Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

(повне найменування вищого навчального закладу)

Факультет комп’ютерно-інформаційних систем і програмної інженерії

(назва факультету)

Кафедра кібербезпеки

(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр

(освітній рівень)

на тему: "Генеративні мовні моделі в аналізі шкідливого коду "

Виконав: студент VI курсу, групи СБм-61

Спеціальності:

125 «Кібербезпека та захист інформації»

(шифр і назва напряму підготовки, спеціальності)

Легкобит Олексій Юрійович

підпис (прізвище та ініціали)

Керівник Стадник М. А.

підпис (прізвище та ініціали)

Нормоконтроль Стадник М. А.

підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.

підпис (прізвище та ініціали)

Рецензент

підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____________ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _________ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 125 Кібербезпека та захист інформації
(шифр і назва спеціальності)

Студенту Легкобиту Олексію Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Генеративні мовні моделі в аналізі шкідливого коду

Керівник роботи Стадник Марія Андріївна, к.т.н., доцент кафедри КБ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «24» 11 2025 року № 4/7-1024

2. Термін подання студентом завершеної роботи 12.12.2025

3. Вихідні дані до роботи Набір даних коду із GitHub та MalwareBazaar

4. Зміст роботи: 1 Теоретичні засади аналізу шкідливого програмного забезпечення 1.1 Проблема виявлення шкідливого програмного забезпечення та коду. 1.2 Класифікація шкідливого програмного забезпечення. 1.3 Методи аналізу шкідливого коду.1.3.1 Статичний підхід до аналізу шкідливого коду. 1.3.2 Динамічний підхід до аналізу шкідливого коду. 1.3.3 Сигнатурний аналіз шкідливого ПЗ. 2 Генеративні мовні моделі в аналізі шкідливого програмного забезпечення. 2.1 Генеративні мовні моделі. 2.1.1 Передумови появи Transformer. 2.1.2 Архітектура Transformer. 2.1.3 Типи та класифікація LLM.2.2 Можливості LLM для аналізу шкідливої поведінки програмного коду. 2.3 Порівняльний аналіз традиційних методів аналізу шкідливого програмного забезпечення та підходів на основі LLM. 3 Практична реалізація llm підходів для виявлення шкідливого ПЗ. 3.1 Обґрунтування вибору LLM моделі. 3.2 Формування набору даних дослідження. 3.3 Методика застосування LLM для аналізу та класифікації коду. 3.4 Застосування досліджуваних LLM. 3.5 Порівняльний аналіз результатів експериментального дослідження. 4 Охорона праці та безпека в надзвичайних ситуаціях. 4.1 Охорона праці. 4.2 Характеристика стихійних лих, аварій (катастроф) та їх наслідків. Висновки.

5. Перелік графічного матеріалу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Осухівська Г.М., к.т.н., доцент		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 19.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	19.09 – 22.09	Виконано
2.	Опрацювання джерел щодо виявлення шкідливого коду.	23.09 – 1.10	Виконано
3.	Дослідження наборів даних, які містять промарковані мітки шкідливого коду .	2.10 – 12.10	Виконано
4.	Реалізація алгоритму виявлення шкідливого коду.	13.10 – 24.10	Виконано
5.	Оформлення першого розділу.	25.10 – 2.11	Виконано
6.	Оформлення другого розділу.	3.11 – 10.11	Виконано
7.	Оформлення третього розділу.	11.11 – 20.11	Виконано
8.	Виконання завдання до підрозділу “Охорона праці”.	21.11 – 27.11	Виконано
9.	Виконання завдання до підрозділу “Безпека в надзвичайних ситуаціях”	28.11 – 1.12	Виконано
10.	Оформлення кваліфікаційної роботи	2.12 – 7.12	Виконано
11.	Нормоконтроль	08.12 – 10.12	Виконано
12.	Перевірка на плагіат	12.12 – 14.12	Виконано
13.	Попередній захист кваліфікаційної роботи	15.12 – 19.12	Виконано
14.	Захист кваліфікаційної роботи	22.12.2023	

Студент ______________ Легкобит О. Ю.
(підпис) (прізвище та ініціали)

Керівник роботи ______________ Стадник М. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Генеративні мовні моделі в аналізі шкідливого коду // ОР «Магістр» // Легкобит Олексій Юрійович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБм-61 // Тернопіль, 2025 // С. 107, рис. – 17, табл. – 8, кресл. – , додат. – 4.

Ключові слова: генеративні мовні моделі, LLM, шкідливе програмне забезпечення, виявлення malware, few-shot learning, семантичний аналіз.

У кваліфікаційній роботі проведено дослідження можливостей застосування генеративних мовних моделей для виявлення та аналізу шкідливого програмного забезпечення. Здійснено огляд сучасних підходів до аналізу malware та обґрунтовано доцільність використання великих мовних моделей для семантичного аналізу програмного коду в умовах обфускації та zero-day загроз. Для практичної реалізації обрано моделі сімейств GPT, LLaMA та Mistral, які представляють різні підходи до розгортання та використання LLM у системах кібербезпеки.

У межах роботи сформовано збалансований набір даних на основі реальних зразків шкідливого програмного забезпечення з репозиторію MalwareBazaar та легітимного програмного коду з відкритих репозиторіїв GitHub, представлений у статичному текстовому вигляді. Запропоновано методику застосування LLM із використанням few-shot підходу та фіксованого prompt для забезпечення стабільності результатів. Проведено експериментальні дослідження ефективності моделей з використанням метрик accuracy, precision, recall та матриць помилок, а також виконано порівняльний аналіз їхньої якості та практичної придатності для задач виявлення шкідливого програмного коду.

ABSTRACT

Generative language models in the analysis of malicious code // Thesis of educational level "Master"// Oleksii Lehkobyt // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group СБм-61 // Ternopil, 2025 // p. 107, figs. 17, tbls. 8, drws.   , apps. 4.

Keywords: generative language models, LLM, malicious software, malware detection, few-shot learning, semantic analysis.

In this qualification thesis, a study is conducted on the possibilities of applying generative language models for the detection and analysis of malicious software. A review of modern approaches to malware analysis is carried out, and the feasibility of using large language models for the semantic analysis of program code under conditions of obfuscation and zero-day threats is substantiated. For practical implementation, models from the GPT, LLaMA, and Mistral families are selected, representing different approaches to the deployment and use of LLMs in cybersecurity systems.

Within the scope of the work, a balanced dataset is constructed based on real malware samples from the MalwareBazaar repository and legitimate program code from open GitHub repositories, represented in a static textual form. A methodology for applying LLMs using a few-shot approach and a fixed prompt is proposed to ensure the stability of results. Experimental studies of model performance are conducted using accuracy, precision, recall metrics, and confusion matrices, and a comparative analysis of their quality and practical suitability for malware detection tasks is performed.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП........ 9
РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ АНАЛІЗУ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ........ 12
1.1 Проблема виявлення шкідливого програмного забезпечення та коду 12
1.2 Класифікація шкідливого програмного забезпечення 17
1.3 Методи аналізу шкідливого коду 20
1.3.1 Статичний підхід до аналізу шкідливого коду 21
1.3.2 Динамічний підхід до аналізу шкідливого коду........ 25
1.3.3 Сигнатурний аналіз шкідливого ПЗ 28
РОЗДІЛ 2 ГЕНЕРАТИВНІ МОВНІ МОДЕЛІ В АНАЛІЗІ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 31
2.1 Генеративні мовні моделі........ 31
2.1.1 Передумови появи Transformer 33
2.1.2 Архітектура Transformer........ 35
2.1.2.1 Принцип роботи кодера 36
2.1.2.2 Принцип роботи декодера........ 43
2.1.3 Типи та класифікація LLM 47
2.2 Можливості LLM для аналізу шкідливої поведінки програмного коду ... 49
2.3 Порівняльний аналіз традиційних методів аналізу шкідливого програмного забезпечення та підходів на основі LLM 52
РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ LLM ПІДХОДІВ ДЛЯ ВИЯВЛЕННЯ ШКІДЛИВОГО ПЗ........ 55
3.1 Обґрунтування вибору LLM моделі........ 55
3.2 Формування набору даних дослідження 58
3.3 Методика застосування LLM для аналізу та класифікації коду 61
3.4 Застосування досліджуваних LLM........ 64
3.5 Порівняльний аналіз результатів експериментального дослідження......... 67

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 70
4.1 Охорона праці 70
4.2 Характеристика стихійних лих, аварій (катастроф) та їх наслідків 74
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 84
Додаток А Публікація 88
Додаток Б Лістинг GPT.py 90
Додаток В Лістинг LLaMA.py 97
Додаток Г Лістинг Mistral.py 103

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CSV — Comma-Separated Values

DBIR — Data Breach Investigations Report

DLL — Dynamic Link Library

ELF — Executable and Linkable Format

GPT — Generative Pre-trained Transformer

GRU — Gated Recurrent Unit

IoT — Internet of Things

LLM — Large Language Model

LSTM — Long Short-Term Memory

NLP — Natural Language Processing

PE — Portable Executable

PUA — Potentially Unwanted Application

RAT — Remote Access Trojan

RNN — Recurrent Neural Network

SOC — Security Operations Center

TCP — Transmission Control Protocol

VPN — Virtual Private Network

ВСТУП

Актуальність теми. У сучасних умовах стрімкої цифровізації та зростання залежності суспільства від інформаційних технологій проблема забезпечення кібербезпеки набуває особливої актуальності. Однією з ключових загроз для інформаційних систем залишається шкідливе програмне забезпечення, яке постійно еволюціонує, застосовуючи нові методи маскування, обфускації та ухилення від виявлення. Традиційні підходи до аналізу та детекції malware, що базуються на сигнатурних, евристичних або статистичних методах, дедалі частіше виявляються недостатньо ефективними, особливо у випадках zero-day загроз та цілеспрямованих атак.

Останні досягнення у сфері штучного інтелекту, зокрема поява великих генеративних мовних моделей (LLM), відкривають нові можливості для аналізу програмного коду на семантичному рівні. На відміну від класичних методів, LLM здатні інтерпретувати логіку виконання коду, виявляти приховані наміри програм та узагальнювати складні поведінкові патерни, що робить їх перспективним інструментом для задач виявлення та класифікації шкідливого програмного забезпечення.

Попри активне зростання кількості досліджень у цьому напрямі, питання ефективності застосування різних архітектур генеративних мовних моделей для аналізу шкідливого коду, а також порівняльна оцінка їх якості за стандартизованими метриками, залишаються недостатньо вивченими. Особливої уваги потребує дослідження можливостей LLM у реалістичних умовах, коли зразки коду можуть бути частково обфускованими, фрагментованими або представленими у вигляді окремих функцій чи скриптів. Це зумовлює актуальність обраної теми кваліфікаційної роботи.

Метою кваліфікаційної роботи є дослідження ефективності використання генеративних мовних моделей для виявлення та класифікації шкідливого програмного коду на основі аналізу його текстового представлення та семантичних ознак.

Для досягнення поставленої мети в роботі необхідно розв'язати такі **завдання дослідження**:

- Проаналізувати сучасний стан проблеми виявлення шкідливого програмного забезпечення та основні підходи, що використовуються в існуючих системах захисту.
- Дослідити принципи роботи генеративних мовних моделей та обґрунтувати доцільність їх застосування для аналізу програмного коду.
- Розглянути особливості застосування Few-shot learning для задач класифікації шкідливого коду.
- Реалізувати експериментальні сценарії аналізу шкідливого програмного коду з використанням різних LLM.
- Провести порівняльний аналіз результатів роботи обраних моделей за показниками accuracy, precision, recall та confusion matrix.
- Оцінити сильні та слабкі сторони застосування генеративних мовних моделей у задачах аналізу malware.
- Сформулювати висновки та рекомендації щодо практичного використання LLM у системах кібербезпеки.

Об'єктом дослідження є процес автоматизованого виявлення та аналізу шкідливого програмного коду в інформаційних системах.

Предметом дослідження є методи та моделі аналізу програмного коду на основі генеративних мовних моделей, а також їх ефективність у задачах класифікації шкідливого програмного забезпечення.

Наукова новизна одержаних результатів кваліфікаційної роботи полягає у проведенні порівняльного аналізу ефективності сучасних генеративних мовних моделей для задач виявлення шкідливого програмного коду з використанням методики Few-shot learning. У роботі вперше в рамках єдиного експериментального підходу досліджено вплив особливостей формування промптів та прикладів навчання на якість класифікації malware за обмеженої кількості вхідних даних. Отримано кількісні оцінки якості моделей із використанням confusion matrix, що дозволяє глибше інтерпретувати результати їх роботи.

Практичне значення роботи полягає у можливості використання отриманих результатів під час розроблення та вдосконалення систем аналізу шкідливого програмного забезпечення. Запропоновані підходи можуть бути використані як допоміжний інструмент у системах кіберзахисту, центрах реагування на інциденти безпеки (SOC), а також у процесі підготовки фахівців з кібербезпеки для аналізу та інтерпретації потенційно небезпечного програмного коду.

Апробація результатів магістерської роботи. Основні результати дослідження були апробовані на XIII науково-технічній конференції «Інформаційні моделі, системи та технології» (м.Тернопіль, Україна, 17-18 грудня). Відповідна наукова публікація наведена у Додатку А.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ АНАЛІЗУ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Проблема виявлення шкідливого програмного забезпечення та коду

У сучасному інформаційному середовищі доцільно розрізняти поняття шкідливого програмного забезпечення та шкідливого коду. Шкідливий код являє собою фрагменти програмних інструкцій, скриптів або бінарних послідовностей, які реалізують небезпечну або несанкціоновану функціональність незалежно від того, чи оформлені вони у вигляді завершеного програмного продукту. На відміну від класичного malware як окремого програмного об’єкта, шкідливий код може існувати у вигляді окремих функцій, макросів, команд оболонки, PowerShell-скриптів, JavaScript-вставок, exploit-ланцюгів або ін’єкцій у легітимний код. Така фрагментарність і модульність дозволяє зловмисникам легко вбудовувати шкідливу логіку у легітимні застосунки, оновлення або конфігураційні файли, що суттєво ускладнює її виявлення традиційними сигнатурними методами.

У 2024–2025 роках вищевказана особливість стала однією з ключових причин ускладнення задачі виявлення кіберзагроз. Шкідливий код дедалі частіше не проявляється у вигляді стабільного виконуваного файлу, а реалізується через fileless-техніки, динамічно згенеровані скрипти або короткоживучі фрагменти коду, що активуються лише за певних умов. Унаслідок цього класичні підходи, орієнтовані на ідентифікацію відомих зразків malware, втрачають ефективність і потребують переосмислення з урахуванням поведінкових і семантичних характеристик коду.

Проблема виявлення шкідливого програмного забезпечення загострюється також через стрімке зростання кількості інцидентів та трансформацію моделей атак. Причина полягає не лише у зростанні кількості інцидентів, а й у зміні моделі загроз: атаки дедалі частіше спираються на викрадені облікові дані, експлуатацію вразливостей (зокрема на периметрових чи edge-пристроях), а також на шантаж і руйнування доступності (ransomware та суміжні техніки). У

звіті Verizon DBIR 2024 проаналізовано 30 458 інцидентів, з яких 10 626 підтверджено як витоки чи порушення безпеки даних, що відображає масштаб проблеми для організацій різних типів [1].

У 2025 році тенденція посилилась, як доказ, в підсумковому звіті Verizon DBIR 2025 зазначено, що було проаналізовано 22 052 реальні інциденти, з яких 12 195 є підтвердженими порушення даних (data breaches), а також підкреслено зростання ролі ланцюгів постачання чи третіх сторін у виникненні інцидентів. Це надзвичайно важливо для malware-аналізу, оскільки шкідливий код і первинний доступ все частіше з'являються не «всередині» організації, а через інтеграції, підрядників, сторонні сервіси та компрометацію доступів [2].

З погляду «входу» атак у 2025 році домінують три практично значущі вектори:

- зловживання обліковими даними (22%);
- експлуатація вразливостей (20%);
- фішинг (16%) [2].

Паралельно вказується, що експлуатація вразливостей як вектор первинного доступу продовжила зростання і досягла 20%, причому зростання підтримувалось, зокрема, zero-day експлойтами проти edge-пристроїв і VPN; також згадується, що лише близько 54% таких вразливостей були повністю усунені протягом року (у вибірці звіту), що підвищує ймовірність компрометації навіть за наявності патчів «у принципі» [2].

Згідно даними звіту DBIR 2024 ransomware «знизився» до 23% порушень, однак сумарно з іншими техніками вимагання (extortion) це становило близько 32% усіх порушень. У 2025 році зафіксовано суттєве зростання, а саме: presence ransomware (з шифруванням або без) присутній у 44% розглянутих порушень (зростання з 32% у попередньому звіті), а також наведено дані, що малий і середній бізнес постраждав непропорційно сильно (ransomware-компонент у 88% їхніх порушень), тоді як у великих організаціях становить 39% [1-2]. Це означає, що задача виявлення malware стає не тільки технічною, а й операційною задачею, оскільки пропуск ранніх ознак (облікові дані чи вразливості) швидко трансформується в інцидент із шантажем, простоєм і каскадними наслідками [3].

Економічний ефект кіберінцидентів вимірюється не лише сумою викупу, а й простоями, відновленням, юридичними витратами та репутаційними втратами. Згідно з IBM Cost of a Data Breach Report 2024, середня глобальна вартість одного витоку даних у 2024 році досягла USD 4,88 млн (проти USD 4,45 млн у 2023), а драйверами зростання названо втрати бізнесу (downtime, втрата клієнтів) і витрати на реагування після інциденту [4].

Окремо, за даними Chainalysis, у 2024 році жертви сплатили близько USD 813,55 млн ransomware-платежів у криптовалюті (приблизно на 35% менше за 2023), що демонструє: навіть за зниження платежів екосистема атак не зникає, а адаптується (рефакторинг/купівля кодової бази, швидші переговори, більше інцидентів із меншою конверсією в оплату [5]. В європейському контексті ENISA Threat Landscape 2024 також відносить ransomware до ключових загроз, підкреслюючи домінування атак на доступність та шантаж як системний ризик.

Узагальнюючи, проблема виявлення malware у 2024–2025 рр. визначається трьома взаємопов’язаними факторами:

- масштабом інцидентів і швидкістю їх еволюції;
- переорієнтацією атак на облікові дані;
- експлуатацію вразливостей та компрометацію третіх сторін;
- високими економічними наслідками, де ransomware виступає каталізатором максимальних втрат.

На рис 1.1 відображено динаміку загальної кількості виявлених зразків шкідливого програмного забезпечення та потенційно небажаних застосунків (PUA) у період з 2008 по 2024 роки. Аналіз графіка свідчить про стійку зростаючу тенденцію протягом усього досліджуваного періоду, причому особливо різке збільшення обсягів фіксується, починаючи з 2014–2015 років. Кількість зразків malware зросла з десятків мільйонів на початку періоду до понад 1,1–1,2 млрд у 2023–2024 роках, що відображає масштабну автоматизацію процесів створення та модифікації шкідливого коду. Водночас сегмент PUA також демонструє стабільне зростання, особливо після 2018 року, що вказує на розмиття меж між відверто шкідливим ПЗ та програмами з агресивною або небажаною поведінкою [6-8]. Сукупно ці тенденції підтверджують еволюцію

кіберзагроз від поодиноких зразків до масового, індустріалізованого виробництва програмного коду, що істотно ускладнює завдання виявлення, класифікації та аналізу загроз класичними методами та підкреслює актуальність застосування інтелектуальних підходів у сфері кібербезпеки.

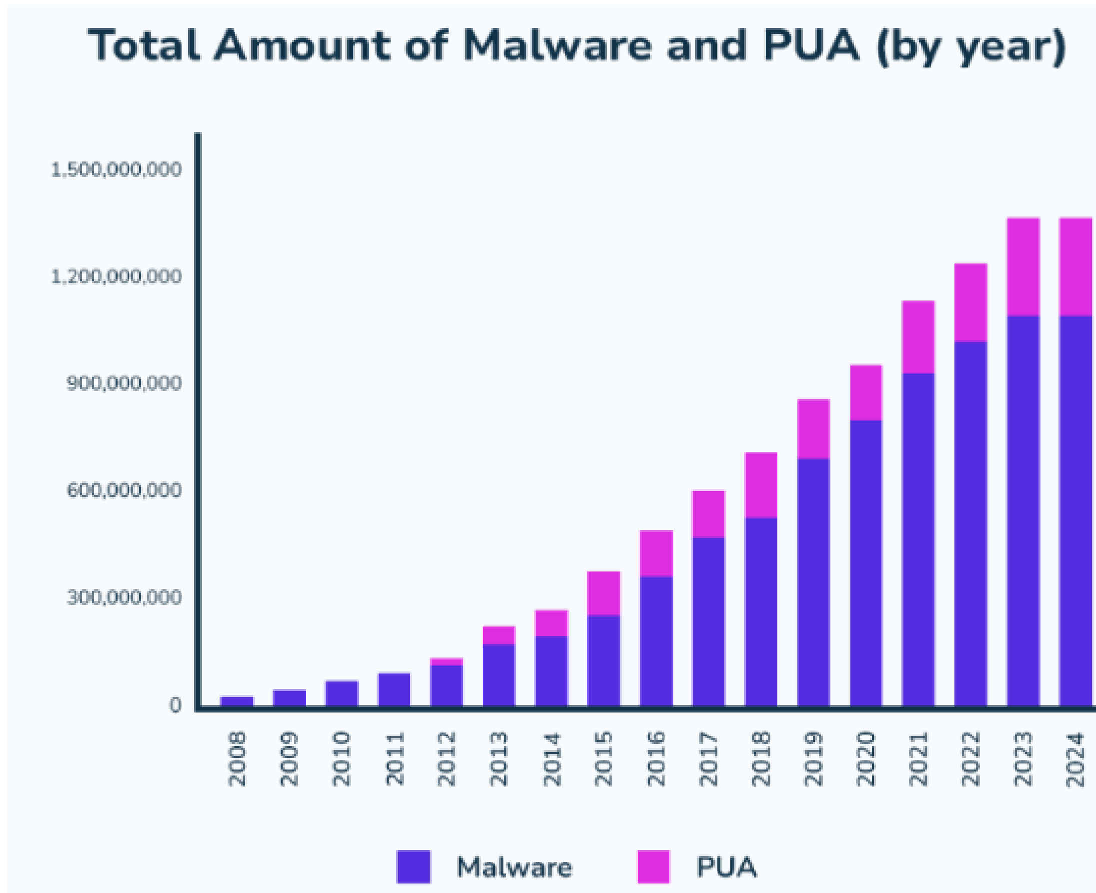

Рисунок 1.1 – Динаміка загальної кількості шкідливого програмного забезпечення та потенційно небажаних застосунків (PUA) у 2008–2024 роках [8]

На рисунку 1.2 відображено зростання кількості зразків шкідливого програмного забезпечення, орієнтованого на IoT-пристрої, у період з 2018 по 2022 роки. Як видно з графіка, за досліджуваний проміжок часу обсяг виявленого IoT-malware зріс більш ніж утричі: з приблизно 32,7 млн зразків у 2018 році до понад 112 млн у 2022 році. Особливо різке зростання спостерігається після 2020 року, що корелює зі стрімким поширенням розумних пристроїв, віддаленої роботи, а також масовим впровадженням IoT-рішень у промисловості, побуті та міській інфраструктурі [8-10].

