

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

*Кваліфікаційна наукова
праця на правах рукопису*

ТЕРЕЩЕНКО ОЛЕКСАНДР ІГОРОВИЧ

УДК 004.85:004.056.55

ДИСЕРТАЦІЯ

**МЕТОДИ І МОДЕЛІ МАШИННОГО НАВЧАННЯ ДЛЯ ВИЯВЛЕННЯ ТА
КЛАСИФІКАЦІЇ ВРАЗЛИВОСТЕЙ СМАРТ-КОНТРАКТІВ**

Спеціальність: 122 – «Комп'ютерні науки»

Галузь знань: 12 – «Інформаційні технології»

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ О. І. Терещенко

Науковий керівник:

Комлева Наталія Олегівна, кандидат технічних наук, доцент

АНОТАЦІЯ

Терещенко О.І. Методи і моделі машинного навчання для виявлення та класифікації вразливостей смарт-контрактів. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 – Комп’ютерні науки. – Національний університет «Одеська політехніка», МОН України, Одеса, 2025.

У **вступі** обґрунтовано актуальність проблеми безпеки смарт-контрактів у блокчейн-екосистемах (зокрема у децентралізованих фінансах), які акумулюють значні фінансові активи та залишаються вразливими до помилок коду. Сформульовано **мету дослідження** – підвищити ефективність за відповідними метриками виявлення, класифікації та локалізації вразливостей у смарт-контрактах за рахунок розробки моделей та методів машинного навчання, які становлять цілісний технологічний стек інтелектуального аналізу вихідного коду смарт-контрактів.

Визначено **об’єкт дослідження** – процеси виявлення, класифікації та локалізації вразливостей у смарт-контрактах. **Предметом дослідження** є методи та моделі машинного навчання для багатоміткового виявлення, класифікації та локалізації вразливостей у смарт-контрактах.

Сформульовано комплекс взаємопов’язаних завдань, серед яких: створення збалансованого та репрезентативного навчального датасету смарт-контрактів із анотацією вразливостей; розроблення моделей і методів багатоміткової класифікації вразливостей смарт-контрактів; підвищення надійності й стабільності моделей за рахунок калібрування впевненості та аналізу стійкості до збурень та змін коду; розроблення слабосупервізованих методів локалізації вразливих фрагментів коду без токенної розмітки; апробація запропонованих методів шляхом їх реалізації у програмному засобі та перевірки ефективності на реальних смарт-контрактах.

У **першому розділі** здійснено систематичний огляд підходів до виявлення вразливостей у смарт-контрактах, що охоплює формальні методи, статичний та динамічний аналіз, а також сучасні методи на основі машинного навчання. Показано,

що кожен клас методів має суттєві обмеження, які стримують їхнє використання у великих блокчейн-екосистемах. Формальні методи забезпечують максимальну строгість і дозволяють строго довести коректність коду, однак є надзвичайно дорогими, потребують значної експертної участі та масштабуються лише на обмежену кількість критичних контрактів. Статичний аналіз є найбільш продуктивним підходом, здатним швидко сканувати великі репозиторії, але страждає від значної кількості хибнопозитивних спрацювань і часто не охоплює нетипові або нові патерни вразливостей. Динамічний аналіз наближають перевірку до реальних сценаріїв атак і добре виявляють логічні помилки, але є повільними, не гарантують повного покриття й сильно залежать від заданих сценаріїв взаємодії. Методи машинного навчання демонструють хорошу масштабованість і здатність виявляти складні закономірності, проте у практичних сценаріях працюють як «чорна скринька», що ускладнює інтерпретацію рішень і локалізацію вразливих фрагментів.

Окремо проведено аналіз трансформерних моделей для коду (CodeBERT/GraphCodeBERT), графові нейронні мережі (Graph Neural Networks, GNN) та гібридних трансформерно-рекурентних архітектур. Виявлено ключові практичні обмеження їхнього застосування: нестачу якісної фрагментної розмітки для локалізації, низьку стійкість до обфускації та рефакторингу без застосування спеціальних методів підвищення стійкості до збурень та змін, а також схильність моделей до надмірної впевненості у прогнозах під час роботи в production-середовищах. Сформовано рамку оцінювання: для детекції використано micro-F1 (мікроусереднену F1-міру) та macro-F1 (макроусереднену F1-міру), а також ROC-AUC (площу під кривою робочої характеристики приймача, Receiver Operating Characteristic – Area Under the Curve); для локалізації застосовано mAP (mean Average Precision – середню точність), IoU@ τ (Intersection over Union при пороговому значенні τ – міру перекриття передбачених та істинних областей) і Hit@K (частоту потрапляння релевантного фрагмента серед перших K прогнозів); для оцінювання надійності використано метрику очікуваної калібрувальної похибки (Expected Calibration Error, ECE) та пов'язані показники.

У **другому розділі** обґрунтовано й реалізовано метод формування збалансованого навчального датасету смарт-контрактів для задач виявлення вразливостей. Запропоновано комбінований підхід, що поєднує детерміноване інжектування типових вразливостей у безпечний код (pattern-based + статичний аналіз) та контекстно узгоджену генерацію/модифікацію прикладів за допомогою великої мовної моделі. Обидва канали супроводжуються валідацією (компіляція, перевірка статичними аналізаторами, контроль формату та відповідності міток). На виході сформовано збалансований корпус із п'яти категорій (чотири класи вразливостей та безпечні контракти) по 1000 прикладів кожної.

Запропонований комбінований метод формування датасету забезпечує одночасно відтворюваність і контрольованість (детерміновані зміни з фіксованими правилами), реалістичну різноманітність синтаксичних реалізацій (LLM-модифікації, що зберігають логіку та стилістику коду) й збалансованість за класами, що підвищує якість подальшого навчання та порівняльної оцінки моделей.

У **третьому розділі** розроблено комплексний підхід до багатоміткової детекції та слабосупервізованої локалізації вразливостей у смарт-контрактах. Як базові розглянуто моделі BiGRU-ATT та CodeBERT-BiGRU для багатоміткової класифікації. Далі запропоновано модель CodeBERT-GDLA (Graph-aware Dual-Level Attention), що поєднує токенний рівень (CodeBERT + BiGRU) з блоковим рівнем і графовим зсувом. Удосконалено механізм уваги: введено графовий зсув на основі AST (Abstract Syntax Tree), CFG (Control Flow Graph), DFG (Data Flow Graph) графів та навчувані коефіцієнти β , γ для токенного й блокового рівнів, які керують часткою структурної інформації в логітах уваги й підсилюють чутливість до далеких/непрямих залежностей у коді.

Запропоновано CAM-узгоджену увагу: увага моделі узгоджується з Grad-CAM через λ -комбінацію, що забезпечує послідовні, інтерпретовані карти важливості на токенному та блоковому рівнях. Створено метод слабосупервізованої локалізації, який за допомогою ковзного вікна з top-K агрегацією перетворює логіти на просторові карти релевантності, а причинні регуляризатори дозволяють виділяти мінімально достатні та критично необхідні фрагменти коду без ручної токенної розмітки.

Експериментальні дослідження засвідчили, що запропонована модель CodeBERT-GDLA перевершує базову CodeBERT-BiGRU за macro-F1, macro-AUC і втратою Хеммінга (Hamming loss), а запропонований локалізатор вразливостей досягає високих показників метрик mAP та Hit@K (зокрема для вразливості повторного входу – $mAP \approx 82\%$, $Hit@5 \approx 77\%$, $Hit@7-10 \approx 100\%$) із низькою дисперсією, що підтверджує стабільність локалізації. Проведено абляційний аналіз внеску графових зсувів, дворівневої уваги, причинних регуляризаторів та CAM-узгодженої уваги, а також комплексне дослідження стійкості до збурень та змін, а саме до рефакторингів, перейменувань, форматування й часткової обфускації.

Щоб зменшити ризики надмірної впевненості та підвищити надійність моделей у задачах багатоміткової детекції, у тренувальний процес інтегровано запропонований метод регуляризації для калібрування ймовірнісних оцінок моделей глибокого навчання, який знижує метрику ECE (Expected Calibration Error) і робить ймовірнісні оцінки узгодженішими з реальною частотою подій. Крім того, проведено дослідження стійкості моделі до типових модифікацій коду (рефакторинг, перейменування ідентифікаторів, стилістичні зміни, часткова обфускація), що підтвердило стабільність як класифікації, так і локалізації вразливостей у реалістичних сценаріях використання.

У **четвертому розділі** представлено практичну реалізацію й впровадження програмного засобу аудиту смарт-контрактів на основі моделі CodeBERT-GDLA. Описано вимоги, архітектуру, REST-інтерфейси, користувацький інтерфейс та кейси використання в реальних сценаріях аудиту. Розроблений інструмент інтегрується в CI/CD-конвеєри та може доповнювати існуючі статичні аналізатори.

Наукова новизна одержаних результатів полягає в наступному:

– **вперше запропоновано комбінований метод формування навчального датасету** за рахунок поєднання автоматичного інжектування аномальних конструкцій у коректний код смарт-контрактів із генерацією додаткових прикладів великою мовною моделлю з подальшою верифікацією, що дозволило забезпечити збалансованість вибірки;

- **вперше побудовано модель CodeBERT-GDLA** на основі вдосконалення архітектур BiGRU-ATT і CodeBERT-BiGRU за допомогою запропонованих методів графового зсуву і дворівневої графово-орієнтованої уваги, що дозволило підвищити ефективність виявлення та класифікації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів;
- **удосконалено метод регуляризації для калібрування ймовірнісних оцінок багатоміткової класифікації вразливостей смарт-контрактів** за рахунок застосування оптимізації очікуваної калібрувальної похибки, температурного масштабування та фокального приглушення надмірної впевненості безпосередньо в процесі навчання моделі CodeBERT-GDLA, що забезпечує зниження її надмірної впевненості та підвищує надійність результатів інтелектуального аналізу вихідного коду смарт-контрактів;
- **отримав подальший розвиток метод слабосупервізованого виділення вразливих фрагментів коду** за рахунок інтеграції CAM-узгодженої уваги з причинно-орієнтованою регуляризацією, що забезпечує підвищення ефективності локалізації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів.

Практичне значення результатів полягає в розробленні програмно-алгоритмічних засобів для автоматизованого аудиту смарт-контрактів, які забезпечують багатоміткове виявлення вразливостей, пояснювану локалізацію фрагментів коду та надійні ймовірнісні оцінки, а також у створенні відтворюваного конвеєра даних і збалансованого датасету для навчання й порівняльної оцінки моделей. Запропоновані моделі й методи впроваджено у діяльності ТОВ НВО «Діскрет» та НВП «КАРЕ», а також використано в навчальному процесі Національного університету «Одеська політехніка».

Ключові слова: смарт-контракти; машинне навчання; глибоке навчання; автоматизація аудиту; безпека програмного коду; виявлення вразливостей; локалізація вразливостей; багатоміткова класифікація; слабосупервізоване навчання; пояснюваний ШІ; графові подання коду; механізм уваги; причинні регуляризатори; калібрування ймовірностей.

Список публікацій здобувача за темою дисертації

1. Komleva, N. O., & Tereshchenko, O. I. (2023). Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform. *Herald of Advanced Information Technology*, 6(1), 54–68. DOI: <https://doi.org/10.15276/hait.06.2023.4> (**Scopus**)
2. Tereshchenko, O., & Komleva, N. (2023). Vulnerability detection of smart contracts based on bidirectional GRU and attention mechanism. In *Communications in Computer and Information Science* (Vol. 1980). Springer, Cham. DOI: https://doi.org/10.1007/978-3-031-48325-7_21 (**Scopus**)
3. Tereshchenko, O. I., & Komleva, N. O. (2024). Identification and localization of vulnerabilities in smart contracts using attention vectors analysis in a BERT-based model. *Radio Electronics, Computer Science, Control*, (3), 173–184. DOI: <https://doi.org/10.15588/1607-3274-2024-3-15> (**Web of Science**)
4. Терещенко, О. І. (2025). Комбінований метод локалізації вразливостей у смарт-контрактах на основі Attention та Grad-CAM. *Таврійський науковий вісник. Серія: Технічні науки*, 1(4), 306-316. DOI: <https://doi.org/10.32782/tnv-tech.2025.4.1.30>
5. Терещенко, О. І. (2025). Методи ін'єкції вразливостей у смарт-контракти для формування збалансованих датасетів. *Вісник Херсонського національного технічного університету*. № 3(94), Ч. 2, 456-462. DOI: <https://doi.org/10.35546/kntu2078-4481.2025.3.2.58>
6. Терещенко, О. І. (2025). Виявлення та локалізація вразливостей у смарт-контрактах на основі дворівневої уваги з графовим підкріпленням. *Вісник Кременчуцького національного університету імені Михайла Остроградського*, 4, 272-281. DOI: <https://doi.org/10.32782/1995-0519.2025.4.30>
7. Терещенко, О. І., & Комлева, Н. О. (2023). Огляд наявних інструментів виявлення вразливостей смарт-контрактів на основі машинного навчання. In *Modern Information Technology 2023 / Сучасні Інформаційні Технології 2023* (Одеса, 18–19 травня 2023 р.). URL: <http://dspace.opu.ua/jspui/handle/123456789/14202>

8. Tereshchenko, O. (2024). Analysis of the efficacy of using machine learning for detecting vulnerabilities in smart contracts compared to traditional methods. In Proceedings of the XIII International Scientific and Practical Conference “Social Ways of Training Specialists in the Social Sphere and Inclusive Education” (pp. 327–329). Prague, Czech Republic, April 1–3, 2024. URL: <https://eu-conf.com/en/events/social-ways-of-training-specialists-in-the-social-sphere-and-inclusive-education/>
9. Tereshchenko, O. (2024). Improving the security of smart contracts through the integration of machine learning: Challenges and prospects. In Proceedings of the VI International Scientific Conference “Traditional and Innovative Approaches to Scientific Research” (pp. 124–125). Vinnytsia, Ukraine, March 8, 2024. URL: <https://archive.mcnd.org.ua/index.php/conference-proceeding/issue/view/08.03.2024>
10. Tereshchenko, O. I. (2024). Integration of NLP and machine learning methods for smart contract security: A comparison with traditional approaches. In Proceedings of the X International Scientific and Practical Conference “Informatics. Culture. Technology”, September 2024. DOI: <https://doi.org/10.15276/ict.01.2024.31>
11. Терещенко, О. І. (2025). Побудова збалансованого корпусу смарт-контрактів шляхом контрольованої ін’єкції вразливостей. In Proceedings of the VIII International Scientific Conference “Розвиток наукової думки постіндустріального суспільства: сучасний дискурс”, 2025. DOI: <https://doi.org/10.62731/mcnd-17.10.2025>

ABSTRACT

Tereshchenko O.I. Machine Learning Methods and Models for Detecting and Classifying Smart-Contract Vulnerabilities. – Qualification scientific work in the form of manuscript.

Thesis for the PhD degree in specialty 122 Computer science. – Odessa Polytechnic National University, Ministry of Education and Science of Ukraine, Odesa, 2025.

In the introduction, the relevance of the smart contract security problem in blockchain ecosystems (in particular, decentralized finance) is substantiated, as these systems accumulate substantial financial assets while remaining vulnerable to software defects. **The aim of the research** is formulated as improving the effectiveness, according to relevant evaluation metrics, of detecting, classifying, and localizing vulnerabilities in smart contracts through the development of machine learning methods and models that constitute an integrated technological stack for intelligent analysis of smart contract source code.

The object of the research is the processes of detecting, classifying, and localizing vulnerabilities in smart contracts. **The subject of the research** is machine learning methods and models for multi-label detection, classification, and localization of vulnerabilities in smart contracts.

A set of interrelated **research tasks** is formulated, including: the creation of a balanced and representative training dataset of smart contracts with vulnerability annotations; the development of models and methods for multi-label vulnerability classification in smart contracts; improving model reliability and stability through confidence calibration and robustness analysis with respect to code changes; the development of weakly supervised methods for localizing vulnerable code fragments without token-level annotations; and validation of the proposed methods through their implementation in a software tool and evaluation of their effectiveness on real-world smart contracts.

The first chapter presents a systematic review of approaches to smart contract vulnerability detection, covering formal methods, static and dynamic analysis, as well as

modern machine learning-based techniques. It is shown that each class of methods has significant limitations that constrain their use in large-scale blockchain ecosystems. Formal methods provide maximal rigor and enable formal proofs of correctness, but are extremely costly, require substantial expert involvement, and scale only to a limited number of critical contracts. Static analysis is the most productive approach in terms of throughput and can rapidly scan large repositories, but suffers from a high rate of false positives and often fails to capture atypical or novel vulnerability patterns. Dynamic analysis brings testing closer to real attack scenarios and is effective at detecting logical flaws, but is slow, does not guarantee full coverage, and heavily depends on predefined interaction scenarios. Machine learning methods demonstrate good scalability and the ability to capture complex patterns; however, in practical settings they often operate as a “black box”, complicating decision interpretation and the localization of vulnerable fragments.

A separate analysis is conducted of transformer-based models for code (CodeBERT/GraphCodeBERT), graph neural networks (GNNs), and hybrid transformer-recurrent architectures. Key practical limitations of their application are identified, including the lack of high-quality fragment-level annotations for localization, low robustness to obfuscation and refactoring without dedicated robustness-enhancing techniques, and a tendency toward overconfident predictions in production environments. An evaluation framework is defined: for detection, micro-F1 and macro-F1, as well as ROC-AUC (Receiver Operating Characteristic – Area Under the Curve), are used; for localization, mAP (mean Average Precision), IoU@ τ (Intersection over Union at threshold τ), and Hit@K (the frequency with which a relevant fragment appears among the top-K predictions) are applied; for reliability assessment, Expected Calibration Error (ECE) and related metrics are employed.

The second chapter substantiates and implements a method for forming a balanced training dataset of smart contracts for vulnerability detection tasks. A combined approach is proposed that integrates deterministic injection of typical vulnerabilities into safe code (pattern-based methods with static analysis) and context-consistent generation/modification of examples using a large language model. Both channels are accompanied by validation procedures (compilation, checks with static analyzers, and format/label consistency

control). As a result, a balanced corpus of five categories (four vulnerability classes and safe contracts) with 1,000 examples per category is produced.

The proposed combined dataset construction method simultaneously ensures reproducibility and controllability (deterministic changes governed by fixed rules), realistic diversity of syntactic implementations (LLM-based modifications that preserve code logic and style), and class balance, thereby improving the quality of subsequent training and comparative model evaluation.

In the third chapter, a comprehensive approach to multi-label detection and weakly supervised localization of vulnerabilities in smart contracts is developed. BiGRU-ATT and CodeBERT-BiGRU models are considered as baselines for multi-label classification. Subsequently, the CodeBERT-GDLA (Graph-aware Dual-Level Attention) model is proposed, which combines a token-level component (CodeBERT + BiGRU) with a block-level component and graph-based biasing. The attention mechanism is enhanced by introducing graph bias derived from AST (Abstract Syntax Tree), CFG (Control Flow Graph), and DFG (Data Flow Graph), along with learnable coefficients β and γ for token- and block-level attention that control the contribution of structural information in attention logits and increase sensitivity to long-range and indirect dependencies in code.

A CAM-aligned attention mechanism is proposed, in which model attention is aligned with Grad-CAM via a λ -combination, yielding consistent and interpretable importance maps at both token and block levels. A weakly supervised localization method is introduced that converts logits into spatial relevance maps using a sliding window with Top-K aggregation, while causal regularizers enable the identification of minimally sufficient and critically necessary code fragments without manual token-level annotation.

Experimental studies demonstrate that the proposed CodeBERT-GDLA model outperforms the baseline CodeBERT-BiGRU in terms of macro-F1, macro-AUC, and Hamming loss, while the proposed vulnerability localizer achieves high mAP and Hit@K scores (in particular for reentrancy vulnerabilities: mAP $\approx$ 82%, Hit@5 $\approx$ 77%, Hit@7–10 $\approx$ 100%) with low variance, confirming localization stability. An ablation study evaluates the contributions of graph biasing, dual-level attention, causal regularizers, and CAM-

aligned attention, along with a comprehensive robustness analysis with respect to refactoring, renaming, formatting, and partial obfuscation.

To mitigate the risks of overconfidence and improve model reliability in multi-label detection tasks, a proposed regularization method for calibrating probabilistic outputs of deep learning models is integrated into the training process. This method reduces Expected Calibration Error (ECE) and aligns predicted probabilities with empirical event frequencies. Additionally, robustness to common code modifications (refactoring, identifier renaming, stylistic changes, and partial obfuscation) is evaluated, confirming the stability of both classification and localization under realistic usage scenarios.

The fourth chapter presents the practical implementation and deployment of a smart contract auditing tool based on the CodeBERT-GDLA model. System requirements, architecture, REST interfaces, user interface, and real-world audit use cases are described. The developed tool integrates into CI/CD pipelines and can complement existing static analyzers.

The **scientific novelty** of the obtained results is as follows:

- **for the first time, a combined method for forming a training dataset is proposed**, based on the integration of automatic injection of anomalous constructs into correct smart contract code with the generation of additional samples using a large language model followed by their verification, which ensures dataset balance;
- **for the first time, the CodeBERT-GDLA model is constructed** based on the enhancement of the BiGRU-ATT and CodeBERT-BiGRU architectures through the proposed graph shifting methods and a two-level graph-aware attention mechanism, which improves the effectiveness of vulnerability detection and classification during intelligent analysis of smart contract source code;
- **the regularization method for calibrating probabilistic estimates in multi-label vulnerability classification of smart contracts is improved** by integrating Expected Calibration Error optimization, temperature scaling, and focal suppression of overconfidence directly into the training process of the CodeBERT-GDLA model, which reduces excessive model confidence and increases the reliability of intelligent smart contract source code analysis results;

– **the method of weakly supervised identification of vulnerable code fragments is further developed** through the integration of CAM-aligned attention with causality-oriented regularization, which enhances the effectiveness of vulnerability localization during intelligent analysis of smart contract source code.

The practical significance of the results lies in the development of software and algorithmic tools for automated smart contract auditing that provide multi-label vulnerability detection, explainable localization of code fragments, and reliable probabilistic estimates, as well as in the creation of a reproducible data pipeline and a balanced dataset for training and comparative evaluation of models. The proposed models and methods have been implemented in the activities of LLC RPA “Diskret” and RPC “KARE”, and have also been used in the educational process of Odesa Polytechnic National University.

Keywords: smart contracts; machine learning; deep learning; audit automation; code security; vulnerability detection; vulnerability localization; multi-label classification; weakly supervised learning; explainable AI; graph representations of code; attention mechanism; causal regularizers; probability calibration.

