсов.
- **Скрытые способности LLM.** Доказано наличие у генеративных сетей способности к анализу простых шифров подстановки.
- **Подтверждение стойкости AES.** Экспериментально доказано, что лавинный эффект AES делает данные неуязвимыми для детекции современными нейросетевыми методами.

СПАСИБО ЗА ВНИМАНИЕ!