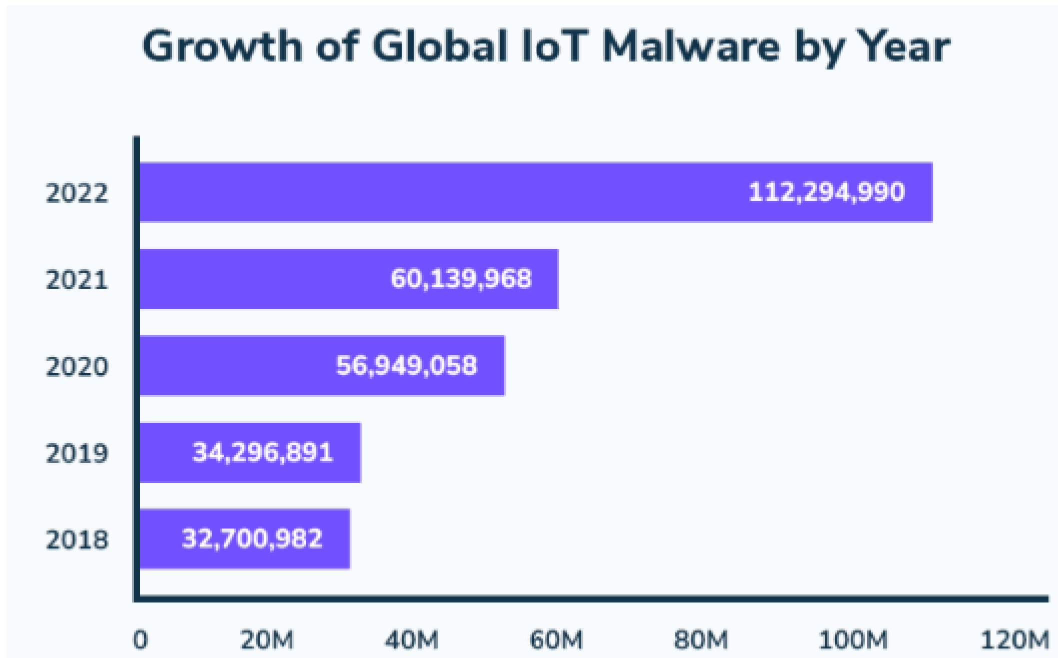

Рисунок 1.2 – Еволюція обсягів IoT-орієнтованого шкідливого програмного забезпечення [8]

Наведені дані підкреслюють, що безпека IoT є критично важливим, але часто недооціненим компонентом загальної системи кіберзахисту. Багато IoT-пристроїв мають обмежені обчислювальні ресурси, використовують застаріле програмне забезпечення, стандартні облікові дані або взагалі не отримують регулярних оновлень безпеки, що робить їх доступною ціллю для зловмисників. Аналіз сучасних тенденцій розвитку malware свідчить про необхідність врахування специфіки IoT-середовищ і впровадження комплексних підходів до захисту, оскільки ігнорування безпеки IoT-пристроїв створює системні ризики для всієї інформаційної інфраструктури.

Підведемо підсумок щодо вище написаного: у 2024–2025 роках проблема виявлення шкідливого програмного забезпечення та шкідливого коду полягає не лише у зростанні кількості загроз, а й у зміні їхньої природи. Сучасний malware дедалі частіше постає як сукупність динамічних, контекстно залежних фрагментів коду, що уникають детектування класичними засобами. Це обумовлює необхідність розвитку нових підходів до аналізу, здатних враховувати семантику, поведінкові патерни та приховані наміри програмного

коду, що безпосередньо зумовлює актуальність подальшого розгляду методів аналізу шкідливого ПЗ.

1.2 Класифікація шкідливого програмного забезпечення

Сучасне шкідливе програмне забезпечення є результатом тривалої еволюції кіберзагроз і характеризується високим рівнем автоматизації, адаптивності та активним використанням складних технік приховування (обфускації, пакування, шифрування, fileless-виконання). На відміну від ранніх форм malware, орієнтованих на масове зараження, сучасні зразки шкідливого ПЗ часто є частиною цілеспрямованих багатоступеневих атак, у межах яких шкідливий код динамічно змінює поведінку залежно від середовища виконання, наявності засобів захисту та профілю жертви. Це суттєво ускладнює його виявлення та аналіз із використанням традиційних сигнатурних і rule-based підходів.

Залежно від функціонального призначення, способу поширення та характеру впливу на інформаційну систему, шкідливе програмне забезпечення поділяється на кілька основних категорій, що представлено на рис. 1.3.

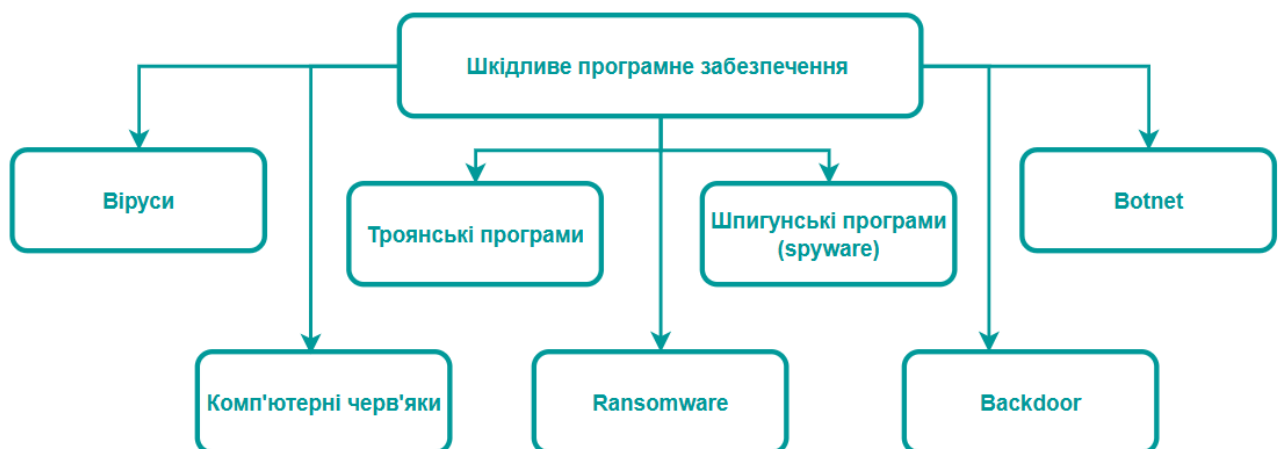

Рисунок 1.4 – Класифікація шкідливого програмного забезпечення

Віруси – це програми, здатні до самовідтворення шляхом зараження інших файлів або виконуваних модулів, активація яких відбувається під час запуску зараженого об’єкта. Класичні файлові віруси в сучасних атаках

використовуються рідше, однак їхні механізми реплікації активно застосовуються в складніших формах malware. Прикладом може слугувати використання вірусних технік у шкідливих макросах Microsoft Office, які маскуються під легітимні документи.

Автономні програми, що поширюються мережею без безпосередньої участі користувача, експлуатуючи вразливості у сервісах та протоколах називаються червами (worms). У 2024 році численні інциденти були пов’язані з мережевими червами, які використовували вразливості у VPN-шлюзах, серверних компонентах та edge-пристроях. Історично відомі приклади, такі як WannaCry або NotPetya, і надалі впливають на архітектуру сучасних мережевих атак.

Троянські програми маскуються під легітимне програмне забезпечення або корисні утиліти, проте виконують приховані шкідливі дії для викрадення облікових даних, завантаження додаткових модулів або надання віддаленого доступу зловмиснику. У 2024 році значну частку інцидентів становили трояни-дропери та лоадери (наприклад, Emotet, QakBot, IcedID), які слугували початковою точкою для розгортання ransomware або APT-кампаній.

Програмне забезпечення призначене для прихованого збору інформації про користувача, його дії, введені дані та конфіденційну інформацію складає групу шпигунського програмного забезпечення (spyware). До цієї категорії належать кейлогери, інформаційні стилери (infostealers) та комерційні spyware-рішення. У 2024 році особливої поширеності набули стилери на кшталт RedLine, Raccoon та Vidar, які масово використовувалися для викрадення облікових даних і подальшого перепродажу доступів на тіньових майданчиках.

Ransomware є одним із найнебезпечніших типів шкідливого ПЗ і реалізує блокування доступу до системи або шифрування даних із метою вимагання викупу. У 2024 році ransomware залишався домінуючим компонентом значної частини підтверджених інцидентів, причому атаки часто поєднували шифрування з попереднім викраденням даних (double extortion). Відомими прикладами таких кампаній є сімейства LockBit, ALPHV (BlackCat) та Cl0p, які завдали багатомільйонних збитків організаціям у різних секторах економіки.

Ще одним видом шкідливого програмного забезпечення є backdoor. По суті backdoor це приховані механізми доступу до інформаційної системи, що дозволяють обходити стандартні процедури автентифікації та контролю доступу. Backdoor-компоненти широко використовуються у цілеспрямованих атаках і часто впроваджуються після первинної компрометації для забезпечення довготривалої присутності зловмисника в системі.

Botnet-клієнти є складовими розподілених мереж заражених пристроїв, що керуються централізовано або децентралізовано. У 2024 році ботнети активно застосовувалися для проведення DDoS-атак, розповсюдження шкідливого ПЗ та фішингових кампаній. Відомими прикладами є модифіковані версії Mirai та Mozi, орієнтовані на IoT-пристрої.

Окрему та особливо небезпечну категорію становить поліморфне та метаморфне шкідливе програмне забезпечення, яке змінює свою структуру, сигнатури або навіть логіку виконання при кожному запуску чи поширенні, зберігаючи при цьому функціональну еквівалентність. Саме такі зразки malware створюють найбільші труднощі для традиційних систем виявлення, оскільки практично не мають стабільних ознак для сигнатурного аналізу.

У 2024 році домінуючу частку серед зафіксованих загроз становили троянські програми та лоадери (рис. 1.5), які використовуються як початковий етап складних багатоступеневих атак. Значну частку також займає шпигунське програмне забезпечення, орієнтоване на викрадення облікових даних. Частка ransomware залишається високою, що підтверджує актуальність загроз, пов’язаних із цифровим шантажем та порушенням доступності інформаційних ресурсів.

Представлені графічно статистичні дані підтверджують тенденцію переходу від масових однотипних загроз до адаптивного, фрагментованого та контекстно залежного шкідливого коду, що суттєво ускладнює його детектування класичними методами.

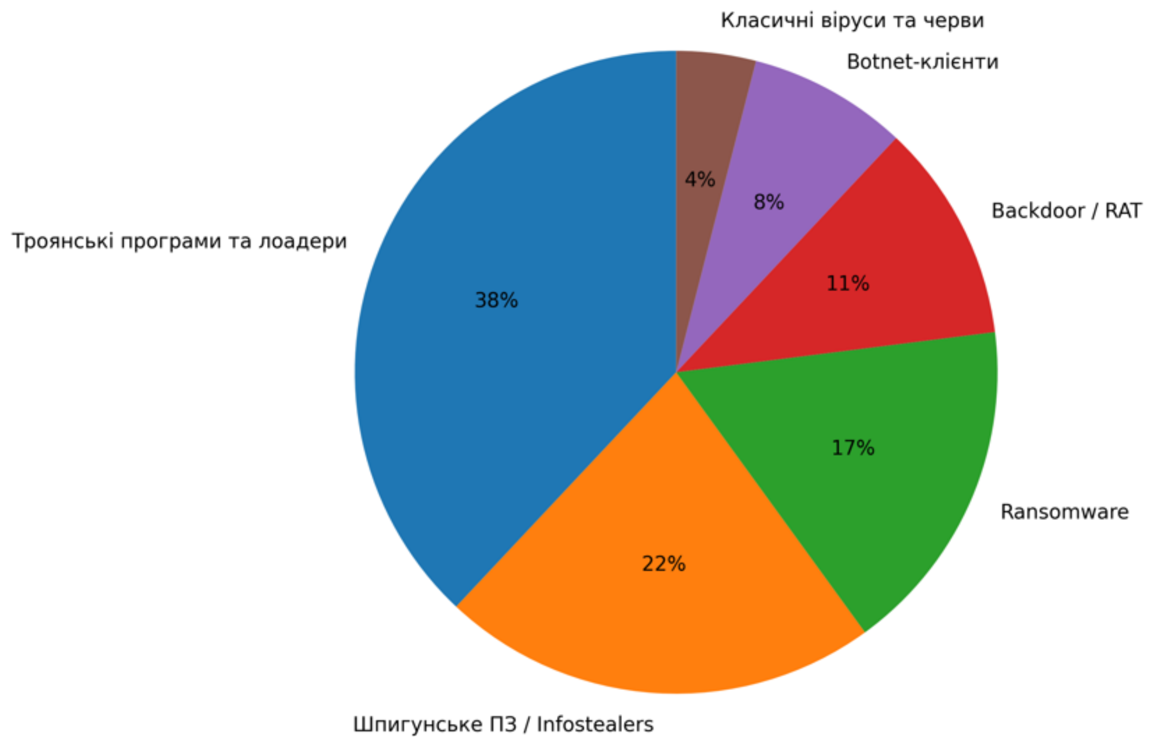

Рисунок 1.5 – Розподіл зафіксованих типів шкідливого програмного забезпечення у 2024 році (за узагальненими даними аналітичних звітів AV-TEST, ENISA, SonicWall та Verizon DBIR) [2-8]

З урахуванням різноманіття типів шкідливого програмного забезпечення та еволюції форм шкідливого коду, особливо зростання частки троянських програм, spyware, ransomware, backdoor і IoT-орієнтованого malware, актуальним напрямом досліджень стає застосування LLM для задач аналізу та виявлення загроз. Потенціал LLM полягає у здатності узагальнювати поведінкові патерни різних класів malware, виявляти приховані зв’язки між ознаками та адаптуватися до нових або модифікованих зразків шкідливого ПЗ, що робить такі моделі перспективним інструментом для подолання обмежень класичних методів аналізу в умовах стрімкого зростання та ускладнення сучасних кіберзагроз.

1.3 Методи аналізу шкідливого коду

Аналіз шкідливого програмного забезпечення є ключовим етапом у процесі кіберзахисту, що дозволяє визначити поведінку malware, його функціональні

можливості та потенційні загрози. У практиці інформаційної безпеки застосовуються три основні підходи до аналізу шкідливого коду:

- статичний;
- динамічний;
- гібридний [11-12].

У наступних підрозділах кваліфікаційної роботи буде здійснено детальний розгляд кожного з методів, а також проаналізовано їхні переваги та обмеження.

1.3.1 Статичний підхід до аналізу шкідливого коду

Статичний аналіз передбачає дослідження програмного коду без його виконання. Він включає аналіз бінарних файлів, дизасемблювання, вивчення структури виконуваних файлів, рядків, імпортованих бібліотек та сигнатур [11-12]. Перевагами статичного аналізу є безпечність та можливість швидкого дослідження великої кількості зразків. Водночас цей метод є малоефективним проти зашифрованого, обфускованого або поліморфного коду [13].

Статичний аналіз шкідливого програмного забезпечення є одним із базових методів дослідження malware і полягає у вивченні програмного коду без його фактичного виконання. Такий підхід дозволяє дослідити структуру, логіку та потенційні функціональні можливості програми, мінімізуючи ризики зараження системи під час аналізу. Статичний аналіз широко застосовується на початкових етапах дослідження шкідливого коду та є ефективним для швидкої класифікації відомих і частково модифікованих зразків malware.

Одним із ключових елементів статичного аналізу є дослідження викликів API. Аналіз використовуваних програмних інтерфейсів дозволяє зробити висновки щодо потенційної поведінки програми. Наприклад, виклики API для роботи з файловою системою, мережею, процесами або реєстром можуть свідчити про спроби модифікації системних ресурсів, встановлення з’єднань із віддаленими серверами або впровадження механізмів стійкості (persistence). У контексті аналізу шкідливого програмного забезпечення вивчення API-викликів

є важливим індикатором наявності троянських функцій, backdoor-компонентів або шпигунських механізмів.

Важливою складовою статичного аналізу є побудова та дослідження графа потоку керування (Control Flow Graph, CFG). CFG відображає всі можливі шляхи виконання програми та взаємозв'язки між базовими блоками інструкцій. Аналіз такого графа дозволяє виявляти складні логічні конструкції, умовні переходи, цикли, а також приховані або рідко активовані гілки коду. У випадку malware це дає змогу ідентифікувати антианалізні механізми, логіку активації шкідливих функцій за певних умов або спроби обходу засобів захисту.

Ще одним аспектом статичного аналізу є вивчення процесорних інструкцій, що виконуються програмою. Дизасемблювання бінарного коду дозволяє дослідити низькорівневу реалізацію алгоритмів, використання системних викликів, маніпуляції з пам'яттю та регістрами процесора. Аналіз інструкцій особливо важливий для виявлення обфускації, шифрування, упаковки коду або застосування експлойт-технік, які часто використовуються сучасним шкідливим програмним забезпеченням для ускладнення аналізу [12-13].

Окрему роль у статичному аналізі відіграє N-грамний аналіз, який передбачає дослідження послідовностей інструкцій, байтів або опкодів фіксованої довжини. Такий підхід дозволяє виявляти статистичні закономірності, характерні для певних сімейств malware, навіть у випадку часткової модифікації коду. N-грамні моделі широко застосовуються у поєднанні з методами машинного навчання для автоматизованої класифікації шкідливих програм і є ефективними проти поліморфних та метаморфних загроз [14-15].

Одним із ранніх і широко цитованих досліджень у цій галузі є робота Kolter та Maloof (2006), у якій запропоновано застосування n-грамного аналізу байтових послідовностей у поєднанні з методами машинного навчання для автоматичної класифікації шкідливого програмного забезпечення. Автори продемонстрували, що статистичні ознаки, отримані зі статичного аналізу, дозволяють досягати високої точності класифікації навіть у випадку поліморфних загроз [14].

Окрім вищезазначених характеристик, у межах статичного аналізу шкідливого програмного забезпечення застосовується низка додаткових ознак, які підвищують точність класифікації та попереднього виявлення загроз. До таких ознак належать розмір виконуваного файлу, кількість і довжина функцій, а також щільність коду, що може свідчити про використання пакувальників або обфускації.

Важливу роль відіграє аналіз мережевих параметрів, зокрема виявлення жорстко закодованих TCP/UDP-портів, IP-адрес джерела та отримувача, а також HTTP-запитів, які можуть містити характерні шаблони звернення до командно-керуючих серверів (C2). Навіть без виконання програми наявність таких даних у коді є сильним індикатором потенційної шкідливої активності.

Додатково аналізуються статистичні характеристики розподілу інструкцій, ентропія секцій файлу та співвідношення коду до даних, що дозволяє виявляти зашифровані або стиснені ділянки.

Подальший розвиток статичного підходу представлений у дослідженні Santos et al. (2013), де використано статистичні характеристики опкодів і структурні метрики виконуваних файлів для виявлення шкідливого програмного забезпечення. Результати дослідження підтвердили ефективність статистичного аналізу як інструменту попереднього фільтрування malware перед глибшим динамічним дослідженням [15].

У роботі Anderson et al. (2018) статистичні ознаки статичного аналізу були поєднані з глибокими нейронними мережами, що дозволило автоматизувати процес виявлення шкідливого коду без необхідності ручного формування сигнатур. Це дослідження є показовим прикладом еволюції статистичних підходів у бік сучасних інтелектуальних моделей, зокрема тих, що концептуально близькі до LLM [16].

У роботі Dhilung Kirat та Giovanni Vigna представлено результати масштабного аналізу шкідливого програмного забезпечення з точки зору протидії сигнатурним методам детектування. Дослідження охоплює 2810 зразків malware та пропонує 78 унікальних технік, що дозволяють не застосувати сигнатурні механізми виявлення [17].

З метою узагальнення ключових складових статичного аналізу та ілюстрації їх практичного застосування у виявленні шкідливого ПЗ в таблиці 2 представлено відповідні елементи аналізу та приклади загроз.

Таблиця 1.1 – Елементи статичного аналізу та приклади загроз

Елемент аналізу	Предмет аналізу	Приклад загроз
Виклики API	Набір та послідовність системних функцій	Трояни, spyware, backdoor
CFG (граф потоку керування)	Логіка виконання, умовні переходи, цикли	Anti-analysis, dormant payload
Процесорні інструкції	Низькорівнева логіка, робота з пам’яттю	Exploit-код, shellcode
N-грами	Частотні послідовності байтів / опкодів	Поліморфний malware
Імпортні бібліотеки	Використовувані DLL / модулі	RAT, ransomware
Структура PE/ELF	Секції, заголовки, аномалії	Упаковане ПЗ
Рядкові константи	URL, команди, ключові слова	C2-комунікація
Метадані файлу	Таймстемпи, компілятор	Маскування походження

Загалом, статичний аналіз забезпечує глибоке розуміння структури та потенційної логіки шкідливого програмного забезпечення, однак має певні обмеження. Зокрема, його ефективність знижується у випадку використання складної обфускації, динамічного завантаження коду або fileless-технік. Саме тому в сучасних системах кібербезпеки статичний аналіз зазвичай використовується у поєднанні з динамічними та інтелектуальними методами, зокрема з підходами, заснованими на великих мовних моделях, що дозволяє підвищити точність і повноту виявлення шкідливого коду.

1.3.2 Динамічний підхід до аналізу шкідливого коду

Динамічний аналіз шкідливого програмного забезпечення полягає у дослідженні поведінки програмного коду під час його виконання у контрольованому та ізольованому середовищі, такому як віртуальна машина або пісочниця (sandbox). На відміну від статичного аналізу, динамічний підхід дозволяє спостерігати реальні дії програми, незалежно від рівня обфускації або упаковки коду, що робить його особливо ефективним для аналізу сучасних загроз.

Одним із ключових аспектів динамічного аналізу є дослідження викликів функцій, які здійснює програма під час виконання. Аналіз послідовності викликів системних та бібліотечних функцій дає змогу виявити спроби доступу до файлової системи, мережевих ресурсів, процесів операційної системи або механізмів автозапуску. Такі виклики є характерними для троянських програм, spyware та ransomware і дозволяють ідентифікувати шкідливу активність навіть у разі відсутності статичних сигнатур [11-13].