List of publications of the applicant on the topic of the dissertation

1. Komleva, N. O., & Tereshchenko, O. I. (2023). Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform. *Herald of Advanced Information Technology*, 6(1), 54–68. <https://doi.org/10.15276/hait.06.2023.4> (**Scopus**)
2. Tereshchenko, O., & Komleva, N. (2023). Vulnerability detection of smart contracts based on bidirectional GRU and attention mechanism. In *Communications in Computer and Information Science* (Vol. 1980). Springer, Cham. https://doi.org/10.1007/978-3-031-48325-7_21 (**Scopus**)
3. Tereshchenko, O. I., & Komleva, N. O. (2024). Identification and localization of vulnerabilities in smart contracts using attention vectors analysis in a BERT-based model. *Radio Electronics, Computer Science, Control*, (3), 173–184. <https://doi.org/10.15588/1607-3274-2024-3-15> (**Web of Science**)
4. Tereshchenko, O. I. (2025). A combined method for vulnerability localization in smart contracts based on attention mechanisms and Grad-CAM. *Tavria Scientific Bulletin. Series: Technical Sciences*, 1(4), 306–316. <https://doi.org/10.32782/tnv-tech.2025.4.1.30>
5. Tereshchenko, O. I. (2025). Methods for vulnerability injection in smart contracts for building balanced datasets. *Bulletin of Kherson National Technical University*, 3(94), Part 2, 456–462. <https://doi.org/10.35546/kntu2078-4481.2025.3.2.58>
6. Tereshchenko, O. I. (2025). Detection and localization of smart contract vulnerabilities using a two-level attention mechanism with graph-based reinforcement. *Bulletin of Kremenchuk Mykhailo Ostrohradskyi National University*, 4, 272–281. <https://doi.org/10.32782/1995-0519.2025.4.30>
7. Tereshchenko, O. I., & Komleva, N. O. (2023). A review of existing machine learning-based tools for smart contract vulnerability detection. In *Modern Information Technology 2023* (Odesa, May 18–19, 2023). <http://dspace.opu.ua/jspui/handle/123456789/14202>

8. Tereshchenko, O. (2024). Analysis of the effectiveness of machine learning approaches for smart contract vulnerability detection compared to traditional methods. In Proceedings of the XIII International Scientific and Practical Conference “Social Ways of Training Specialists in the Social Sphere and Inclusive Education” (pp. 327-329). Prague, Czech Republic, April 1–3, 2024. <https://eu-conf.com/en/events/social-ways-of-training-specialists-in-the-social-sphere-and-inclusive-education/>
9. Tereshchenko, O. (2024). Improving smart contract security through the integration of machine learning: Challenges and prospects. In Proceedings of the VI International Scientific Conference “Traditional and Innovative Approaches to Scientific Research” (pp. 124-125). Vinnytsia, Ukraine, March 8, 2024. <https://archive.mcnd.org.ua/index.php/conference-proceeding/issue/view/08.03.2024>
10. Tereshchenko, O. I. (2024). Integration of NLP and machine learning methods for smart contract security: A comparison with traditional approaches. In Proceedings of the X International Scientific and Practical Conference “Informatics. Culture. Technology”, September 2024. <https://doi.org/10.15276/ict.01.2024.31>
11. Tereshchenko, O. I. (2025). Construction of a balanced smart contract corpus through controlled vulnerability injection. In Proceedings of the VIII International Scientific Conference “Development of Scientific Thought in the Post-Industrial Society: Contemporary Discourse”, 2025. <https://doi.org/10.62731/mcnd-17.10.2025>

ЗМІСТ

ВСТУП.....	19
РОЗДІЛ 1. АНАЛІЗ СУЧАСНИХ МЕТОДІВ І МОДЕЛЕЙ ВИЯВЛЕННЯ ТА КЛАСИФІКАЦІЇ ВРАЗЛИВОСТЕЙ У СМАРТ-КОНТРАКТАХ.....	25
1.1 Архітектура блокчейну.....	25
1.2 Смарт-контракти як предмет аналізу у контексті безпеки	27
1.2.1 Історичний розвиток концепції смарт-контрактів.....	27
1.2.2 Технічне визначення смарт-контракту	28
1.2.3 Види смарт-контрактів	30
1.2.4 Життєвий цикл смарт-контрактів.....	31
1.2.5 Роль смарт-контрактів у Web3 та децентралізованих екосистемах	33
1.2.6 Статистика смарт-контрактів в мережі Ethereum	34
1.3 Актуальність проблеми безпеки смарт-контрактів	35
1.4 Види вразливостей смарт-контрактів на рівні Solidity-коду	41
1.4.1 Вразливість повторного входу.....	43
1.4.2 Вразливість цілочисельного переповнення/недоповнення	44
1.4.3 Вразливість контролю доступу.....	45
1.4.4 Вразливість неправильно обробленого винятку з зовнішньої функції	46
1.4.5 Вразливість відмови в обслуговуванні	47
1.4.6 Вразливість невідповідності типу	50
1.4.7 Вразливість виклику невідомої функції	50
1.4.8 Вразливість незахищеного виклику функції самознищення смарт-контракту	50
1.4.9 Вразливість використання джерела транзакції.....	51
1.4.10 Вразливість суворого порівняння балансу	53
1.4.11 Вразливість залежності від часової мітки (timestamp dependency).....	53
1.5 Класифікація сучасних методів виявлення вразливостей у смарт-контрактах .	54
1.5.1 Формальний аналіз смарт-контрактів.....	55
1.5.2 Статичний аналіз смарт-контрактів	57
1.5.3 Динамічний аналіз смарт-контрактів.....	58
1.5.4 Підходи на основі машинного навчання	60
1.5.5 Гібридні методи.....	63

1.6 Формалізована постановка проблеми	65
1.7 Висновки до першого розділу.....	66
РОЗДІЛ 2. МЕТОД ФОРМУВАННЯ ЗБАЛАНСОВАНОГО ДАТАСЕТУ ДЛЯ	
ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ У СМАРТ-КОНТРАКТАХ.....	68
2.1 Аналіз існуючих датасетів смарт-контрактів.....	68
2.2 Метод формування датасету	76
2.2.1 Збір даних.....	78
2.2.2 Детермінований метод ін'єкції вразливостей у смарт-контракти	80
2.2.3 Метод ін'єкції вразливостей у смарт-контракти на основі великих мовних моделей.....	86
2.2.4 Створення збалансованого датасету та його переваги.....	91
2.2.5 Попередня обробка даних	92
2.3 Висновки до другого розділу	94
РОЗДІЛ 3. ПОБУДОВА МЕТОДІВ І МОДЕЛЕЙ ВИЯВЛЕННЯ ТА КЛАСИФІКАЦІЇ	
ВРАЗЛИВОСТЕЙ ТА ЇХ ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ	96
3.1 Початкова модель: BiGRU-ATT	97
3.1.1 Обґрунтування вибору GRU перед LSTM.....	97
3.1.2 Двонаправлена обробка (BiGRU) для покращення розуміння коду	101
3.1.3 Механізм уваги (Attention) для фокусування на важливих частинах коду	102
3.1.4 Векторне представлення коду смарт-контрактів.....	103
3.1.5 Загальна архітектура моделі BiGRU-ATT	105
3.1.6 Результати моделі BiGRU-ATT	106
3.1.7 Обмеження моделі BiGRU-ATT	108
3.2 Багатоміткова модель CodeBERT-BiGRU	108
3.2.1 CodeBERT для семантичного аналізу програмного коду на основі NLP ..	110
3.2.2 Архітектура CodeBERT-BiGRU	112
3.2.3 Переваги моделі CodeBERT-BiGRU перед BiGRU-ATT	119
3.3 Виявлення та локалізація вразливостей у смарт-контрактах на основі	
дворівневої уваги з графовим підкріпленням	126
3.3.1 Метод впровадження графового зсуву у матриці уваги	128
3.3.2 Метод побудови ієрархічної (дворівневої) уваги	131
3.3.3 Метод побудови клас-орієнтованих карт важливості	135

3.3.4 Метод причинних регуляризаторів	139
3.3.5 Метод побудови САМ-узгодженої уваги	140
3.3.6 Метод слабосупервізованої локалізація	143
3.3.7 Абляційні дослідження.....	146
3.3.8 Аналіз стійкості моделі до збурень та змін.....	151
3.3.9 Метод регуляризації для калібрування ймовірнісних оцінок моделей глибокого навчання.....	153
3.4 Висновки до третього розділу.....	158
РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАСОБУ АУДИТУ СМАРТ-КОНТРАКТІВ НА ОСНОВІ CODEBERT-GDLA	160
4.1 Мета впровадження та вимоги.....	160
4.2 Архітектура інструмента	164
4.3 Потoki даних і сценарії	166
4.4 API та формати даних	167
4.5 Інтерфейс користувача	168
4.6 Приклад застосування.....	172
4.7 Апробація запропонованих рішень на практиці	176
4.7 Висновки до четвертого розділу.....	178
ВИСНОВКИ.....	180
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	184
ДОДАТОК А АКТИ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОГО ДОСЛІДЖЕННЯ.....	199
ДОДАТОК Б ДОВІДКА ПРО ВИКОРИСТАННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОЇ РОБОТИ У НАВЧАЛЬНОМУ ПРОЦЕСІ НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ «ОДЕСЬКА ПОЛІТЕХНІКА»	201
ДОДАТОК В ФРАГМЕНТИ ВИХІДНОГО КОДУ ПРОГРАМНИХ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ.....	203
ДОДАТОК Г СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА	221
ДОДАТОК Д ВІДОМОСТІ ПРО АПРОБАЦІЮ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОЇ РОБОТИ.....	222

ВСТУП

Інтенсивний розвиток блокчейн-технологій призвів до широкого впровадження смарт-контрактів у фінансовій, юридичній, управлінській та інших сферах. Смарт-контракти, зокрема на платформі Ethereum, забезпечують децентралізоване виконання угод без участі посередників, що дозволяє значно знизити витрати та підвищити ефективність бізнес-процесів. Однак зростання популярності смарт-контрактів супроводжується також зростанням кількості атак та фінансових втрат через наявність вразливостей у їхньому програмному коді.

За даними аналітичних звітів SlowMist та Chainalysis, щорічні втрати внаслідок експлуатації вразливостей у смарт-контрактах вимірюються мільярдами доларів США. За даними Chainalysis, у 2024 р. хакери викрали 2,2 млрд доларів у криптоактивах; основною ціллю залишаються DeFi-протоколи. Згідно зі звітом SlowMist, лише у 2022 р. зафіксовано 303 інциденти безпеки блокчейнів із сукупними втратами $\approx 3,8$ млрд доларів, переважно в децентралізованих фінансах та крос-чейн містках. Окремі експлойти досягають сотень мільйонів: Ronin bridge (~ 600 млн доларів), Mango Markets (~ 100 млн доларів), Bybit (2025 р.) – 1,5 млрд доларів.

Попри аудити та статичні аналізатори, численні контракти з критичними вразливостями вже проходили аудит, а інструменти на основі правил не встигають за новими класами атак. Це обумовлює надзвичайну актуальність дослідження проблем виявлення вразливостей у смарт-контрактах ще на етапі їх розробки. Особливої ваги набуває застосування сучасних методів машинного навчання та глибоких нейронних мереж, здатних автоматизувати аналіз програмного коду та виявляти потенційні загрози.

Разом із тим існує протиріччя: підвищення точності виявлення вразливостей за рахунок експертного (ручного) аудиту неминуче знижує швидкість аналізу та масштабованість процесу, тоді як наявні автоматизовані засоби забезпечують високу продуктивність і охоплення, але поступаються за точністю та повнотою детекції, а також обмежені у пояснюваності результатів. Це створює потребу в розробленні нових моделей і методів, здатних поєднати переваги обох підходів – зберегти точність

і інтерпретованість, притаманні експертному аудиту, та досягти швидкодії й відтворюваності автоматизованих систем.

Таким чином, дослідження методів і моделей виявлення вразливостей у смарт-контрактах на основі машинного навчання є актуальним як з теоретичної, так і з практичної точки зору, оскільки сприяє підвищенню рівня безпеки блокчейн-систем та зниженню фінансових ризиків користувачів.

Зв'язок роботи з науковими програмами, темами, планами. Чинну дисертаційну роботу виконано у відповідності до пріоритетних напрямків науково-дослідних робіт Національного університету «Одеська політехніка», згідно координаційних планів Міністерства освіти і науки України, зокрема, в рамках наукових досліджень за держбюджетними науково-дослідними роботами (НДР): «Методи та програмні засоби інтерпретації моделей машинного навчання непараметричних динамічних об'єктів» № 210-63, 2021-2025 р. р. (№ держ. реєстрації 0122U002161) та «Моделі та методи штучного інтелекту та розподілених систем обробки великих даних» № 265-73, 2025-2027 р.р. (№ держ. реєстрації 0125U003871).

Метою дослідження є підвищення ефективності за відповідними метриками виявлення, класифікації та локалізації вразливостей у смарт-контрактах за рахунок розробки моделей та методів машинного навчання, які становлять цілісний технологічний стек інтелектуального аналізу вихідного коду смарт-контрактів.

Для досягнення поставленої мети дисертаційного дослідження було визначено комплекс взаємопов'язаних задач, що охоплюють повний цикл побудови, навчання, інтерпретації та оцінювання моделей глибинного навчання для багатоміткового виявлення та локалізації вразливостей у смарт-контрактах:

1. створити збалансований і репрезентативний навчальний датасет смарт-контрактів із анотацією типів вразливостей, включно з інжектуванням і генерацією прикладів;
2. розробити методи і моделі багатоміткової класифікації вразливостей у смарт-контрактах на основі глибоких моделей з урахуванням синтаксичних та семантичних структур коду;

3. підвищити надійність і стабільність моделей через калібрування й оцінку стійкості до модифікацій коду;
4. розробити слабосупервізовані методи локалізації вразливих фрагментів коду, які працюють без токенної розмітки та забезпечують інтерпретованість результатів;
5. апробація запропонованих методів шляхом їх реалізації у програмному засобі та перевірки ефективності на реальних смарт-контрактах.

Об'єктом дослідження є процеси виявлення, класифікації та локалізації вразливостей у смарт-контрактах.

Предметом дослідження є методи та моделі машинного навчання для багатоміткового виявлення, класифікації та локалізації вразливостей у смарт-контрактах.

Методи дослідження. Для розв'язання поставлених задач застосовано методи глибинного навчання для багатоміткової класифікації коду: трансформерні представлення (CodeBERT) у поєднанні з рекурентним моделюванням (BiGRU) та двошаровими механізмами уваги; методи пояснюваності рішень (attention-карти, Grad-CAM) для локалізації вразливостей на рівні токенів/рядків; калібрування імовірностей для підвищення надійності прогнозів. Використано комбіновані підходи до формування датасету (детерміноване інжектування вразливостей і LLM-генерація/аугментація прикладів), оптимізаційні процедури навчання (AdamW, підбір гіперпараметрів, рання зупинка), а також методи математичної статистики для оцінювання якості (precision, recall, F1, ROC-AUC, IoU/overlap@k для локалізації), перевірки значущості покращень та аналізу стійкості до збурень та змін (мутації коду, перейменування, зміни форматування).

Наукова новизна отриманих результатів. У дисертаційній роботі отримано наступні наукові результати, що визначають її новизну:

1. **вперше запропоновано комбінований метод формування навчального датасету** за рахунок поєднання автоматичного інжектування аномальних конструкцій у коректний код смарт-контрактів із генерацією додаткових

прикладів великою мовною моделлю з подальшою верифікацією, що дозволило забезпечити збалансованість вибірки;

2. **вперше побудовано модель CodeBERT-GDLA** на основі вдосконалення архітектур BiGRU-ATT і CodeBERT-BiGRU за допомогою запропонованих методів графового зсуву і дворівневої графово-орієнтованої уваги, що дозволило підвищити ефективність виявлення та класифікації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів;

3. **удосконалено метод регуляризації для калібрування ймовірнісних оцінок багатоміткової класифікації вразливостей смарт-контрактів** за рахунок застосування оптимізації очікуваної калібрувальної похибки, температурного масштабування та фокального приглушення надмірної впевненості безпосередньо в процесі навчання моделі CodeBERT-GDLA, що забезпечує зниження її надмірної впевненості та підвищує надійність результатів інтелектуального аналізу вихідного коду смарт-контрактів;

4. **отримав подальший розвиток метод слабосупервізованого виділення вразливих фрагментів коду** за рахунок інтеграції CAM-узгодженої уваги з причинно-орієнтованою регуляризацією, що забезпечує підвищення ефективності локалізації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів.

Практичне значення одержаних результатів. Результати дисертаційного дослідження мають прикладне значення для підвищення ефективності систем аналізу безпеки програмного коду на етапах розробки та передрозгортальної перевірки. Розроблені моделі та методи забезпечують точніше й своєчасніше виявлення ризиків, зменшують обсяг ручної перевірки та сприяють зниженню імовірності експлуатації вразливостей. Запропоновані рішення реалізовано у вигляді модулів, придатних для інтеграції в системи аудиту смарт-контрактів.

У процесі дослідження сформовано спеціалізований збалансований датасет смарт-контрактів з анотацією основних типів вразливостей, який може використовуватися для навчання, тестування та порівняльної оцінки моделей

автоматизованого аналізу безпеки, а також як еталонна вибірка для подальших наукових досліджень.

Розроблені в дисертаційній роботі моделі та методи отримали впровадження у діяльності товариства з обмеженою відповідальністю НВО «Діскрет» та товариства з обмеженою відповідальністю НВП «КАРЕ», а також знайшли відображення у навчальному процесі Національного університету «Одеська політехніка».

Особистий внесок здобувача. Основні наукові положення, висновки і рекомендації, що містяться в дисертації та виносяться на захист, отримано здобувачем особисто у період 2023-2025 рр. У статті [1] та роботах [7-10] внесок здобувача полягає у проведенні аналізу предметної області, класифікації вразливостей і порівнянні традиційних підходів та підходів на основі машинного навчання до їх виявлення; у статтях [2, 6] – запропоновано архітектури багатоміткової детекції на основі BiGRU з механізмами уваги та дворівневою увагою з графовим підкріпленням; у статті [3] – розроблено підхід до ідентифікації та локалізації вразливостей на базі аналізу векторів уваги в BERT-подібній моделі; у статті [4] – запропоновано комбінований метод локалізації з використанням уваги та наочних карт важливості; у роботах [5] та [11] – запропоновано методики контрольованої ін'єкції вразливостей і побудови збалансованих навчальних вибірок для багатоміткової класифікації.

Апробація результатів дисертації. Основні положення та практичні результати дисертаційної роботи доповідалися й одержали схвалення на таких наукових заходах: міжнародна науково-практична конференція Modern Information Technology 2023 / Сучасні Інформаційні Технології 2023 (м. Одеса, 18-19 травня 2023 р.); XIII International Scientific and Practical Conference “Social Ways of Training Specialists in the Social Sphere and Inclusive Education” (м. Прага, Чеська Республіка, 1-3 квітня 2024 р.); VI International Scientific Conference “Traditional and Innovative Approaches to Scientific Research” (м. Вінниця, Україна, 8 березня 2024 р.); X International Scientific and Practical Conference “Informatics. Culture. Technology” (вересень 2024 р.); VIII International Scientific Conference “Розвиток наукової думки постіндустріального суспільства: сучасний дискурс” (жовтень 2025 р.).

Публікації. Основні результати дисертаційної роботи викладено в 11 публікаціях, з них: 6 статей у наукових фахових виданнях України з технічних наук та 5 публікацій у працях і матеріалах наукових конференцій. Серед них: стаття [1] опублікована у виданні, включеному до Переліку фахових видань України (Категорія "Б", Scopus), стаття [2] у Springer CCIS (індексується у наукометричній базі Scopus) та стаття [3] у журналі Radio Electronics, Computer Science, Control (Категорія "А", індексується у наукометричній базі Web of Science).

Структура і обсяг роботи.

Дисертація складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи – 222 сторінки, містить 58 рисунка, 11 таблиць, 5 додатків. Список використаних джерел налічує 126 найменувань. У додатках наведено акти впровадження результатів дисертаційного дослідження, довідку про використання результатів роботи у навчальному процесі, фрагменти вихідного коду програмних інструментальних засобів, список наукових публікацій здобувача та відомості про апробацію результатів дисертаційної роботи.

РОЗДІЛ 1. АНАЛІЗ СУЧАСНИХ МЕТОДІВ І МОДЕЛЕЙ ВИЯВЛЕННЯ ТА КЛАСИФІКАЦІЇ ВРАЗЛИВОСТЕЙ У СМАРТ-КОНТРАКТАХ

Наведено теоретико-методичні аспекти та концептуальні засади аналізу безпеки смарт-контрактів у блокчейн-екосистемах, що функціонують в умовах високої варіативності коду та постійного зростання кількості вразливостей. Проаналізовано найбільш відомі вразливості смарт-контрактів. Представлені і проаналізовані сучасні методи та моделі виявлення помилок і вразливостей у смарт-контрактах, включно з формальними, статичними та динамічними підходами, а також методами машинного навчання й глибинного аналізу коду. Розглянуті існуючі інструменти та технології, які забезпечують аудит і верифікацію смарт-контрактів на різних стадіях їх життєвого циклу. Визначено ключові проблеми, пов'язані з низькою інтерпретованістю, обмеженою масштабованістю та високим рівнем хибнопозитивних результатів у традиційних системах аналізу.

1.1 Архітектура блокчейну

Блокчейн – це система розподіленого програмного забезпечення, яка дозволяє обробляти транзакції без допомоги довіреної третьої сторони. Транзакція – це операція збереження даних у блокчейні, під час якої відбувається передача криптоактивів чи іншої інформації між учасниками [1]. Завдяки таким характеристикам блокчейну, як прозорість, децентралізація та захист від втручання, забезпечується довіра зі сторони користувачів, оскільки майже неможливо підробити будь-які транзакції, що зберігаються в блокчейні, бо всі минулі транзакції піддаються перевірці та відстеженню.

Блокчейн представляє собою послідовність записів, які групуються в блоки, хешуються і пов'язуються з попереднім блоком. На рисунку 1.1 наведено структуру блокчейну.

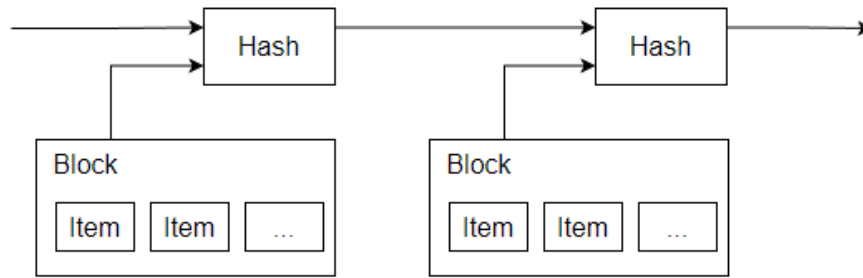


Рисунок 1.1 – Структура ланцюжка блоків

Традиційна модель транзакцій є централізованою. Учасники довіряють центральному органу з розрахунком, що він залишатиметься чесним під час перевірки операцій. Екземпляри записів зберігаються посередником. Навмисна чи ненавмисна компрометація центрального органу може завдати значних збитків системі. Модель блокчейну усуває посередника у вигляді центрального органу, натомість розповсюджуючи копії записів усім сторонам [2]. Блокчейн складається з учасників (вузлів), кожен з яких підключений через мережу P2P (Peer-to-Peer) та має власний екземпляр записів. Щоб забезпечити цілісність копії записів у кожного вузла, потрібен алгоритм консенсусу. Алгоритм консенсусу дозволяє кожному вузлу переконатися, що кожен доданий блок є законним. Це також запобігає від компрометації та розгалуження блокчейну [3]. Сатосі Накамото запропонував алгоритм консенсусу proof-of-work, згідно з яким майнери мають розгадувати складну криптографічну головоломку [4]. Майнери витрачають обчислювальні ресурси та винагороджуються за свої зусилля у вигляді криптовалюти. Інші алгоритми, такі як proof-of-stake, proof-of-burn, proof-of-elapsed-time та proof-of-access, були запропоновані для подолання деяких недоліків алгоритму proof-of-work. Вузли транслують зміни, формуючи нові блоки та проводять перевірку на основі правил моделі консенсусу. Після перевірки, блок додається до кожного ланцюжка [5]. Такий процес потенційно безпечніший, ніж традиційна модель, та не потребує агента-посередника. На рисунку 1.2 наведено порівняння традиційної моделі транзакцій та моделі транзакцій блокчейну.

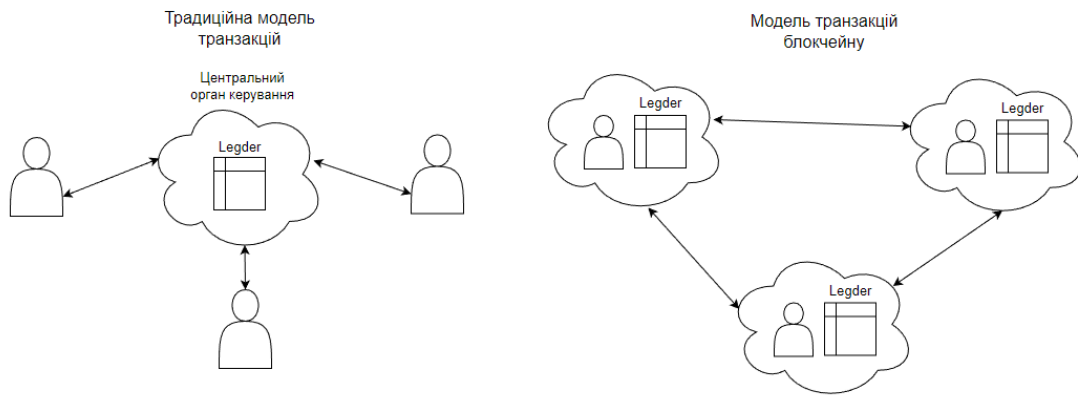


Рисунок 1.2 – Традиційна модель транзакцій та модель транзакцій блокчейну

Ethereum – це перша блокчейн-платформа, яка підтримує роботу смарт-контрактів [6]. Вона використовує мову програмування Solidity [7] для розробки смарт-контрактів і надає велику кількість прикладних програмних інтерфейсів (API). Облікові записи Ethereum поділяються на дві категорії: зовнішні облікові записи (EOA) [8], які мають баланс і можуть здійснювати виклики смарт-контрактів, і внутрішні облікові записи, які, на відміну від зовнішніх облікових записів, мають баланс і код смарт-контракту, який можна виконувати на додаток до викликів інших смарт-контрактів. Смарт-контракти виконуються у віртуальній машині Ethereum (EVM), яка є повною за Тьюрінгом та є стековою віртуальною машиною для виконання байт-кодів смарт-контрактів [9].

1.2 Смарт-контракти як предмет аналізу у контексті безпеки

1.2.1 Історичний розвиток концепції смарт-контрактів

Поняття смарт-контракту було вперше запропоноване в 1990-х роках Ніком Сабо – дослідником у галузі криптографії та права. Уже в 1994 році Сабо описав смарт-контракт як «комп'ютеризований протокол транзакцій, який виконує умови контракту» [10]. Цим терміном він прагнув підкреслити, що програми можуть автоматично забезпечувати виконання зобов'язань угоди та зменшувати потребу в посередниках. Ранні роботи Сабо (1997) і Міллера (1997) заклали теоретичний фундамент для таких «розумних» договорів. Однак практична реалізація цієї ідеї

тривалий час залишалася обмеженою – наприклад, у біткоїні використовувались скрипти, але їхні можливості були дуже вузькими [11].

Прорив у розвитку смарт-контрактів стався з появою платформи Ethereum. Її засновник – програміст Віталік Бутерін – у кінці 2013 року опублікував White Paper Ethereum, в якому назвав Bitcoin «слабкою версією» концепції смарт-контракту за Сабо і запропонував реалізацію потужнішої платформи зі спеціальною мовою програмування Solidity, що є Тюринг-повною [12]. Проєкт Ethereum був запущений у 2015 році й на практиці втілює ідею децентралізованого виконання програмного коду на блокчейні. Таким чином, Ethereum став першою масовою реалізацією блокчейн-платформи для смарт-контрактів, яка дозволяла розробникам створювати децентралізовані програми (dApp) будь-якої логіки, що виконуються у розподіленому середовищі. У White Paper Ethereum підкреслювалося, що Ethereum можна розглядати як загальну реалізацію "crypto-law" системи, яка виконує умови контрактів у блокчейні. З часів запуску Ethereum поняття смарт-контрактів стало міцно асоціюватися з цією платформою, а інші блокчейни почали впроваджувати схожі можливості.

1.2.2 Технічне визначення смарт-контракту

З технічної точки зору смарт-контракт – це програма, що зберігається та виконується у розподіленій мережі блокчейна. В оригінальному визначенні Сабо смарт-контракт – протокол, що формалізує та автоматично забезпечує виконання умов договору [13]. У контексті Ethereum це означає, що контракт – це спеціальний код, розгорнутий у блокчейні, який самостійно виконує запрограмовані інструкції при настанні визначених умов. В офіційній документації Solidity зазначено, що контракт – це сукупність коду (функцій) і даних (стану), яка знаходиться за певною адресою в блокчейні Ethereum [14]. Іншими словами, смарт-контракт на Ethereum виступає окремим об'єктом – обліковим записом блокчейна – який має свій власний адрес, баланс та внутрішній стан. Такий контракт може приймати транзакції від користувачів і інших контрактів, змінювати свій стан відповідно до закладеної логіки

та відправляти транзакції (наприклад, переказ коштів або виклик функцій) згідно з умовами, прописаними в його коді. Суть смарт-контракту подібна до звичайного контракту в тому, що він встановлює правила і гарантує їх виконання, але робить це автоматично через програмний код без довіри до сторонніх осіб.

Дуже вдалим є порівняння смарт-контракту з торговим автоматом (vending machine): як і автомат, смарт-контракт автоматично виконує закладену домовленість – достатньо внести правильні вхідні дані (наприклад, оплату або інші транзакції) і результат (вихід) гарантовано буде отримано. Ця аналогія, запропонована самим Сабо, допомагає зрозуміти принцип: код смарт-контракту містить логіку, яка самостійно стежить за виконанням визначених умов і забезпечує запрограмований результат [15]. Формально смарт-контракти в Ethereum виконуються у середовищі Ethereum Virtual Machine (EVM) – ізольованій віртуальній машині, яка працює на всіх вузлах мережі (рисунок 1.3). Код контракту після розгортання стає незмінним, а взаємодія з ним необоротно записується в блокчейн. Така детермінована природа виконання забезпечує довіру до результатів роботи смарт-контракту: жодна зі сторін не може підробити або відмінити виконання коду, оскільки воно гарантовано більшістю вузлів мережі [16].

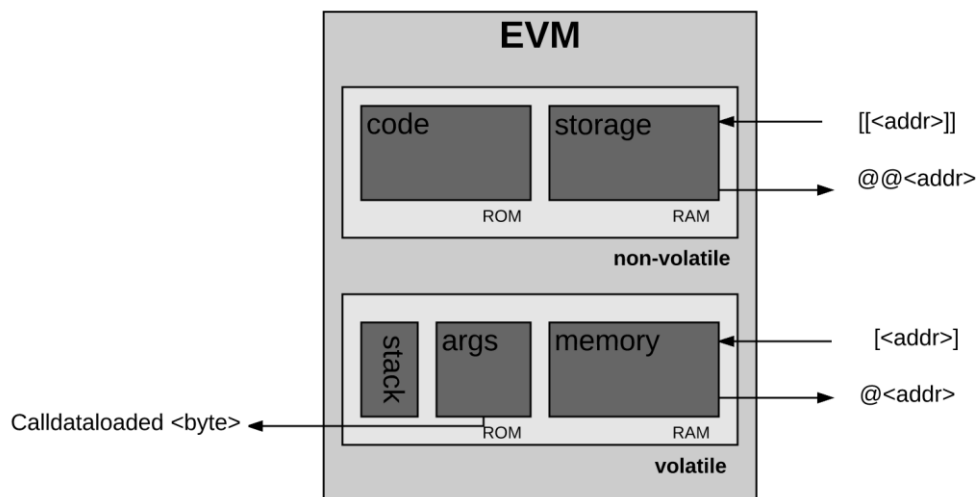


Рисунок 1.3 – Архітектура EVM

Можна виділити наступні переваги смарт-контрактів порівняно зі звичайними контрактами:

1. зменшення ризиків. Через незмінність блокчейну смарт-контракти не можуть бути довільно змінені після їх створення. Крім того, усі транзакції, які зберігаються та дублюються у всій розподіленій системі блокчейну, можна відстежувати та перевіряти;

2. скорочення витрат на адміністрування та обслуговування. Блокчейн забезпечує довіру до всієї системи за допомогою механізмів розподіленого консенсусу без використання посередника;

3. підвищення ефективності бізнес-процесів. Усунення залежності від посередника може значно підвищити ефективність бізнес-процесу.

1.2.3 Види смарт-контрактів

Розрізняють декілька видів смарт-контрактів відповідно до їх застосування, а саме:

1. юридичні смарт-контракти. Ці контракти є юридично обов'язковими, і сторони зобов'язані виконувати свої договірні зобов'язання. Якщо вони не виконають вимог, вони можуть зіткнутися з правовими наслідками. Такі смарт-контракти використовуються для обміну криптовалютами, в децентралізованих фінансах і Game-Fi проектах, та різноманітними платформами на блокчейні, від ринків NFT до метавсесвітів та торгівлі нерухомістю [17, 18];

2. децентралізовані автономні організації (DAO). Децентралізовані автономні організації є організаційною формою, за якої координація діяльності учасників та управління ресурсами відбувається відповідно до заздалегідь узгодженого та формалізованого набору правил, контроль за дотриманням яких виконується автоматично. Правила роботи DAO описуються в смарт-контрактах. Записи фінансових транзакцій DAO та програмні правила таких контрактів зберігаються у блокчейні. Прикладами DAO є протоколи керування Decentraland, Uniswap, Polkadot і MakerDAO. Згідно з правилами цих проектів, ними керують власники токенів, які можуть вносити різні пропозиції (наприклад, визначати структуру комісій) і

голосувати за них. Смарт-контракти DAO у цьому випадку відповідають за голосування та підрахунок голосів [19];

3. смарт-контракти логіки додатків (ALC). Ці контракти включають програмний код, який зазвичай синхронізується з іншими смарт-контрактами. Вони також забезпечують зв'язок між пристроями Інтернету речей (IoT) і технологією блокчейн. Крім того, такі смарт-контракти можуть обробляти зв'язок між блокчейном і блокчейн-оракулами [20].

1.2.4 Життєвий цикл смарт-контрактів

Смарт-контракти виконують різні задачі на різних етапах життєвого циклу, і кожен етап може призвести до потенційних проблем безпеки. Таким чином, підвищення безпеки смарт-контрактів вимагає від розробників і користувачів глибокого розуміння характеристик кожної фази життєвого циклу смарт-контракту.

Весь життєвий цикл смарт-контрактів складається з чотирьох послідовних фаз, а саме створення, розгортання, виконання та знищення смарт-контракту [21]. На рисунку 1.4 наведено життєвий цикл смарт-контракту. Він складається з наступних етапів:

1. створення смарт-контракту. Кілька залучених сторін спочатку ведуть переговори щодо зобов'язань, прав і заборон в контракті. Після ретельних обговорень і переговорів можна досягти угоди. Юристи або консультанти допоможуть сторонам скласти первинну договірну угоду. Програмісти потім перетворюють цю угоду, написану на природних мовах, у смарт-контракт, написаний на комп'ютерних мовах, включаючи декларативні мови. Подібно до розробки комп'ютерного програмного забезпечення, процедура перетворення смарт-контракту складається з проектування, впровадження та тестування. Варто зазначити, що створення смарт-контрактів є ітеративним процесом, що включає кілька раундів переговорів та ітерацій, в які залучені зацікавлені сторони, а саме юристи та розробники програмного забезпечення. Наразі існує понад 40 платформ, які підтримують розгортання смарт-контрактів, усі з яких мають відповідні мови розробки контрактів. Крім Solidity, існує

більше 10 мов програмування для розробки смарт-контрактів, таких як Vyper, Idris і Rust [22]. Розроблений код смарт-контракту необхідно скомпілювати. Віртуальна машина Ethereum перетворює байт-код у відповідний код операції для виконання смарт-контракту. Після компіляції компілятором, смарт-контракт Ethereum також створить двійковий інтерфейс програми (ABI) для блокчейну, який дає можливість іншим смарт-контрактам та користувачам взаємодіяти з цим смарт-контрактом [23];

2. розгортання смарт-контракту. Смарт-контракти працюють на платформі блокчейн і їх потрібно розгорнути в блокчейні, щоб синхронізувати їх з кожним вузлом для виклику. Контракт розгортається шляхом надсилання транзакції до блокчейну з порожнім одержувачем, що містить байт-код та іншу інформацію, і пакується майнером в блок та повертає адресу створеного смарт-контракту. Ця адреса є унікальним ідентифікатором контракту та за допомогою цієї адреси здійснюється виклик або доступ до контракту;

3. виконання смарт-контракту. Виклик смарт-контракту вимагає використання адреси контракту та двійкового інтерфейсу програми (ABI). Існує два способи виклику смарт-контракту: шляхом надсилання “транзакції” або “повідомлення”. “Транзакція” визначається в офіційному документі Ethereum як рядок даних, підписаний зовнішнім обліковим записом (EOA) [24], тому зовнішній обліковий запис може викликати контракт, надсилаючи транзакційний виклик. Визначення “повідомлення” в Ethereum Yellow Book наступне: це дані та “ефір”, що пересилаються між двома обліковими записами. Внутрішній обліковий запис контракту робить виклик інших контрактів через “повідомлення”. “Повідомлення” передаються внутрішньо і не синхронізуються з блокчейном, тому виклики між контрактами не записуються в блокчейні;

4. знищення смарт-контракту. Смарт-контракти реалізують функцію самознищення, яку потрібно прописати в контракті під час розробки. Оскільки дані блокчейну неможливо видалити, знищений контракт все ще існує в блокчейні, але його не можна викликати знову і його статус позначено як знищений.



Рисунок 1.4 – Життєвий цикл смарт-контракту

1.2.5 Роль смарт-контрактів у Web3 та децентралізованих екосистемах

Запровадження смарт-контрактів у Ethereum започаткувало цілу парадигму Web3 – децентралізованого Інтернету, де бекенд-логіка додатків працює на блокчейні. Смарт-контракти стали фундаментальним будівельним блоком сучасних децентралізованих застосунків. Зокрема, саме завдяки смарт-контрактам стрімко розвинувся сектор децентралізованих фінансів (DeFi) – фінансових платформ без посередників [25]. Смарт-контракти Ethereum вважаються базисом DeFi-протоколів – на них працюють децентралізовані біржі, кредитні платформи, стейблкоїни та інші фінансові інструменти нового покоління. Також смарт-контракти лежать в основі феномену невзаємозамінних токенів (NFT) – токенизації цифрових активів і творів, де контракт гарантує унікальність і право власності на актив [26]. Крім фінансів, смарт-контракти уможливили створення DAO (децентралізованих автономних організацій) – спільнот, що керуються правилами, закодованими у контрактах, без традиційної ієрархії. Фактично, статuti і “бюджети” DAO реалізовані смарт-контрактами, які автоматично виконують рішення спільноти (наприклад, виділення коштів або голосування) на основі заздалегідь визначених правил. Таким чином, значна частина екосистеми Web3 працює завдяки смарт-контрактам: саме вони є ядром Web3, забезпечуючи виконання різноманітних завдань – від фінансових транзакцій до

управління організаціями – без потреби в централізованому контролі. Практично всі ключові інновації блокчейн-екосистем (DeFi-протоколи, NFT-маркетплейси, DAO-платформи) або виникли на базі Ethereum, або тісно з ним інтегруються [27]. Це підтверджує тезу, що Ethereum і його смарт-контракти стали “серцем” децентралізованої інфраструктури, на яку рівняються інші платформи (рисунк 1.5).

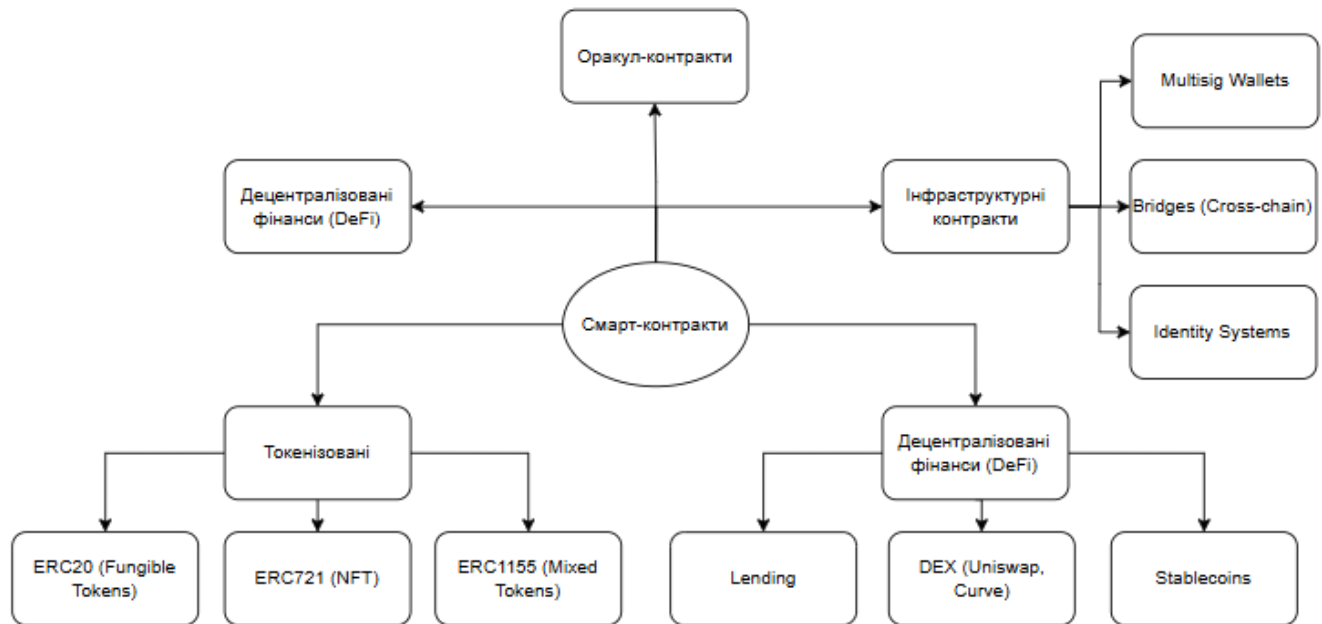


Рисунок 1.5 – Типи смарт-контрактів

1.2.6 Статистика смарт-контрактів в мережі Ethereum

За минуле десятиліття Ethereum перетворився на найбільшу платформу смарт-контрактів, що підтверджується її статистичними показниками. Кількість розгорнутих смарт-контрактів в мережі неухильно зростала і на 2024 рік сумарно перевищує 5 мільйонів контрактів від запуску Ethereum у 2015 році [28]. Це число ілюструє глибину та активність екосистеми: мільйони програм різного призначення були задеплойовані на блокчейн, формуючи основу для тисяч децентралізованих додатків. Кожен із цих контрактів – від простих токенів до складних протоколів – працює автономно у мережі, забезпечуючи ті чи інші сервіси. Обсяг транзакцій у мережі Ethereum також вражає. Станом на 2024 рік блокчейн Ethereum обробляє в середньому понад 1,16 мільйона транзакцій на день. Щоденна активність, як правило,

коливається в межах від ~1,0 до 1,2 млн транзакцій, причому у пікові періоди показники можуть наближатися до 2 мільйонів транзакцій за добу [28]. Для порівняння, ще декілька років тому такі обсяги були нижчими – мережа демонструє стабільне зростання навантаження. Зокрема, середнє число щоденних транзакцій у 2024 році збільшилося на $\approx 11-12\%$ у порівнянні з 2023 роком, що свідчить про сталий висхідний тренд використання Ethereum. Не менш показовим є й зростання кількості активних користувачьких адрес: щоденно в мережі взаємодіють сотні тисяч унікальних адрес (в середньому 300-400 тисяч, зі сплесками понад 500 тисяч під час значущих подій) [29]. Це вказує на розширення спільноти користувачів і розробників, що залучені до екосистеми смарт-контрактів.

Також Ethereum утримує лідерство у сфері децентралізованих фінансах – на його контрактах зосереджено понад \$50 млрд заблокованої вартості (TVL) в різних протоколах. Значні обсяги проходять і через NFT-платформи на Ethereum (для прикладу, лише за 1-й квартал 2024 року обсяг торгів NFT перевищив \$1,2 млрд) [29]. Усі ці цифри підтверджують, що смарт-контракти Ethereum є надзвичайно активними як за кількістю, так і за цінністю виконуваних операцій. Динаміка зростання мережі вказує на її зрілість та широкий попит на децентралізовані програми.

1.3 Актуальність проблеми безпеки смарт-контрактів

Смарт-контракти стали критичним компонентом сучасної блокчейн-екосистеми, особливо з бурхливим розвитком децентралізованих фінансів (DeFi). Вони керують значними обсягами цифрових активів і виконують фінансову логіку без участі людини. Це означає, що будь-яка вразливість у кодї смарт-контракту може миттєво призвести до прямих фінансових втрат, адже виконання контракту незворотне і не потребує довіри до централізованих посередників. Як наслідок, безпека смарт-контрактів є надзвичайно важливою: експлуатація навіть однієї помилки може поставити під загрозу кошти тисяч користувачів і підірвати довіру до всієї платформи. Наприклад, у випадку крос-чейн містків – спеціальних смарт-контрактів для переносу активів між блокчейнами – в одному контракті можуть

концентруватися величезні суми. Будь-яка помилка в такому коді фактично гарантовано буде знайдена й використана зловмисниками, адже ці «медові горщики» зберігають сотні мільйонів доларів [30].

Фінансові втрати, пов'язані з атаками на смарт-контракти, сягають мільярдів доларів, що підкреслює критичність проблеми. За даними аналітичної компанії Chainalysis, лише у 2022 році хакери викрали еквівалент \$3,8 млрд, що стало рекордом за всю історію криптовалют [31]. Більшість цих втрат (понад 80%) припадає саме на протоколи DeFi, які працюють на смарт-контрактах. Подібні оцінки наводить і звіт компанії SlowMist: за 2022 рік зафіксовано 303 інциденти безпеки, що призвели до збитків у розмірі приблизно \$3,77 млрд (за цінами на момент інцидентів). Хоча ця сума менша, ніж рекордні ~\$9,8 млрд втрат у 2021 році, загальний тренд залишається невтішним – щороку користувачі та інвестори втрачають мільярди через вразливості смарт-контрактів. Згідно зі SlowMist, понад половину втрат 2022 року (~55%) становлять інциденти саме в секторі децентралізованих фінансів на різних блокчейнах, а близько третини – атаки на крос-чейн містки (~32% від загальних збитків). Це свідчить, що атакуючі усе частіше націлюються на складні багато компонентні протоколи, де сконцентровано максимальні суми коштів [32].