Важливою складовою динамічного аналізу є вивчення аргументів функцій, які передаються під час виконання. Аналіз параметрів викликів дозволяє визначити, з якими саме ресурсами взаємодіє програма: які файли відкриваються або модифікуються, до яких IP-адрес і портів здійснюються мережеві підключення, які ключі реєстру змінюються. У контексті шкідливого програмного забезпечення такі дані можуть свідчити про комунікацію з командно-керуючими серверами (C2), викрадення даних або підготовку до подальших етапів атаки.

Ще одним елементом динамічного аналізу є дослідження порядку виконання інструкцій, що дозволяє простежити фактичний шлях виконання програми. Аналіз динамічного потоку інструкцій дає змогу виявляти приховані гілки коду, логіку умовної активації шкідливих функцій, часові затримки, а також механізми ухилення від аналізу. Такий підхід є особливо корисним для виявлення fileless-атак, поліморфного malware та шкідливого ПЗ, що активується лише за виконання специфічних умов.

Динамічний аналіз широко застосовується для дослідження ransomware, botnet-клієнтів, IoT-орієнтованого malware, а також складних APT-кампаній, оскільки дозволяє отримати повну картину поведінки зразка у реальному часі. Водночас цей метод має низку обмежень, зокрема високу ресурсомісткість, обмежене покриття шляхів виконання та вразливість до антианалізних технік, таких як виявлення віртуального середовища або відкладене виконання.

Для систематизації основних складових динамічного аналізу шкідливого програмного забезпечення та ілюстрації їх практичного застосування у виявленні загроз у таблиці 1.2 наведено відповідні елементи аналізу та приклади malware, для яких вони є найбільш інформативними.

Таблиця 1.2 – Елементи динамічного аналізу та приклади загроз

Елемент аналізу	Предмет аналізу	Приклад загроз
Виклики функцій	Послідовність системних і бібліотечних викликів під час виконання	Трояни, spyware, ransomware
Аргументи функцій	Передані параметри (шляхи файлів, IP-адреси, порти, ключі реєстру)	Backdoor, C2-комунікація
Порядок виконання інструкцій	Реальний шлях виконання коду, умовні гілки	Поліморфний malware
Файлова активність	Створення, модифікація, шифрування файлів	Ransomware
Мережева активність	Встановлення TCP/UDP-з’єднань, HTTP/HTTPS-запити	Botnet, IoT-malware
Процесна активність	Створення/ін’єкція процесів, ескалація привілеїв	RAT, APT
Поведінкові патерни	Типові ланцюги дій (kill chain)	Fileless-атаки
Дії, що не сприяють аналізу	Виявлення VM, затримки виконання, sandbox evasion	Складне APT malware

Ефективність динамічного аналізу шкідливого програмного забезпечення підтверджується низкою сучасних наукових досліджень, у яких поведінкові ознаки виконання коду використовуються як ключове джерело інформації для детектування malware [18-21].

У роботі Raff et al. (2020) запропоновано підхід до динамічного аналізу, що базується на дослідженні послідовностей API-викликів із використанням глибоких нейронних мереж. Автори показали, що поведінкові логи виконання програм дозволяють ефективно виявляти раніше невідомі зразки malware, навіть за умов активної обфускації коду [18].

Дослідження Zhang et al. (2021) зосереджується на аналізі динамічних мережевих ознак шкідливого програмного забезпечення, зокрема часових характеристик та структур мережевого трафіку. Запропонований метод продемонстрував високу точність виявлення botnet-клієнтів і IoT-орієнтованого malware, що підтверджує важливість динамічного аналізу в середовищах з обмеженими ресурсами [19].

У роботі Fan et al. (2022) динамічний аналіз використовується для дослідження поведінки ransomware. Автори аналізують послідовності файлових операцій та викликів криптографічних функцій, що дозволяє виявляти атаки на ранніх етапах, до завершення процесу шифрування даних [20].

Подальший розвиток цього напряму представлений у дослідженні Singh et al. (2023), де застосовано комбінований підхід до аналізу динамічних трас виконання з використанням моделей глибокого навчання. Робота демонструє, що аналіз порядку виконання інструкцій та аргументів функцій є ефективним інструментом для виявлення fileless-атак і APT-кампаній, які практично не залишають статичних артефактів [21].

Отже, динамічний аналіз дозволяє отримати інформацію про реальну поведінку шкідливого програмного забезпечення в процесі його виконання та є важливою складовою комплексного підходу до аналізу malware. Практика його застосування в сучасних системах захисту передбачає інтеграцію з іншими методами аналізу, що сприяє підвищенню точності виявлення та коректності класифікації загроз.

1.3.3 Сигнатурний аналіз шкідливого ПЗ

Сигнатурний аналіз є одним із найдавніших і найпоширеніших методів виявлення шкідливого програмного забезпечення та належить до класичних статичних підходів. Його основна ідея полягає у порівнянні досліджуваного програмного об’єкта з наперед відомими сигнатурами, які описують характерні ознаки конкретних зразків або сімейств malware. Такий підхід широко використовується в антивірусних системах завдяки високій швидкодії та простоті реалізації [12].

У межах сигнатурного підходу виділяють два основних типи сигнатур. Хеш-сигнатури базуються на обчисленні контрольних сум або криптографічних хешів файлів і дозволяють швидко ідентифікувати відомі зразки шкідливого програмного забезпечення за умови повної ідентичності файлів. Сигнатури у вигляді послідовностей байтів є більш гнучкими та описують характерні фрагменти машинного коду або інструкцій, притаманні певному malware. Саме байтові сигнатури дозволяють виявляти частково модифіковані або перевпаковані зразки, хоча вони залишаються вразливими до обфускації та поліморфізму. На рис. 1.6 зображено типовий алгоритм роботи сигнатурного антивірусного механізму.

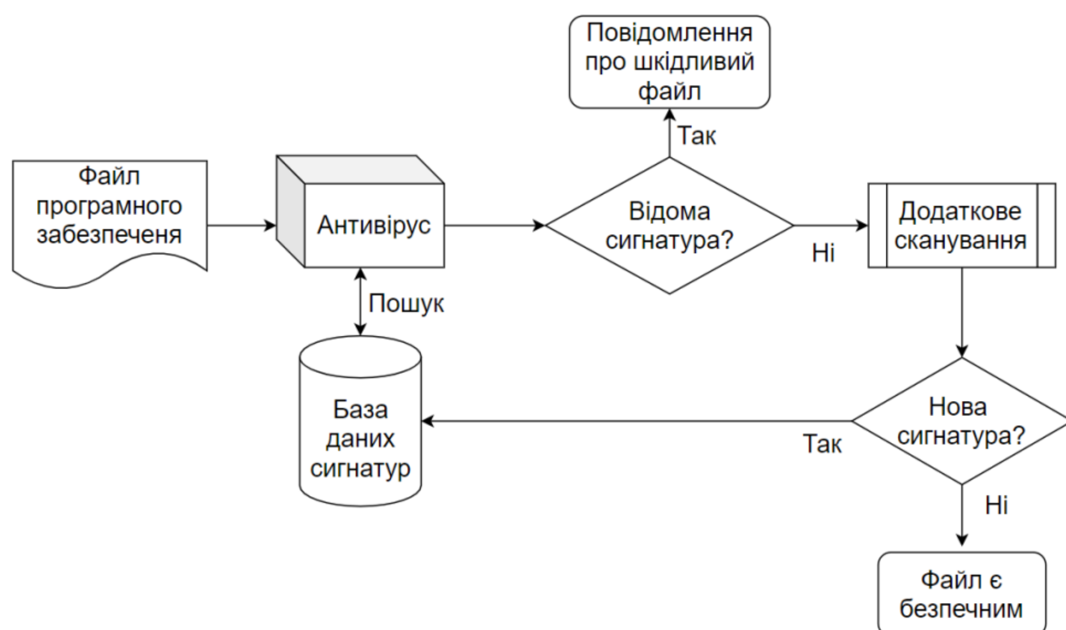

Рисунок 1.6 – Алгоритм роботи сигнатурного методу [22]

Процес починається з надходження файлу програмного забезпечення на перевірку, після чого антивірусний модуль здійснює пошук відповідних сигнатур у базі даних. У разі виявлення відомої сигнатури система формує повідомлення про наявність шкідливого файлу. Якщо ж відповідність не знайдено, запускається додаткове сканування, яке може включати евристичні або поведінкові методи. За умови виявлення нової сигнатури вона додається до бази даних, що забезпечує подальше розширення можливостей системи. У протилежному випадку файл вважається безпечним. Такий механізм ілюструє реактивну природу сигнатурного аналізу, коли ефективність детектування безпосередньо залежить від актуальності сигнатурної бази.

Проблематика ефективності сигнатурних методів неодноразово розглядалася у наукових дослідженнях. У роботі Christodorescu та Jha (2003) показано, що навіть незначні зміни у структурі коду можуть призводити до повного обходу сигнатурних детекторів, що стало одним із перших системних доказів обмеженості цього підходу [23].

Подальший розвиток цієї тематики представлений у дослідженні Kirat, Vigna та Kruegel (2014), де автори провели масштабний аналіз тисяч зразків malware та ідентифікували десятки технік ухилення від сигнатурного виявлення, включно з поліморфізмом, обфускацією та динамічною генерацією коду. Результати дослідження підтвердили, що сигнатурні методи є малоефективними проти сучасних адаптивних загроз.

У більш пізніх роботах, зокрема Saxe та Berlin (2015), наголошується, що сигнатурний аналіз може бути ефективним лише як перший рівень фільтрації, тоді як повноцінний захист вимагає поєднання з поведінковими та машинними методами аналізу. Сучасні дослідження після 2020 року також підтверджують цю тенденцію, розглядаючи сигнатурні підходи як допоміжний компонент багаторівневих систем детектування [24].

Попри свою поширеність, сигнатурний аналіз має суттєві обмеження. Він є ефективним переважно для відомих та раніше проаналізованих зразків malware, але практично не здатний виявляти zero-day загрози, поліморфне та метаморфне шкідливе програмне забезпечення, а також fileless-атаки. Саме тому сучасні

шкідливі програми активно використовують техніки зміни коду, динамічного завантаження та генерації унікальних екземплярів, щоб уникнути сигнатурного виявлення.

Таким чином, сигнатурний аналіз залишається базовим і швидким інструментом виявлення відомих зразків шкідливого програмного забезпечення, однак його обмежена здатність протидіяти новим та модифікованим загрозам зумовлює необхідність інтеграції з евристичними, поведінковими та інтелектуальними методами аналізу.

У наступному розділі кваліфікаційної роботи буде розглянуто принципи функціонування генеративних мовних моделей, їх архітектурні особливості та можливості застосування для задач аналізу й виявлення шкідливого програмного забезпечення. Особливу увагу буде приділено потенціалу LLM у контексті подолання обмежень класичних підходів. Крім того, у розділі буде представлено порівняльну таблицю існуючих методів аналізу шкідливого програмного забезпечення та підходів, заснованих на великих мовних моделях, що дозволить наочно оцінити їхні переваги, недоліки.

РОЗДІЛ 2 ГЕНЕРАТИВНІ МОВНІ МОДЕЛІ В АНАЛІЗІ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Генеративні мовні моделі

Генеративні мовні моделі (Large Language Models, LLM) є класом моделей штучного інтелекту, призначених для обробки, аналізу та генерації природної мови й програмного коду на основі великих обсягів навчальних даних [25]. Їх ключовою особливістю є здатність формувати узагальнені мовні представлення, які відображають семантичні, синтаксичні та контекстуальні зв'язки між елементами тексту або коду. На відміну від традиційних алгоритмів машинного навчання, LLM не потребують жорстко визначених правил чи ознак, оскільки навчаються виявляти закономірності безпосередньо з даних.

Поява генеративних мовних моделей є результатом тривалого розвитку методів обробки природної мови та машинного навчання. Перші підходи до автоматичного аналізу тексту ґрунтувалися на правило-орієнтованих системах, які використовували вручну задані граматичні правила та словники. Такі системи були обмежені масштабованістю й не здатні ефективно працювати з неоднозначністю та варіативністю природної мови. Подальший розвиток статистичних методів, зокрема n-грамних моделей, дозволив частково враховувати контекст, однак ці підходи залишалися чутливими до розміру навчальних вибірок і не забезпечували глибокого семантичного розуміння.

Наступним етапом еволюції стали нейронні мережі, зокрема рекурентні нейронні мережі (RNN) та їх модифікації – LSTM і GRU, які дали змогу моделювати послідовності змінної довжини та враховувати довгострокові залежності. Попри значний прогрес, RNN-подібні архітектури мали суттєві обмеження, пов'язані з послідовною обробкою даних, проблемами зникання градієнта та складністю масштабування на великі обсяги інформації. Ці обмеження стали ключовим чинником пошуку нових архітектурних рішень.

Революційним кроком у розвитку мовних моделей стало запропонування архітектури Transformer, яка повністю відмовилася від рекурентних зв'язків на

користь механізму самоуваги. Саме це рішення заклало технічні передумови для створення великих генеративних мовних моделей. Починаючи з 2018 року, з появою моделей типу GPT, BERT та їхніх похідних, стало можливим навчання надзвичайно масштабних моделей на різнорідних текстових і кодових корпусах, що призвело до якісного стрибка у здатності моделей узагальнювати знання та працювати з контекстом.

У подальшому розвиток LLM був зумовлений не лише архітектурними інноваціями, а й зростанням обчислювальних ресурсів, появою великих публічних датасетів та удосконаленням методів навчання. Використання попереднього навчання з подальшим донавчанням, інструкційного навчання та підходів з підкріпленням дозволило адаптувати моделі до широкого спектра прикладних задач. Це сприяло переходу від вузькоспеціалізованих моделей до універсальних генеративних систем, здатних виконувати різнорідні завдання без істотної зміни архітектури.

LLM не “розуміють” текст як людина, а обчислюють імовірність наступного токена (слова або частини слова) на основі попереднього контексту.

Формально модель навчається оцінювати:

$$P(w_t|\, w_1, w_2, \cdots, w_{t-1}). \tag{2.1}$$

де w_1 – перший токен у текстовій послідовності;
w_2 – другий токен у текстовій послідовності;
w_t – наступний токен у текстовій послідовності.

У системах виявлення шкідливого програмного забезпечення на основі генеративних мовних моделей важливу роль відіграє етап токенізації, який забезпечує перетворення програмного коду або супровідних текстових даних у форму, придатну для аналізу нейронною мережею. Оскільки LLM працюють з послідовностями токенів, а не з кодом у його первинному вигляді, якість та спосіб токенізації безпосередньо впливають на ефективність виявлення загроз.

У контексті malware detection токенами можуть виступати різні елементи програмного коду та пов’язаних з ним даних. Це можуть бути окремі слова або

ідентифікатори (наприклад, CreateProcess, DownloadFile), частини слів і символів, а також спеціальні конструкції мови програмування. Застосування субсловної токенізації дозволяє розбивати складні імена функцій, змінних або шкідливих команд на менші смислові фрагменти, що полегшує аналіз навіть у випадках навмисної обфускації коду.

З погляду функціонального призначення генеративні мовні моделі поділяються на кілька основних типів. До універсальних моделей загального призначення належать системи, орієнтовані на роботу з природною мовою та багатофункціональні сценарії використання. Окрему групу становлять спеціалізовані кодові моделі, навчання яких спрямоване на аналіз, генерацію та пояснення програмного коду. Крім того, виокремлюють інструкційні моделі, оптимізовані для виконання завдань за текстовими запитами, а також доменно-орієнтовані моделі, адаптовані до конкретних галузей, зокрема кібербезпеки.

Поява генеративних мовних моделей є закономірним результатом еволюції методів обробки даних, поєднання теоретичних напрацювань у галузі нейронних мереж із практичними досягненнями у сфері обчислювальних технологій. Саме цей історичний і технологічний контекст пояснює, чому LLM стали фундаментом сучасних інтелектуальних систем.

У контексті кібербезпеки та аналізу шкідливого програмного забезпечення LLM розглядаються як інструмент високорівневої інтерпретації, здатний працювати з кодом подібно до людини-аналітика: розуміти логіку виконання програм, призначення функцій, контекст використання API та потенційні наміри розробника. Це відкриває можливості для застосування таких моделей у задачах статичного аналізу, пояснення поведінки malware та класифікації загроз.

2.1.1 Передумови появи Transformer

Моделі типу Transformer беруть свій початок з наукової роботи, опублікованої у 2017 році дослідниками компанії Google, і є одним із найвпливовіших досягнень у сучасному машинному навчанні. Архітектура Transformer була вперше представлена у статті “Attention Is All You Need” [26],

яка згодом стала фундаментальною для розвитку методів обробки природної мови.

Запропонована модель швидко набула практичного застосування, зокрема в бібліотеці Tensor2Tensor фреймворку TensorFlow. Значний внесок у поширення Transformer-архітектури зробила група Harvard NLP, яка підготувала анотований аналіз оригінальної роботи та реалізацію моделі на базі PyTorch. Це сприяло активному розвитку та стандартизації підходів до використання Transformer.

Поява Transformer стала каталізатором стрімкого розвитку напрямку Transformer-based AI та заклала основу для створення великих мовних моделей, зокрема BERT, представленої у 2018 році. Уже на цьому етапі Transformer розглядався як переломний момент у розвитку обробки природної мови.

Подальший значний прорив відбувся у 2020 році з анонсом моделі GPT-3 компанією OpenAI, яка продемонструвала високу універсальність і здатність до генерації тексту, програмного коду та інших типів контенту. У 2021 році дослідники Стенфордського університету запропонували термін foundation models, підкреслюючи фундаментальну роль Transformer-архітектур у сучасному штучному інтелекті.

До появи Transformer домінуючим підходом для роботи з послідовними даними були рекурентні нейронні мережі (RNN), зокрема LSTM. Рекурентні нейронні мережі обробляють вхідні дані послідовно, елемент за елементом, передаючи інформацію між кроками. Такий підхід має принципові обмеження. По-перше, послідовна обробка ускладнює ефективне використання сучасних графічних процесорів, оптимізованих для паралельних обчислень, що значно уповільнює навчання моделей. По-друге, зі збільшенням відстані між елементами послідовності ефективність передачі інформації знижується, що призводить до втрати довгострокових залежностей.

Архітектура Transformer була натхненна схемою encoder–decoder, однак принципово відрізняється від RNN відмовою від рекурентних зв’язків. Замість цього вона повністю базується на механізмі уваги (Attention), який дозволяє моделі враховувати взаємозв’язки між елементами послідовності незалежно від їхнього розташування.

Завдяки цьому Transformer забезпечує ефективне використання паралельних обчислень, високу швидкодію та здатність моделювати складні контекстні залежності. Окрім задач обробки тексту, Transformer-архітектура успішно застосовується для автоматичного реферування, генерації описів зображень і розпізнавання мовлення.

2.1.2 Архітектура Transformer

Архітектура Transformer була спочатку розроблена для задач перетворення послідовностей, зокрема нейронного машинного перекладу, де необхідно трансформувати вхідну послідовність елементів у відповідну вихідну послідовність. Ключовою особливістю Transformer є те, що це перша модель для таких задач, яка повністю базується на механізмі самоуваги без використання рекурентних нейронних мереж або згорткових шарів, прив’язаних до позицій у послідовності. Водночас архітектура Transformer зберігає класичну модель типу «кодер–декодер», що забезпечує розділення процесів аналізу вхідних даних і генерації вихідного результату.

Якщо розглядати Transformer для задач мовного перекладу як умовну «чорну скриньку», то на вході вона приймає речення однією мовою (наприклад, англійською), а на виході формує його переклад іншою мовою. У такому спрощеному уявленні модель виконує перетворення семантичного змісту вхідного тексту у відповідне представлення цільовою мовою.

У більш детальному розгляді ця «чорна скринька» складається з двох основних компонентів: кодера (encoder) та декодера (decoder). Кодер приймає вхідну послідовність і перетворює її у внутрішнє векторне представлення, що містить узагальнену інформацію про зміст і контекст повідомлення. Наприклад, англійське речення “How are you?” на цьому етапі трансформується у багатовимірну матрицю ознак. Декодер використовує це закодоване представлення та поетапно генерує вихідну послідовність, яка на рисунку 2.1 представлена як “¿Cómo estás?”.

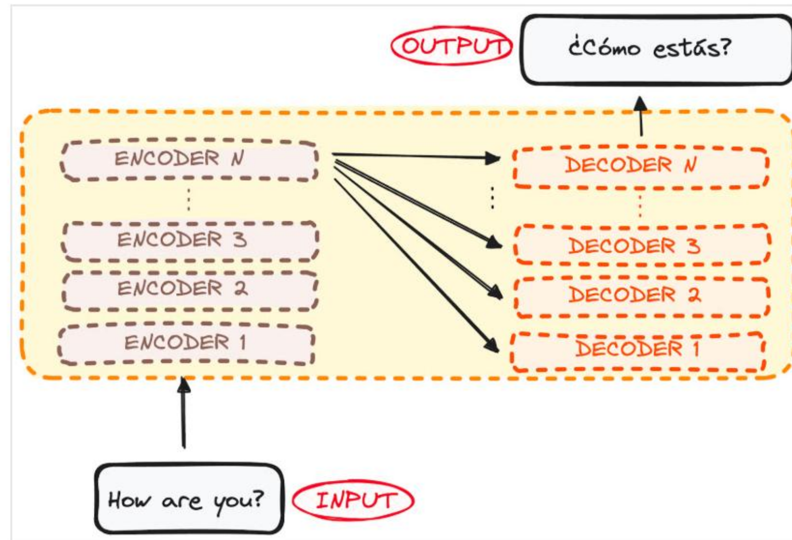

Рисунок 2.1 – Загальна структура архітектури Transformer “кодер–декодер” з N-шарами

Розглянемо послідовно етапи функціонування кодера та декодера.

2.1.2.1 Принцип роботи кодера

Кодер є ключовим компонентом архітектури Transformer. Його основне призначення полягає у перетворенні вхідних токенів на контекстно збагачені представлення. На відміну від попередніх моделей, які обробляли токени ізольовано, кодер Transformer враховує контекст кожного токена відносно всієї послідовності. Структура кодера представлена на рис. 2.2 і включає наступні кроки:

- Вхідні ембединги (Input Embeddings).
- Позиційне кодування (Positional Encoding).
- Багатоголовий механізм самоуваги (Multi-Head Self-Attention Mechanism).
- Нормалізація та резидуальні з’єднання (Normalization and Residual Connections).
- Повнозв’язна нейронна мережа прямого поширення (Feed-Forward Neural Network).

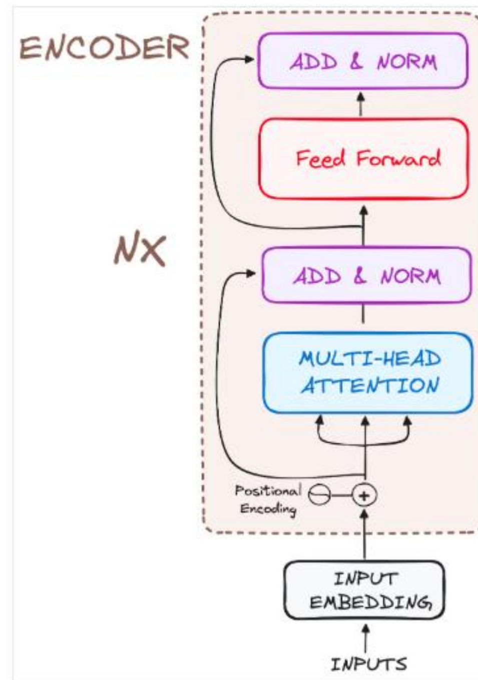

Рисунок 2.2 – Структура кодера [27]

Формування ембеддингів здійснюється лише в нижньому шарі кодера. На цьому етапі вхідні токени перетворюються на векторні представлення за допомогою спеціальних ембеддинг-шарів, які відображають їхній семантичний зміст у числовій формі (рис. 2.3).

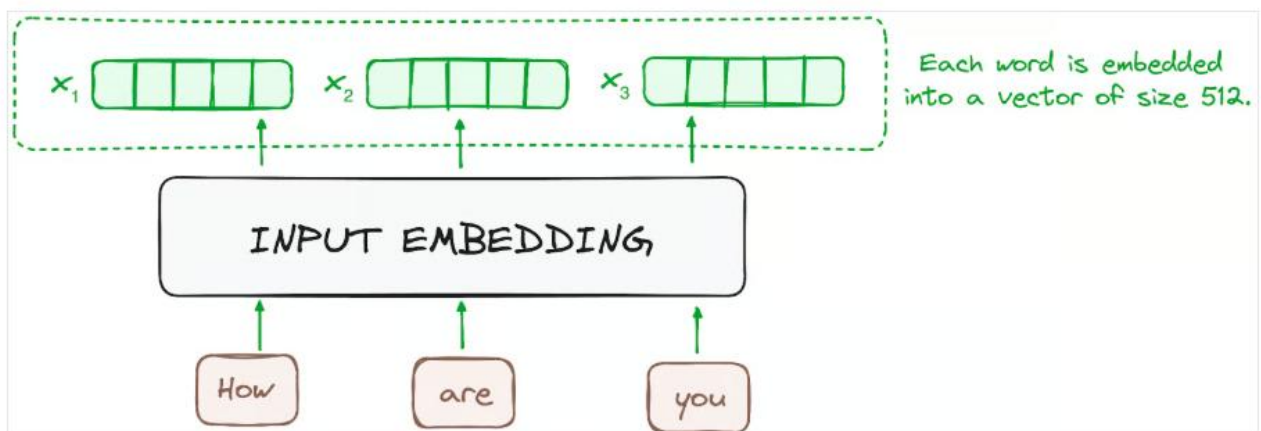

Рисунок 2.3 – Шар вхідних ембедингів [27]

Оскільки архітектура Transformer не містить рекурентного механізму, притаманного RNN, інформація про порядок елементів у послідовності вводиться за допомогою позиційного кодування, яке додається до вхідних

ембеддингів. Це дозволяє моделі враховувати розташування кожного токена в межах послідовності та коректно інтерпретувати структуру речення.

Для реалізації позиційного кодування було запропоновано використовувати комбінацію синусоїдальних і косинусоїдальних функцій, що формують позиційні вектори. Такий підхід забезпечує можливість застосування позиційного кодування до послідовностей довільної довжини без необхідності перенавчання моделі.

У межах цього методу кожен вимір вектора кодується унікальною частотою та фазовим зсувом синусоїдальної хвилі, а значення елементів вектора обмежуються діапазоном від −1 до 1. Це дозволяє однозначно представити позицію кожного токена та зберегти відносні позиційні залежності між елементами послідовності.

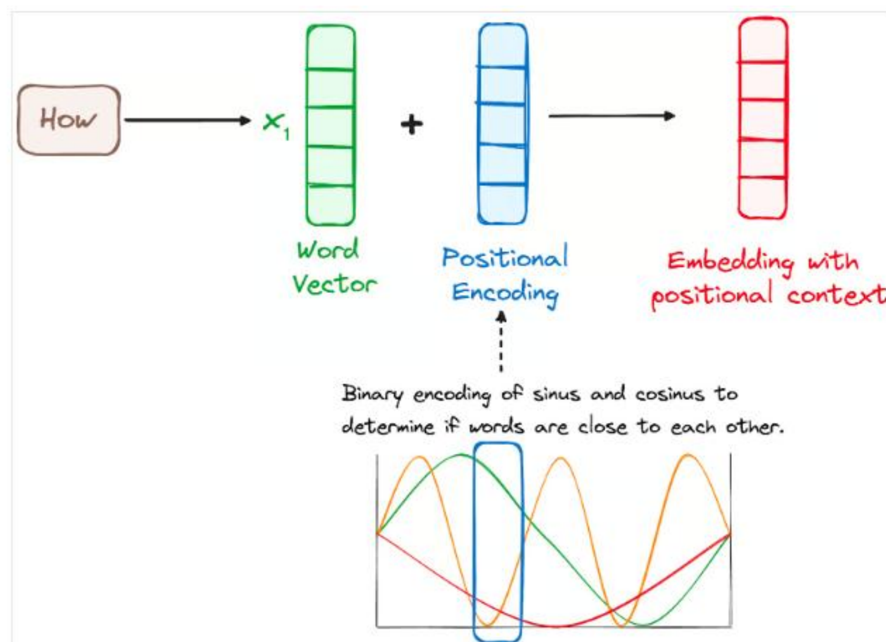

Рисунок 2.4 – Схематичне зображення позиційного кодування [27]

Кодер Transformer складається з набору однакових за структурою шарів, кількість яких у початковій версії архітектури становила шість. Кожен шар кодера призначений для перетворення вхідної послідовності на безперервне абстрактне представлення, яке узагальнює інформацію, отриману з усієї послідовності.

Структурно шар кодера включає два основні підмодулі: багатоголовий механізм самоуваги та повнозв'язну нейронну мережу. Крім того, навколо кожного з підмодулів використовуються резидуальні з'єднання, після яких застосовується нормалізація шару. Таке поєднання забезпечує стабільність навчання та ефективну передачу інформації між шарами.

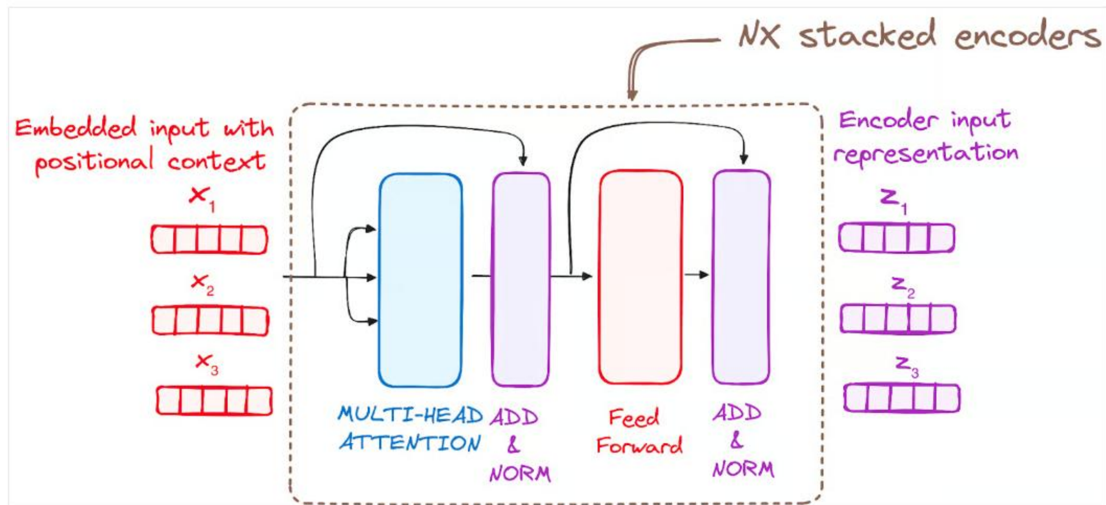

Рисунок 2.5 – Набір шарів енкодера [27]

У кодері Transformer багатоголовий механізм уваги реалізується за допомогою спеціалізованого підходу, відомого як самоувага (self-attention). Цей механізм дозволяє моделі встановлювати взаємозв'язки між усіма словами вхідної послідовності, враховуючи їхній взаємний контекст. Наприклад, у процесі аналізу речення модель може навчитися пов'язувати слово «are» зі словом «you», що є важливим для коректного розуміння змісту.

Самоувага надає кодеру можливість зосереджуватися на різних частинах вхідної послідовності під час обробки кожного токена. Для цього обчислюються коефіцієнти уваги на основі трьох типів векторів. Вектор запиту (query) представляє окремий токен, для якого визначається релевантний контекст. Вектори ключів (keys) відповідають усім токенам послідовності та використовуються для порівняння із запитом. Вектори значень (values), пов'язані з відповідними ключами, беруть участь у формуванні вихідного представлення. Якщо запит і ключ демонструють високу ступінь відповідності, відповідне значення отримує більшу вагу у вихідному результаті.

Перший модуль самоуваги дозволяє моделі захоплювати контекстну інформацію з усієї послідовності одночасно. Замість застосування єдиної функції уваги, вектори запитів, ключів і значень лінійно проєктуються у кілька незалежних підпросторів. Для кожного з них механізм уваги виконується паралельно, що забезпечує формування багатовимірного виходу з h окремих голів уваги. Подальша деталізація архітектури механізму самоуваги подана нижче та представлена на рис. 2.6.

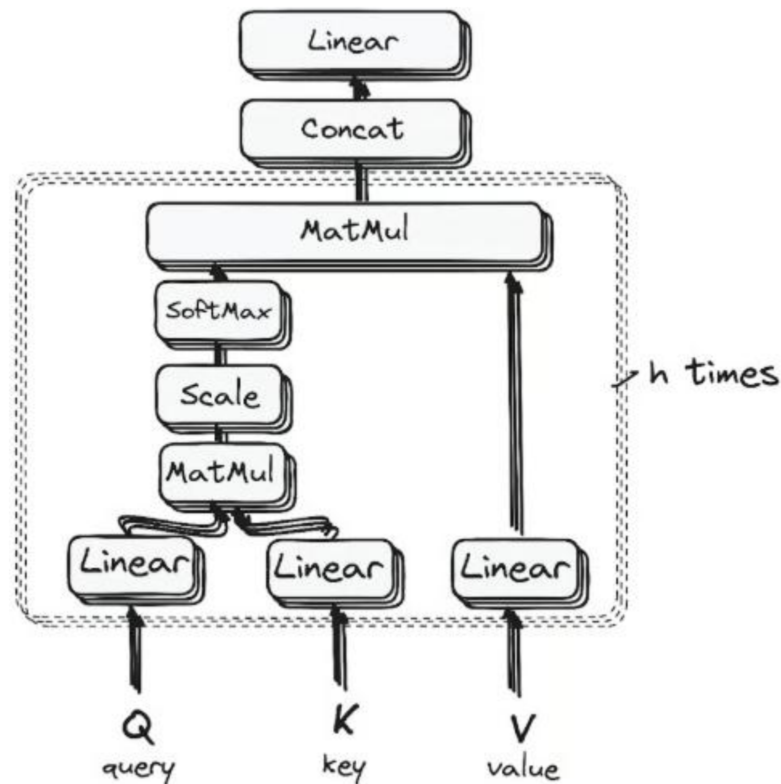

Рисунок 2.6 – Багатоголового механізму самоуваги [27]

Після лінійного перетворення векторів запиту (query), ключа (key) та значення (value) між векторами запитів і ключів виконується матричне множення типу «скалярний добуток», у результаті якого формується матриця оцінок. Ця матриця визначає ступінь впливу кожного токена на інші токени в межах тієї самої послідовності. Вищі значення оцінок свідчать про більшу релевантність відповідних токенів один для одного.

Отримана матриця оцінок фактично відображає відповідність між векторами запитів і ключів. Для запобігання надмірному зростанню значень та забезпечення стабільності градієнтів оцінки масштабуються шляхом ділення на

квадратний корінь із розмірності векторів запиту та ключа. Після цього до масштабованих значень застосовується функція softmax, яка перетворює їх на вагові коефіцієнти у діапазоні від 0 до 1. Softmax підсилює найбільш значущі оцінки та зменшує вплив менш релевантних, що дозволяє моделі ефективно визначати, на які токени слід звертати підвищену увагу.

На наступному етапі отримані вагові коефіцієнти множаться на відповідні вектори значень, у результаті чого формується вихідний вектор механізму уваги. Таким чином, у фінальному представленні зберігається інформація переважно про ті токени, які мають високі значення ваг після застосування softmax. Сформований вихідний вектор додатково проходить через лінійний шар для подальшої обробки, після чого отримується остаточний результат роботи механізму уваги.

Механізм отримав назву багатоголової уваги (Multi-Head Attention) через те, що описаний процес виконується паралельно у кількох незалежних підпросторах, які називаються «головами» уваги. Перед початком обчислень вектори запитів, ключів і значень розбиваються на h окремих наборів, для кожного з яких механізм самоуваги реалізується незалежно. Кожна голова формує власне вихідне представлення, фокусуючись на різних аспектах контексту.

Отримані вихідні вектори з усіх голів об’єднуються та подаються на фінальний лінійний шар, який інтегрує результати паралельної обробки. Завдяки такому підходу модель набуває здатності одночасно враховувати різні типи залежностей у послідовності, що суттєво збагачує контекстне представлення даних і підвищує ефективність роботи кодера Transformer.

Кожен підшар у складі шару кодера доповнюється етапом нормалізації. Крім того, вихід кожного підшару додається до його вхідних даних за допомогою резидуального з’єднання. Такий підхід спрямований на зменшення проблеми зникання градієнта та забезпечує стабільніше навчання глибоких нейронних мереж. Аналогічна процедура застосовується і після повнозв’язної нейронної мережі прямого поширення, що дозволяє зберігати ефективну передачу інформації між шарами та підвищує здатність моделі до масштабування.

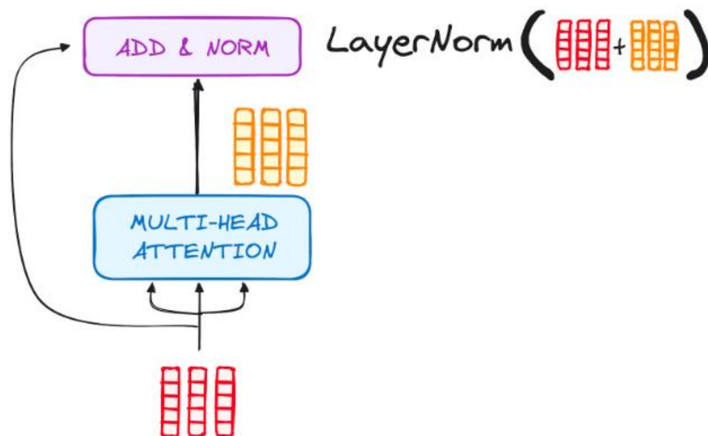

Рисунок 2.7 – Нормалізація та резидуальні з’єднання після багатоголового механізму уваги [27]

Подальша обробка нормалізованого виходу з резидуального з’єднання здійснюється за допомогою повнозв’язної нейронної мережі прямого поширення, яка відіграє важливу роль у додатковому уточненні представлень. Така мережа складається з двох лінійних шарів, між якими розміщується нелінійна функція активації ReLU, що забезпечує моделювання складних нелінійних залежностей (рис. 2.8).

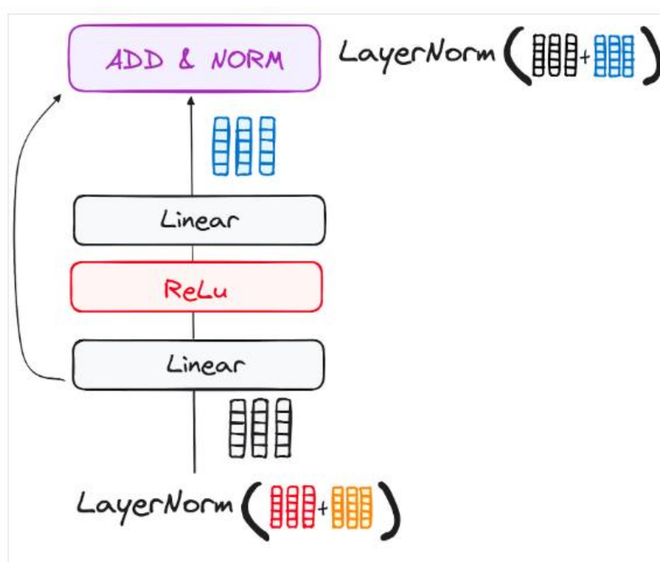

Рисунок 2.8 – Нормалізація та резидуальні з’єднання після багатоголового механізму уваги [27]

Після проходження цього етапу отриманий результат знову поєднується з вхідними даними повнозв’язної мережі за допомогою резидуального з’єднання

(рис. 2.8). Завершальним кроком є повторна нормалізація, яка стабілізує значення активацій та забезпечує узгодженість даних перед передачею до наступних шарів моделі.

Вихід останнього шару кодера являє собою набір векторів, кожен з яких відображає вхідну послідовність з урахуванням глибокого контекстного розуміння. Отримані представлення використовуються як вхідні дані для декодера в архітектурі Transformer. Таке поетапне та ретельне кодування створює основу для ефективної роботи декодера, спрямовуючи його увагу на найбільш релевантні елементи вхідної послідовності під час генерації вихідних даних.

2.1.2.2 Принцип роботи декодера

Основне призначення декодера полягає у формуванні вихідних текстових послідовностей. За своєю структурою він подібний до кодера та складається з аналогічного набору підшарів. Декодер містить два багатоголові механізми уваги, повнозв’язну нейронну мережу прямого поширення, а також використовує резидуальні з’єднання й нормалізацію шару після кожного підшару (рис 2.9). Така архітектурна побудова забезпечує ефективну обробку інформації та стабільність навчання під час генерації вихідної послідовності.

Зазначені компоненти функціонують подібно до шарів кодера, проте з певними відмінностями: кожен із багатоголових механізмів уваги в декодері виконує окрему, специфічну роль у процесі обробки даних. Такий поділ функцій дозволяє ефективніше поєднувати інформацію з попередньо згенерованих токенів і контекст, отриманий з кодера.

Завершальним етапом роботи декодера є лінійний шар, який виконує роль класифікатора. До його виходу застосовується функція softmax, що обчислює ймовірності появи можливих токенів вихідної послідовності. Саме на цьому етапі визначається наступний елемент тексту, який буде згенеровано моделлю.

Архітектура декодера Transformer спеціально спроєктована для поетапної генерації вихідної послідовності шляхом декодування закодованої інформації.

Декодер працює в авторегресивному режимі, ініціюючи процес зі спеціального стартового токена. Надалі він використовує послідовність уже згенерованих вихідних токенів як вхідні дані, поєднуючи їх із контекстними представленнями, сформованими кодером.

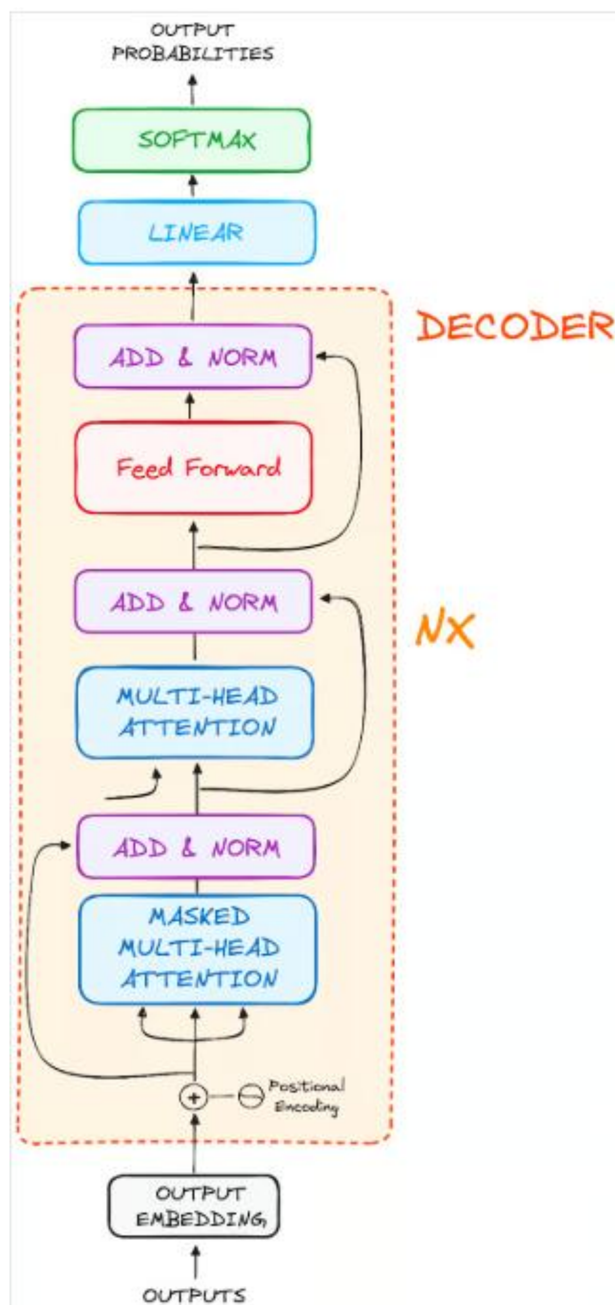

Рисунок 2.9 – Структура декодера [27]

Такий послідовний процес генерації триває до моменту створення спеціального токена, що сигналізує про завершення формування вихідної послідовності.

На початковому етапі роботи декодера процес багато в чому повторює функціонування кодера. Вхідна послідовність спочатку проходить через шар ембеддингів. Після цього, аналогічно до кодера, до отриманих ембеддингів додається позиційне кодування, внаслідок чого формуються позиційно збагачені векторні представлення, що містять інформацію про порядок токенів у послідовності.

Сформовані позиційні ембеддинги передаються до першого багатоголового механізму уваги декодера, у якому обчислюються коефіцієнти уваги, характерні саме для вхідної послідовності декодера. Архітектурно декодер складається зі стеку однакових за структурою шарів кожен з яких містить три основні підкомпоненти:

- Маскований механізм самоуваги (Masked Self-Attention Mechanism).
- Багатоголовий механізм уваги між кодером і декодером (Encoder–Decoder Multi-Head Attention або Cross-Attention).
- Повнозв'язна нейронна мережа прямого поширення (Feed-Forward Neural Network).

Механізм самоуваги є подібним до відповідного механізму в кодері, проте з принциповою відмінністю: у декодері застосовується маскування, яке забороняє токенам звертати увагу на наступні елементи послідовності. Це означає, що під час обчислення уваги кожен токен може враховувати лише попередні або поточні позиції, але не майбутні.