Яскраві приклади атак на смарт-контракти демонструють різноманітність вразливостей і масштаби потенційних збитків. Нижче наведено кілька резонансних інцидентів та типових технічних причин, що до них призвели:

1. атака повторного входу (reentrancy). Ця вразливість дозволяє зловмиснику багатократно викликати вразливу функцію контракту до того, як попередній виклик завершить виконання, фактично «входячи» в одну і ту ж функцію повторно без належного оновлення стану. Класичним прикладом є злам проєкту The DAO у 2016 році – перший гучний інцидент в історії Ethereum. Менш ніж за три місяці після запуску The DAO хакер використав атаку повторного входу, щоб поступово вилучити з контракту організації близько \$60 млн в Ether [33]. Вразливий смарт-контракт дозволяв повторно викликати функцію виведення коштів до оновлення балансу, через що зловмисник викрав більшу частину фонду. Наслідки були катастрофічними: спільнота була змушена піти на хард-форк Ethereum задля

відновлення вкрадених коштів, що призвело до розділення мережі на Ethereum та Ethereum Classic. Цей випадок продемонстрував, наскільки серйозними можуть бути наслідки однієї помилки в смарт-контракті;

2. маніпулювання оракулом (oracle manipulation). Багато децентралізованих протоколів покладаються на зовнішні оракули цін, які надають смарт-контрактам актуальні ціни активів. Якщо нападник зможе викривити ці цінові дані, система почне приймати рішення на основі фальшивих курсів. Зазвичай для цього застосовуються флеш-позики (миттєві необезпечені позики в межах однієї транзакції) з метою штучно змінити ціну на децентралізованій біржі. Перший гучний випадок такої атаки стався у 2020 році на протоколі bZx, де зловмисник через маніпуляцію ціною в Uniswap зміг прибуткувати на ~\$350 тис [34]. Надалі атаки цього типу почастишали й стали приносити на кілька порядків більші прибутки хакерам. Так, у жовтні 2022 року протокол Mango Markets (Solana) було експлуатовано шляхом маніпуляції ціною малоліквідного токена MNGO: нападник розігнав курс токена, штучно завищивши заставну вартість, і отримав позику, не забезпечену реальними активами. В результаті з Mango Markets було виведено близько \$100 млн користувацьких коштів [35]. Цей інцидент підтвердив, що атаки на основі маніпуляції оракульними даними здатні завдати протоколам суттєвих фінансових збитків;

3. переповнення цілого (integer overflow). Це класична помилка програмування, яка в ранніх версіях Solidity призводила до непередбачуваної поведінки без використання безпечних математичних бібліотек. Переповнення виникає, коли при додаванні або множенні чисел результат перевищує максимально допустиме значення змінної, що «перевертало» число (наприклад, $255 + 1$ ставало 0 в типі `uint8`). В контексті смарт-контрактів така вразливість може дозволити атакувальнику збільшити свій баланс до астрономічного рівня або генерувати нові токени з повітря. Показовим є інцидент з токеном BeautyChain (BEC) у квітні 2018 року: через помилку переповнення в коді функції `batchTransfer` зловмисникам вдалося випустити надзвичайно велику кількість tokenів BEC практично з нічого [36]. Це призвело до обвалу вартості токена і значних втрат для інвесторів. Згодом розробники Solidity усунули цю уразливість, впровадивши автоматичні перевірки переповнення

(починаючи з версії 0.8), однак історія з ВЕС наголосила на необхідності ретельного тестування навіть базових арифметичних операцій у смарт-контрактах.

Перелічені атаки – лише мала частина зростаючого арсеналу зловмисників. Щороку фіксуються все нові види експлойтів, у тому числі складні комбіновані атаки на цілі екосистеми. Це відбувається на тлі стрімкого ускладнення самих смарт-контрактів і протоколів. Сучасні DeFi-програми часто складаються з цілої мережі інтерактивних контрактів, що працюють у різних блокчейнах, взаємодіють з зовнішніми даними та містять все більш складну бізнес-логіку. Така підвищена складність розширює простір можливих вразливостей і сценаріїв атак. Достатньо згадати, що у 2022 році чотири з десяти найбільших криптовалютних пограбувань були пов'язані саме з експлойтами крос-чейн містків (Ronin, Binance Smart Chain, Wormhole тощо), кожен з яких приніс атакувальникам від \$300 до \$600 млн. Зокрема, взлом мосту Ronin у березні 2022 року – випадок, коли атакувальники скомпрометували валідатори сайдчейну проєкту Axie Infinity – призвів до крадіжки еквівалента \$620 млн в криптовалюті. Це найбільша сума, будь-коли вкрадена через вразливість смарт-контрактів, що підкреслює реальні ризики для індустрії. Попри усвідомлення цих загроз, наявні засоби захисту смарт-контрактів усе ще виявляються недостатніми перед сучасними атаками. Стандартною практикою для команд розробників є аудит смарт-контрактів – ручна перевірка коду фахівцями з безпеки перед розгортанням протоколу. Однак аудити не гарантують повної безпеки. Згідно з дослідженням AnChain.AI (2023), серед проаналізованих випадків зламу смарт-контрактів на суму ~\$2,8 млрд, майже 92% зламаних контрактів вже пройшли хоча б один аудит перед експлуатацією вразливості [37]. Іншими словами, навіть залучення провідних аудиторських фірм не убезпечило більшість проєктів від критичних помилок у коді. Причини криються як у людському факторі (аудитори можуть щось пропустити), так і в обмежених рамках одноразового аналізу – нові версії протоколів та складна взаємодія між контрактами здатні породити непередбачені уразливості, які не були очевидними під час початкового аудиту.

Окрім аудитів, розробники покладаються на автоматизовані засоби аналізу безпеки – статичні аналізатори коду (на кшталт Mythril, Slither) та інструменти

формальної верифікації [38]. Статичний аналіз дозволяє виявити типові помилки (наприклад, відсутність перевірки граничних умов, можливість повторного входу тощо) за заданим набором правил. Проте такі інструменти теж мають суттєві обмеження. Вони часто генерують численні хибні спрацювання або пропускають нестандартні баги, оскільки покладаються на наперед визначені шаблони вразливостей і не розуміють контексту програмної логіки. Правила, закладені в аналізатор, не встигають за появою нових видів атак і швидко втрачають актуальність. Як результат, статичний аналіз може дати розробникам помилкове відчуття безпеки – відсутність попереджень не гарантує відсутності вразливостей. Формальна верифікація, яка математично доказує коректність контракту, теж не панацея: вона потребує значних зусиль і експертних знань, а застосовується переважно до окремих критичних контрактів через складність і високу вартість. У масштабах же цілої DeFi-екосистеми формальні методи практично не покривають всі можливі взаємодії між контрактами.

Таким чином, зростаюча складність смарт-контрактів та частота успішних атак демонструють, що сучасні підходи до безпеки потребують якісного посилення. Існує нагальна потреба в нових методах захисту, здатних проактивно виявляти потенційні вразливості та адаптуватися до появи нових класів атак. Перспективними напрямками є підвищення автоматизації аналізу та застосування методів штучного інтелекту. Зокрема, сучасні дослідження пропонують використовувати машинне навчання для пошуку вразливостей у смарт-контрактах, що дозволяє моделі самостійно навчитися шаблонам небезпечної поведінки без явного програмування правил. Глибокі нейронні мережі можуть враховувати семантику коду та виявляти приховані залежності, які не під силу виявити простими лінтерами. Автоматизовані інтелектуальні системи аналізу здатні працювати безперервно, перевіряючи оновлення контрактів і модулів великих протоколів у режимі реального часу, що особливо актуально для динамічної DeFi-сфери [39].

Отже, загальні тенденції вказують на нарощування обсягів інвестицій в безпеку смарт-контрактів і одночасне збільшення складності атак. Це обґрунтовує актуальність досліджень із застосуванням навчальних алгоритмів для виявлення

вразливостей до розгортання контрактів у мережі [40]. Причини, чому смарт-контракти особливо вразливі до атак зловмисників, можна підсумувати в наступних чотирьох аспектах:

1. сучасні мови програмування та інструменти для смарт-контрактів (наприклад, Solidity) все ще знаходяться на стадії зародження та примітивні. Смарт-контракти, написані такими мовами програмування, відносно складніше перевірити. Зокрема, смарт-контракту дозволено взаємодіяти із функціями або інтерфейсами інших смарт-контрактів, що може призвести до повторних зовнішніх викликів і неочікуваних вразливостей;

2. розробники смарт-контрактів не можуть повністю зрозуміти основну логіку виконання нових мов програмування та інструментів аудиту, тому не завжди вдається передбачити всі можливі стани та середовища, з якими смарт-контракт може стикнутися в майбутньому, що робить їх схильними до помилок;

3. у традиційних програмах розробники можуть виправляти помилки, коли виявляють проблеми з безпекою. Однак, на відміну від звичайних програм, смарт-контракти та їхні стани зберігаються в незмінній мережі блокчейну, тому вони є незмінними. Таким чином, контракти, розгорнуті в мережі блокчейн, можуть мати потенційні проблеми з безпекою, але їх неможливо виправити;

4. оскільки смарт-контракти містять цифрові активи вартістю мільйони доларів, це робить їх привабливими для багатьох зловмисників. Високі прибутки спонукають зловмисників використовувати вразливі місця в смарт-контрактах для крадіжки грошей.

Все вищезазначене підкреслює, що проблема безпеки смарт-контрактів сьогодні є надзвичайно актуальною і комплексною. Незважаючи на прогрес у засобах захисту, кількість успішних атак свідчить про необхідність нових підходів та поглиблених досліджень у цій галузі.

1.4 Види вразливостей смарт-контрактів на рівні Solidity-коду

Для створення надійних смарт-контрактів, розробники мають орієнтуватися в найбільш поширених вразливостях смарт-контрактів. Цей підрозділ надає аналіз найбільш поширених вразливостей смарт-контрактів, а також наводить приклади експлуатації цих вразливостей з боку злоумисників та підходи до уникнення цих вразливостей в Solidity-кодi смарт-контрактів на платформі Ethereum.

З технічної точки зору, вразливості смарт-контрактів Ethereum можна розділити на три рівні, тобто рівень коду Solidity, рівень виконання EVM і рівень залежностей блоків. Організація NCC Group створила аналог проекту OWASP TOP 10 для вразливостей смарт-контрактів, який отримав назву Decentralized Application Security Project (DASP) TOP 10. У проекті представлено 10 найбільш критичних вразливостей смарт-контрактів. Крім цього, існує реєстр вразливостей смарт-контрактів SWC Registry, який постійно оновлюється [41]. На даний момент до нього було додано більш ніж 30 виявлених вразливостей. В дисертації розглядаються вразливості на рівні Solidity-коду.

У межах цієї дисертації основну увагу зосереджено на вразливостях, що поєднують високу поширеність, значний практичний ризик та репрезентативність для трьох зазначених рівнів. Зокрема, детально проаналізовано:

- повторний вхід (reentrancy) як класичний приклад міжрівневої вразливості, що проявляється на етапі виконання EVM через помилки в порядку операцій у кодi;
- залежність від часової мітки (timestamp dependency) як представника рівня залежностей від параметрів блоку;
- цілочисельне переповнення/недоповнення (integer overflow/underflow) як уразливість, що виникає через помилки арифметики та конфігурації перевірок у кодi і впливає на безпеку виконання.

Для забезпечення повноти та методичної цінності огляду, попри те що експериментальна частина роботи зосереджується на трьох репрезентативних категоріях (атака повторного входу, залежність від часової мітки та цілочисельні

переповнення/недоповнення), у цьому підрозділі також наведено узагальнений опис одинадцяти найпоширеніших вразливостей смарт-контрактів. Такий розширений перелік виконує дві функції: демонструє повний контекст типових помилок на всіх трьох рівнях (рівень коду Solidity, рівень виконання EVM, рівень залежностей від параметрів блокчейну), забезпечує єдність термінології та відповідність галузевим класифікаціям (на кшталт DASP/SWC) для подальшого зіставлення результатів.

У подальшому викладі, окрім детального розгляду трьох базових категорій, компактно охарактеризовано такі вразливості: повторний вхід (reentrancy); цілочисельне переповнення/недоповнення; помилки контролю доступу; неправильна обробка винятків зовнішніх викликів; відмова в обслуговуванні; невідповідність типів; виклик невідомої функції; незахищений виклик самознищення; використання джерела транзакції (tx.origin); суворе порівняння балансу; залежність від часової мітки (timestamp dependency) (рисунок 1.6). Для кожної категорії подано коротке визначення, типовий сценарій експлуатації, рівень віднесення у тришаровій класифікації та практичні рекомендації щодо запобігання помилкам у Solidity-кодi.

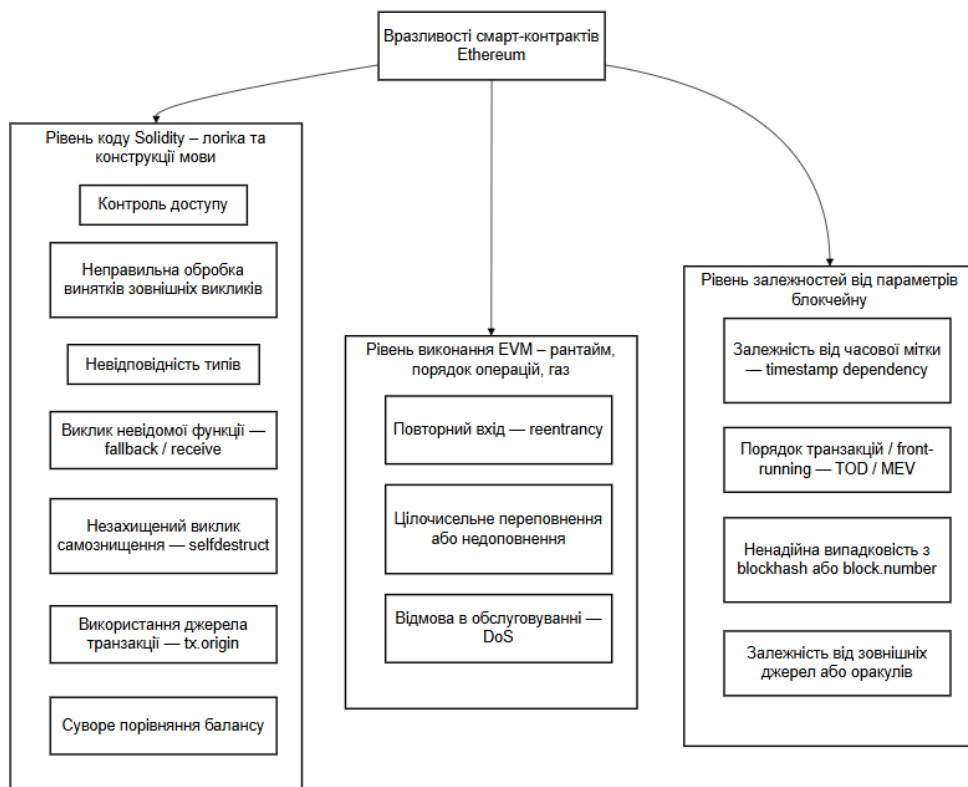


Рисунок 1.6 – Перелік вразливостей смарт-контрактів

1.4.1 Вразливість повторного входу

Вразливість повторного входу є найпоширенішою вразливістю смарт-контрактів. Виконання смарт-контрактів не є атомарним і послідовним, через що виникають певні шпарини в їх безпеці. Зловмисники можуть повторно увійти до викликаної функції під час поточного виконання програми [42]. Подібно до більшості мов програмування, смарт-контракти використовують міжфункціональні або міжконтрактні виклики для обробки бізнес-логіки. Але різниця полягає в тому, що такі виклики мають на меті передачу деяких цінних активів. Виклик операції *transfer* у контракті відправника неминуче запустить fallback-функцію у контракті одержувача. Коли смарт-контракт виконує міжконтрактну операцію переказу грошей, зловмисники можуть перехопити такий зовнішній виклик і виконати деякі зловмисні операції. Прикладом такої операції є впровадження зловмисником шкідливого коду у своїй fallback-функції, завдяки якому реалізується рекурсивний вхід в контракт жертви, щоб повторно викликати функцію *transfer* для викрадення ефіру. Вразливість повторного входу призвела до найбільшого інциденту з безпекою в історії смарт-контрактів (атаки на “The DAO”), який не лише призвів до втрат майже 60 мільйонів доларів, але й спричинив хардфорк Ethereum. На рисунку 1.7 наведено схему атаки повторного входу на “The DAO”.

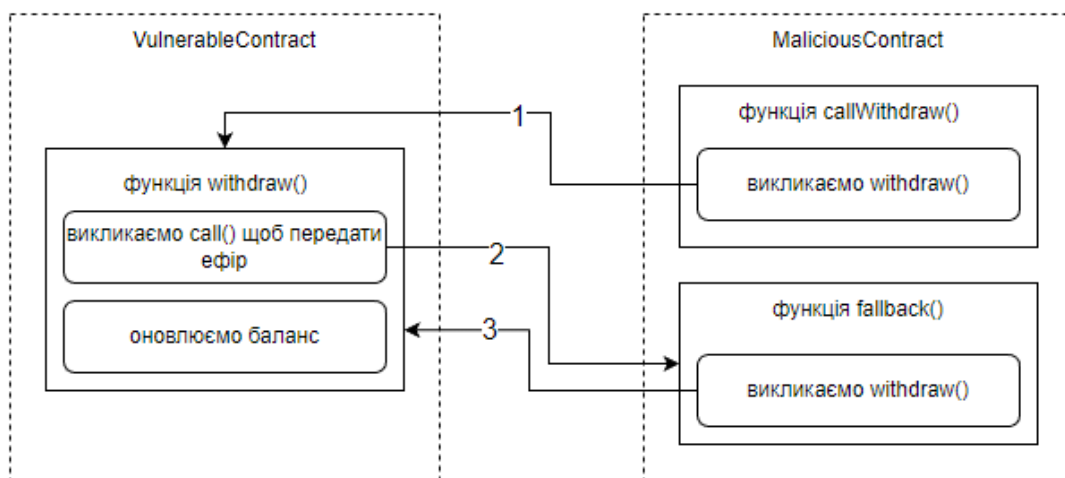


Рисунок 1.7 – Атака повторного входу

Одним з підходів до запобігання вразливості повторного входу є зміна балансу користувача до виконання будь-яких взаємодій з іншими смарт-контрактами. На

рисунку 1.8 наведені приклади вразливої та безпечної функцій. Так як баланс користувача в безпечній функції анулюється заздалегідь, при рекурсивному виклику методу *withdraw* криптоактиви не будуть надіслані повторно на адресу зловмисника.

```

1  // вразливий контракт
2  function vulnerableWithdraw() public {
3      uint256 amount = balances[msg.sender];
4      require(msg.sender.call.value(amount)());
5      balances[msg.sender] = 0;
6  }
7
8  // невразливий контракт
9  function withdraw() external {
10     uint256 amount = balances[msg.sender];
11     balances[msg.sender] = 0; // змінюємо баланс перед викликом методу call
12     require(msg.sender.call.value(amount)());
13 }

```

Рисунок 1.8 – Приклади вразливої та безпечної функцій

1.4.2 Вразливість цілочисельного переповнення/недоповнення

Цілочисельне переповнення/недоповнення є поширеною вразливістю в багатьох програмах. У смарт-контрактах є три типи вразливостей переповнення/недоповнення цілих чисел, а саме переповнення множення, переповнення додавання та недоповнення віднімання. У вихідному коді смарт-контрактів, цілі числа розглядаються як беззнакові числа з фіксованим розміром цілого типу. Очевидно, що якщо цілочисельна змінна перевищує певний діапазон, виникне помилка переповнення цілого числа [43].

Смарт-контракти Ethereum написані мовами високого рівня, такими як Solidity, які підтримують діапазон цілих чисел від *uint8* до *uint256*. Наприклад, якщо число має тип *uint8*, його значення зберігається як 8-бітне число без знаку в діапазоні від 0 до 256–1. Якщо значення поза цим діапазоном присвоюється змінній типу *uint8*, віртуальна машина Ethereum (EVM) автоматично скоротить старші цифри. Для прикладу уявімо, що в нас є змінна типу *uint8*, в якій зберігається значення 255 в десятковій системі або 11111111 в двійковій системі. Якщо ми захочемо додати 1 до цієї змінної, то в двійковій системі отримаємо число 100000000, яке займає 9 біт. Так

як наш тип даних займає лише 8 біт, то старший біт буде відкинуто. Таким чином, ми отримаємо значення змінної 0, а не 256 як планувалося.

На відміну від інших програм, втрати, спричинені вразливістю цілочисельного переповнення в смарт-контрактах, величезні та непоправні. Наприклад, вразливість цілочисельного переповнення використовувалася зломисниками, які нескінченно копіювали токени ВЕС, що призвело до того, що ціна токenu ВЕС впала до нуля. Функція *mintToken* смарт-контракту токenu Coinstar (CSTR) має цілочисельне переповнення, яке дозволяє власнику контракту встановити будь-яке значення балансу довільного користувача. Наразі, щоб запобігти цілочисельному переповненню смарт-контрактів, розробникам потрібно не лише перевіряти код вручну, але й використовувати бібліотеку SafeMath [44] для перевірки арифметичної логіки або компілятор Solidity, починаючи з версії 0.8, який автоматично дбає про перевірку коду на вразливість цілочисельного переповнення/недоповнення.

1.4.3 Вразливість контролю доступу

Основна причина вразливості, пов'язаної з контролем доступу, полягає в тому, що розробники забувають явно встановлювати обмеження доступів до функцій у смарт-контракті, що дозволяє зломисникам використовувати функції або змінні, до яких вони не повинні мати доступу. Вразливість контролю доступу зазвичай відображається на двох рівнях: на рівні коду та логічному рівні. На рівні коду існує чотири типи обмежень доступу до функцій і змінних смарт-контрактів, а саме публічні, приватні, зовнішні та внутрішні. На логічному рівні використовується модифікатори для обмеження прав доступу до функцій у смарт-контрактах, такі як *onlyOwner* і *onlyAdmin*. Функції без модифікаторів обмежень показують, що будь-хто має право на доступ до них, що може призвести до маніпулювання ключовими функціями зломисниками, ставлячи під загрозу безпеку смарт-контрактів [45].

1.4.4 Вразливість неправильно обробленого винятку з зовнішньої функції

Правильна обробка винятків є одним із найважливіших механізмів підвищення надійності та стійкості програм. Загалом смарт-контракт обробляє ненормальну поведінку за допомогою відкату, а саме припиняє поточне виконання контракту, відновлює попередній стан і повертає ідентифікатор помилки [46]. Однак, смарт-контракт, що викликає зовнішню функцію, може не мати змоги отримати інформацію про виняток з викликаного контракту. В більшості випадків, виняток, згенерований у функції зовнішнього смарт-контракту, повертається контракту, що його викликав. Тим не менш, деякі низькорівневі виклики функцій, такі як *call*, *callcode*, *send* та *delegatecall*, повертають лише значення *false*, а не саме виключення. Таким чином, неможливо судити про успішне виконання контракту на основі того, чи був згенерований виняток в смарт-контракті. В результаті, виконання контракту буде продовжене навіть у випадку виникнення виключення в зовнішньому контракті, що призведе до порушення програмної логіки, яким можуть скористатися зломисники.

Отже, слід обов'язково обробляти значення, що повертається під час виклику подібних функцій. На рисунку 1.9 наведено код вразливого смарт-контракту, який не обробляє значення, що повертає метод *call* при виклику методу *malicious_func* зовнішнього смарт-контракту.

```

1  pragma solidity 0.4.25;
2
3  contract VulnerableContract {
4      address MaliciousContract;
5
6      function callnotchecked() public {
7          // викликаємо вразливу функцію контракту VulnerableContract, яка повертає false
8          MaliciousContract.call(bytes4(sha3("malicious_func())));
9      }
10 }

```

Рисунок 1.9 – Вразливий смарт-контракт

На рисунку 1.10 наведено приклад смарт-контракту зломисника, який при виклику сторонніми контрактами методу *malicious_func* генерує виключення.

```

1  pragma solidity 0.4.25;
2
3  contract MaliciousContract {
4      function malicious_func() public {
5          require(false); // генеруємо виключення
6      }
7  }

```

Рисунок 1.10 – Смарт-контракт зловмисників

Одним з підходів до боротьби з цією вразливістю є обертання низькорівневих функцій, таких як *call*, функцією *require*.

1.4.5 Вразливість відмови в обслуговуванні

Відмова в обслуговуванні (DoS, Denial of service) є поширеною вразливістю смарт-контрактів на Ethereum [47]. Зловмисники зазвичай використовують таку вразливість, щоб зламати початкову логіку програми, що робить контракт нездатним надавати звичні послуги на деякий час або навіть назавжди. Зазвичай існує два типи DoS-атак проти смарт-контрактів.