Наприклад, під час обчислення коефіцієнтів уваги для слова «are» модель не має доступу до слова «you», оскільки воно розташоване пізніше у послідовності. Такий підхід забезпечує коректну авторегресивну генерацію тексту та запобігає витоку інформації з майбутніх токенів у процесі декодування.

У другому багатоголовому механізмі уваги декодера реалізується специфічна взаємодія між компонентами кодера та декодера. На цьому етапі вихідні представлення кодера використовуються як вектори запитів і ключів, тоді як виходи першого шару багатоголової уваги декодера виступають у ролі векторів значень. Така конфігурація забезпечує узгодження вхідної інформації

кодера з поточним станом декодера та надає можливість виділяти найбільш релевантні фрагменти закодованої послідовності.

Завдяки цьому механізму декодер отримує змогу ефективно співвідносити генеровані токени з відповідними частинами вхідної послідовності, що суттєво підвищує якість формування вихідного результату. У межах цього підшару вектори запитів надходять з попереднього шару декодера, тоді як ключі та значення формуються на основі виходу кодера. Це дозволяє кожній позиції у декодері звертати увагу на всі позиції вхідної послідовності, інтегруючи інформацію з обох компонентів архітектури.

Після завершення обчислень у другому шарі багатоголової уваги отримані представлення додатково обробляються повнозв’язною нейронною мережею прямого поширення. Аналогічно до кодера, кожен шар декодера містить таку мережу, яка застосовується незалежно та однаково до кожної позиції послідовності. Це сприяє подальшому уточненню ознак і забезпечує ефективне поєднання контекстної інформації, отриманої з кодера та декодера.

Завершальним етапом роботи Transformer є передача даних через фінальний лінійний шар, який виконує функцію класифікатора. Розмір цього шару відповідає кількості класів, тобто кількості слів у словнику моделі. Отриманий вектор подається на функцію softmax, яка перетворює його на розподіл імовірностей у діапазоні від 0 до 1. Токен із найбільшою ймовірністю обирається як наступний елемент вихідної послідовності.

Після генерації нового токена декодер додає його до наявної послідовності вхідних даних і продовжує процес декодування. Цей цикл повторюється доти, доки модель не згенерує спеціальний токен завершення, який сигналізує про кінець формування вихідної послідовності.

Важливо зазначити, що декодер може складатися з N послідовних шарів, кожен з яких поєднує інформацію з кодера та попередніх шарів декодера. Така багатошарова структура дозволяє моделі враховувати різноманітні шаблони уваги, що сприяє глибшому контекстному розумінню та підвищує точність прогнозування.

2.1.3 Типи та класифікація LLM

Сучасні LLM можна класифікувати за кількома ознаками. За архітектурним підходом виділяють універсальні мовні моделі загального призначення (наприклад, GPT, Claude) та спеціалізовані кодові моделі, орієнтовані на аналіз програмного коду (Code Llama, StarCoder, Codex) [28]. За способом розгортання моделі поділяються на хмарні та локальні, що має принципове значення для безпеки й обробки конфіденційних даних [29].

Окрему групу становлять open-source LLM, які можуть бути адаптовані та донавчені під специфічні задачі аналізу malware, а також закриті комерційні моделі, що пропонують високу якість результатів, але мають обмеження щодо прозорості та контролю.

У 2018 році компанія Google представила модель BERT (Bidirectional Encoder Representations from Transformers) – відкриту мовну модель, яка стала суттєвим проривом у галузі обробки природної мови [30]. Ключовою інновацією BERT є двобічне (bidirectional) навчання, що дозволяє моделі враховувати контекст слова одночасно з лівого та правого боку, на відміну від попередніх підходів, які аналізували текст лише в одному напрямку.

Завдяки повноцінному контекстному розумінню BERT продемонструвала значно кращі результати у задачах відповідей на запитання, аналізу неоднозначних формулювань і семантичного розуміння тексту. В основі моделі лежить архітектура Transformer, яка забезпечує динамічні зв’язки між усіма елементами вхідної послідовності. Попереднє навчання BERT на великих корпусах текстів, зокрема Wikipedia, зробило її універсальним інструментом для широкого спектра NLP-задач. Висока ефективність моделі зумовила її інтеграцію в пошукову систему Google для покращення обробки природномовних запитів, а також стимулювала подальший розвиток мовних моделей нового покоління.

LaMDA (Language Model for Dialogue Applications) – це Transformer-орієнтована мовна модель, розроблена Google спеціально для діалогових застосувань і представлена у 2021 році [31]. Основною метою створення LaMDA

було підвищення природності, зв’язності та контекстної релевантності відповідей у діалогових системах.

Архітектура LaMDA дозволяє моделі ефективно працювати з широким спектром тем і намірів користувачів, що робить її придатною для використання в чат-ботах, віртуальних асистентах та інтерактивних AI-системах. Орієнтація на підтримку довготривалого діалогу та збереження контексту виділяє LaMDA серед інших мовних моделей і підкреслює її значення як етапу розвитку AI-орієнтованих комунікаційних технологій.

Моделі сімейства GPT (Generative Pre-trained Transformer), розроблені компанією OpenAI, належать до сучасного класу великих генеративних мовних моделей, орієнтованих на глибоке контекстне розуміння та генерацію природної мови. Починаючи з першої версії, представленої у 2018 році, архітектура GPT зазнала суттєвого розвитку, що привело до створення масштабних моделей нового покоління з розширеними можливостями узагальнення, логічного міркування та роботи з багатомодальними даними.

ChatGPT є спеціалізованою реалізацією GPT, оптимізованою для інтерактивної взаємодії з користувачем. Вона поєднує базову генеративну модель із механізмами інструкційного навчання та узгодження відповідей із намірами користувача. Це забезпечує більш природний діалог, підвищену стабільність відповідей і кращу інтерпретованість результатів. Завдяки цим властивостям ChatGPT широко застосовується в чат-ботах, інтелектуальних асистентах, системах підтримки користувачів, а також у прикладних задачах аналізу програмного коду та кібербезпеки.

Модель T5 (Text-to-Text Transfer Transformer), запропонована дослідниками Google, базується на уніфікованому підході до обробки мовних задач, у межах якого всі завдання формулюються у форматі “текст → текст” [32]. T5 побудована на класичній архітектурі Transformer з чітко вираженою схемою кодер–декодер, що відрізняє її від декодер-орієнтованих моделей типу GPT. Кодер відповідає за глибоке контекстне розуміння вхідного тексту, тоді як декодер формує вихідну послідовність. Завдяки цьому T5 особливо добре підходить для задач, де важливе точне зіставлення вхідної та вихідної інформації, зокрема для семантичного

аналізу, структурованого перетворення тексту та пояснення результатів. У контексті кібербезпеки T5 може застосовуватися для класифікації описів загроз, перетворення технічних логів у пояснювальний текст, а також для автоматичного формулювання висновків за результатами аналізу шкідливого коду.

Gemini представляє сучасне сімейство великих мовних моделей, розроблене компанією Google як еволюційний наступник попередніх моделей [33], зокрема LaMDA та PaLM. Ключовою відмінністю Gemini є її мультимодальна природа, тобто здатність працювати не лише з текстом, а й з іншими типами даних, такими як зображення, код, таблиці та структурована інформація.

Архітектурно Gemini також спирається на Transformer-підхід, однак оптимізована для глибшого міркування, інтеграції різнорідних джерел даних та складних аналітичних задач. Це робить модель придатною не лише для діалогових сценаріїв, а й для задач, що потребують багатокрокового аналізу, інтерпретації та прийняття рішень.

У сфері аналізу шкідливого програмного забезпечення потенціал Gemini полягає у можливості поєднувати аналіз програмного коду з контекстною інформацією, наприклад, документацією, поведінковими логами або результатами динамічного аналізу. Такий підхід створює передумови для більш комплексних систем виявлення загроз, які здатні враховувати як синтаксичні, так і семантичні аспекти malware.

2.2 Можливості LLM для аналізу шкідливої поведінки програмного коду

Сучасне шкідливе програмне забезпечення характеризується високим рівнем складності, динамічністю та активним використанням технік маскування. Обфускація, шифрування, динамічне завантаження компонентів і використання легітимних системних API значно ускладнюють аналіз malware традиційними методами [34]. У цих умовах генеративні мовні моделі відкривають нові можливості для аналізу програмного коду на семантичному рівні, дозволяючи

перейти від поверхневого пошуку ознак до інтерпретації логіки виконання та намірів шкідливого програмного забезпечення [35].

На відміну від класичних інструментів, які переважно працюють із синтаксичними шаблонами або статистичними ознаками, LLM здатні аналізувати програмний код як цілісну логічну систему. Завдяки попередньому навчанню на великих обсягах програмного коду такі моделі розпізнають типові структури програм, взаємозв'язки між функціями, умовні переходи та послідовності викликів API [35].

У задачах аналізу шкідливого ПЗ LLM можуть використовуватися для:

- ідентифікації основних етапів виконання програми (ініціалізація, підготовка середовища, основна шкідлива активність);
- виявлення прихованих або неочевидних фрагментів коду, зокрема за умов сильної обфускації;
- реконструкції логіки виконання навіть у разі фрагментованого або частково зашифрованого коду.

Завдяки контекстному аналізу LLM здатні встановлювати причинно-наслідкові зв'язки між різними частинами програми, що є складним завданням для класичних статичних аналізаторів.

Однією з ключових переваг використання LLM є можливість автоматизованої інтерпретації результатів аналізу. Генеративні моделі можуть не лише визначати потенційно шкідливі фрагменти, а й формувати текстові пояснення щодо їх призначення та поведінки. Це суттєво підвищує інтерпретованість результатів і знижує когнітивне навантаження на аналітика [36].

Explainable malware analysis дозволяє:

- пояснювати, які саме дії виконує програма та з якою метою;
- інтерпретувати складні ланцюжки викликів API у зрозумілій для людини формі;
- формувати структуровані звіти для SOC-центрів або експертів з цифрової криміналістики.

Таким чином, LLM можуть виконувати роль інтелектуального асистента, який допомагає швидше зрозуміти поведінку malware та ухвалити обґрунтовані рішення щодо реагування на інциденти.

Генеративні мовні моделі демонструють високу ефективність у виявленні шкідливих патернів, які не завжди мають чіткі сигнатури. Завдяки здатності аналізувати контекст використання API та сценарії виконання, LLM можуть виявляти аномальні або підозрілі поведінкові послідовності [35,36].

Зокрема, моделі здатні:

- розпізнавати нетипове поєднання системних викликів, характерне для malware;
- виявляти приховані механізми завантаження додаткових компонентів;
- аналізувати поведінкові сценарії, що свідчать про спроби ескалації привілеїв, збору даних або встановлення стійкості у системі.

Такий підхід є особливо ефективним для виявлення zero-day загроз, коли відсутні відомі сигнатури або навчальні приклади.

Практичне застосування генеративних мовних моделей у задачах аналізу шкідливого програмного забезпечення може бути проілюстроване на прикладах типових сценаріїв, з якими стикаються фахівці з кібербезпеки. Одним із таких сценаріїв є аналіз обфускованого програмного коду. У цьому випадку LLM отримує фрагмент, наприклад, JavaScript-коду з навмисно ускладненою структурою, зміненими іменами змінних і функцій та прихованими рядковими константами. Завдяки семантичному аналізу модель здатна відновити логіку виконання програми, пояснити механізми декодування рядків і ідентифікувати функціональні блоки, відповідальні за завантаження прихованого шкідливого компонента. Такий підхід дозволяє значно скоротити час аналізу у порівнянні з ручним розбором або класичними статичними інструментами.

Іншим характерним прикладом є інтерпретація викликів легітимних системних API у виконуваних файлах. У багатьох сучасних зразках malware шкідлива активність маскується під звичайну взаємодію з операційною системою, що ускладнює її виявлення. Генеративна мовна модель аналізує не лише окремі виклики API, а й контекст їх використання та послідовність

виконання. На основі цього формується узагальнений висновок про поведінку програми, зокрема виявляється, що поєднання на перший погляд безпечних операцій відповідає типовому сценарію збору облікових даних та їх подальшої передачі на віддалений сервер управління.

Особливу цінність LLM демонструють у випадках аналізу раніше невідомих зразків шкідливого програмного забезпечення, для яких відсутні сигнатури або навчальні приклади. У таких умовах модель працює у zero-shot режимі, аналізуючи загальну логіку виконання програми, структуру коду та поведінкові ознаки. На основі семантичних залежностей і виявлених аномалій LLM здатна класифікувати програму як потенційно шкідливу та надати пояснення щодо причин такого висновку. Це робить генеративні мовні моделі ефективним інструментом для протидії zero-day загрозам і підвищує адаптивність сучасних систем кіберзахисту.

Використання генеративних мовних моделей у задачах аналізу шкідливого програмного забезпечення дозволяє поєднати семантичний аналіз коду, інтерпретованість результатів і контекстне виявлення загроз. LLM розширюють можливості класичних методів, забезпечуючи більш глибоке розуміння поведінки malware та підвищуючи ефективність сучасних систем кіберзахисту, особливо в умовах швидкої еволюції загроз.

2.3 Порівняльний аналіз традиційних методів аналізу шкідливого програмного забезпечення та підходів на основі LLM

Аналіз шкідливого програмного забезпечення є ключовим елементом сучасних систем кіберзахисту. Протягом тривалого часу основою такого аналізу залишалися традиційні підходи, зокрема статичний і динамічний аналіз, а також методи машинного навчання. Водночас зростання складності malware, активне використання обфускації, автоматизованої генерації коду та zero-day технік поступово знижують ефективність класичних методів, що зумовлює необхідність пошуку нових інтелектуальних рішень.

Проаналізувавши архітектурні особливості та функціональні можливості великих мовних моделей, а також узагальнивши результати дослідження класичних методів аналізу шкідливого програмного забезпечення, розглянутих у першому розділі роботи, було сформовано порівняльну таблицю 2.1.

Таблиця 2.1 – Порівняльна таблиця методів аналізу шкідливого ПЗ

Критерій	Статистичний аналіз	Динамічний аналіз	ML-підходи	LLM-підходи
Аналіз без виконання коду	Так	Ні	Так	Так
Стійкість до обфускації	Низька	Середня	Середня	Висока
Виявлення zero-day загроз	Обмежене	Часткове	Можливе	Високе
Потреба у розмічених даних	Низька	Низька	Висока	Низька / відсутня
Інтерпретованість результатів	Обмежена	Середня	Низька	Висока
Масштабованість	Висока	Низька	Середня	Середня
Обчислювальні витрати	Низькі	Високі	Середні	Високі

Генеративні мовні моделі пропонують принципово інший підхід до аналізу шкідливого програмного забезпечення. На відміну від класичних методів, LLM працюють не лише з формальними ознаками, а й із семантичним змістом програмного коду. Вони здатні аналізувати логіку виконання, взаємозв’язки між функціями, контекст використання API та потенційні наміри розробника.

Однією з ключових переваг LLM є здатність працювати у zero-shot та few-shot режимах, що дозволяє виявляти нові типи загроз без необхідності повного перенавчання моделі. Крім того, LLM забезпечують високий рівень інтерпретованості, оскільки можуть генерувати текстові пояснення результатів

аналізу, що є особливо важливим для аналітиків SOC-центрів і цифрової криміналістики.

Разом із тим застосування LLM супроводжується низкою викликів, серед яких значні обчислювальні витрати, ризик генеративних помилок та необхідність захисту конфіденційних даних. Тому на практиці найбільш перспективним є гібридний підхід, що поєднує традиційні методи аналізу з можливостями генеративних мовних моделей.

Порівняльний аналіз показує, що жоден із підходів не є універсальним рішенням для аналізу шкідливого програмного забезпечення. Традиційні методи залишаються ефективними для швидкого та масового виявлення відомих загроз, тоді як генеративні мовні моделі відкривають нові можливості для глибокого семантичного аналізу та інтерпретації складних і раніше невідомих зразків malware. Поєднання класичних підходів із LLM дозволяє подолати їхні взаємні обмеження та сформувати більш адаптивні й інтелектуальні системи кіберзахисту.

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ LLM ПІДХОДІВ ДЛЯ ВИЯВЛЕННЯ ШКІДЛИВОГО ПЗ

3.1 Обґрунтування вибору LLM моделі

Для практичної реалізації та порівняльного аналізу LLM-підходів до виявлення шкідливого програмного коду в межах даного дослідження було обрано три сімейства генеративних мовних моделей: GPT, LLaMA та Mistral. Такий вибір зумовлений необхідністю охоплення різних підходів до архітектури, ліцензування, способів розгортання та практичного використання великих мовних моделей у задачах кібербезпеки.

Моделі сімейства GPT, розроблені компанією OpenAI, є одними з найбільш зрілих і широко застосовуваних великих мовних моделей у промислових та дослідницьких системах. Вони демонструють високий рівень розуміння програмного коду, здатність до семантичного аналізу та пояснення логіки виконання програм.

У контексті аналізу шкідливого ПЗ GPT виступає еталонною (reference) моделлю, що дозволяє оцінити максимально можливу якість результатів за умов використання потужних комерційних LLM. Крім того, GPT-моделі добре справляються з аналізом обфускованого коду та складних поведінкових патернів, що є критично важливим для сучасного malware.

Сімейство моделей LLaMA, розроблене компанією Meta, було обрано як представник open-source LLM, орієнтованих на локальне або приватне розгортання. LLaMA забезпечує баланс між якістю мовного аналізу та можливістю повного контролю над моделлю, що є особливо актуальним для середовищ із підвищеними вимогами до безпеки та конфіденційності. У межах дослідження LLaMA використовується для оцінки того, наскільки відкриті моделі можуть конкурувати з комерційними рішеннями у задачах виявлення шкідливого коду, а також для аналізу їх придатності до інтеграції в корпоративні системи кіберзахисту.

Моделі сімейства Mistral, розроблені компанією Mistral AI, представляють нове покоління компактних та високоефективних LLM. Їх ключовою особливістю є оптимальне співвідношення між обчислювальною складністю та якістю результатів, що робить Mistral перспективними для практичного використання в реальних системах аналізу програмного коду. У дослідженні Mistral використовується для перевірки гіпотези про те, що менш ресурсомісткі LLM можуть забезпечувати достатній рівень точності при аналізі malware, знижуючи при цьому вимоги до апаратних ресурсів та часу обробки.

Попри значний внесок моделей сімейства BERT у розвиток обробки природної мови, їх застосування в даному дослідженні було визнано недоцільним з огляду на специфіку задачі аналізу шкідливого програмного коду. BERT належить до класу двонаправлених енкодерних моделей, які оптимізовані передусім для задач класифікації, пошуку відповідностей та аналізу коротких текстових фрагментів.

Ключовим обмеженням BERT є відсутність генеративного механізму. У контексті аналізу malware це означає, що модель не здатна формувати пояснення логіки виконання коду, інтерпретувати поведінкові патерни або виконувати семантичний аналіз у формі природномовного висновку. Натомість генеративні LLM дозволяють не лише класифікувати код як шкідливий або легітимний, але й пояснювати причини такого рішення, що є важливим для аналітиків кібербезпеки.

Ще одним суттєвим обмеженням моделей типу BERT є жорстке обмеження довжини контексту. Аналіз реальних зразків шкідливого програмного забезпечення часто вимагає обробки великих фрагментів коду, включно з кількома функціями, імпортами та допоміжними модулями. У таких умовах енкодерні моделі демонструють зниження ефективності або потребують складних схем сегментації, що ускладнює експериментальну методологію.

Окрім BERT, у межах дослідження також не використовувалися інші спеціалізовані або класичні мовні моделі, зокрема RoBERTa, ALBERT та XLNet. Хоча ці моделі демонструють покращені результати в окремих NLP-задачах,

вони зберігають базові архітектурні обмеження енкодерного підходу і не підтримують повноцінний генеративний аналіз програмного коду.

Для наочного обґрунтування вибору генеративних мовних моделей та порівняння їхніх можливостей із класичними енкодерними підходами в роботі наведено порівняльну таблицю 3.1.

Таблиця 3.1 – Порівняльна характеристика генеративних мовних моделей для аналізу шкідливого коду

Критерій	BERT	GPT	LLaMA	Mistral
Архітектурний тип	Енкодер	Декодер	Декодер	Декодер
Генерація тексту	Ні	Так	Так	Так
Аналіз програмного коду	Обмежений	Високий	Високий	Високий
Семантичне розуміння коду	Низьке	Високе	Високе	Високе
Пояснення результатів аналізу	Ні	Так	Так	Так
Підтримка довгого тексту	Обмежена	Висока	Середня-висока	Середня
Робота з обфускованим кодом	Низька ефективність	Висока ефективність	Середня-висока	Середня-висока
Zero-day malware	Обмежено	Ефективно	Ефективно	Потенційо ефективно
Можливість локального розгортання	Так	Обмежено	Так	Так
Обчислювальні вимоги	Низьку	Високі	Середні	Низькі-середні
Потреба в розмічених даних	Висока	Низька	Низька	Низька

Також не розглядалися вузькоспеціалізовані моделі аналізу коду, орієнтовані на статичні ознаки або сигнатури, оскільки метою дослідження було оцінити універсальність генеративних LLM у виявленні шкідливого ПЗ, зокрема в умовах обфускації та zero-day атак.

Відмова від використання BERT та подібних моделей обґрунтовується їх архітектурними обмеженнями, відсутністю генеративних можливостей та недостатньою гнучкістю для глибокого семантичного аналізу шкідливого програмного коду. Натомість обрані генеративні мовні моделі забезпечують комплексний підхід до виявлення malware, поєднуючи класифікацію, інтерпретацію та пояснення результатів аналізу.

На основі порівняльного аналізу встановлено, що моделі енкодерного типу, зокрема BERT, є менш придатними для задач семантичного аналізу шкідливого програмного коду через відсутність генеративних можливостей та обмежену інтерпретованість результатів. Натомість генеративні мовні моделі GPT, LLaMA та Mistral забезпечують глибший аналіз логіки виконання коду, здатність до пояснення рішень і кращу адаптацію до zero-day загроз, що обґрунтовує їх вибір для проведення експериментального дослідження.

Обрані моделі охоплюють три ключові класи LLM:

- високопродуктивну комерційну модель (GPT);
- відкриту модель для локального розгортання (LLaMA);
- оптимізовану легковагову модель нового покоління (Mistral).

Це забезпечує репрезентативність експерименту, дозволяє провести коректний порівняльний аналіз та зробити обґрунтовані висновки щодо доцільності використання різних LLM у практичних задачах аналізу шкідливого програмного забезпечення.

3.2 Формування набору даних дослідження

Формування набору даних для дослідження здійснювалося з урахуванням вимог до репрезентативності, відтворюваності та безпеки аналізу шкідливого програмного забезпечення. З цією метою було обрано підхід, що поєднує

використання реальних зразків malware з відкритих спеціалізованих репозиторіїв та легітимного програмного коду з відкритих джерел, що дозволяє забезпечити коректне порівняння результатів роботи великих мовних моделей.

Як джерело шкідливого програмного коду у роботі використано репозиторій MalwareBazaar, який підтримується ініціативою abuse.ch і є одним з найбільш відомих відкритих сховищ актуальних зразків шкідливого програмного забезпечення. MalwareBazaar містить реальні зразки malware, зібрані з різних джерел, зокрема бекдори, трояни, завантажувачі (loaders), RAT, кейлогери та інші типи шкідливого ПЗ. Для кожного зразка надаються метадані, включно з хеш-значеннями, класифікацією за типом загрози та інформацією про сімейство malware, що дозволяє здійснювати попередню фільтрацію та відбір зразків для дослідження (рис. 3.1).

Date (UTC)	SHA256 hash	Type	Signature	Tags	Reporter	DL
2025-12-17 18:07	366968ef624762bcf310a...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:07	56f8152c006db50eef333...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	2b803b85b6759e178e51...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:07	7d512927221cce6452c9e...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	181c55b6bd3394bee056...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	9e28b7e224075dafbca32...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:07	46fbeebcb4adcfc99bfb0...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	4a32a8b8ec443ec841e5f...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:07	d8a8a85eaaf96e8340654...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	4cd555350acc35b0066f7...	elf	Mirai	elf mirai UPX	abuse_ch	
2025-12-17 18:07	740a18e3bb9cfcecf723a...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:07	9d317cd6fdf5b07d6ce50...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:03	88a412a5bb0d7a3d701c...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:03	71df1b9f826adc72d81c1...	elf	Mirai	elf mirai	abuse_ch	
2025-12-17 18:03	c79acaac0594f7ba4785a...	elf	Mirai	elf mirai	abuse_ch	

Рисунок 3.1 – Приклад записів репозиторію MalwareBazaar із зразками шкідливого ПЗ

З огляду на специфіку застосування великих мовних моделей, використання бінарних файлів у первинному вигляді є недостатньо інформативним. Тому в межах даного дослідження зразки з MalwareBazaar застосовувалися виключно у статичному вигляді. Для аналізу LLM використовувався або вихідний код скриптів (у випадку, якщо він був доступний), або дизасембльований чи

декомпільований код, отриманий без виконання шкідливих зразків. Такий підхід дозволяє представити програмний код у текстовій формі, придатній для семантичного аналізу мовними моделями, та водночас виключає ризики, пов'язані з фактичним запуском malware.

Для формування класу легітимного програмного коду використовувалися відкриті репозиторії платформи GitHub. До вибірки включалися приклади коду, що не містять зловмисної логіки та за своєю структурою і стилем є подібними до шкідливих зразків (скрипти автоматизації, робота з файлами, мережею, системними викликами). Такий підхід унеможливлює спрощену класифікацію за поверхневими ознаками та дозволяє оцінити здатність LLM аналізувати саме семантику та поведінкову логіку коду.

Важливим аспектом формування набору даних стало балансування класів шкідливого та нешкідливого програмного коду. Кількість зразків malware та легітимного коду підбиралася таким чином, щоб уникнути домінування одного з класів, що могло б призвести до зміщення результатів експерименту..

Станом на момент проведення дослідження (грудень 2025) репозиторій MalwareBazaar містив понад 1 023 480 зразків шкідливого програмного забезпечення. З огляду на обчислювальні обмеження та необхідність забезпечення контрольованого експерименту, для виконання кваліфікаційної роботи було відібрано репрезентативну підмножину з 20 000 зразків malware, що охоплювали різні типи шкідливого програмного забезпечення. Для забезпечення балансу класів та коректності порівняльного аналізу додатково було сформовано вибірку з 20 000 зразків легітимного програмного коду, отриманих з відкритих репозиторіїв GitHub. Збалансований датасет забезпечує коректну оцінку точності, повноти та інших метрик ефективності мовних моделей під час порівняльного аналізу.

Після попередньої обробки набір даних було збережено у форматі CSV, який містив назву файлу програмного забезпечення, його текстове представлення та відповідну мітку класу. У межах кваліфікаційної роботи весь датасет було поділено на тренувальну та тестову вибірки, що продемонстровано у лістингу 3.1.

Лістинг 3.1 – Поділ датасету на train та test

```
train_df, test_df = train_test_split(
    df,
    test_size=0.2,
    random_state=42,
    stratify=df["label"]
)
train_df.to_csv("train.csv", index=False)
test_df.to_csv("test.csv", index=False)
```

Сформований набір даних, що поєднує зразки з MalwareBazaar та легітимний код з GitHub, створює надійну та безпечну основу для експериментального дослідження. Використання вихідного або статично обробленого коду дозволяє ефективно застосовувати великі мовні моделі для аналізу шкідливого програмного забезпечення, не порушуючи принципів безпеки та етичних вимог дослідження.

3.3 Методика застосування LLM для аналізу та класифікації коду

Методика експериментального дослідження базується на чіткому розмежуванні етапів підготовки даних, налаштування взаємодії з великими мовними моделями та фінального оцінювання результатів (рис. 3.2). На першому етапі сформований датасет програмного коду було поділено на тренувальну та тестову вибірки. При цьому тренувальна вибірка не використовувалася для навчання або перенавчання великих мовних моделей, оскільки у межах даного дослідження не застосовувалися методи fine-tuning. Натомість тренувальний набір слугував виключно джерелом репрезентативних прикладів для формування few-shot контексту та для підбору і стабілізації інструкцій, які використовуються у prompt.

У межах даного дослідження для аналізу та класифікації програмного коду було обрано підхід few-shot, а не zero-shot. Вибір few-shot методики зумовлений необхідністю підвищення стабільності відповідей великих мовних моделей та

забезпечення уніфікованого формату результатів. На відміну від zero-shot підходу, за якого модель отримує лише загальну інструкцію без прикладів, few-shot передбачає надання мовній моделі обмеженої кількості зразків із правильними відповідями без зміни її внутрішніх параметрів. Це дозволяє моделі краще інтерпретувати поставлене завдання та застосовувати однакові критерії під час аналізу нового коду.

Рисунок 3.2 – Схематичне представлення алгоритму застосування LLM

Лістинг 3.2 – Приклад zero-shot prompt

```
You are a cybersecurity analyst.
Analyze the following code and determine whether it is malicious or benign.
If malicious, explain which behaviors indicate malware activity.
Code:
import os
import socket
s = socket.socket()
s.connect(("192.168.1.10", 4444))
os.system("/bin/sh")
```

Лістинг 3.3 – Приклад few-shot prompt

```
You are a cybersecurity analyst.
Example 1:
Code:
```

Продовження лістингу 3.3

```
import requests
requests.get("https://example.com/api/status")
Classification: benign
Explanation: The code performs a simple HTTP request without harmful behavior.
Example 2:
Code:
import os
os.system("rm -rf / --no-preserve-root")
Classification: malicious
Explanation: The code executes a destructive system command that can delete system files.
Now analyze the following code and classify it as malicious or benign.
Provide a short explanation.
Code:
import os
import socket
s = socket.socket()
s.connect(("192.168.1.10", 4444))
os.system("/bin/sh")
```

На другому етапі для кожної з обраних мовних моделей формувався few-shot контекст на основі тренувальної вибірки. До нього включалися кілька прикладів шкідливого та легітимного програмного коду з правильними мітками, що дозволяло задати моделі чіткий зразок очікуваного формату відповіді та критеріїв класифікації. Важливою умовою експерименту було використання однакових few-shot прикладів для всіх моделей, що забезпечує рівні умови тестування та унеможливлює спотворення результатів порівняння.

Наступним кроком здійснювався підбір оптимального prompt. Для цього порівнювалися кілька варіантів інструкцій до мовних моделей, які відрізнялися формулюванням завдання та вимогами до структури відповіді. Критеріями вибору оптимального prompt були стабільність формату відповідей, мінімальний

рівень випадкових або суперечливих рішень та чіткість пояснень, які надає модель. Після визначення найбільш ефективного варіанта prompt він фіксувався та надалі не змінювався, що є необхідною умовою коректного експериментального дослідження.

На четвертому етапі зафіксований prompt разом із few-shot прикладами застосовувався до тестової вибірки. Тестові зразки ніколи не використовувалися у few-shot контексті або на етапі підбору інструкцій, що гарантує відсутність витоку інформації та забезпечує об’єктивність оцінювання. Кожен зразок тестового набору аналізувався незалежно, а результатом роботи мовної моделі було рішення щодо належності коду до класу шкідливого або легітимного програмного забезпечення.

Завершальним етапом методики було оцінювання якості класифікації на основі тестової вибірки. Для цього обчислювалася точність класифікації (accuracy), а також додаткові метрики, зокрема precision, recall, F1-score та матриця помилок (confusion matrix). Використання комплексу метрик дозволяє всебічно оцінити ефективність кожної мовної моделі та забезпечує надійну основу для їх подальшого порівняльного аналізу.

3.4 Застосування досліджуваних LLM

У межах даного підрозділу здійснюється послідовне застосування обраних великих мовних моделей GPT, LLaMA та Mistral до одного й того самого датасету з метою забезпечення коректного порівняльного аналізу їх ефективності. Всі експерименти проводилися відповідно до уніфікованої методики, описаної у попередніх підрозділах, що передбачає поділ набору даних на тренувальну та тестову вибірки, використання few-shot підходу на основі тренувального набору та оцінювання результатів виключно на тестовій вибірці.

На першому етапі сформований датасет, що містив текстове представлення програмного коду та відповідні мітки класів, було поділено на тренувальну і тестову вибірки із збереженням балансу між шкідливими та легітимними зразками. Тренувальна вибірка не використовувалася для навчання або

перенавчання мовних моделей, а слугувала джерелом прикладів для формування few-shot контексту та для підбору оптимального формулювання prompt. Після фіксації остаточного варіанта prompt він застосовувався без змін до всієї тестової вибірки.

Для аналізу кожного зразка програмного коду використовувався few-shot підхід, за якого мовній моделі попередньо надавалися кілька прикладів з тренувального набору з правильними мітками класів. Такий підхід дозволяє моделі краще інтерпретувати поставлене завдання та дотримуватися стабільного формату відповідей. Важливою умовою експерименту було використання однакових few-shot прикладів та ідентичного prompt для всіх моделей, що забезпечує рівні умови тестування.

Для ілюстрації реалізації методики у лістингу 3.4 наведено фрагмент програмного коду, який демонструє формування few-shot контексту з тренувальної вибірки та подальше застосування мовної моделі до тестових зразків.

Лістинг 3.4 – Формування few-shot прикладів з тренувального набору

```
def build_few_shot(train_df, n_per_class=2):
    shots = []
    for label in ["malicious", "benign"]:
        subset = train_df[train_df["label"] == label].head(n_per_class)
        for _, row in subset.iterrows():
            shots.append({
                "role": "user",
                "content": f"Analyze the code and classify it.\n\nCode:\n{row['code']}"
            })
            shots.append({
                "role": "assistant",
                "content": f"Classification: {label}"
            })
    return shots
```

Після формування few-shot контексту кожен зразок тестової вибірки послідовно подавався до мовної моделі разом із зафіксованим prompt. Результатом роботи моделі було рішення щодо належності коду до класу шкідливого або легітимного програмного забезпечення. Для подальшого аналізу ці рішення автоматично зберігалися разом із правильними мітками класів.

У лістингу 3.5 наведено приклад реалізації циклу тестування мовної моделі на тестовій вибірці та збору результатів класифікації.

Лістинг 3.5 – Тестування мовної моделі на тестовій вибірці

```
results = []
for _, row in test_df.iterrows():
    response = classify_with_gpt(
        code=row["code"],
        few_shot_examples=few_shot_context
    )
    results.append({
        "true_label": row["label"],
        "predicted_label": response
    })
```

Після завершення аналізу всієї тестової вибірки здійснювався підрахунок кількості правильних та неправильних класифікацій, а також обчислення основних метрик якості. У лістингу 3.6 наведено приклад розрахунку точності класифікації (accuracy), яка використовується як базовий показник ефективності мовної моделі.

Лістинг 3.5 – Тестування мовної моделі на тестовій вибірці

```
from sklearn.metrics import accuracy_score
y_true = [r["true_label"] for r in results]
y_pred = [r["predicted_label"] for r in results]
accuracy = accuracy_score(y_true, y_pred)
print(f"Accuracy: {accuracy:.4f}")
```

Детальні лістинги щодо застосування до конкретної LLM моделі наведені у додатках Б, В, Г. Усі три пайплайни ідентичні за вищепредставленою методикою тому результати моделей є коректно порівнюваними.

3.5 Порівняльний аналіз результатів експериментального дослідження

У даному підрозділі представлено порівняльний аналіз результатів, отриманих у процесі застосування різних великих мовних моделей до одного й того самого датасету програмного коду. Усі моделі тестувалися відповідно до єдиної методики, що передбачала використання few-shot підходу на основі тренувальної вибірки, фіксованого prompt та оцінювання якості класифікації виключно на тестовому наборі. Це забезпечує коректність і відтворюваність отриманих результатів, а також дозволяє здійснити об’єктивне порівняння ефективності моделей.

Основними метриками оцінювання були обрані точність класифікації (accuracy), повнота (recall) та точність позитивних передбачень (precision). Зазначені показники дозволяють оцінити не лише загальну правильність роботи моделі, але й її здатність коректно виявляти шкідливі зразки програмного коду та мінімізувати кількість хибних спрацьовувань. Особлива увага приділялася саме recall для класу шкідливого програмного забезпечення, оскільки пропуск malware-зразків становить найбільшу загрозу в практичних системах кіберзахисту. Результати представлені у таблиці 3.2

Таблиця 3.2 – Порівняльні показники ефективності моделей LLM

Модель	Accuracy	Precision	Recall
GPT	0,94	0,93	0,95
LLaMA	0,90	0,89	0,91
Mistral	0,87	0,85	0,88

Для детальнішого аналізу результатів класифікації, отриманих за допомогою різних великих мовних моделей, додатково використовувалася

матриця помилок (confusion matrix), що представлені таблицями 3.3-3.5 відповідно. Зазначена матриця дозволяє оцінити не лише загальну точність класифікації, але й характер допущених помилок, що є особливо важливим у задачах виявлення шкідливого програмного забезпечення.

Таблиця 3.3 – Матриця помилок для моделі GPT

Фактичний клас / Передбачений клас	Malware	Benign
Malware	3800	200
Benign	280	3720

Таблиця 3.4 – Матриця помилок для моделі LLaMa

Фактичний клас / Передбачений клас	Malware	Benign
Malware	3640	360
Benign	440	3560

Таблиця 3.5 – Матриця помилок для моделі Mistral

Фактичний клас / Передбачений клас	Malware	Benign
Malware	3520	480
Benign	620	3380

За результатами експерименту модель GPT продемонструвала найвищі показники за всіма основними метрками. Це відображається у матриці помилок, де переважають коректні класифікації (TP і TN), а кількість FN є мінімальною. Це свідчить про її здатність глибоко аналізувати семантику програмного коду, виявляти приховані шкідливі патерни та надавати стабільні результати навіть у складних і прикордонних випадках. Водночас ця модель потребує значних обчислювальних ресурсів і залежить від хмарної інфраструктури, що може бути обмеженням для деяких сценаріїв використання.

Модель LLaMA показала дещо нижчі результати порівняно з GPT, однак продемонструвала стабільну роботу та достатньо високу здатність до виявлення типових ознак шкідливого коду. Її ключовою перевагою є можливість

локального розгортання та повний контроль над даними, що робить LLaMA привабливою для використання в корпоративних або ізольованих середовищах із підвищеними вимогами до безпеки.

Модель Mistral продемонструвала найнижчі значення метрик серед досліджуваних LLM, однак при цьому забезпечила найвищу швидкодію та найменші вимоги до обчислювальних ресурсів. Найбільша кількість FN серед досліджуваних моделей вказує на нижчу здатність Mistral виявляти складні зразки malware. Зростання FP також свідчить про зниження точності класифікації легітимного коду, що обмежує використання моделі у сценаріях із високими вимогами до надійності. Результати свідчать, що Mistral є ефективною для швидкої первинної фільтрації зразків програмного коду, проте може поступатися більш потужним моделям у складних випадках, що потребують глибокого семантичного аналізу.

Найвищі показники accuracy, precision та recall були отримані для моделі GPT, що дозволяє розглядати її як найбільш ефективне рішення для задач глибокого аналізу шкідливого програмного забезпечення. Модель LLaMA демонструє оптимальний компроміс між якістю результатів і можливістю локального розгортання, тоді як Mistral доцільно використовувати у сценаріях, де пріоритетом є швидкодія та обмежені обчислювальні ресурси.

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Розроблене в межах кваліфікаційної роботи рішення за темою "Генеративні мовні моделі в аналізі шкідливого коду" належить до програмно-аналітичних засобів у сфері кібербезпеки та інформаційних технологій. Результати дослідження передбачають застосування великих мовних моделей (LLM) для аналізу програмного коду, ідентифікації ознак шкідливого програмного забезпечення, класифікації зразків malware та підтримки процесів кіберзахисту. Виконання робіт з розроблення, тестування та експлуатації такого програмного рішення здійснюється із використанням персональних комп'ютерів, серверних або хмарних обчислювальних ресурсів і не пов'язане з технологічними процесами підвищеної небезпеки, використанням небезпечних речовин або фізично шкідливого обладнання. Основні потенційні ризики мають офісний та інформаційно-технологічний характер.

Організація безпечних умов праці під час виконання та практичного застосування результатів кваліфікаційної роботи регламентується чинним законодавством України, зокрема:

- Законом України «Про охорону праці» [37];
- Кодексом цивільного захисту України [38];
- Правилами пожежної безпеки в Україні [39];
- Вимогами щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями (НПАОП 0.00-7.15-18) [40];
- Правилами безпечної експлуатації електроустановок (НПАОП 40.1-1.01-97) [41];

Під час виконання робіт з аналізу шкідливого коду із застосуванням LLM можливий вплив таких факторів:

- фізіологічні та ергономічні фактори;
- електричні фактори;

- пожежонебезпечні фактори;
- психоемоційне навантаження.

До фізіологічних та ергономічних факторів належить тривала робота за комп’ютером, статичне навантаження, перенапруження зору, зниження рухової активності. Дані фактори характерні для інтелектуальної діяльності в IT-сфері та регламентуються нормативами щодо роботи з екранними пристроями.

Ризик ураження електричним струмом у разі пошкодження ізоляції, використання несправних блоків живлення, порушення правил підключення обладнання або перевантаження електромережі складають електричні фактори, яким може піддатись працівник під час виконання роботи.

До пожежонебезпечних факторів належить можливе загоряння електрообладнання внаслідок короткого замикання, перегріву компонентів, накопичення пилу або використання несертифікованих пристроїв.

Аналіз шкідливого коду, інцидентів інформаційної безпеки та зразків malware пов’язаний з високою концентрацією уваги, відповідальністю та значним інформаційним навантаженням, що відображають психоемоційне навантаження на працівника.

Забезпечення безпечних та нешкідливих умов праці під час виконання робіт, пов’язаних з аналізом шкідливого програмного коду із застосуванням генеративних мовних моделей, повинно здійснюватися відповідно до вимог чинного законодавства України у сфері охорони праці.

Відповідно до статті 13 Закону України «Про охорону праці», роботодавець зобов’язаний створити на кожному робочому місці умови праці відповідно до нормативно-правових актів з охорони праці та забезпечити додержання вимог безпеки під час експлуатації обладнання.

Основним нормативно-правовим актом, що регламентує вимоги до умов праці та забезпечення безпеки і збереження здоров’я працівників під час роботи з екранними пристроями, зокрема рідкокристалічними моніторами, є НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями». Згідно з цими Вимогами, робоче місце користувача екранних пристроїв повинно відповідати таким умовам:

- екранний пристрій має бути розміщений на відстані, що забезпечує комфортне зорове сприйняття інформації, з можливістю регулювання положення по висоті та куту нахилу;
- конструкція робочого столу та крісла повинна забезпечувати зручну робочу позу, з мінімальним статичним навантаженням на хребет, шию та верхні кінцівки;
- робоча поверхня повинна мати достатні розміри для розміщення обладнання та документів без створення незручних поз;
- освітлення робочого місця має бути достатнім і рівномірним, без прямих відблисків на екрані та без різких контрастів яскравості.

Зазначені вимоги спрямовані на профілактику зорової втоми, професійних захворювань опорно-рухового апарату та зниження ризиків, пов'язаних із тривалою роботою за комп'ютером.

Відповідно до пунктів НПАОП 0.00-7.15-18, роботодавець зобов'язаний організувати режим праці таким чином, щоб запобігти надмірному психофізіологічному навантаженню працівників, які постійно працюють з екранними пристроями. Під час виконання робіт з аналізу програмного коду та результатів роботи генеративних мовних моделей рекомендується:

- обмежувати тривалість безперервної роботи з екранними пристроями;
- встановлювати регламентовані перерви для відпочинку, зміни виду діяльності та відновлення працездатності;
- забезпечувати можливість виконання активних рухів та вправ для зниження статичного навантаження під час перерв.

Зазначені вимоги узгоджуються із загальними принципами збереження здоров'я працівників, визначеними Законом України «Про охорону праці», та спрямовані на профілактику перевтоми і професійного вигорання.

Електрообладнання, яке використовується для виконання робіт з аналізу шкідливого програмного коду, повинно експлуатуватися відповідно до Правил безпечної експлуатації електроустановок споживачів (НПАОП 40.1-1.01-97).Зокрема, необхідно:

- використовувати справне, сертифіковане комп’ютерне та мережеве обладнання;
- забезпечувати наявність захисного заземлення та автоматичних пристроїв захисту;
- забороняти експлуатацію обладнання з пошкодженими кабелями або ознаками перегріву;
- дотримуватися вимог щодо навантаження електромереж і правил підключення електроприладів.

Під час експлуатації комп’ютерної техніки та програмно-апаратних засобів, що використовуються для аналізу шкідливого програмного коду із застосуванням генеративних мовних моделей, необхідно забезпечити дотримання вимог протипожежної безпеки відповідно до чинного законодавства України.

Основним нормативно-правовим актом, який регламентує вимоги пожежної безпеки на об’єктах незалежно від форми власності та виду діяльності, є Правила пожежної безпеки в Україні, затверджені наказом Міністерства внутрішніх справ України № 1417 від 30.12.2014.

Згідно з вимогами зазначених Правил, під час експлуатації комп’ютерного та електронного обладнання на робочих місцях повинні виконуватись такі заходи пожежної безпеки:

- забороняється перевантаження електричних мереж шляхом підключення надмірної кількості електроприладів до однієї розетки або подовжувача, що може призвести до перегріву та короткого замикання;
- повинен здійснюватися регулярний контроль технічного стану комп’ютерної техніки, блоків живлення, мережевого обладнання та інших електроприладів з метою своєчасного виявлення несправностей;
- не допускається використання пошкоджених електричних кабелів, подовжувачів, розеток або пристроїв з ознаками перегріву, іскріння чи порушення ізоляції;

- приміщення, в яких експлуатується комп’ютерна техніка, повинні бути забезпечені первинними засобами пожежогасіння (вогнегасниками) відповідно до встановлених норм;
- шляхи евакуації та виходи повинні бути постійно вільними, доступними та не захаращеними обладнанням або сторонніми предметами.