Першим типом є DoS-атака, запущена через несподіваний виклик функції *revert* у зовнішній функції. Смарт-контракт буде сприйнятливим до DoS-атаки у випадку, коли він намагається надіслати кошти користувачеві, і подальша робота смарт-контракту залежить від того, що цей переказ коштів буде успішним. Проблема може виникнути у тому випадку, якщо кошти надсилаються на смарт-контракт, створений зловмисником, оскільки він може просто створити *callback*-функцію, яка скасовує всі платежі.

На рисунку 1.11 наведено код, вразливий до DoS-атаки, запущеної через несподіваний виклик функції *revert* у зовнішній функції. Як можна бачити в цьому прикладі, якщо зловмисник робить ставку зі смарт-контракту з *callback*-функцією, яка скасовує всі платежі, кошти ніколи не будуть йому відшкодовані, і, отже, ніхто ніколи не зможе зробити вищу ставку.

```

1  contract Auction {
2      address currentLeader;
3      uint highestBid;
4
5      function bid() payable {
6          require(msg.value > highestBid);
7          // Виплата минулому лідеру не проходить через
8          // виклик функції revert у зовнішньому контракті
9          require(currentLeader.send(highestBid));
10
11         currentLeader = msg.sender;
12         highestBid = msg.value;
13     }
14 }

```

Рисунок 1.11 – Смарт-контракт, вразливий до DoS-атаки, запущеної через несподіваний виклик функції *revert* у зовнішній функції

Ефективним способом боротьби з такою вразливістю, є виділення операції відшкодування в окрему функцію, яку учаснику потрібно буде викликати самостійно. Приклад безпечного коду наведено на рисунку 1.12.

```

1  contract Auction {
2      address highestBidder;
3      uint highestBid;
4      mapping(address => uint) refunds;
5
6      function bid() payable external {
7          require(msg.value >= highestBid);
8
9          if (highestBidder != address(0)) {
10             // записуємо в окремий масив кошти, які слід відшкодувати
11             refunds[highestBidder] += highestBid;
12         }
13
14         highestBidder = msg.sender;
15         highestBid = msg.value;
16     }
17
18     // користувач викликає метод для отримання відшкодування
19     function withdrawRefund() external {
20         uint refund = refunds[msg.sender];
21         refunds[msg.sender] = 0;
22         (bool success, ) = msg.sender.call.value(refund)("");
23         require(success);
24     }
25 }

```

Рисунок 1.12 – Приклад коду смарт-контракту, який убезпечує від DoS-атаки

Другим типом є DoS-атака запущена через перевищення ліміту газу блоку. Ліміт газу для блоку був впроваджений для того, щоб запобігати створенню зловмисниками нескінченного циклу при транзакціях. Якщо використання газу для транзакції перевищує цей ліміт, транзакція буде заблокована. Прикладом, коли

виникає подібна вразливість, є виконання логіки в циклі з дуже великою кількістю ітерацій. Таким чином, навіть якщо немає зловмисної атаки, смарт-контракт може мати проблеми через вихід за ліміт газу. Наприклад, надто великий набір користувачів, яким треба надіслати кошти, може перевищити ліміт газу та перешкодити успішній транзакції, потенційно назавжди заблокувавши кошти [48]. На рисунку 1.13 наведено код, який описує подібну ситуацію. Більш серйозною є проблема, коли зловмисник маніпулює кількістю використаного контрактом газу так, щоб вона досягала ліміту та процес транзакції завершувався невдачею.

```

1  struct Payee {
2      address addr;
3      uint256 value;
4  }
5
6  Payee[] payees;
7
8  function payOut() {
9      while (i < payees.length) {
10         payees[i].addr.send(payees[i].value);
11         i++;
12     }
13 }

```

Рисунок 1.13 – Приклад вразливого коду

Для того, щоб уникнути такої вразливості слід розбити таку транзакцію на декілька та перевіряти в передумові циклу, чи не було перевищено ліміт газу. Безпечний код наведено на рисунку 1.14.

```

1  struct Payee {
2      address addr;
3      uint256 value;
4  }
5
6  Payee[] payees;
7  uint256 nextPayeeIndex;
8
9  function payOut() {
10     uint256 i = nextPayeeIndex;
11     // перевіряємо, щоб кількість використаного газу
12     // не перевищувала ліміт для блоку
13     while (i < payees.length && msg.gas > 200000) {
14         payees[i].addr.send(payees[i].value);
15         i++;
16     }
17     nextPayeeIndex = i;
18 }

```

Рисунок 1.14 – Приклад коду смарт-контракту, який убезпечує від DoS-атаки

1.4.6 Вразливість невідповідності типу

Solidity – це строго типізована мова програмування, яка може автоматично перевіряти, чи є в програмі невідповідність типу. Наприклад, якщо значення рядкового типу присвоєно цілочисельній змінній, виникне помилка невідповідності типу. Однак у смарт-контрактах, навіть якщо тип не збігається в деяких випадках, контракт не може згенерувати виняток під час виконання програми. Відсутність ручного аудиту смарт-контракту може привести до виникнення вразливості невідповідності типів [49].

1.4.7 Вразливість виклику невідомої функції

Подібно до більшості мов програмування, смарт-контракт забезпечує унікальність функцій шляхом відповідності імені функції та кількості параметрів. Коли контракт викликає функцію в зовнішньому контракті та ім'я функції і кількість параметрів не відповідають жодній функції в контракті, що викликається, то в ньому автоматично запускається fallback-функція. У цей час може несподівано виникнути проблеми з безпекою, якщо зловмисний код прихований у fallback-функції [50].

1.4.8 Вразливість незахищеного виклику функції самознищення смарт-контракту

Смарт-контракти можуть використовувати функцію *delegatecall* для виклику функцій у зовнішньому контракті. Функція, яку викликали за допомогою *delegatecall*, виконується в контексті контракту, який її викликав. Тому, якщо викликана зовнішня функція має такі операції самознищення, як *selfdestruct* або *suicide*, “ефір” у поточному контракті (який викликає функцію через *delegatecall*), ймовірно, буде заморожено назавжди через виконання такої операції з самознищення. На рисунку 1.15 наведено два смарт-контракти: смарт-контракт *MaliciousContract* був створений

зловмисником та *VulnerableContract*, який має вразливість незахищеного виклику функції самознищення смарт-контракту. Якщо *VulnerableContract* через функцію *call* викликає метод *getMoney* зовнішнього контракту *MaliciousContract*, то контракт *VulnerableContract* буде самознищений, бо метод *selfdestruct* в методі *getMoney* буде викликаний в контексті контракту *VulnerableContract* [51].

```

1  contract MaliciousContract {
2      function getMoney() {
3          // знищуємо контракт VulnerableContract
4          selfdestruct(payable(address(this)));
5      }
6  }
7
8  contract VulnerableContract {
9      function call(address a) {
10         // викликаємо метод getMoney контракту MaliciousContract
11         a.delegatecall(bytes4(sha3("getMoney()")));
12     }
13 }

```

Рисунок 1.15 – Приклад коду, вразливого до незахищеного виклику функції самознищення

Ця атака була використана в протоколі Parity. Анонімний користувач знайшов і використав вразливість у смарт-контракті бібліотеки, зробивши себе власником контракту. Потім зловмисник приступив до самознищення контракту. Це призвело до блокування коштів у 587 унікальних гаманцях із загальною кількістю 513 774,16 ефіру [52]. Таким чином, функцію *delegatecall* слід використовувати для контрактів, до яких існує повна довіра.

1.4.9 Вразливість використання джерела транзакції

Смарт-контракт Ethereum має глобальну змінну, а саме *tx.origin*, яка може відстежувати весь стек викликів і повертати адресу, яка ініціювала транзакцію. Якщо контракт використовує цю глобальну змінну для авторизації та автентифікації, зловмисники можуть використати цю особливість *tx.origin* для розробки відповідної атаки для викрадення “ефіру”. Наприклад, інший контракт може використовувати fallback-функцію для повторного виклику смарт-контракту та проходження авторизації. На рисунку 1.16 наведений вразливий смарт-контракт

VulnerableContract, який під час свого створення записує адресу контракта, який його створив, в змінну *owner*. При виклику методу *transferTo* спочатку відбувається авторизація, шляхом порівняння значення *tx.origin* та змінної *owner*, а потім передача ефіру на адресу смарт-контракту зломисників [53].

```

1  contract VulnerableContract {
2      address owner;
3
4      constructor() public {
5          owner = msg.sender;
6      }
7
8      function transferTo(address payable dest, uint amount) public {
9          require(tx.origin == owner);
10         dest.transfer(amount);
11     }
12 }

```

Рисунок 1.16 – Вразливий смарт-контракт

На рисунку 1.17 наведено код зломисного контракту *MaliciousContract*. Коли контракт *VulnerableContract* пересилає “ефір” цьому контракту, в ньому викликається callback-функція, яка повторно викликає метод *transferTo* вразливого смарт-контракту. Таким чином, коли метод *transferTo* буде виконуватись, значення *tx.origin* залишиться минулим, тобто значенням буде адреса контракту *VulnerableContract*, а не контракту *MaliciousContract*. Отже, зломисники зможуть пройти авторизації і передавати “ефір” з контракту *VulnerableContract* до тих пір, поки він не закінчиться.

```

1  interface TxUserWallet {
2      function transferTo(address payable dest, uint amount) external;
3  }
4
5  contract MaliciousContract {
6      address payable owner;
7
8      constructor() public {
9          owner = msg.sender;
10     }
11
12     function() external {
13         TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
14     }
15 }

```

Рисунок 1.17 – Смарт-контракт зломисників, який експлуатує вразливість

Тому розробникам слід використовувати глобальну змінну *msg.sender*, а не *tx.origin* для авторизації користувачів.

1.4.10 Вразливість суворого порівняння балансу

Суворе порівняння балансу означає, що логіка виконання контракту покладається на те, що баланс контракту дорівнює точному значенню, але значення балансу контракту є прозорим і може бути змінено будь-яким користувачем. Наприклад, коли контракт викликає функцію самознищення, можна надіслати ефір до будь-якого контракту, і контракт-одержувач не може відхилити передачу. Якщо в контракті є така вразливість, як суворе порівняння балансу, зловмисник може змінити баланс контракту та спричинити помилку в логіці виконання контракту. Як показано на рисунку 1.18, для порівняння *this.balance* з *3ether* безпечніше використовувати оператор порівняння «більше або дорівнює», ніж порівняння «строого рівно» [54].

```
1  if (this.balance == 3ether): // поганий підхід
2      dosomething();
3  if (this.balance >= 3ether): // хороший підхід
4      dosomething();
```

Рисунок 1.18 – Код з вразливістю суворого порівняння балансу та без неї

1.4.11 Вразливість залежності від часової мітки (timestamp dependency)

Залежність від часової мітки визначається як клас вразливостей, за якого критичні для безпеки гілки виконання або результати обчислень контракту істотно залежать від значення *block.timestamp*. Оскільки часову мітку формально визначає виробник блоку (валідатор/майнер) у припустимих межах консенсусу, вона не є строго детермінованою зовнішньою константою: її можна зміщувати в невеликому часовому інтервалі без порушення протоколу. За таких умов використання *block.timestamp* для прийняття високоризикових рішень (визначення переможця, розподіл винагород, відкриття/закриття доступу, актування станів на «точний момент часу», генерація «випадкових» значень тощо) створює канал впливу на поведінку контракту. Навіть малий зсув часової мітки здатен змінити логічну гілку, спровокувати передчасне або відкладене спрацювання умови й, відповідно, трансформувати економічний результат або розподіл прав доступу. Ця вразливість

належить до рівня залежностей від параметрів блоку (block/chain environment) і часто маскується під «часову бізнес-логіку», коли точкові пороги або парності/модульні співвідношення за timestamp помилково трактуються як надійна ентропія чи об'єктивний маркер моменту події. З погляду безпечного проектування, block.timestamp може використовуватися лише для некритичних часових перевірок у вигляді інтервалів із достатнім запасом (window-based), де невелике зміщення не впливає на справедливість і безпеку, тоді як для задач, що потребують випадковості або точного фіксування «миті істини», слід застосовувати криптографічно обґрунтовані схеми (commit–reveal, перевірювані джерела випадковості, зовнішні оракули з доказовою перевіркою). Додатково до проєктних практик доцільним є впровадження правил статичного аналізу для виявлення використань block.timestamp у критичних ділянках коду, а також тестів граничних умов, що емулюють допустимі зсуви часової мітки. Сукупно такі заходи мінімізують простір для маніпуляцій із боку виробника блоку та унеможлиблюють підміни логіки, зумовлені недетермінованістю часових параметрів середовища виконання.

1.5 Класифікація сучасних методів виявлення вразливостей у смарт-контрактах

Різноманіття підходів до виявлення вразливостей у смарт-контрактах зумовлює потребу в їх систематизації. Існуючі методи можна поділити на кілька груп: формальні методи перевірки, статичний аналіз коду, динамічний аналіз, підходи на основі штучного інтелекту/машинного навчання (AI/ML), а також гібридні методи, що поєднують елементи кількох підходів [55]. Кожен з цих методів має власний принцип роботи, переваги та обмеження. У цьому підрозділі проведено системну класифікацію таких підходів, наведено короткий опис принципів їх роботи, приклади відомих інструментів та актуальних досліджень для кожної категорії (рисунок 1.19). В таблиці 1.1 наведено порівняння підходів до виявлення вразливостей у смарт-контрактах.

Таблиця 1.1 – Порівняння підходів до виявлення вразливостей у смарт-контрактах

Підхід	Інструменти	Переваги	Обмеження
Формальна верифікація	VeriSolid, Certora	Висока точність	Складність, обмеження за об'ємом
Статичний аналіз	Slither, SmartCheck	Швидкість, покриття	Багато хибно позитивних результатів
Динамічний аналіз	Manticore, Echidna	Реалістичність тестів	Повільність, неповнота
Глибоке навчання	SmartBERT, Mandi	Автоматизація, масштабованість	Потреба у даних, explainability
Гібридні методи	ContractWard	Баланс точність/швидкість	Складність реалізації

1.5.1 Формальний аналіз смарт-контрактів

Формальні методи передбачають побудову математичної моделі смарт-контракту та строгий доказ його коректності відносно формально заданих специфікацій. Іншими словами, розробник задає властивості безпеки або інваріанти, яким повинен задовольняти контракт (відсутність конкретних вразливостей або дотримання бізнес-логіки), і спеціалізовані інструменти перевіряють виконання цих властивостей у всіх можливих шляхах виконання програми. Формальна верифікація може здійснюватися методами автоматичного теоремодоказування або модельного перевіряння (model checking), які вичерпно обходять простір станів контракту в пошуках порушень специфікації. На практиці це часто реалізується шляхом

перекладу смарт-контракту в проміжне подання та розв’язання логічних формул за допомогою SMT- або SAT-розв’язувачів [56].

Популярні приклади включають платформу ZEUS – фреймворк автоматичної формальної верифікації смарт-контрактів, що використовує комбінацію абстрактної інтерпретації та символічної модельної перевірки [57]. Інструмент ZEUS переводить код контракту (написаний на Solidity чи іншій мові) у проміжний LLVM-байткод і автоматично перевіряє задані політики безпеки (наприклад, відсутність переповнення чисел або атака повторного входу) за допомогою логічних запитів до SMT-розв’язувача. Іншим напрямом є формальні семантичні моделі Ethereum, як-от KEVM (реалізація Ethereum Virtual Machine у формальному фреймворку K) чи використання мови F^* для специфікації контрактів. Компанії також розробляють власні інструменти: приклад – система Certora для формального доказу відповідності контракту специфікаціям (хоча вона є комерційною). Загалом, формальні методи дозволяють математично довести відсутність певних вразливостей, охоплюючи всі гілки виконання програми, що теоретично дає змогу виявляти навіть невідомі раніше вразливі сценарії.

В останні роки з’явилися огляди і мета-дослідження, що узагальнюють прогрес у формальній верифікації смарт-контрактів. Зокрема, Tolmach et al. (2021) виконали великий огляд формальних моделей та методів верифікації для смарт-контрактів Ethereum [58]. Інше дослідження (2024) класифікує методи аналізу безпеки смарт-контрактів за категоріями, виокремлюючи формальну верифікацію як один із ключових напрямів поряд зі статичним аналізом, fuzz-тестуванням та іншими [59].

Слід зазначити, що ефективність формальних методів залежить від якості специфікації: інструмент виявить тільки ті вразливості, які порушують явно задані властивості. Якщо якась небезпечна поведінка не формалізована в специфікації, формальна перевірка її не помітить. Крім того, процес підготовки формальних специфікацій і моделей може бути трудомістким і потребує високої кваліфікації розробників. Це обмежує застосування формальної верифікації на практиці, особливо для складних контрактів з нетривіальною логікою.

1.5.2 Статичний аналіз смарт-контрактів

Статичний аналіз передбачає перевірку коду смарт-контракту без його виконання. Інструменти статичного аналізу аналізують вихідний код або байткод, шукаючи шаблони відомих вразливостей або підозріні конструкції. Основою є заздалегідь визначені правила та патерни для небезпечних практик програмування. Статичні аналізатори будують внутрішнє подання коду (синтаксичні дерева, графи потоку управління чи залежностей) і перевіряють виконання певних умов (правил безпеки) у цьому поданні. На відміну від динамічного аналізу, статичні методи не запускають контракт, а тому працюють швидко і можуть бути застосовані на ранніх етапах розробки (до деплою).

Серед open-source інструментів статичного аналізу найвідомішими є Slither, Securify, SmartCheck та інші. Slither – це статичний аналізатор від Trail of Bits, написаний на Python; він виконує швидке сканування Solidity-коду, щоб знайти поширені уразливості (наприклад, повторний вхід, небезпечне використання tx.origin, незахищені функції тощо) [60]. Slither застосовує набір правил і перевірок, здатен виявляти понад 70 різних проблем за лічені секунди, а також надає розробникам рекомендації щодо оптимізації коду. Securify (Tsankov et al., 2018) – інший відомий інструмент, що поєднує статичний аналіз із елементами формальної перевірки: він будує граф залежностей змінних контракту і перевіряє комплаєнтні та порушуючі шаблони (compliance/violation patterns), які гарантують виконання або порушення певної властивості [61]. Securify здатен формально доводити, що контракт дотримується властивості (або містить уразливість), якщо виконані відповідні умови у коді; цей інструмент було застосовано для аналізу тисяч реальних контрактів Ethereum. SmartCheck – ще один статичний сканер, який використовує набір регулярних виразів і шаблонів для пошуку відомих багів у Solidity-коді (наприклад, назви небезпечних функцій чи шаблони арифметичного переповнення) [62].

Статичний аналіз залишається активною сферою досліджень. Новітні роботи намагаються підвищити його ефективність, зокрема, за рахунок кращого розв’язання символічних посилань у байткодi чи поєднання з машинним навчанням. З’являються

праці, що оцінюють надійність статичних аналізаторів: так, у роботі Ferreira et al. (2020) було запропоновано ін'єкцію помилок у смарт-контракт для перевірки, які вразливості реально ловлять різні аналізатори (таким чином виявлено, що жоден інструмент не є вичерпним) [63]. В огляді 2023 р. також зазначено, що ~70% інструментів статичного аналізу, розглянутих у дослідженні, вже інтегрують методи глибокого навчання для поліпшення виявлення вразливостей – це свідчить про тенденцію до розширення класичного статичного аналізу інтелектуальними підходами [64].

Основна вада традиційного статичного аналізу – залежність від заздалегідь відомих шаблонів. Інструменти на кшталт Slither або Oyente охоплюють типовий набір вразливостей, але можуть давати помилкові спрацювання (false positives) і пропускати нетривіальні або нові баги. Через спрощення моделі виконання можливі як хибні спрацювання, так і пропуски: наприклад, аналізатори на основі правил можуть не врахувати складні взаємодії контрактів або контекст виконання і видати попередження про проблему там, де її насправді немає, або навпаки пропустити вразливість, що не відповідає жодному правилу. Крім того, статичний аналіз не дозволяє побачити реальну поведінку контракту при виконанні, через що важко оцінити експлуатованість виявлених проблем.

1.5.3 Динамічний аналіз смарт-контрактів

Динамічний аналіз полягає у дослідженні смарт-контракту шляхом його виконання (реального чи емульованого) з різними вхідними даними. Мета – простежити поведінку контракту в рантаймі та виявити небезпечні ситуації під час виконання. До методів динамічного аналізу належать символічне виконання, fuzz-тестування (генерація випадкових або спрямованих тестів), динамічне відстеження даних (taint analysis) тощо. На відміну від статичного аналізу, динамічні методи генерують конкретні послідовності транзакцій чи викликів функцій і спостерігають за наслідками, що дозволяє виявляти вразливості типу атаки повторного входу або гонки умов у реальному виконанні контракту.

Піонером динамічного підходу став інструмент Oyente (Luu et al., 2016) – перший символічний виконувач для EVM-байткоду, призначений для пошуку вразливостей у Ethereum-контрактах [65]. Oyente моделює виконання байткоду контракту із символічними (невизначеними) вхідними даними, розгалужуючи всі можливі шляхи виконання і перевіряючи ряд типових проблем (повторний вхід, виклики зовнішніх контрактів без перевірки, переповнення арифметики тощо). За допомогою Oyente автори виявили критичну вразливість повторного входу в реальному контракті TheDAO ще до атаки 2016 року. Недоліком оригінального Oyente було наближення виконання (неповна емуляція середовища Ethereum), що призводило до втрати точності. Подальший розвиток динамічного аналізу пішов шляхом fuzz-тестування. Інструмент ContractFuzzer (Jiang et al., 2018) став першим генератором випадкових тестів для смарт-контрактів Ethereum. Він автоматично створює послідовності транзакцій, намагаючись викликати типові помилки, і виявляє 7 видів вразливостей [66]. Перевага fuzzing-інструментів – здатність знайти експлойт у вигляді конкретної послідовності викликів, зменшуючи кількість пропущених вразливостей порівняно з чисто символічним аналізом. Сучасний і популярний fuzz-інструмент Echidna (Trail of Bits) застосовує property-based testing: розробник формулює інваріанти або умови безпеки як Solidity-функції, а Echidna генерує тисячі псевдовипадкових транзакцій для перевірки, чи не порушуються ці умови [67]. Це дозволяє ефективно знаходити сценарії, що призводять до зриву інваріантів (наприклад, виявлення функції, яка при певній послідовності викликів порушує баланс або права доступу). Fuzzing активно використовується для пошуку нових вразливостей, оскільки може випробувати неочікувані послідовності дій, які не розглядалися при ручному аналізі. Ще один різновид динамічного аналізу – динамічне відстеження даних (taint analysis). Інструмент EasyFlow (Gao et al., 2019) застосував аналіз «зараження даних» для відстеження потоків значень у смарт-контракті, зокрема для виявлення арифметичних переповнень: він автоматично класифікує контракти на безпечні і такі, що містять переповнення, і навіть генерує транзакції, які викликають переповнення для перевірки експлойта [68].

Динамічні методи продовжують розвиватися. Так, Huang et al. (2020) представили EOSFuzzer – перший fuzz-тестер для платформи EOSIO, орієнтований на смарт-контракти поза Ethereum [69]. Eshghie et al. (2020) запропонували Dynamit – фреймворк моніторингу виконання Ethereum-контрактів із використанням машинного навчання: транзакції проганяються у симульованому середовищі, монітор збирає дані, а модуль-детектор на базі ML класифікує поведінку як безпечну чи шкідливу [70]. Оскільки екосистема блокчейну розширюється, з’являються інструменти для інших платформ (наприклад, HFContractFuzzer для Hyperledger Fabric). Свіжа тенденція – застосування підкріплювального навчання для керування fuzz-тестуванням: Su et al. (2021) розробили RLF (Reinforcement Learning Fuzzer), який навчений спрямовувати генерування тестових транзакцій у найбільш вразливі ділянки коду [71].

Динамічний аналіз, попри високу точність у відтворенні реальних атак, не гарантує повноти. Рішення на основі fuzz-тестування і символічні виконувачі не можуть вичерпно перевірити всі можливі вхідні дані та шляхи виконання, тож деякі вразливості можуть залишитися невиявленими. Крім того, динамічні методи часто є ресурсозатратними (вимагають часу на випробування багатьох сценаріїв) і можуть потребувати налаштування під конкретний контракт (наприклад, визначення інваріантів для Echidna). Символьне виконання страждає від проблеми комбінаторного вибуху станів, що обмежує його застосування для дуже складних контрактів. Відтак на практиці динамічні інструменти іноді комбінують зі статичними аналізаторами, щоби охопити якомога ширший спектр можливих вразливостей.

1.5.4 Підходи на основі машинного навчання

Методи на основі штучного інтелекту та машинного навчання застосовують алгоритми для автоматичного навчання ознакам вразливого коду з наявних даних. Замість ручного програмування правил, ці підходи тренують моделі (нейронні мережі, дерева рішень тощо) на множині прикладів контрактів – з вразливостями і без

– щоб модель самостійно навчилася розпізнавати приховані шаблони, притаманні вразливому коду. На вході моделі подається подання коду контракту: текст, векторизоване представлення байткоду, абстрактне синтаксичне дерево (AST) або навіть граф залежностей. В сучасних працях використовуються як класичні архітектури (наприклад, рекурентні нейронні мережі LSTM для аналізу послідовностей коду), так і спеціалізовані моделі, розроблені для коду, зокрема графові нейронні мережі (Graph Neural Networks, GNN) та трансформери, які враховують структурні особливості програм [72].

Попри стрімкий поступ ML у сфері безпеки смарт-контрактів, низка ранніх оглядів або зовсім оминула машинне навчання, або висвітлила його лише частково. Зокрема, у [73] виконано систематичний аналіз 35 праць за 2015-2019 рр., де домінувала методологія формального доведення, тоді як ML-підходи не розглянуто. Огляд [74] зосередив увагу лише на двох ML-роботах (2018 і 2019 рр.). У [75] передбачено окремий підрозділ про ML-методи, проте фактично враховано лише шість підходів (п'ять за 2019 р. та один за 2020 р.). Подібно, [76] охоплює різні аспекти безпеки смарт-контрактів (зокрема детекцію вразливостей), але серед 49 проаналізованих публікацій лише 21 використовує ML.

Частина оглядів фокусується передусім на інструментах детекції, а не на алгоритмах [77, 78, 79, 80]. Переважна більшість таких інструментів ґрунтується на статичному аналізі; навіть за наявності ML усередині вони часто менш гнучкі та гірше налаштовуються, тож результати можуть поступатися «чистим» ML-конвеєрам. Новітні огляди вже звертаються до ML, однак нерідко лишаються неповними за охопленням. Наприклад, [81] окреслює кілька ML-підходів і формує підґрунтя для подальших досліджень, але не є систематичним і не забезпечує повної картини. Аналогічно, [82] розглядає застосування ML для виявлення вразливостей, класифікує типові дефекти та аналізує формальні інструменти, підсвічуючи сильні й слабкі сторони ML у безпеці смарт-контрактів, однак не претендує на вичерпність. Додатково [83] подає загальний огляд смарт-контрактів і атак на блокчейн-інфраструктури та роль AI-підходів у їхньому пом'якшенні, проте й тут бракує повної інвентаризації саме ML-праць із детекції.

Переходячи від оглядової перспективи до конкретних технічних результатів, зауважимо, що в академічних роботах активно з'являються експериментальні ML-інструменти. Наприклад, Zhang et al. (2020) застосували Bi-LSTM для векторизації байткоду Ethereum і виявлення вразливостей, досягаючи високої точності на відомих датасетах [84]. Yang et al. (2019) запропонували підхід з використанням AST: код контракту розбивається на піддерева (за функціями, модифікаторами, змінними), з яких витягаються ознаки для навчання моделі, що дозволило покращити виявлення деяких вразливостей [85]. Graph Neural Networks продемонстрували ще кращі результати – Zhuang et al. (2020) розробили GCN-модель, що будує граф виконання контракту та навчилася виявляти уразливості за допомогою аналізу структурних зв'язків у коді [86]. Удосконалення цієї ідеї – модель GVD (Graph Vulnerability Detection), де змінні Solidity-контракту та їх взаємозв'язки моделюються неевклідовим графом, і для отримання векторного представлення коду використовується алгоритм Node2Vec. Окремо слід виділити використання трансформерів та попередньо навчених мовних моделей для коду. Моделі CodeBERT та GraphCodeBERT (Microsoft Research) – це аналоги BERT, навчені на великих об'ємах програмного коду (CodeBERT – на паралельних даних код-коментарі, GraphCodeBERT – додатково враховує граф потоку даних у коді). Такі моделі вже володіють розумінням синтаксису і семантики коду, тож їх можна донавчити (fine-tune) на завдання виявлення вразливостей. Нещодавні роботи показують високу ефективність цього підходу. Зокрема, Deng et al. (2024) побудували модель Lightning Cat на основі оптимізованого CodeBERT, яка перевершила традиційні статичні аналізатори (наприклад, Slither) за точністю виявлення вразливостей [87]. В іншому дослідженні (Sapkota, 2024) fine-tune CodeBERT на вибірці з ~47 тисяч Solidity-контрактів дозволив досягти ~85% точності класифікації з $AUC \approx 0.93$ на тестових даних, що підтверджує придатність таких мовних моделей для задачі пошуку вразливостей [88].

GraphCodeBERT – більш просунута модель, яка враховує структурні зв'язки в коді – потенційно може ще підвищити результативність, і деякі роботи вже інтегрують її в конвеєри аналізу вразливостей. У роботі Wu et al. (2023)

запропоновано контрастивне навчання для покращення розпізнавання вразливого коду, що у комбінації з GraphCodeBERT дало змогу підвищити точність класифікації за рахунок кращого розрізнення уявлень вразливого/невразливого коду [89]. Інші дослідники експериментують з мультимодальними моделями, поєднуючи текстові ознаки коду з іншими даними (наприклад, описами вразливостей або історією транзакцій) для ухвалення рішення. Важливим напрямом є й розв’язання проблеми дисбалансу даних: оскільки кількість відомих вразливих контрактів менша за невразливі, моделі можуть упереджено навчатися. Для боротьби з цим застосовують методи аугментації даних та спеціальні функції втрат, як-от запропонована Li et al. (2023) функція CESCL, що ущільнює кластери embedding’ів вразливого коду, зменшуючи помилкові класифікації [90].

Попри успіхи, AI/ML-підходи мають свої виклики. По-перше, необхідність великого обсягу даних: для навчання глибокої моделі потрібна репрезентативна вибірка контрактів з точно розміченими вразливостями, що потребує зусиль зі збору й анотації датасетів. Помилки та шум у даних можуть призвести до того, що модель вчитиметься неправильним ознакам. По-друге, моделі нейронних мереж – це «чорний ящик»: важко інтерпретувати, чому саме модель позначила той чи інший фрагмент як уразливий. Це ускладнює довіру до результатів та аналіз хибних спрацювань. По-третє, модель, навчена на одних типах вразливостей, може пропустити нові або неочевидні вразливості, якщо їх патерни не були представлені в навчальних даних. Хоча на відміну від жорстко правилкових систем ML-моделі здатні адаптуватися до нових даних (їх можна перенавчити, додавши нові випадки), цей процес не миттєвий. В підсумку, ML-інструменти доцільно використовувати в комплексі з іншими методами, щоб експерти могли перевіряти результати і отримувати пояснення для виявлених проблем.

1.5.5 Гібридні методи

Жоден з розглянутих підходів не забезпечує ідеального покриття й точності, тому актуальним є поєднання методів у гібридні рішення. Гібридні методи прагнуть

об'єднати сильні сторони різних підходів – наприклад, швидкість і повноту статичного аналізу з конкретністю динамічних тестів, або ж строгість формальної верифікації з гнучкістю машинного навчання. Прикладом є комбінування статичного та динамічного аналізу в межах одного інструменту.

Li et al. (2021) запропонували систему HFCCT (Hyperledger Fabric Chaincode Test) для аналізу смарт-контрактів Hyperledger (написаних на Go), що поєднує динамічне символічне виконання з аналізом статичного синтаксичного дерева для виявлення вразливостей [91]. Ідея полягає в тому, що статичний аналіз швидко звужує коло підозрілих місць у коді, після чого динамічний модуль глибше досліджує ці місця, генеруючи конкретні виконувані сценарії. За оцінками авторів, HFCCT показав значно вищу ефективність порівняно з чисто динамічним fuzz-тестером HFContractFuzzer, виявив більше вразливостей та працював швидше. Водночас повністю усунути всі недоліки не вдалося: навіть HFCCT пропустив дві категорії вразливостей (специфічних для Hyperledger) і потребує подальшої оптимізації. Це демонструє, що інтеграція методів – нетривіальне завдання, але може дати вигоду у продуктивності. Інший напрям гібридизації – включення ML-модулів у традиційні інструменти. До прикладу, можна уявити статичний аналізатор, який використовує нейронну мережу для прогностичного ранжування підозрілих ділянок коду, на які варто звернути увагу аудитуру. Або fuzz-тестер, який підбудовує генерацію випадкових тестів на основі підказок від ML-моделі (такий підхід вже реалізовано у згаданому RLF-фреймворку з підкріплювальним навчанням). Компанія ConsenSys у своєму комплексному сервісі MythX свого часу об'єднала кілька рушіїв (статичний аналіз, символічне виконання, fuzzing), що працювали паралельно для аналізу контракту, намагаючись досягти якнайповнішого покриття [92]. Подібні багатокomпонентні рішення здатні знаходити більше вразливостей, проте вони складніші у реалізації і споживають більше ресурсів. Загалом, тренд розвитку інструментів і досліджень йде в напрямі гібридизації: поєднання формальних методів із евристичними, інтеграція правил і навчених моделей, комбінування аналізу коду з аналізом виконання. Такий підхід обіцяє підвищити і точність, і повноту виявлення вразливостей.

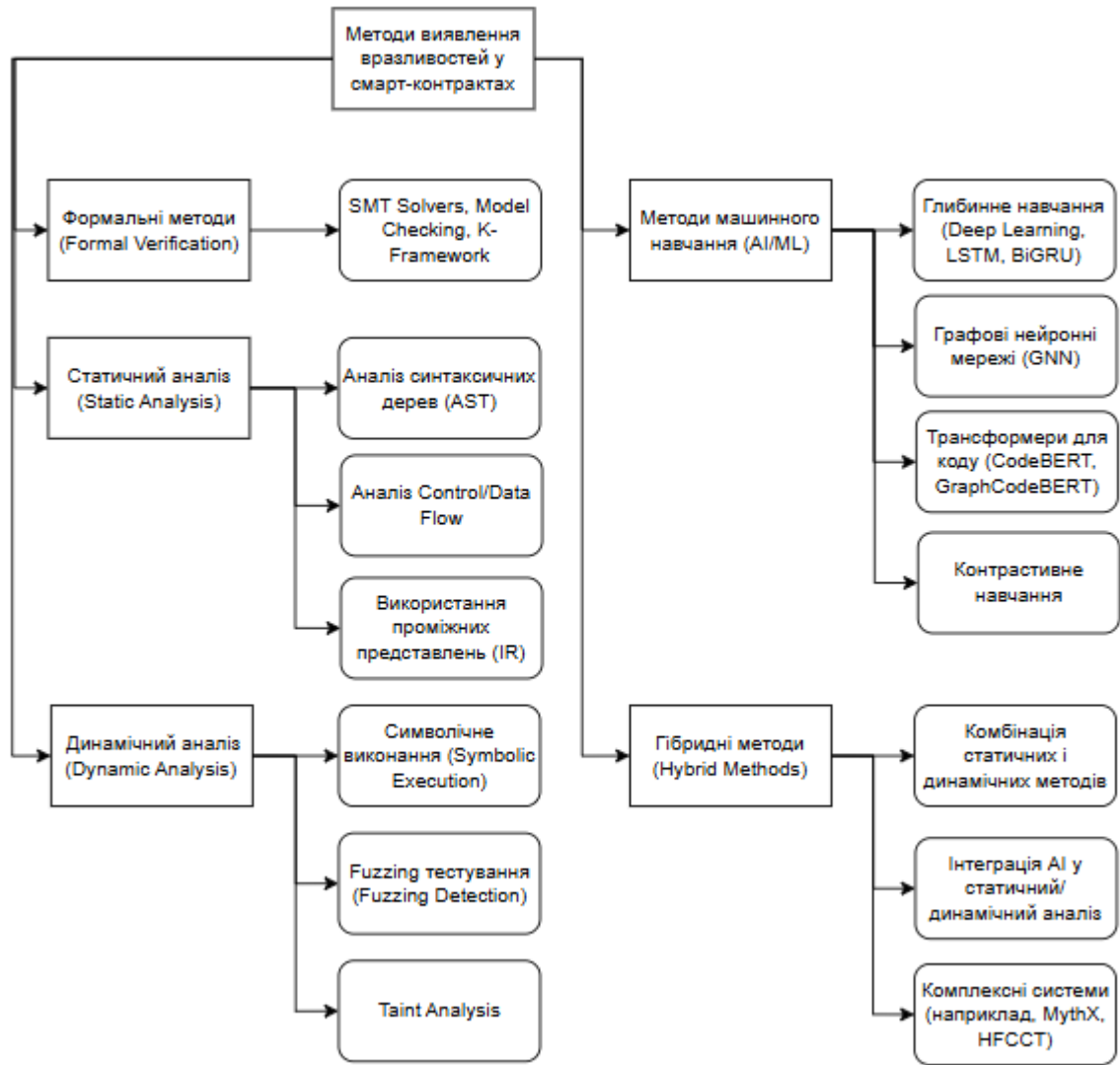


Рисунок 1.19 – Класифікація сучасних методів виявлення вразливостей у смарт-контрактах

1.6 Формалізована постановка проблеми

У результаті проведеного аналізу встановлено, що смарт-контракти активно використовуються в екосистемах блокчейн, зокрема у фінансових протоколах, де вони управляють значними грошовими потоками. При цьому кількість атак, пов'язаних з вразливостями у таких контрактах, невідомо зростає. Більшість вразливостей мають складну логічну природу, не виявляються класичними статичними чи динамічними методами аналізу і потребують глибшого семантичного розуміння коду. Практика засвідчує потребу в автоматизованих, інтерпретованих

системах аналізу безпеки смарт-контрактів, здатних адаптуватися до нових типів вразливостей.

Водночас наявні підходи мають низку істотних обмежень:

1. ручні методи аудиту забезпечують високу точність і здатність до локалізації вразливостей, але є трудомісткими, повільними та немасштабованими;
2. автоматизовані інструменти статичного/динамічного аналізу дозволяють здійснювати перевірку швидко, але мають обмежене охоплення сценаріїв і часто продукують хибні спрацювання;
3. моделі машинного навчання демонструють перспективу щодо автоматизації процесу, але здебільшого не забезпечують достатньої точності локалізації уразливих ділянок коду та потребують спеціально сформованих репрезентативних навчальних вибірок.

Таким чином, протиріччя полягає в тому, що підвищення точності виявлення вразливостей у смарт-контрактах при використанні експертного (ручного) аналізу призводить до значного зниження швидкодії та масштабованості, тоді як автоматизовані підходи забезпечують високу швидкодію, але не досягають необхідного рівня точності і не забезпечують локалізації вразливостей. У свою чергу, більшість моделей машинного навчання, незважаючи на здатність до узагальнення, не забезпечують інтерпретованості та точного вказування на вразливі фрагменти коду, що обмежує їх практичну застосовність у задачах аудиту.

Це обумовлює необхідність розробки інтелектуальної моделі, яка поєднує переваги глибокого навчання з можливістю локалізації уразливих ділянок у коді, забезпечує багатоміткову класифікацію типів вразливостей та дозволяє досягти балансу між точністю, швидкодією та інтерпретованістю результатів.

1.7 Висновки до першого розділу

1. Смарт-контракти визначено як важливий об'єкт дослідження через їх стрімке поширення у різних галузях (фінансовій, юридичній, управлінській тощо) та критичну роль у децентралізованих застосунках. Їхня незмінність після розгортання

обумовлює особливу увагу до безпеки: будь-які вразливості в коді можуть призводити до значних фінансових втрат і підриву довіри користувачів.

2. Сучасні методи виявлення вразливостей у смарт-контрактах класифіковано на кілька основних категорій: статичний аналіз (перевірка коду без виконання), динамічний аналіз (тестування під час виконання), формальна верифікація (математичне доведення коректності) та методи на основі машинного навчання. Відзначається поява гібридних підходів, які поєднують елементи різних методів задля підвищення ефективності виявлення вразливостей.

3. Проведено порівняльний аналіз зазначених підходів, результати якого зведено в узагальнювальну таблицю. Виявлено ключові переваги й недоліки кожного класу методів. Статичні та динамічні аналізатори працюють автоматично й швидко, проте можуть пропускати деякі помилки або давати хибнопозитивні спрацювання. Формальні методи забезпечують найвищу строгість перевірки смарт-контрактів, однак потребують значних обчислювальних ресурсів та експертного втручання. Методи машинного навчання здатні адаптивно виявляти нові шаблони вразливостей, але не надають формальних гарантій і характеризуються низькою прозорістю моделі (принцип “чорного ящика”).

4. Здійснено ґрунтовний аналіз літературних джерел (в тому числі сучасних оглядових досліджень) щодо безпеки смарт-контрактів, що засвідчив високу актуальність проблеми та різноманітність підходів до її розв’язання. Узагальнення результатів попередніх досліджень дозволило систематизувати відомі вразливості й методи їх виявлення, а також виявити невирішені завдання та обмеження існуючих рішень у сфері безпеки смарт-контрактів.

5. Встановлено, що жоден із наявних підходів сам по собі не забезпечує повного розв’язання проблеми вразливостей смарт-контрактів. Це зумовлює потребу в подальшому удосконаленні методів і комбінуванні різних підходів для досягнення більш повного й надійного виявлення вразливостей.

РОЗДІЛ 2. МЕТОД ФОРМУВАННЯ ЗБАЛАНСОВАНОГО ДАТАСЕТУ ДЛЯ ВІЯВЛЕННЯ ВРАЗЛИВОСТЕЙ У СМАРТ-КОНТРАКТАХ

У розділі представлено метод формування збалансованого датасету смарт-контрактів для задачі виявлення вразливостей. Проведено аналіз існуючих публічних корпусів і виявлено їхні обмеження, зокрема дисбаланс класів, неповноту охоплення та відсутність негативних прикладів. Описано процес збору даних із відкритих джерел, їхню перевірку за допомогою статичного аналізу та первинну класифікацію за типами вразливостей. Запропоновано та реалізовано два методи аугментації даних – детермінований алгоритм ін'єкції вразливостей і метод на основі великих мовних моделей, що забезпечили контрольоване та різноманітне розширення корпусу. Сформовано збалансований набір із п'яти категорій контрактів (чотири основні класи вразливостей та безпечні приклади), який пройшов процедури передобробки та нормалізації. Отриманий датасет забезпечує необхідні умови для подальшого навчання та тестування моделей глибинного аналізу коду смарт-контрактів.

2.1 Аналіз існуючих датасетів смарт-контрактів

У процесі дослідження та побудови моделей машинного навчання для виявлення вразливостей у смарт-контрактах критично важливою є наявність якісного та репрезентативного навчального датасету. Враховуючи складність проблематики та обмежену кількість публічно доступних даних, було проведено ретельний огляд існуючих датасетів, що використовуються в наукових роботах і практичних реалізаціях систем аналізу безпеки смарт-контрактів.

У таблиці 2.1 та 2.2 представлено порівняльний аналіз найбільш відомих публічних датасетів, які містять Solidity-контракти з розміткою вразливостей.

Таблиця 2.1 – Порівняльний аналіз розмірів найбільш відомих публічних датасетів

Датасет	Кількість контрактів	Кількість вразливих прикладів
SmartBugs Curated	143 контрактів (Solidity файлів)	208 вразливостей (9 категорій DASP; напр., повторний вхід – 31 контракт, інші ~10–30 в кожній)
SolidiFI Benchmark	350 контрактів	7 типів: повторний вхід, таймінг, невід.винятки, unchecked send, TOD, overflow, tx.origin
DAppSCAN	682 DApps (21457 файлів)	1646 знайдених вразливостей (~25 типів SWC; найбільше: NoEffects – 291, Overflow – 204, Reentrancy – 138, UncheckedCall – 116, тощо)
ScrawlD	6780 контрактів (Ethereum mainnet)	~ Ордени тисяч випадків (8 типів; напр., ~1000 контрактів з Arithmetic, ~600 Reentrancy, ~600 UncheckedEx, ~500 TimeManipulation, тощо)
VeriSmart Benchmarks	~616 контрактів (487 CVE, 25 Zeus, 104 Leak/Suic.)	616 відомих вразливих прикладів (типи відомі з експлойтів: Reentrancy, Overflow, Delegatecall, Access/Auth, Selfdestruct, LockedEther,...)

Таблиця 2.2 – Порівняльний аналіз параметрів найбільш відомих публічних датасетів

Датасет	Типова розмітка	Типи вразливостей	Баланс (позитивн./негативн.)
SmartBugs Curated	Вручну ановано в коді коментарями	9 категорій (Reentrancy, Access Control, Arithmetic, Unchecked Calls, DoS, Bad Randomness, Front Running, Time, Short Addr.)	Тільки позитивні (всі приклади вразливі)
SolidiFI Benchmark	Автоматична ін'єкція; логи з типом багу і місцем	7 типів (Reentrancy, Timestamp dep., Unhandled Exception, Unchecked Send, Tx Order, Integer O/F, Use of tx.origin)	Переважно позитивні (всі ін'єктовані – вразливі); оригінальні “безпечні” версії окремо
DAppSCAN	Вручну (з аудиторських звітів) + SWC мітки	~25+ типів (повний спектр SWC: переповнення, реентрансі, рандом, доступ, DoS, та ін.)	Переважно позитивні (у кожному DApp шукано баги; 1646 випадків у 608 звітах); ~274 DApp без багів (негативні)

Продовження таблиці 2.2

ScrawlID	Авто-аналіз 5 інструментами; мітка при збігу більшості	8 типів (ARTHM, RENT, UE, DOS, LE, TimeM, TimeO, Tx-Origin) – див. опис (SWC-101,107,104,128/113,—,116,114,115)	~38% позитивних (мають хоч одну вразливість), ~62% негативних (жодна не підтверджена)
VeriSmart Benchmarks	Вручну (на основі CVE та аналізу авторів)	~10 типів (конкретні випадки: TheDAO реентрансі, Parity баг (auth), BatchOverflow, suicidal, leaking, та ін.)	Тільки позитивні (усі містять відому вразливість); безпечні контракти не включені

Першим набором даних, який було проаналізовано, є SmartBugs Curated (SB Curated) [93]. Це відібраний корпус смарт-контрактів, що містить 143 Solidity-файли з вручну анотованими вразливостями. Усього в цих контрактах позначено 208 випадків різних вразливостей, які віднесено до категорій відповідно до таксономії DASP. Особливість цього набору полягає в тому, що всі приклади є позитивними, тобто кожен контракт містить принаймні одну вразливість.