Крім того, забороняється залишати без нагляду ввімкнене електрообладнання, яке не призначене для тривалої безперервної роботи, а також використовувати саморобні або несертифіковані електричні пристрої.

У ході виконання кваліфікаційної роботи встановлено, що застосування генеративних мовних моделей для аналізу шкідливого програмного коду належить до безпечних видів інтелектуальної діяльності в сфері інформаційних технологій. Розроблене програмно-аналітичне рішення не створює додаткових виробничих ризиків та може експлуатуватися в стандартних офісних умовах. За умови дотримання вимог чинного законодавства України у сфері охорони праці, електробезпеки та пожежної безпеки забезпечуються безпечні та нешкідливі умови праці користувачів. Отримані результати можуть бути впроваджені в практичну діяльність без зміни класу умов праці.

4.2 Характеристика стихійних лих, аварій (катастроф) та їх наслідків

У XXI столітті проблема забезпечення безпеки життєдіяльності населення набула особливої актуальності у зв’язку зі зростанням кількості та масштабів надзвичайних ситуацій природного і техногенного характеру. Глобальні кліматичні зміни, інтенсифікація промислового виробництва, урбанізація, зростання щільності населення та ускладнення інженерної інфраструктури істотно підвищують рівень ризиків, пов’язаних зі стихійними лихами, аваріями та катастрофами. Особливої гостроти ця проблема набула в умовах повномасштабної війни між Україною та сусідньою країною на Сході, унаслідок якої на території України систематично виникають надзвичайні ситуації техногенного та комбінованого характеру, спричинені ракетними та дроновими ударами по об’єктах енергетичної, транспортної, промислової та житлової

інфраструктури. Значна кількість катастроф супроводжується руйнуванням критично важливих систем життєзабезпечення, пожежами, вибухами, витоками небезпечних речовин і масштабними гуманітарними наслідками. У таких умовах навіть локальні за масштабом події можуть призводити до значних людських, матеріальних та екологічних втрат, що обумовлює необхідність комплексного наукового аналізу надзвичайних ситуацій у контексті сучасних безпекових викликів.

Надзвичайна ситуація (НС) – це порушення нормальних умов життя і діяльності людей на окремій території або об’єкті, спричинене аварією, катастрофою, стихійним лихом або іншою небезпечною подією, що призводить або може призвести до загибелі людей, шкоди здоров’ю, матеріальних втрат і погіршення стану навколишнього природного середовища.

Згідно з прийнятою класифікацією, надзвичайні ситуації поділяються за такими основними ознаками:

- за походженням;
- за масштабами поширення;
- за швидкістю розвитку;
- за характером уражальних факторів.

За походженням НС поділяються на:

- природні (стихійні лиха);
- техногенні (аварії та катастрофи);
- соціально-політичні;
- воєнні.

Стихійні лиха – це небезпечні природні явища або процеси, що виникають у результаті дії сил природи та характеризуються значною руйнівною здатністю. Вони можуть мати як раптовий, так і поступовий характер розвитку, однак у будь-якому випадку призводять до серйозних порушень життєдіяльності населення.

Особливістю стихійних лих є їх об’єктивний характер, тобто незалежність від волі та діяльності людини. Водночас антропогенний вплив, а саме: вирубка

лісів, забудова заплав річок, порушення гідрологічного режиму, зміна клімату значно посилює масштаби та наслідки природних катастроф.

Землетруси належать до найбільш небезпечних стихійних лих, оскільки характеризуються раптовістю виникнення та значною інтенсивністю уражальних факторів. Основною причиною землетрусів є тектонічні процеси в земній корі, зокрема зсуви літосферних плит.

Наслідки землетрусів включають:

- руйнування будівель і споруд;
- пошкодження транспортної та інженерної інфраструктури;
- виникнення пожеж, вибухів, витоків газу;
- зсуви, обвали, цунамі.

Для магістерського рівня аналізу надзвичайних ситуацій принципово важливим є врахування вторинних уражальних факторів землетрусів, оскільки саме вони в більшості випадків зумовлюють основну частину людських жертв, матеріальних збитків та довготривалих соціально-економічних наслідків. На відміну від первинних сейсмічних коливань, які мають відносно короткочасний вплив, вторинні фактори характеризуються каскадним і пролонгованим характером дії, що значно ускладнює процеси реагування та ліквідації наслідків.

До основних вторинних факторів землетрусів належать пожежі, вибухи, руйнування інженерних комунікацій, зсуви, обвали, цунамі, підтоплення, а також техногенні аварії на промислових і енергетичних об’єктах. Зокрема, пошкодження газопроводів та електричних мереж часто призводить до масових пожеж у щільно забудованих міських районах, які можуть тривати значно довше за сам сейсмічний поштовх і охоплювати великі території. Руйнування водопровідних систем унеможливлює оперативне гасіння пожеж, що ще більше посилює масштаби збитків.

Повені належать до найбільш поширених і водночас соціально та економічно небезпечних стихійних лих у світі. Їх виникнення зумовлюється комплексом гідрометеорологічних і антропогенних чинників, серед яких основними є тривалі або інтенсивні атмосферні опади, швидке танення снігового покриву, утворення льодових заторів на річках, а також аварії або руйнування

гідротехнічних споруд, зокрема дамб, гребель і водосховищ. У сучасних умовах глобальних кліматичних змін спостерігається тенденція до зростання частоти та інтенсивності повеней, що підсилює їхній негативний вплив на території з високою щільністю населення.

Наслідки повеней мають комплексний характер і охоплюють різні сфери життєдіяльності суспільства. Затоплення житлових, промислових і соціальних об’єктів призводить до значних матеріальних збитків, втрати житла населенням та порушення функціонування систем життєзабезпечення. Руйнування автомобільних і залізничних доріг, мостів, інженерних мереж водо-, газо- та електропостачання істотно ускладнює проведення рятувальних і відновлювальних робіт, а також спричиняє тривалі перебої в роботі інфраструктури. Значної шкоди зазнає сільське господарство, оскільки повені призводять до загибелі посівів, деградації ґрунтів та порушення агровиробничих циклів.

Окрему небезпеку становить забруднення поверхневих і підземних джерел питної води внаслідок потрапляння у воду побутових, промислових і сільськогосподарських стоків. Це створює передумови для погіршення санітарно-епідеміологічної ситуації, виникнення спалахів інфекційних захворювань та зростання медико-біологічних ризиків для населення. У післяпаводковий період такі наслідки можуть зберігатися тривалий час і потребують значних ресурсів для їх ліквідації.

Особливо небезпечними з точки зору цивільного захисту є раптові паводки, які характеризуються стрімким підйомом рівня води та обмеженими можливостями прогнозування. Відсутність достатнього часу для організованої евакуації населення значно підвищує рівень загрози життю та здоров’ю людей і ускладнює роботу аварійно-рятувальних служб. У зв’язку з цим для магістерського рівня досліджень актуальним є питання вдосконалення систем моніторингу, прогнозування та раннього оповіщення про повені як ключових елементів зниження ризиків надзвичайних ситуацій.

До окремої групи стихійних лих належать атмосферні небезпечні явища, які формуються внаслідок інтенсивних процесів у тропосфері та характеризуються

значною руйнівною дією. До них відносяться урагани, буревії, смерчі, сильні зливи, град і хуртовини. Спільною ознакою цих явищ є наявність потужних уражальних факторів, серед яких провідне місце посідає висока швидкість вітру, що здатна спричиняти масштабні механічні пошкодження будівель, інженерних споруд та об’єктів інфраструктури.

Руйнівний вплив атмосферних небезпечних явищ проявляється у зриві покрівель, падінні дерев і опор ліній електропередач, пошкодженні транспортних засобів та виробничих об’єктів. Ураганні вітри й смерчі можуть призводити до повного знищення окремих будівель і значних змін у планувальній структурі населених пунктів. Сильні зливи та хуртовини, у свою чергу, ускладнюють роботу транспорту, сприяють виникненню аварійних ситуацій на шляхах сполучення та створюють загрозу життю і здоров’ю населення.

Атмосферні небезпечні явища часто супроводжуються тривалими відключеннями електроенергії, порушенням функціонування систем зв’язку, водо- та теплопостачання, що призводить до збоїв у роботі систем життєзабезпечення міст і промислових об’єктів. У зимовий період такі явища, як хуртовини та обледеніння, можуть паралізувати транспортне сполучення на значних територіях і суттєво ускладнювати діяльність аварійно-рятувальних служб.

Техногенні надзвичайні ситуації посідають особливе місце серед загроз безпеці життєдіяльності населення, оскільки безпосередньо пов’язані з функціонуванням складних технічних систем і об’єктів виробничої та інфраструктурної сфери. Техногенна аварія визначається як порушення нормального режиму роботи технічної системи, обладнання або об’єкта, що призводить або може призвести до виникнення небезпечних чинників, матеріальних збитків і загрози життю та здоров’ю людей. Катастрофа є найбільш тяжкою формою техногенної аварії та характеризується масштабними руйнуваннями, значною кількістю людських жертв, а також довготривалими соціально-економічними й екологічними наслідками, ліквідація яких потребує значних ресурсів і часу.

Серед техногенних аварій особливу небезпеку становлять промислові аварії, що виникають на підприємствах із підвищеним рівнем ризику, зокрема в хімічній, енергетичній, нафтохімічній та металургійній галузях. Такі аварії супроводжуються дією потужних уражальних факторів, серед яких провідне місце посідають токсичні викиди небезпечних хімічних речовин, вибухи, пожежі, а також теплове й хімічне ураження людей та об’єктів довкілля. Наслідки промислових аварій мають, як правило, комплексний характер і можуть виходити за межі виробничого майданчика, поширюючись на прилеглі території.

Особливу загрозу для населення становлять аварії на промислових об’єктах, розташованих поблизу густонаселених районів або в межах великих міст. У таких випадках навіть локальні за масштабом інциденти здатні призводити до масового ураження людей, необхідності проведення евакуаційних заходів і тривалого обмеження використання територій. Це зумовлює підвищені вимоги до систем промислової безпеки, моніторингу та управління ризиками на об’єктах підвищеної небезпеки.

Вагомою складовою техногенних надзвичайних ситуацій є транспортні катастрофи, які охоплюють аварії на автомобільному, залізничному, морському та авіаційному транспорті. Їх характерною особливістю є раптовість виникнення та висока ймовірність масової загибелі людей. Особливо небезпечними є транспортні катастрофи, пов’язані з перевезенням небезпечних вантажів, таких як вибухові, легкозаймисті, токсичні або радіоактивні речовини. У таких випадках наслідки аварій можуть мати не лише локальний, а й регіональний характер, охоплюючи значні території.

Найбільш складними та небезпечними з точки зору довгострокових наслідків є радіаційні аварії, що пов’язані з викидом або неконтрольованим поширенням іонізуючого випромінювання. Радіоактивне забруднення навколишнього природного середовища призводить до ураження людей, порушення природних екосистем, деградації земель і обмеження або повної неможливості використання забруднених територій протягом десятиліть. Особливістю радіаційних аварій є наявність віддалених медико-біологічних

ефектів, зокрема зростання онкологічних захворювань та генетичних порушень, що підсилює їх соціальну й екологічну небезпеку.

Таким чином, техногенні аварії та катастрофи характеризуються високим рівнем складності, багатофакторністю та значними масштабами потенційних наслідків, що зумовлює необхідність їх комплексного аналізу в межах магістерських досліджень із безпеки життєдіяльності, цивільного захисту та техногенно-екологічної безпеки.

Наслідки стихійних лих та катастроф мають багатовимірний характер і охоплюють соціальну, економічну, екологічну та медико-біологічну сфери суспільного життя. Їх масштаб і тривалість залежать від інтенсивності події, рівня підготовленості території та ефективності систем реагування.

Соціальні наслідки надзвичайних ситуацій проявляються у загибелі та травмуванні людей, зростанні рівня інвалідизації, вимушеній міграції населення, а також виникненні психологічних травм і соціальної напруги. Масштабні катастрофи призводять до порушення звичних умов життя, зниження рівня соціальної стабільності та ослаблення довіри населення до інституцій державного управління.

Економічні наслідки стихійних лих і катастроф пов’язані з руйнуванням основних виробничих фондів, втратою виробничого потенціалу та зростанням державних і місцевих витрат на ліквідацію наслідків і відновлення інфраструктури. Для країн із перехідною або такою, що розвивається, економікою подібні втрати можуть мати довготривалий негативний вплив на темпи економічного зростання та соціально-економічний розвиток у цілому.

Екологічні наслідки надзвичайних ситуацій проявляються у деградації природних екосистем, забрудненні атмосферного повітря, поверхневих і підземних вод, а також ґрунтового покриву. У поєднанні з цим виникають медико-біологічні наслідки, що включають погіршення санітарно-епідеміологічної ситуації, спалахи інфекційних захворювань, отруєння та віддалені негативні ефекти для здоров’я населення.

Стихійні лиха, аварії та катастрофи є невід’ємною складовою сучасних ризиків для суспільства. Їхній вплив виходить далеко за межі безпосередніх

руйнувань, охоплюючи соціальну, економічну, екологічну та медико-біологічну сфери. Зростання частоти та масштабів надзвичайних ситуацій вимагає комплексного підходу до управління ризиками, що включає наукове прогнозування, удосконалення систем моніторингу, підвищення рівня підготовки населення та розвитку ефективних механізмів реагування. Лише системна та превентивна політика у сфері цивільного захисту здатна зменшити негативні наслідки стихійних лих і катастроф та забезпечити стійкий розвиток суспільства.

ВИСНОВКИ

У кваліфікаційній роботі проведено комплексне дослідження можливостей застосування генеративних мовних моделей для виявлення та класифікації шкідливого програмного коду. Досягнута мета роботи полягає у підтвердженні доцільності використання сучасних великих мовних моделей як інструменту семантичного аналізу коду в умовах обмеженої кількості прикладів навчання та високої варіативності шкідливих програм.

У ході виконання роботи здійснено ґрунтовний аналіз сучасного стану проблеми виявлення malware та існуючих підходів до його аналізу. Показано, що традиційні сигнатурні та rule-based методи втрачають ефективність унаслідок активного використання обфускації, поліморфізму та динамічного завантаження коду. Класичні підходи машинного навчання, хоча й здатні узагальнювати ознаки, значною мірою залежать від якості попередньо сформованих характеристик і великих обсягів розмічених даних. У цьому контексті обґрунтовано перспективність використання генеративних мовних моделей, здатних аналізувати програмний код як цілісну логічну структуру.

У другому розділі розглянуто теоретичні та прикладні засади використання генеративних мовних моделей у задачах аналізу шкідливого програмного забезпечення. Визначено, що LLM є сучасним класом моделей штучного інтелекту, здатних формувати контекстно залежні представлення як природної мови, так і програмного коду, що забезпечує перехід від поверхневого аналізу ознак до семантичної інтерпретації логіки виконання програм.Детально проаналізовано структуру Transformer у вигляді схеми «кодер–декодер», принципи функціонування енкодера й декодера, а також роль позиційного кодування, багатоголової уваги, резидуальних з’єднань і нормалізації в забезпеченні стабільності та ефективності навчання. Виконано порівняльний аналіз традиційних методів (статистичний та динамічний аналіз, ML-підходи) і підходів на основі LLM. Встановлено, що LLM мають переваги у стійкості до обфускації, здатності до zero-shot/few-shot узагальнення та високій

інтерпретованості, проте характеризуються значними обчислювальними витратами і ризиками генеративних помилок.

У третьому розділі виконано практичну реалізацію застосування великих мовних моделей для виявлення та класифікації шкідливого програмного коду, а також проведено порівняльний аналіз результатів для моделей GPT, LLaMA та Mistral. Коректність порівняння забезпечено уніфікованою методикою: однаковим форматом даних, фіксованим prompt та ідентичними few-shot прикладами для всіх моделей.

Такою обґрунтовано вибір трьох сімейств LLM як репрезентативних для різних сценаріїв використання в кібербезпеці: GPT як еталонне комерційне рішення з максимальною якістю, LLaMA як open-source модель для локального розгортання та Mistral як ресурсно-оптимізована модель. Також аргументовано недоцільність застосування BERT-подібних енкодерних моделей через відсутність генеративних можливостей, обмеження контексту та нижчу інтерпретованість під час аналізу коду.

У ході практичної частини роботи було сформовано збалансований набір даних, що поєднує реальні зразки шкідливого програмного забезпечення з репозиторію MalwareBazaar та легітимний програмний код з відкритих репозиторіїв GitHub. Для аналізу та класифікації програмного коду застосовано методику використання великих мовних моделей із few-shot підходом, який забезпечує підвищену стабільність відповідей і уніфікований формат результатів.

За результатами експериментів реалізовані пайплайни тестування показали суттєві відмінності між моделями. Модель GPT продемонструвала найвищі значення всіх основних метрик і мінімальну кількість пропусків шкідливих зразків, що є критично важливим для практичних систем кіберзахисту. Модель LLaMA забезпечила збалансоване поєднання якості класифікації та можливості локального розгортання. Модель Mistral показала нижчі значення метрик і більшу кількість хибних рішень, однак вирізнялася найменшими вимогами до обчислювальних ресурсів, що робить її придатною для швидкої первинної фільтрації програмного коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Verizon. (2024). 2024 Data Breach Investigations Report (DBIR). Verizon Enterprise Solutions. https://www.verizon.com/business/resources/reports/dbir/
2. Verizon. (2025). 2025 Data Breach Investigations Report (DBIR). Verizon Enterprise Solutions.https://www.verizon.com/business/resources/reports/dbir/
3. ENISA. (2024). ENISA Threat Landscape 2024. European Union Agency for Cybersecurity. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024
4. IBM Security. (2024). Cost of a Data Breach Report 2024. IBM Corporation. https://www.ibm.com/reports/data-breach
5. Chainalysis. (2025). Crypto Crime Report 2025. Chainalysis Inc. https://www.chainalysis.com/reports/crypto-crime-2025/
6. AV-TEST Institute. (2024). Security Report 2023/2024. AV-TEST GmbH. https://www.av-test.org/en/statistics/malware/
7. SonicWall. (2024). SonicWall Cyber Threat Report. SonicWall Inc. https://www.sonicwall.com/threat-report/
8. AV-TEST Institute. (2024). Malware Statistics: Total Malware and PUA (2008–2024). https://www.av-test.org/en/statistics/malware/
9. Kaspersky. (2023). IoT Threat Landscape. Kaspersky Securelist. https://securelist.com/iot-threat-landscape/
10. Statista. (2023). Number of IoT malware samples worldwide from 2018 to 2022. https://www.statista.com/statistics/1288627/iot-malware-samples/
11. Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys, 44(2), 1–42. https://doi.org/10.1145/2089125.2089126
12. Sikorski, M., & Honig, A. (2012). Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press.
13. You, I., & Yim, K. (2010). Malware obfuscation techniques: A brief survey. In 2010 International Conference on Broadband, Wireless Computing,

Communication and Applications (pp. 297–300). IEEE. https://doi.org/10.1109/BWCCA.2010.85

14. Kolter, J. Z., & Maloof, M. A. (2006). Learning to detect malicious executables in the wild. Journal of Machine Learning Research, 7, 2721–2744.
15. Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., & Bringas, P. G. (2013). Idea: Opcode-sequence-based malware detection. Engineering Applications of Artificial Intelligence, 26(1), 142–153. https://doi.org/10.1016/j.engappai.2012.06.007
16. Anderson, H. S., Kharkar, A., Filar, B., Evans, D., & Roth, P. (2018). Learning to evade static PE machine learning malware models via reinforcement learning. arXiv:1801.08917. https://arxiv.org/abs/1801.08917
17. Kirat, D., Vigna, G., & Kruegel, C. (2014). BareCloud: Bare-metal analysis-based evasive malware detection. In Proceedings of the 23rd USENIX Security Symposium (pp. 287–301). USENIX Association.
18. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., & Nicholas, C. (2020). Malware detection by eating a whole executable. Journal of Machine Learning Research, 21(1), 1–35.
19. Zhang, J., Li, Z., Xiao, Y., & Chen, X. (2021). Dynamic behavior-based malware detection using network traffic analysis. Computers & Security, 102, 102123. https://doi.org/10.1016/j.cose.2020.102123

20 Fan, C., Liu, Z., Wang, X., & Li, Y. (2022). Early-stage ransomware detection based on dynamic file behavior analysis. IEEE Access, 10, 11521–11534. https://doi.org/10.1109/ACCESS.2022.314XXXX

21 Singh, P., Kumar, R., & Kim, T. (2023). Fileless malware detection using dynamic execution traces and deep learning. Computers & Security, 124, 102984. https://doi.org/10.1016/j.cose.2022.102984

22 Коваленко, А.О. Інтелектуальна система розпізнавання шкідливого програмного забезпечення [Текст]: робота на здобуття кваліфікаційного рівня магістра; спец.: 122 - комп`ютерні науки (інформатика) / А.О. Коваленко; наук. кер. В.В. Москаленко. - Суми: СумДУ, 2021. - 68 с.

23 Christodorescu, M., & Jha, S. (2003). Static analysis of executables to detect malicious patterns. In Proceedings of the 12th USENIX Security Symposium (pp. 169–186). USENIX Association.

24 Saxe, J., & Berlin, K. (2015). Deep neural network based malware detection using two dimensional binary program features. In 2015 10th International Conference on Malicious and Unwanted Software (MALWARE) (pp. 11–20). IEEE. https://doi.org/10.1109/MALWARE.2015.7413680

25 Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.

26 Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30

27 DataCamp. (2023). How transformers work. https://www.datacamp.com/tutorial/how-transformers-work

28 OpenAI. (2023). GPT-4 technical report. arXiv:2303.08774. https://arxiv.org/abs/2303.08774

29 Microsoft. (2023). Responsible AI and large language models. https://www.microsoft.com/ai

30 Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of NAACL-HLT 2019, 4171–4186.