Набір охоплює дев'ять основних типів вразливостей, серед яких вразливість повторного входу, помилки контролю доступу, арифметичні переповнення та заниження, неконтрольовані низькорівневі виклики, відмова в обслуговуванні, небезпечна випадковість, атаки випередження транзакцій, залежність від часових міток та вразливість коротких адрес. Кількісно категорії представлені нерівномірно: наприклад, 31 контракт містить вразливість повторного входу, тоді як інші види зустрічаються рідше. Баланс негативних і позитивних прикладів відсутній, оскільки набір створювався саме для демонстрації вразливих випадків.

Важливою перевагою SB Curated є висока якість анотацій: дослідники вручну додавали коментарі безпосередньо в код, а також створили JSON-опис усіх вразливостей. Це робить набір придатним для тестування статичних аналізаторів і систем виявлення помилок, адже він містить реальні приклади з відкритих джерел, таких як Etherscan та GitHub. Разом із тим обмеженим фактором цього корпусу є його невеликий обсяг і відсутність безпечних контрактів, що знижує його корисність для задач навчання моделей машинного навчання, але робить цінним для оцінки точності й повноти інструментів аналізу.

Наступним розглянутим набором даних є SolidiFI Benchmark, створений для оцінки ефективності інструментів статичного аналізу смарт-контрактів [94]. Він включає 350 унікальних контрактів, у які цілеспрямовано було ін'єктовано вразливості. Загалом у набір додано 9369 прикладів помилок семи різних типів, серед яких повторний вхід, залежність від часових міток, неперехоплені винятки, використання `send` чи `call` без перевірки, залежність від порядку транзакцій, арифметичні переповнення та заниження, а також помилки автентифікації, пов'язані з використанням `tx.origin`.

Усі контракти у наборі є вразливими: кожен із них містить кілька ін'єктованих багів, які було детально задокументовано в журналах ін'єкцій із вказанням типу та місця вставки. Хоча оригінальні «безпечні» версії контрактів існують, основна версія корпусу поширює саме модифіковані файли, що містять вразливості. Такий підхід дозволяє створити рівні умови для порівняння різних інструментів, оскільки всі вони тестуються на заздалегідь відомих і чітко позначених помилках.

Особливістю SolidiFI є його орієнтація на штучно згенеровані вразливості, що гарантує контроль над процесом формування датасету та дає можливість забезпечити паритет між різними категоріями помилок. Це робить набір особливо корисним для експериментів, спрямованих на вимірювання точності та повноти інструментів статичного аналізу. Таким чином, SolidiFI Benchmark посідає важливе місце серед бенчмарків для дослідження вразливостей, хоча його головним обмеженням є відсутність реальних негативних прикладів і надмірна штучність ін'єктованих помилок.

Ще одним важливим набором є DAppSCAN, який орієнтований на вразливості, виявлені під час професійних аудитів децентралізованих застосунків [95]. Він охоплює вихідний код 682 проєктів, що у сукупності містять понад 21 тисячу файлів на Solidity. Ці дані зібрано з 608 аудиторських звітів різних компаній, що надає корпусу унікальної практичної цінності, адже він відображає результати реальних процесів перевірки безпеки.

Усі виявлені проблеми класифіковано відповідно до стандарту SWC-ID (Smart Contract Weakness Classification) [96]. Загалом датасет охоплює понад 25 типів вразливостей, серед яких найбільш поширеними є помилки бездіяльного коду (SWC-135), арифметичні переповнення та заниження (SWC-101), вразливість повторного входу (SWC-107), неперевірені результати зовнішніх викликів (SWC-104), використання застарілих версій компілятора (SWC-102), «плаваючі» директиви pragma (SWC-103), відмова в обслуговуванні через газові обмеження (SWC-128) та залежність від порядку транзакцій (SWC-114). Така широта охоплення дозволяє аналізувати як поширені, так і рідкісні класи вразливостей.

Загалом у проєктах зафіксовано 1646 випадків вразливостей. Більшість застосунків містили хоча б одну помилку, хоча близько 274 проєктів можна було визнати безпечними. Таким чином, набір містить як позитивні, так і частково негативні приклади, що створює більш реалістичне уявлення про розподіл вразливостей у практичних умовах.

Особливістю DAppSCAN є те, що він поєднує у собі не лише вихідний код, а й супровідну документацію: у складі набору містяться оригінальні аудиторські звіти у форматі PDF, зкомпільований байткод, а також допоміжні таблиці з детальною статистикою. Така багаторівнева структура робить датасет цінним джерелом як для дослідження поширеності різних класів помилок, так і для відтворення реальних сценаріїв аудиту безпеки. У результаті DAppSCAN вважається одним із наймасштабніших і найрепрезентативніших наборів для аналізу вразливостей у смарт-контрактах.

Серед сучасних наборів особливе місце посідає ScrawlID, представлений у 2022 році як один із найбільших корпусів смарт-контрактів із реального блокчейну

Ethereum [97]. Він містить 6780 контрактів, отриманих безпосередньо з Etherscan API, тобто відображає «польові» дані без попередньої підготовки чи відбору за конкретними вразливостями. Такий підхід надає корпусу високої репрезентативності, оскільки він охоплює широкий спектр практичних прикладів коду.

Розмітка вразливостей у ScrawlID здійснювалася не вручну, а за допомогою автоматизованого аналізу. Для цього було використано п'ять популярних інструментів: Slither, Mythril, SmartCheck, Oyente та Osiris. Виявлені результати охоплюють вісім основних категорій вразливостей, серед яких арифметичні переповнення, вразливість повторного входу, неперехоплені винятки, відмова в обслуговуванні, блокування ефіру, маніпуляція часом, залежність від порядку транзакцій та використання tx.origin для авторизації. Таким чином, датасет фокусується на ключових проблемах безпеки, які найчастіше перевіряються інструментами статичного аналізу.

За результатами обробки близько 38% контрактів отримали мітки хоча б однієї з вразливостей. Наприклад, понад тисяча контрактів була віднесена до категорії арифметичних помилок, близько шістсот – до категорії вразливості повторного входу, ще кілька сотень – до інших типів. Водночас у корпусі залишилася значна кількість контрактів, де жоден інструмент не виявив критичних проблем. Це робить ScrawlID особливо корисним, оскільки він містить як позитивні, так і негативні приклади, що забезпечує збалансованість набору для навчання моделей машинного навчання.

Серед важливих переваг ScrawlID варто відзначити його масштабність і близькість до реального розподілу коду в блокчейні. Проте автоматична розмітка має і свої недоліки: можливі хибні спрацьовування чи пропуск реальних вразливостей, що знижує достовірність міток. Незважаючи на це, набір широко використовується для оцінки інструментів та алгоритмів, оскільки дозволяє тестувати їх у більш «польових» умовах, наближених до реальних сценаріїв використання смарт-контрактів.

Окрему групу становить набір VeriSmart Benchmarks, створений у рамках розробки аналізатора VeriSmart і використаний у дослідженнях, опублікованих на конференціях S&P 2020 та USENIX Security 2021 [98]. Він поєднує кілька піднаборів

смарт-контрактів, що дозволяє оцінювати інструменти на реальних інцидентах і контрольованих експериментах. Основу корпусу становлять 487 контрактів із реєстру CVE, серед яких зафіксовані найбільш відомі вразливості, такі як вразливість повторного входу (TheDAO), переповнення (BatchOverflow), помилки контролю доступу (Parity wallet) та делеговані виклики. Кожному контракту в цій групі присвоєно відповідний CVE-ідентифікатор, що забезпечує прозорість походження прикладу.

До набору також включено 25 контрактів із датасету Zeus [99], який став основою низки попередніх досліджень, а також 104 контракти з категоріями «Ether Leaking» та «Suicidal». Частина з них є реальними експлойтами, зокрема взятими зі SmartBugs, а частина була сконструйована авторами спеціально для експериментів. Це дозволило створити збалансовану колекцію прикладів, які охоплюють як перевірені вразливості з історичних інцидентів, так і синтетично згенеровані сценарії.

Загалом набір налічує понад 600 контрактів, кожен із яких містить принаймні одну підтверджену вразливість. Негативних прикладів тут немає, адже основна мета полягає у відтворенні відомих помилок та перевірці здатності інструментів їх виявляти. У деяких контрактах зустрічається кілька різних вразливостей одночасно, що відображає реалістичність практичних сценаріїв.

Важливою особливістю VeriSmart Benchmarks є його чітка структурованість: для кожного піднабору наведені метадані, мітки та допоміжні файли, що дозволяють ідентифікувати типи вразливостей та їх відповідність конкретним CVE (Common Vulnerabilities and Exposures). Завдяки цьому набір придатний не лише для тестування точності аналізаторів, але й для детальнішого порівняння їхньої ефективності на різних категоріях багів. Таким чином, VeriSmart Benchmarks виступає надійним еталоном для перевірки відтворюваності й повноти результатів інструментів, які працюють із критичними вразливостями смарт-контрактів.

Основні висновки аналізу:

- відсутність універсального датасету. Жоден з існуючих наборів не забезпечує повноцінного покриття усіх основних критеріїв якості: точність, масштабованість, збалансованість, локалізація вразливостей;

- обмеженість вручну розмічених даних. Такі набори, як SmartBugs Curated та VeriSmart, мають високу якість розмітки, але надто малу кількість прикладів, що не дозволяє навчати сучасні моделі ШІ;
- автоматично розмічені набори (як ScrawlID) забезпечують масштабованість, але мають нижчу точність через хибні спрацьовування інструментів;
- ін'єктовані набори (як SolidiFI) придатні для бенчмаркінгу, однак не відображають реальних сценаріїв використання коду;
- брак негативних прикладів у деяких наборах призводить до зміщення моделей у бік виявлення лише вразливих контрактів.

Враховуючи зазначені недоліки, було прийнято рішення про створення власного навчального датасету, який:

- містить збалансовану кількість контрактів з кожним типом цільової вразливості;
- включає як вручну зібрані приклади з реальних джерел, так і згенеровані за допомогою методів на основі машинного навчання;
- проходить верифікацію через статичний аналіз (Slither/Oyente) для підтвердження наявності вразливостей;
- забезпечує наявність безпечних (негативних) прикладів;
- включає мітки з можливістю використання в мультиміткової класифікації.

2.2 Метод формування датасету

Запропонований у роботі метод формування датасету смарт-контрактів орієнтований на побудову репрезентативного, збалансованого та відтворюваного набору даних, придатного для навчання моделей багатоміткової класифікації та локалізації вразливостей. Метод базується на поетапній обробці вихідних даних і поєднує автоматизований аналіз коду, алгоритмічні процедури фільтрації та контроль якості отриманих зразків [100].

На вхід методу подаються сирі дані смарт-контрактів, отримані з відкритих джерел у вигляді верифікованого вихідного коду та супровідних метаданих. Подальша обробка здійснюється у вигляді послідовності взаємопов'язаних етапів, кожен з яких виконує чітко визначену функцію та формує проміжний результат, що використовується на наступному кроці. Така організація дозволяє контролювати якість даних на всіх рівнях і мінімізувати накопичення помилок.

Загальна схема методу включає етапи збору та агрегування даних, фільтрації та нормалізації коду, автоматизованої розмітки вразливостей, формування негативної вибірки, аугментації та балансування класів, а також підготовки фінального датасету до використання в процесі навчання моделей машинного навчання. З метою балансування класів і збільшення кількості навчальних прикладів було застосовано два підходи до генерації даних, а саме детермінований метод ін'єкції вразливостей у смарт-контракти та метод ін'єкції вразливостей у смарт-контракти на основі великих мовних моделей.

На виході методу формується структурований датасет, що складається з контрактів, розмічених за кількома типами вразливостей, із контрольованим розподілом класів і уніфікованим форматом представлення. Завдяки фіксації параметрів усіх етапів обробки та використаних інструментів забезпечується відтворюваність результатів і можливість подальшого розширення або оновлення датасету без порушення його цілісності. Етапи формування датасету наведені на рисунку 2.1.

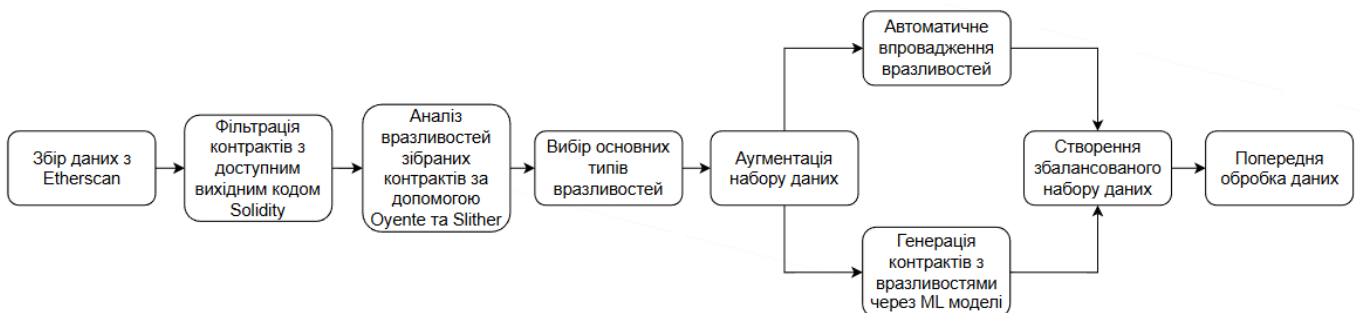


Рисунок 2.1 – Етапи формування датасету

2.2.1 Збір даних

Ефективне навчання глибинних моделей у сфері безпеки смарт-контрактів вимагає великого, збалансованого та точно розміченого корпусу коду, однак наявні публічні набори мають недоліки, які були перелічені в підрозділі 2.1. Тому було розроблено відтворюваний конвеєр формування власного датасету, який починається зі збору вихідних текстів Solidity із Etherscan виключно для контрактів із доступним та верифікованим кодом. На етапі первинної підготовки здійснюється технічна фільтрація: перевіряються валідність зазначеної версії компілятора, адекватність розміру файлів і відсутність тривіальних або порожніх фрагментів. Для уникнення дублювань застосовується нормалізація коду (усунення коментарів і зайвих пробілів) із подальшим хешуванням нормалізованого представлення; за можливості також звіряється байткод для ідентичних версій контрактів. Паралельно фіксуються метадані (адреса, версія Solidity, дата верифікації), що надалі дозволяє контролювати часові зрізи й відтворюваність експериментів.

Анотація вразливостей виконується за допомогою двох комплементарних інструментів статичного аналізу – Oyente та Slither. Перший орієнтований на проблеми повторного входу, залежності від зовнішнього стану та арифметичні помилки, другий забезпечує ширше покриття детекторів і повертає точні координати підозрілих конструкцій у коді. Результати обох аналізаторів агрегуються в уніфікованому форматі з урахуванням типу, важливості та локалізації; далі застосовується узгоджувальна процедура, яка відокремлює підтверджені спрацьовування від імовірних і відбраковує типові хибні позитиви. Отримані ярлики відразу формуються у багатомітковому представленні, оскільки один контракт часто поєднує кілька типів вразливостей; локалізація фіксується у вигляді діапазонів рядків, що є критично важливим для подальшого навчання компонентів точкової індикації.

Особлива увага приділяється наявності «негативних» прикладів. До безпечної підмножини включаються контракти, для яких аналізатори не генерують попереджень, після чого вибірково проводиться ручна ревізія, аби мінімізувати приховані хибні негативи. Розбиття корпусу на train/validation/test здійснюється з

урахуванням часової структури даних та зі збереженням розподілу класів: контракти, верифіковані на пізніших етапах, цілеспрямовано відносяться до валідаційної та тестової вибірок, що зменшує ризик інформаційного витоку між вибірками та краще відображає реальні умови розгортання моделей.

Дані зберігаються у структурі, придатній до відтворення експериментів: вихідні файли Solidity супроводжуються метаданими, уніфікованими звітами інструментів і файлами з мультимітковою анотацією та локалізацією; версії аналізаторів і середовища фіксуються у маніфестах, що дозволяє точно повторити всі стадії конвеєра. Попри вжиті заходи, ймовірність поодиноких FP/FN для статичного аналізу залишається невід’ємною властивістю методу, тому процес анотації передбачає циклічне оновлення списків типових хибних спрацьовувань і періодичну людську валідацію підозрілих випадків. Підсумковий розподіл типів вразливостей у сформованому корпусі наведено в Таблиці 2.3, де для кожного класу подано частоту появ, частку підтверджених спрацьовувань та питому вагу контрактів із множинними ураженнями разом із долею негативних прикладів, що відображає досягнутий баланс вибірки.

Таблиця 2.3 – Розподіл контрактів за категоріями вразливостей

Категорія	Кількість контрактів
Integer Underflow	23 702
Integer Overflow	31 061
Timestamp Dependency	1 460
Re-entrancy Vulnerability	310
Без вразливостей	12 829

З безлічі вразливостей, які можуть бути присутніми в смарт-контрактах, було виділено чотири найпоширеніші:

- Integer Underflow – ситуація, коли значення змінної типу uint зменшується нижче за нуль, що призводить до несподіваної поведінки;
- Integer Overflow – ситуація, коли змінна типу uint перевищує максимальне допустиме значення та "перестрибує" через 0;

- Timestamp Dependency – використання `block.timestamp` у логіці прийняття рішень, що робить контракт уразливим до маніпуляцій з боку майнерів;
- Re-Entrancy Vulnerability – вразливість повторного входу, коли зовнішній контракт може викликати повторні транзакції до завершення поточної операції.

2.2.2 Детермінований метод ін'єкції вразливостей у смарт-контракти

Запропонований підхід реалізує детерміновану автоматизовану ін'єкцію вразливостей у вихідний код смарт-контрактів без залучення методів машинного навчання. Алгоритм працює на основі статичного аналізу та пошуку шаблонів (pattern matching), що дає змогу точно визначати місця, придатні для модифікації, і застосовувати стандартизовані трансформації. Завдяки повній відсутності випадковості процес є передбачуваним і відтворюваним: за однакових вхідних даних результати ін'єкції завжди тотожні. Контракти не виконуються – аналізується лише їхня статична структура, а внесені зміни контролюються на предмет синтаксичної коректності.

Метод спирається на три взаємопов'язані засади:

- детермінованість: алгоритм забезпечує детермінований характер обчислень, завдяки чому для однакових вхідних даних завжди формується ідентичний результат. Поведінка алгоритму є повністю передбачуваною, оскільки в процесі модифікації не використовуються випадкові чинники;
- статичний аналіз: ідентифікація ділянок, придатних для ін'єкції, здійснюється шляхом аналізу структурних властивостей програмного коду без його виконання. Алгоритм виконує парсинг вихідного коду Solidity з метою виявлення характерних конструкцій і патернів, а також контролює синтаксичну коректність внесених модифікацій;
- шаблонно-орієнтований підхід: для кожного класу вразливостей попередньо визначаються типові шаблони програмного коду, відповідно до яких алгоритм систематично здійснює пошук релевантних конструкцій у смарт-контракті

та вносить стандартизовані зміни, що забезпечує уніфікованість ін'єкцій і спрощує подальшу валідацію отриманих результатів.

Практично це реалізовано так: контракт парситься, після чого відбираються функції-кандидати, у яких одночасно присутні релевантні структури стану (наприклад, балансний mapping), операції з коштами (функції на кшталт withdraw/transfer або зовнішні виклики переказу Ether), перевірки перед дією (require/assert) та ефекти оновлення стану (зокрема зменшення балансу). Лише ті функції, що відповідають усім критеріям цільового класу вразливості, передаються на етап модифікації.

На рисунку 2.2 показано алгоритмічний конвеєр детермінованої ін'єкції вразливостей. Процес базується на статичному аналізі та пошуку шаблонів, після чого виконується або цілеспрямована модифікація існуючої функції, або додавання мінімального вразливого шаблону з подальшою компіляційною та інструментальною валідацією.



Рисунок 2.2 – Детермінований метод ін'єкції вразливостей

2.2.2.1 Стратегії модифікації наявної функції на прикладі вразливості повторного входу

У випадках, коли у функції вже присутні перевірка на достатність балансу, операція переказу коштів і подальше оновлення стану, ін'єкція виконується у формі цілеспрямованої перебудови послідовності дій та заміни механізму відправлення. Замість безпечних високорівневих примітивів на кшталт `transfer` чи `send`

підставляється низькорівневий виклик вигляду `call{value: ...}("")` з обов'язковою перевіркою значення `success`. Паралельно свідомо порушується принцип Checks-Effects-Interactions: зовнішня взаємодія з одержувачем відбувається до того, як контракт оновлює власний внутрішній стан (наприклад, списує баланс чи фіксує факт виплати). Така перестановка створює часовий проміжок, у межах якого можливий повторний вхід і повторне виконання критичного фрагмента до синхронізації інваріантів стану.

Зміни впроваджуються мінімально інвазивно: зберігається вихідна логіка перевірок, сигнатури функцій і контракти сумісності з компілятором, а трансформації стосуються лише порядку викликів і типу операції переказу. Завдяки цьому модифікований код лишається синтаксично коректним, але набуває семантичної вразливості, необхідної для навчальних цілей. Ілюстративний приклад такого локальної зміни наведено на рисунку 2.3, де видно, як перенесення зовнішнього виклику перед оновленням стану у поєднанні з `call{value: ...}` відкриває можливість для повторного входу.

```

1 // Було (безпечніше)
2 require(balances[msg.sender] >= amount);
3 payable(msg.sender).transfer(amount);
4 balances[msg.sender] -= amount;
5
6 // Стало (вразливо)
7 require(balances[msg.sender] >= amount);
8 (bool success, ) = msg.sender.call{value: amount}("");
9 require(success, "Transfer failed");
10 balances[msg.sender] -= amount;
```

Рисунок 2.3 – Мінімальний фрагмент зміни (ілюстративно) в коді

2.2.2.2 Стратегії додавання шаблонної функції на прикладі вразливості повторного входу

Орієнтуючись на приклад повторного входу, передбачено стратегію додавання мінімальної «шаблонної» функції тоді, коли у вихідному контракті немає потрібної бізнес-логіки (скажімо, відсутній механізм виведення коштів). У цьому разі до коду вводиться компактна вразлива процедура з коректною сигнатурою, узгодженою з

існуючим інтерфейсом, а також допоміжний фрагмент для поповнення балансу, що відтворює реалістичну динаміку стану. Функція проектується так, щоб мінімально торкатися структури контракту: не змінюється розкладка сховища, не порушуються наявні інтерфейси та модифікатори доступу, зберігається сумісність із цільовою версією компілятора та вже наявними бібліотеками/імпортами. Ілюстративний приклад такого додавання наведено на рисунку 2.4.

```

1 function withdraw(uint256 amount) public {
2     require(balances[msg.sender] >= amount, "Insufficient balance");
3     (bool success, ) = msg.sender.call{value: amount}(""); // зовнішній виклик до
    оновлення стану
4     require(success, "Transfer failed");
5     balances[msg.sender] -= amount; // оновлення після взаємодії
6 }
7
8 function deposit() public payable {
9     balances[msg.sender] += msg.value;
10 }

```

Рисунок 2.4 – Приклад додавання шаблонної функції в код

Щоб не впливати на чинну функціональність, нові ідентифікатори підбираються з урахуванням унікальності в межах контракта, сигнатури узгоджуються з існуючими типами даних, а виклики організовуються так, щоб не перехоплювати контрольні гілки початкової логіки. Втілення шаблону дотримується семантики, яка робить вразливість відтворюваною: зовнішня взаємодія (через *call{value: ...}*) свідомо виконується до оновлення внутрішнього стану (наприклад, до списання балансу), що створює часовий проміжок для повторного входу. Допоміжна логіка поповнення забезпечує правдоподібні умови для запуску вразливої гілки, водночас не нав'язуючи залежностей від решти компонентів контракту.