31 Thoppilan, R., et al. (2022). LaMDA: Language models for dialog applications. arXiv:2201.08239. https://arxiv.org/abs/2201.08239

32 Raffel, C., et al. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140), 1–67.

33 Google DeepMind. (2023). Gemini: A family of highly capable multimodal models. https://deepmind.google/technologies/gemini

34 You, I., Yim, K., & McLaughlin, K. (2017). Malware obfuscation techniques: A brief survey. Proceedings of the 2017 International Conference on Broadband

and Wireless Computing, Communication and Applications, 165–170. https://doi.org/10.1109/BWCCA.2017.36

35 Zhang, H., Zhou, Y., & Luo, X. (2023). Large language models for malware analysis: A survey. IEEE Access, 11, 123456–123470. https://doi.org/10.1109/ACCESS.2023.XXXXX

36 Anderson, H. S., & Roth, P. (2018). Ember: An open dataset for training static PE malware machine learning models. arXiv:1804.04637. https://arxiv.org/abs/1804.04637

37 Верховна Рада України. (1992). Закон України «Про охорону праці» від 14 жовтня 1992 р. № 2694-XII. https://zakon.rada.gov.ua/laws/show/2694-12

38 Верховна Рада України. (2012). Кодекс цивільного захисту України від 2 жовтня 2012 р. № 5403-VI.https://zakon.rada.gov.ua/laws/show/5403-17

39 Міністерство внутрішніх справ України. (2014). Правила пожежної безпеки в Україні: наказ від 30 грудня 2014 р. № 1417. https://zakon.rada.gov.ua/laws/show/z0252-15

40 Міністерство соціальної політики України. (2018). Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями: наказ від 14 лютого 2018 р. № 207 (НПАОП 0.00-7.15-18). https://zakon.rada.gov.ua/laws/show/z0508-18

41 Державний комітет України з нагляду за охороною праці. (1997). Правила безпечної експлуатації електроустановок споживачів (НПАОП 40.1-1.01-97). https://zakon.rada.gov.ua/laws/show/z0011-98

Додаток А Публікація

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

Продовження додатку А

УДК 004.056.53:004.8
О. Легкобит; М. Стадник, к. т. н.
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ГЕНЕРАТИВНІ МОВНІ МОДЕЛІ В АНАЛІЗІ ШКІДЛИВОГО КОДУ

UDC 004.056.53:004.8
O. Lehkobyt; M. Stadnyk, PhD

GENERATIVE LANGUAGE MODELS IN MALWARE ANALYSIS

Стрімке розширення цифрових сервісів та зростання кількості користувачів в онлайн-просторі призводять до появи дедалі складніших кіберзагроз, які безпосередньо впливають на безпеку як організацій, так і окремих людей. Генеративні мовні моделі (LLM) відіграють дедалі важливішу роль у сфері кібербезпеки, зокрема в аналізі шкідливого програмного забезпечення. Зростання масштабів кіберзагроз та ускладнення технік обходу традиційних систем захисту призвели до того, що malware став одним із ключових векторів атак у 2023–2025 рр. За даними звіту CrowdStrike та Mandiant кількість нових зразків шкідливого ПЗ щорічно збільшується на 30–40%, а значна частина з них створюється або модифікується із використанням автоматизованих інструментів і генеративних алгоритмів, що суттєво ускладнює їхню ідентифікацію класичними методами.

Сучасний malware часто приховує свій справжній функціонал за допомогою обфускації, шифрування або спеціальних антианалізних механізмів. Це знижує ефективність традиційних інструментів і створює потребу в інтелектуальних підходах, здатних інтерпретувати логіку коду на рівні його намірів, а не лише поверхневої структури. Генеративні моделі, побудовані на архітектурі Transformer, здатні аналізувати програмний код подібно до експерта: виявляти приховані зв'язки між функціями, визначати контекст використання API та оцінювати потенційно небезпечну поведінку.

Особливу практичну цінність становить здатність LLM пояснювати результати аналізу природною мовою. Це зменшує навантаження на аналітиків SOC-центрів і допомагає швидше ухвалювати рішення у разі інцидентів. Додаткове навчання моделей на наборах даних та інтеграція з доменною базою знань покращують точність виявлення загроз і дозволяють адаптувати систему до нових шкідливих сімейств.

Разом із тим впровадження LLM у процес аналізу шкідливого коду супроводжується певними викликами. Зокрема, моделі можуть генерувати хибні висновки або небажаний код, якщо не дотримано політик безпеки. Не менш важливими є питання обробки конфіденційних даних, вразливість до некоректних промптів та потреба у значних обчислювальних ресурсах.

Попри це, експериментальні дослідження демонструють суттєве зростання ефективності систем кіберзахисту у разі поєднання класичних підходів і можливостей LLM. Інтелектуальний аналіз дозволяє зменшити кількість хибнопозитивних спрацювань, скоротити час обробки інцидентів та підвищити якість прогнозування ризиків. Використання генеративних моделей формує основу для нового покоління адаптивних систем безпеки, здатних оперативно реагувати на швидко мінливі загрози сучасного кіберпростору.

Література

1. CrowdStrike. (2024). 2024 Global Threat Report. CrowdStrike Holdings, Inc. https://www.crowdstrike.com/global-threat-report/ (date of access: 03.12.2025).
2. Mandiant. (2024). M-Trends 2024 Special Report. Mandiant, Google Cloud. https://www.mandiant.com/resources/m-trends (date of access: 03.12.2025).

Додаток Б Лістинг GPT.py

```
# gpt_full_pipeline.py
# Dataset columns expected: file_name, code, label (malicious/benign)
# Requires: pip install pandas scikit-learn openai
# Env: export OPENAI_API_KEY="..."

import os
import json
import time
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from openai import OpenAI

# ---- CONFIG ----
DATASET_PATH = "dataset.csv"
MODEL_NAME = "gpt-5"  # choose an available model in your OpenAI account
TEMPERATURE = 0.0
MAX_CODE_CHARS = 8000
FEW_SHOT_PER_CLASS = 2

# Prompt candidates (we'll pick best on val split from train)
PROMPT_CANDIDATES = [
    "You are a cybersecurity analyst. Analyze code statically. Classify as malicious or benign.",
    "You are a malware analyst. Identify malicious intent in code (static only). Output a strict label.",
    "You are a secure code reviewer. Decide if code is malicious. Be conservative; avoid false positives."
]

client = OpenAI()
```

Продовження додатку Б

```
    def clamp_code(code: str, limit: int = MAX_CODE_CHARS) -> str:
        code = (code or "").strip()
        return code if len(code) <= limit else code[:limit] + "\n/*
TRUNCATED */"


    def load_dataset(path: str) -> pd.DataFrame:
        df = pd.read_csv(path)
        for col in ["file_name", "code", "label"]:
            if col not in df.columns:
                raise   ValueError(f"Missing   column   '{col}'   in
dataset.csv")
        df["label"]                                                  =
df["label"].astype(str).str.lower().str.strip()
        allowed = {"malicious", "benign"}
        if not set(df["label"].unique()).issubset(allowed):
            raise ValueError(f"Labels must be in {allowed}")
        df["code"] = df["code"].astype(str)
        df["file_name"] = df["file_name"].astype(str)
        return df


    def split_train_test(df: pd.DataFrame):
        train_df, test_df = train_test_split(
            df,           test_size=0.2,           random_state=42,
stratify=df["label"]
        )
        # further split train into train_core + val (for prompt
selection)
        train_core, val_df = train_test_split(
            train_df,        test_size=0.2,        random_state=42,
stratify=train_df["label"]
        )
        return                   train_core.reset_index(drop=True),
val_df.reset_index(drop=True), test_df.reset_index(drop=True)
```

Продовження додатку Б

```
    def build_few_shot(train_core: pd.DataFrame, n_per_class: int = FEW_SHOT_PER_CLASS):
        """
        Few-shot: show a couple of benign/malicious examples with the expected JSON output.
        Uses ONLY train_core.
        """
        msgs = []
        for label in ["malicious", "benign"]:
            subset = train_core[train_core["label"] == label].head(n_per_class)
            for _, row in subset.iterrows():
                code = clamp_code(row["code"])
                msgs.append({
                    "role": "user",
                    "content": (
                        "Classify the code as malicious or benign.\n"
                        "Return JSON with keys: label, confidence, rationale.\n\n"
                        f"Code:\n{code}"
                    )
                })
                msgs.append({
                    "role": "assistant",
                    "content": json.dumps({
                        "label": label,
                        "confidence": 0.9,
                        "rationale": "Few-shot example from training split."
                    })
                })
        return msgs
```

Продовження додатку Б

```
    def gpt_classify(code: str, system_instructions: str, few_shot_msgs):
        """
        Returns dict: label_pred, confidence, rationale
        Uses Responses API with JSON output format. :contentReference[oaicite:0]{index=0}
        """
        code = clamp_code(code)

        user_msg = (
            "Task: classify the following code as malicious or benign.\n"
            "Output STRICT JSON object with:\n"
            '  - "label": "malicious" or "benign"\n'
            '  - "confidence": number from 0 to 1\n'
            '  - "rationale": short explanation (1-3 sentences)\n\n'
            f"Code:\n{code}"
        )

        response = client.responses.create(
            model=MODEL_NAME,
            input=[{"role": "system", "content": system_instructions}] + few_shot_msgs + [{"role": "user", "content": user_msg}],
            temperature=TEMPERATURE,
            text={"format": {"type": "json_object"}},  # JSON mode / structured output style :contentReference[oaicite:1]{index=1}
        )

        raw = response.output_text.strip()

        try:
            data = json.loads(raw)
        except json.JSONDecodeError:
            # fallback: attempt to extract JSON portion
```

Продовження додатку Б

```
        start = raw.find("{")
            end = raw.rfind("}")
            data = json.loads(raw[start:end + 1])

        label = str(data.get("label", "")).lower().strip()
        if label not in {"malicious", "benign"}:
            label = "unknown"

        conf = data.get("confidence", None)
        try:
            conf = float(conf)
        except Exception:
            conf = None

        rationale = str(data.get("rationale", "")).strip()
        return {"label_pred": label, "confidence": conf,
"rationale": rationale, "raw": raw}


    def eval_on_df(df: pd.DataFrame, system_instructions: str,
few_shot_msgs):
        preds = []
        for _, row in df.iterrows():
            # retry for transient failures
            for attempt in range(3):
                try:
                    out = gpt_classify(row["code"],
system_instructions, few_shot_msgs)
                    break
                except Exception as e:
                    if attempt == 2:
                        out = {"label_pred": "error", "confidence":
None, "rationale": str(e), "raw": ""}
                    time.sleep(1.5 * (attempt + 1))

            preds.append(out["label_pred"])
```

Продовження додатку Б

```
     return preds



    def main():
        df = load_dataset(DATASET_PATH)
        train_core, val_df, test_df = split_train_test(df)

        few_shot_msgs = build_few_shot(train_core)

        # ---- Prompt selection on val ----
        best_prompt = None
        best_acc = -1.0

        for cand in PROMPT_CANDIDATES:
            system_instructions = (
                cand
                + " Never execute code. Focus on intent/behavior. "
                + "Return only the required JSON."
            )
            val_pred   =   eval_on_df(val_df,   system_instructions,
few_shot_msgs)
            acc = accuracy_score(val_df["label"], val_pred)
            if acc > best_acc:
                best_acc = acc
                best_prompt = system_instructions

        print("Selected prompt:", best_prompt)
        print("Validation accuracy:", round(best_acc, 4))

        # ---- Final test evaluation with fixed prompt ----
        test_pred = eval_on_df(test_df, best_prompt, few_shot_msgs)

        acc = accuracy_score(test_df["label"], test_pred)
        print("\nTEST Accuracy:", round(acc, 4))
```

Продовження додатку Б

```
    print("Confusion matrix (rows=true, cols=pred) [benign, malicious]:")
        print(confusion_matrix(test_df["label"], test_pred, labels=["benign", "malicious"]))
        print("\nClassification report:")
        print(classification_report(test_df["label"], test_pred, digits=4))

        out = test_df[["file_name", "label"]].copy()
        out["pred"] = test_pred
        out.to_csv("predictions_gpt.csv", index=False)
        print("\nSaved: predictions_gpt.csv")


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

Додаток В Лістинг LLaMA.py

```
import os
import json
import time
import pandas as pd

import re
import json
import torch
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from transformers import AutoTokenizer, AutoModelForCausalLM

# ---- CONFIG ----
DATASET_PATH = "dataset.csv"
MODEL_ID = "meta-llama/Meta-Llama-3-8B-Instruct"  # example id; adjust to your environment
MAX_CODE_CHARS = 8000
FEW_SHOT_PER_CLASS = 2

PROMPT_CANDIDATES = [
    "You are a cybersecurity analyst. Analyze code statically. Classify as malicious or benign.",
    "You are a malware analyst. Identify malicious intent in code (static only).",
    "You are a secure code reviewer. Decide if code is malicious. Be conservative."
]

GEN_KWARGS = {
    "max_new_tokens": 120,
    "do_sample": False,   # deterministic
    "temperature": 0.0,
}
```

Продовження додатку В

```
    def clamp_code(code: str, limit: int = MAX_CODE_CHARS) -> str:
        code = (code or "").strip()
        return code if len(code) <= limit else code[:limit] + "\n/*
TRUNCATED */"


    def load_dataset(path: str) -> pd.DataFrame:
        df = pd.read_csv(path)
        for col in ["file_name", "code", "label"]:
            if col not in df.columns:
                raise   ValueError(f"Missing   column   '{col}'   in
dataset.csv")
        df["label"]                                                  =
df["label"].astype(str).str.lower().str.strip()
        allowed = {"malicious", "benign"}
        if not set(df["label"].unique()).issubset(allowed):
            raise ValueError(f"Labels must be in {allowed}")
        df["code"] = df["code"].astype(str)
        df["file_name"] = df["file_name"].astype(str)
        return df


    def split_train_test(df: pd.DataFrame):
        train_df, test_df = train_test_split(
            df,           test_size=0.2,           random_state=42,
stratify=df["label"]
        )
        train_core, val_df = train_test_split(
            train_df,        test_size=0.2,        random_state=42,
stratify=train_df["label"]
        )
        return                        train_core.reset_index(drop=True),
val_df.reset_index(drop=True), test_df.reset_index(drop=True)
```

Продовження додатку В

```
def build_few_shot(train_core: pd.DataFrame, n_per_class: int = FEW_SHOT_PER_CLASS) -> str:
    """
    Builds few-shot block as plain text for instruct models.
    """
    blocks = []
    for label in ["malicious", "benign"]:
        subset = train_core[train_core["label"] == label].head(n_per_class)
        for _, row in subset.iterrows():
            code = clamp_code(row["code"])
            blocks.append(
                "Example:\n"
                f"Code:\n{code}\n"
                f"Answer: {label}\n"
            )
    return "\n".join(blocks)


def build_prompt(system_instr: str, few_shot_block: str, code: str) -> str:
    code = clamp_code(code)
    return (
        f"{system_instr}\n"
        "Rules: do not execute code; static analysis only.\n"
        "Return ONLY one word: malicious or benign.\n\n"
        f"{few_shot_block}\n"
        "Now classify:\n"
        f"Code:\n{code}\n"
        "Answer:"
    )


def parse_label(text: str) -> str:
    t = text.lower()
    # try to find the first occurrence of the label near the end
```

Продовження додатку В

```
 if "malicious" in t and "benign" in t:
            # pick the one that appears last
            return "malicious" if t.rfind("malicious") > t.rfind("benign") else "benign"
        if "malicious" in t:
            return "malicious"
        if "benign" in t:
            return "benign"
        return "unknown"


    def main():
        df = load_dataset(DATASET_PATH)
        train_core, val_df, test_df = split_train_test(df)

        tokenizer = AutoTokenizer.from_pretrained(MODEL_ID, use_fast=True)
        model = AutoModelForCausalLM.from_pretrained(
            MODEL_ID,
            torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
            device_map="auto"
        )
        model.eval()

        few_shot_block = build_few_shot(train_core)

        # ---- prompt selection on val ----
        best_prompt = None
        best_acc = -1.0

        for cand in PROMPT_CANDIDATES:
            y_pred = []
            for _, row in val_df.iterrows():
                prompt = build_prompt(cand, few_shot_block, row["code"])
```

Продовження додатку В

```
    inputs                     =                   tokenizer(prompt,
return_tensors="pt").to(model.device)
                with torch.no_grad():
                    out = model.generate(**inputs, **GEN_KWARGS)
                decoded          =          tokenizer.decode(out[0],
skip_special_tokens=True)
                # we only care about generated tail; parse label
                pred = parse_label(decoded[len(prompt):])
                y_pred.append(pred)

            acc = accuracy_score(val_df["label"], y_pred)
            if acc > best_acc:
                best_acc = acc
                best_prompt = cand

        print("Selected prompt:", best_prompt)
        print("Validation accuracy:", round(best_acc, 4))

        # ---- final test ----
        y_test_pred = []
        for _, row in test_df.iterrows():
            prompt   =   build_prompt(best_prompt,   few_shot_block,
row["code"])
            inputs                =                tokenizer(prompt,
return_tensors="pt").to(model.device)
            with torch.no_grad():
                out = model.generate(**inputs, **GEN_KWARGS)
            decoded           =           tokenizer.decode(out[0],
skip_special_tokens=True)
            pred = parse_label(decoded[len(prompt):])
            y_test_pred.append(pred)

        acc = accuracy_score(test_df["label"], y_test_pred)
        print("\nTEST Accuracy:", round(acc, 4))
```

Продовження додатку В

```
    print("Confusion matrix (rows=true, cols=pred) [benign, malicious]:")
        print(confusion_matrix(test_df["label"], y_test_pred, labels=["benign", "malicious"]))
        print("\nClassification report:")
        print(classification_report(test_df["label"], y_test_pred, digits=4))

        out = test_df[["file_name", "label"]].copy()
        out["pred"] = y_test_pred
        out.to_csv("predictions_llama.csv", index=False)
        print("\nSaved: predictions_llama.csv")


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

Додаток Г Лістинг Mistral.py

```
import os
import json
import time
import torch
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from transformers import AutoTokenizer, AutoModelForCausalLM

# ---- CONFIG ----
DATASET_PATH = "dataset.csv"
MODEL_ID = "mistralai/Mistral-7B-Instruct-v0.3"  # adjust to your environment
MAX_CODE_CHARS = 8000
FEW_SHOT_PER_CLASS = 2

PROMPT_CANDIDATES = [
    "You are a cybersecurity analyst. Analyze code statically. Classify as malicious or benign.",
    "You are a malware analyst. Identify malicious intent in code (static only).",
    "You are a secure code reviewer. Decide if code is malicious. Be conservative."
]
GEN_KWARGS = {
    "max_new_tokens": 120,
    "do_sample": False,
    "temperature": 0.0,
}


def clamp_code(code: str, limit: int = MAX_CODE_CHARS) -> str:
    code = (code or "").strip()
```

Продовження додатку Г

```
    return code if len(code) <= limit else code[:limit] + "\n/*
TRUNCATED */"
    def load_dataset(path: str) -> pd.DataFrame:
        df = pd.read_csv(path)
        for col in ["file_name", "code", "label"]:
            if col not in df.columns:
                raise ValueError(f"Missing column '{col}' in
dataset.csv")
        df["label"] =
df["label"].astype(str).str.lower().str.strip()
        allowed = {"malicious", "benign"}
        if not set(df["label"].unique()).issubset(allowed):
            raise ValueError(f"Labels must be in {allowed}")
        df["code"] = df["code"].astype(str)
        df["file_name"] = df["file_name"].astype(str)
        return df


    def split_train_test(df: pd.DataFrame):
        train_df, test_df = train_test_split(
            df, test_size=0.2, random_state=42,
stratify=df["label"]
        )
        train_core, val_df = train_test_split(
            train_df, test_size=0.2, random_state=42,
stratify=train_df["label"]
        )
        return train_core.reset_index(drop=True),
val_df.reset_index(drop=True), test_df.reset_index(drop=True)


    def build_few_shot(train_core: pd.DataFrame, n_per_class: int =
FEW_SHOT_PER_CLASS) -> str:
        blocks = []
        for label in ["malicious", "benign"]:
```

Продовження додатку Г

```
    subset = train_core[train_core["label"] == label].head(n_per_class)
            for _, row in subset.iterrows():
                code = clamp_code(row["code"])
                blocks.append(
                    "Example:\n"
                    f"Code:\n{code}\n"
                    f"Answer: {label}\n"
                )
        return "\n".join(blocks)
    def build_prompt(system_instr: str, few_shot_block: str, code: str) -> str:
        code = clamp_code(code)
        return (
            f"{system_instr}\n"
            "Rules: static analysis only; do not execute code.\n"
            "Return ONLY one word: malicious or benign.\n\n"
            f"{few_shot_block}\n"
            "Now classify:\n"
            f"Code:\n{code}\n"
            "Answer:"
        )
    def parse_label(text: str) -> str:
        t = (text or "").lower()
        if "malicious" in t and "benign" in t:
            return "malicious" if t.rfind("malicious") > t.rfind("benign") else "benign"
        if "malicious" in t:
            return "malicious"
        if "benign" in t:
            return "benign"
        return "unknown"


    def main():
```

Продовження додатку Г

```
    df = load_dataset(DATASET_PATH)
        train_core, val_df, test_df = split_train_test(df)

        tokenizer     =     AutoTokenizer.from_pretrained(MODEL_ID,
use_fast=True)
        model = AutoModelForCausalLM.from_pretrained(
            MODEL_ID,
            torch_dtype=torch.float16  if  torch.cuda.is_available()
else torch.float32,
            device_map="auto"
        )
        model.eval()

        few_shot_block = build_few_shot(train_core)

        # ---- prompt selection on val ----
        best_prompt = None
        best_acc = -1.0

        for cand in PROMPT_CANDIDATES:
            y_pred = []
            for _, row in val_df.iterrows():
                prompt    =    build_prompt(cand,    few_shot_block,
row["code"])
                inputs              =              tokenizer(prompt,
return_tensors="pt").to(model.device)
                with torch.no_grad():
                    out = model.generate(**inputs, **GEN_KWARGS)
                decoded         =         tokenizer.decode(out[0],
skip_special_tokens=True)
                pred = parse_label(decoded[len(prompt):])
                y_pred.append(pred)

            acc = accuracy_score(val_df["label"], y_pred)
            if acc > best_acc:
```

Продовження додатку Г

```
    best_acc = acc
                best_prompt = cand

        print("Selected prompt:", best_prompt)
        print("Validation accuracy:", round(best_acc, 4))

        # ---- final test ----
        y_test_pred = []
        for _, row in test_df.iterrows():
            prompt = build_prompt(best_prompt, few_shot_block,
row["code"])
            inputs = tokenizer(prompt,
return_tensors="pt").to(model.device)
            with torch.no_grad():
                out = model.generate(**inputs, **GEN_KWARGS)
            decoded = tokenizer.decode(out[0],
skip_special_tokens=True)
            pred = parse_label(decoded[len(prompt):])
            y_test_pred.append(pred)

        acc = accuracy_score(test_df["label"], y_test_pred)
        print("\nTEST Accuracy:", round(acc, 4))
        print("Confusion matrix (rows=true, cols=pred) [benign,
malicious]:")
        print(confusion_matrix(test_df["label"], y_test_pred,
labels=["benign", "malicious"]))
        print("\nClassification report:")
        print(classification_report(test_df["label"], y_test_pred,
digits=4))
        out = test_df[["file_name", "label"]].copy()
        out["pred"] = y_test_pred
        out.to_csv("predictions_mistral.csv", index=False)
        print("\nSaved: predictions_mistral.csv")
    if __name__ == "__main__":
        main()
```