Після внесення змін виконується повний мінімальний цикл перевірок. Спершу здійснюється синтаксична валідація: контракт компілюється *solc* у цільовій версії, що підтверджує коректність вставлених конструкцій і відсутність конфліктів у визначеннях типів чи модифікаторів. Далі проводиться підтвердження властивості вразливості інструментами статичного аналізу: перевіряється наявність зовнішнього виклику до ефектів, відповідність відомим патернам атаки повторного входу та фіксуються координати підозрілих ділянок коду. Насамкінець оцінюється сумісність:

зберігається початкова логіка контракту, ABI (Application Binary Interface) не зазнає небажаних змін, події та модифікатори лишаються валідними, а нові фрагменти не впливають на існуючі контрольні шляхи. Такий підхід дозволяє отримати синтаксично коректний і водночас семантично вразливий приклад, придатний для навчання та бенчмаркінгу без спотворення базової функціональності.

2.2.2.3 Математична модель детермінованого підходу

Нехай C – вихідний смарт-контракт, V – тип вразливості (наприклад, атака повторного входу), $P_V = \{p_1, \dots, p_n\}$ – набір шаблонів для виявлення коду, придатного до інжектування вразливості.

Множина функцій-кандидатів визначається як

$$F_V(C) = \{f \in C \mid \forall p \in P_V: match(p, f) = 1\}, \quad (2.1)$$

Тобто до $F_V(C)$ входять лише ті f , що одночасно задовольняють усі патерни з P_V .

З множини кандидатів детермінованим правилом відбору обирається єдина функція f^* . До неї застосовується оператор модифікації, параметризований типом уразливості V :

$$M(f^*, V) \rightarrow f'_v, \quad (2.2)$$

де f'_v – версія з уразливістю.

Остаточне перетворення контракту задається оператором I :

$$I(C, V) = \begin{cases} C' = (C \setminus f^*) \cup f'_v, & \text{якщо } F_V(C) \neq \emptyset, \\ C' = C \cup F_{new}(V), & \text{інакше.} \end{cases}, \quad (2.3)$$

Таким чином, якщо в контракті наявна придатна функція, виконується її цілеспрямована модифікація $f^* \rightarrow f'_v$; інакше до C детерміновано додається мінімальний уразливий шаблон $F_{new}(V)$. Детермінованість процедури гарантовано фіксованими правилами відбору f^* і незмінними параметрами M та I , тож для однакових вхідних даних завжди отримуємо однаковий результат.

2.2.3 Метод ін'єкції вразливостей у смарт-контракти на основі великих мовних моделей

Запропонований метод реалізує автоматизоване впровадження вразливостей у вихідний код смарт-контрактів із залученням великих мовних моделей (LLM, Large Language Model). На відміну від детермінованих, патерн-орієнтованих методів, він виконує контекстно залежні модифікації, що зберігають первинну функціональність контракту й виглядають природно інтегрованими у вихідну архітектуру коду. Мета полягає у формуванні різноманітних і реалістичних навчальних корпусів для задач виявлення вразливостей у смарт-контрактах.

Процес починається з попередньої обробки коду: контракт нормалізується до цільової версії компілятора, усуваються надлишкові конструкції, уніфікуються імпорти й коментарі, а також виконується фрагментація великих файлів на керовані контекстом блоки – усе це забезпечує відповідність обмеженням обраної моделі та підвищує стабільність подальших перетворень. Далі задіюється інжектор, який керується LLM: моделі подається структурована підказка з описом цільового типу вразливості, обмежень на синтаксис і семантику, вимог щодо збереження існуючих сигнатур та інваріантів контракту. На цьому етапі здійснюються локальні, контекстно узгоджені зміни, що індукують потрібну слабкість, але не руйнують логіку взаємодії модулів.

Отриманий результат проходить парсинг і сувору валідацію: перевіряється синтаксична коректність і сумісність зі специфікованою версією solc, контролюється відсутність небажаних побічних змін інтерфейсу (ABI, Application Binary Interface), підтверджується наявність ознак цільової уразливості за допомогою інструментів статичного аналізу та узгодженість модифікацій із вихідною структурою даних. Лише після успішного проходження цих перевірок модифікований фрагмент вноситься до корпусу.

Завершальний етап – конвеєр генерації датасету: модифіковані контракти зберігаються у стандартизованому форматі разом із метаданими про тип вразливості,

локалізацію змін і результати перевірок; фіксуються параметри виклику моделі та версії інструментів, що забезпечує відтворюваність експериментів і уніфікований обмін даними між етапами дослідження. Така організація дозволяє поєднати гнучкість LLM-підходу з необхідними гарантіями якості, отримуючи різноманітні, але контрольовані приклади для навчання та оцінювання моделей.

З метою дотримання обмежень великих мовних моделей (максимальна довжина запиту та кількість токенів) програмний код попередньо оптимізується шляхом видалення коментарів і порожніх рядків, а також усікання надмірно великих контрактів до допустимого розміру з коректним завершенням структурних конструкцій. Застосована функція оптимізації гарантує синтаксичну цілісність коду навіть після його скорочення.

Система формує спеціалізований запит до великої мовної моделі, який включає:

- рольову інструкцію, відповідно до якої модель інтерпретується як «фахівець з безпеки»;
- чітко визначені обмеження, що передбачають заборону створення нових контрактів, вимогу внесення змін безпосередньо до наявного коду та збереження його логіки;
- вимоги до формату вихідних даних, зокрема подання повністю модифікованого контракту з окремим виділенням змінених фрагментів;
- опис цільового типу вразливості.

Для генерації модифікованого контракту застосовується модель GPT-4.1-mini з такими параметрами:

- `temperature = 0.8` (баланс варіативності та стабільності);
- `max_tokens = 4000`;
- оптимізовані витрати та продуктивність.

Модель інтегрує вразливість згідно з вказівками, намагаючись мінімально змінювати код і зберігати його функціональність.

Після отримання відповіді здійснюється подальша обробка, яка включає:

- виокремлення блоків коду Solidity із застосуванням регулярних виразів;

- нормалізацію програмного коду з метою усунення відмінностей форматування;
- перевірку наявності вразливого фрагмента у повному тексті контракту;
- відхилення результатів, що не відповідають встановленим вимогам.

Основний цикл роботи системи включає:

1. завантаження та перевірку контрактів;
2. пропуск файлів > 10 КБ;
3. виключення вже оброблених контрактів (SHA1-хешування);
4. генерацію вразливої версії через LLM;
5. парсинг і валідацію;
6. Збереження у форматі JSON з полями: `contract` – повний код, `vulnerability` – тип вразливості, `vulnerable_snippet` – змінений фрагмент.

Рисунок 2.5 ілюструє конвеєр LLM-орієнтованої ін'єкції вразливостей. На відміну від детермінованого підходу, модифікації виконуються контекстно залежно від архітектури контракту, однак кожен згенерований приклад проходить сувору багатоступеневу валідацію, що запобігає накопиченню синтаксичних і семантичних помилок.

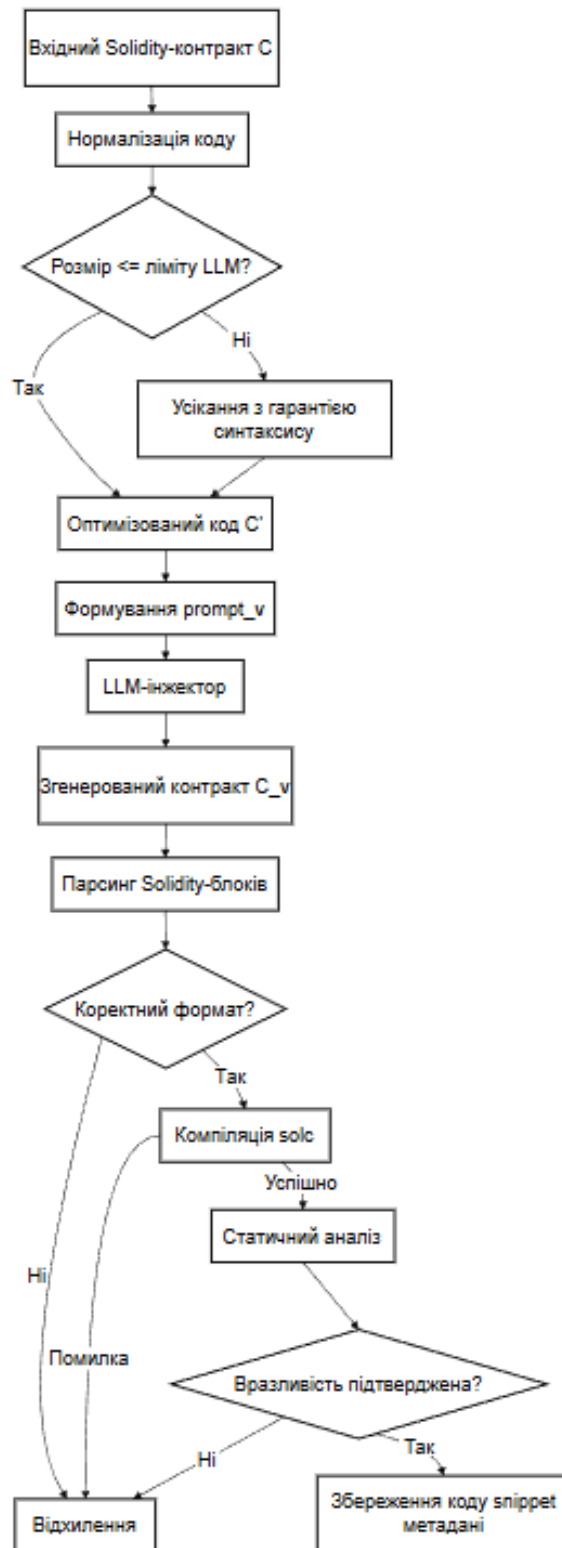


Рисунок 2.5 – LLM-орієнтований метод ін'єкції вразливостей

2.2.3.1 Математична модель

Нехай C позначає вихідний смарт-контракт, V – цільовий тип вразливості, а AI – мовну модель. На першому кроці застосовується функція попередньої оптимізації коду $O(\cdot)$, яка забезпечує відповідність вхідних даних обмеженням моделі (зокрема, за розміром контексту). Формально:

$$O(C) = \begin{cases} C', \text{ якщо } |C| \leq MAX_SIZE \\ truncate(C, MAX_SIZE), \text{ інакше.} \end{cases} \quad (2.4)$$

де C' – версія коду після видалення коментарів/порожніх рядків та інших нормалізацій; у разі перевищення ліміту виконується усікання з коректним закриттям синтаксичних конструкцій.

Далі формується запит до моделі як композиція оптимізованого коду та конфігурації цільової вразливості:

$$P(C, V) = f_{prompt}(O(C), config(V)) \quad (2.5)$$

де f_{prompt} – побудова інструкції/контексту, а $config(V)$ фіксує опис потрібної вразливості та вимоги до формату відповіді.

Відповідь моделі задається відображенням:

$$R = AI(P(C, V)) \quad (2.6)$$

у якому R містить сгенерований модифікований контракт і (за наявності) виділений фрагмент змін.

Перевірка коректності виконується функцією валідації $Val(\cdot)$, що повертає структурований результат лише тоді, коли змінений фрагмент справді є підрядком повного модифікованого коду:

$$Val(R) = \begin{cases} \{contract, snippet\}, \text{ якщо } snippet \in contract, \\ \emptyset, \text{ інакше.} \end{cases} \quad (2.7)$$

Нарешті, успішність ін'єкції на вибірці з n оброблених контрактів оцінюється часткою валідних результатів:

$$S = \frac{\sum_{i=1}^n 1[Val(R_i) \neq \emptyset]}{n}, \quad (2.8)$$

де $1[\cdot]$ – індикаторна функція. Значення S інтерпретується як емпірична ймовірність того, що LLM-модифікація відповідає заданому формату та проходить базову валідацію включення фрагмента у повний контракт.

2.2.4 Створення збалансованого датасету та його переваги

В результаті застосування розроблених методів ін'єкції було сформовано частково штучний збалансований датасет смарт-контрактів, що містить по 1000 прикладів для кожної з п'яти категорій (таблиця 2.4).

Таблиця 2.4 – Структура створеного датасету

Категорія	Кількість контрактів
Integer Overflow	1000
Integer Underflow	1000
Timestamp Dependency	1000
Re-entrancy	1000
Безпечні контракти	1000
Разом	5000

Переваги створеного датасету:

- збалансованість класів. Для кожної категорії передбачено однакову кількість прикладів, що усуває типовий дисбаланс реальних вибірок;
- керованість і прозорість анотацій. Для кожного контракту однозначно визначено тип вразливості (або її відсутність), що спрощує формування навчальних і тестових підмножин;
- реалістичність коду. Модифікації виконано так, щоб зберегти притаманну продакшн-контрактам логіку, структуру та стиль;
- відтворюваність процесу. Ін'єкційний конвеєр повністю автоматизований і може бути повторений для отримання додаткових або розширених версій набору;

– придатність для навчання моделей. Склад і розмітка забезпечують оптимальні умови для тренування та об'єктивного тестування алгоритмів виявлення вразливостей.

Таким чином, збалансований корпус поєднує переваги реальних вихідних кодів із контрольованими умовами ін'єкції та робить вагомий внесок у дослідження безпеки смарт-контрактів.

2.2.5 Попередня обробка даних

Для забезпечення коректного функціонування моделей машинного навчання та підвищення їх здатності до узагальнення було здійснено комплексну попередню обробку вихідного коду смарт-контрактів. Основною метою цього етапу є усунення надмірності, мінімізація варіативності синтаксичних елементів і приведення даних до уніфікованого, структурованого вигляду. Такий підхід дає змогу зменшити шум у даних, уникнути перенавчання моделі на неінформативних особливостях і забезпечити стабільність результатів під час подальшого навчання.

2.2.5.1 Стандартизація імен змінних

Одним із ключових етапів попередньої обробки є стандартизація імен змінних. Вихідний код смарт-контрактів містить численні унікальні позначення змінних, функцій і структур, які не мають безпосереднього відношення до логіки вразливостей, але можуть створювати зайве семантичне навантаження для моделі.

Для усунення цієї проблеми всі змінні було перейменовано у уніфікований формат – наприклад, VAR1, VAR2, ..., VARN. Така нормалізація дозволила:

- зменшити вплив унікальних імен, характерних для конкретного контракту чи розробника;
- підвищити узагальнюваність моделі шляхом усунення надлишкової лексичної інформації, що не впливає на виявлення вразливостей;

– знизити ймовірність перенавчання, коли модель запам'ятовує частотні шаблони імен замість аналізу синтаксичної та логічної структури коду.

В результаті стандартизації код зберіг свою семантичну цілісність – тобто логіка виконання контракту залишилась незмінною, – але став більш однорідним з точки зору структури даних. Це забезпечило підвищення стійкості моделі до нових, раніше невідомих контрактів і покращило її здатність виявляти типові вразливості незалежно від стилю написання коду.

2.2.5.2 Оптимізація коду за рахунок видалення допоміжних функцій

На етапі попередньої обробки вихідного коду було здійснено оптимізацію структури смарт-контрактів з метою зниження складності аналізу та покращення релевантності навчальних прикладів.

Із текстів контрактів виключалися допоміжні функції, що не впливають безпосередньо на логіку виконання чи безпеку програми, зокрема методи логування, службові обгортки, процедури форматування та інші технічні елементи, призначені лише для спрощення коду.

Такий підхід дозволив зменшити структурну надмірність і сконцентрувати навчальну модель на ключових конструкціях, які визначають функціональність і потенційні вразливості смарт-контрактів. Після очищення коду решта функцій була стандартизована за стилем і структурою, що зменшило варіативність синтаксису між різними контрактами та підвищило стабільність процесу навчання.

Оскільки обробка довгих послідовностей у трансформерних архітектурах супроводжується значними обчислювальними витратами, довжина вхідного коду була обмежена 256 токенами. Такий компроміс забезпечив баланс між рівнем деталізації даних і продуктивністю обчислень.

У випадках, коли обсяг коду перевищував зазначене обмеження, надлишкові частини усікалися з кінця, зберігаючи найбільш інформативні сегменти. Для коротших прикладів застосовувалося доповнення нульовими токенами (padding), що дозволило зберегти фіксовану довжину послідовності.

Ця процедура забезпечила стандартизовану форму вхідних даних, відповідність архітектурним вимогам моделі та збереження ключових ознак, необхідних для коректного виявлення вразливостей у коді.

2.3 Висновки до другого розділу

1. Розглянуто й систематизовано наявні публічні датасети смарт-контрактів (SmartBugs Curated, SolidiFI, DAppSCAN, ScrawlD, VeriSmart) та показано, що жоден із них не забезпечує одночасно високу якість розмітки, масштаб, збалансованість класів і близькість до реальних розподілів коду. Вручну анотовані набори є малими (обмежена узагальнюваність), ін'єктовані – зручні для бенчмаркінгу, але менш репрезентативні, а автоматично розмічені – масштабні, проте містять інструментальні похибки. Це створює методичні передумови для формування власного навчального корпусу з контрольованою якістю даних.

2. Запропоновано метод формування збалансованого датасету для задачі виявлення вразливостей у смарт-контрактах: поєднання збору реальних контрактів із відкритих джерел із подальшим маркуванням (статичний аналіз) та двома комплементарними шляхами аугментації – детермінованою ін'єкцією вразливостей за шаблонами та контекстною ін'єкцією на основі великих мовних моделей. Такий підхід водночас забезпечує різноманіття синтаксичних реалізацій, контрольованість типів вразливостей і підвищує достовірність міток.

3. Реалізовано конвеєр збору та первинного маркування даних: автоматизоване вилучення Solidity-контрактів (Etherscan), первинна обробка статичними аналізаторами (Slither, Oyente) та агрегування результатів для формування початкового корпусу. Отримано показовий дисбаланс класів: Integer Overflow – 31 061 приклад, Integer Underflow – 23 702, Timestamp Dependency – 1 460, Re-entrancy – 310, а також 12 829 «безпечних» контрактів, що підтверджує необхідність подальшого балансування набору.

4. Запропоновано та впроваджено детермінований метод ін'єкції вразливостей, заснований на статичному аналізі та пошуку шаблонів (pattern

matching) із подальшою стандартизованою модифікацією коду. Для вразливості повторного входу реалізовано кероване порушення «Checks-Effects-Interactions» і заміну безпечних переказів на низькорівневий `call{value:...}`, з обов’язковою перевіркою компіляції й повторною валідацією статичними інструментами. Метод гарантує відтворюваність (ідентичний результат за однакового входу) та синтаксичну коректність.

5. Розроблено LLM-орієнтований метод ін’єкції вразливостей (prompt-керована модифікація коду), що забезпечує контекстно доцільні, «природні» зміни без руйнування функціональності контракту. Описано модулі попередньої обробки (укорочення/очищення коду з гарантією синтаксичної цілісності), правила формування промптів, вимоги до формату відповіді, а також процедури парсингу та відхилення невідповідних результатів. Це підвищує різноманітність прикладів і зменшує «синтетичний відбиток» ін’єкцій.

6. На основі двох каналів аугментації досягнуто збалансованої структури датасету 5 000 контрактів: по 1 000 прикладів для Integer Overflow, Integer Underflow, Timestamp Dependency, Re-entrancy та 1 000 «безпечних» (negatives). Для кожного екземпляра збережено тип вразливості та фрагмент (snippet), що спрощує подальшу підтримку багатоміткового виявлення і локалізаційні експерименти.

7. Сформовано процедури контролю якості: усунення дублікатів (хеш-фільтрація), перевірка компіляції solc, повторна валідація статичними аналізаторами, уніфікований формат збереження і ведення метаданих. Це підвищує надійність міток і відтворюваність експериментів.

8. Проведено нормалізацію даних для зниження варіативності, не пов’язаної з вразливостями: стандартизацію імен ідентифікаторів, видалення допоміжних (нерелевантних до безпеки) фрагментів, уніфікацію довжин послідовностей (до 256 токенів) із відповідним паддінгом/усіканням. Це зменшує ризик перенавчання на іменах і стилях, концентруючи модель на структурах, релевантних безпеці.

9. Показано, що отриманий датасет відповідає вимогам масштабованості, збалансованості, відтворюваності й практичної релевантності, що робить його придатним як для навчання й порівняння сучасних моделей виявлення вразливостей.

РОЗДІЛ 3. ПОБУДОВА МЕТОДІВ І МОДЕЛЕЙ ВИЯВЛЕННЯ ТА КЛАСИФІКАЦІЇ ВРАЗЛИВОСТЕЙ ТА ЇХ ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

Розділ присвячений побудові моделей і їх експериментальному дослідженню подає послідовну еволюцію підходів до виявлення, класифікації та локалізації вразливостей у смарт-контрактах і створений як цілеспрямована відповідь на сформульовані у «Вступі» завдання. Розділ спирається на збалансований набір даних і єдиний протокол підготовки (нормалізація коду, уніфіковані розбиття на train/val/test), що забезпечує чесне порівняння моделей. Спочатку обґрунтовується і оцінюється базова послідовна модель BiGRU-ATT, яка задає точку відліку та дозволяє кількісно виділити внесок двонапрямної обробки та механізму уваги. Далі демонструється перехід до використання трансформерного представлення коду (CodeBERT) у поєднанні з BiGRU і показується, чи дає така інтеграція стабільний виграш відносно бази. Наступним кроком є постановка задачі у мультимітковій формі для більш точного відображення одночасної наявності кількох вразливостей у контракті та аналіз того, як це впливає на якість прогнозування. Завершує розділ представлення архітектури локалізації (CodeBERT-GDLA) із дворівневою увагою та графовим підкріпленням, спрямованої на точне визначення фрагментів коду з вразливостями; наведено кількісні показники якості локалізації та приклади інтерпретації результатів для практичного аудиту. Усі експерименти виконано в єдиному середовищі з фіксованими налаштуваннями навчання, наведено порівняння з базовими підходами, абляційні дослідження ключових компонентів; наприкінці підсумовано висновки, що безпосередньо підтверджують доцільність обраних рішень і їх придатність до впровадження.

В розділі розв'язуються наступні дослідницькі питання:

1. Якою є ефективність моделей BiGRU-ATT і CodeBERT-BiGRU у задачі багатоміткової класифікації вразливостей та який внесок окремих компонентів (рекурентний шар, трансформерні представлення, механізм уваги) у підвищення метрик macro-F1, ROC-AUC та Hamming loss?

2. Чи забезпечує архітектура CodeBERT-GDLA зі структурно підкріпленою дворівневою увагою та Grad-CAM достатню точність слабосупервізованої локалізації вразливих фрагментів за метриками mAP, Hit@K та IoU?

3. Наскільки впровадження калібрувального регуляризатора L_{cal} та дослідження стійкості до синтаксичних і стилістичних змін коду підвищують надійність, стабільність і практичну придатність моделей у реальних сценаріях аудиту смарт-контрактів?

3.1 Початкова модель: BiGRU-ATT

3.1.1 Обґрунтування вибору GRU перед LSTM

Рекурентні нейронні мережі (RNN) – це різновид глибоких нейронних мереж, пристосованих до роботи з послідовними, впорядкованими за часом даними. До таких даних належать, наприклад, речення, що складаються зі слів, відеопослідовності, що складаються з кадрів тощо. Основною особливістю рекурентних мереж є здатність "запам'ятовувати" попередні елементи послідовності, що вже були опрацьовані [101]. Ці моделі можуть бути навчені та демонструють високі результати при розв'язанні складних задач послідовного навчання, таких як розпізнавання мовлення, машинний переклад та обробка природної мови (рисунок 3.1). Відповідно, рекурентні нейронні мережі є придатними для вирішення задачі виявлення вразливостей у смарт-контрактах на рівні коду мовою Solidity [102].

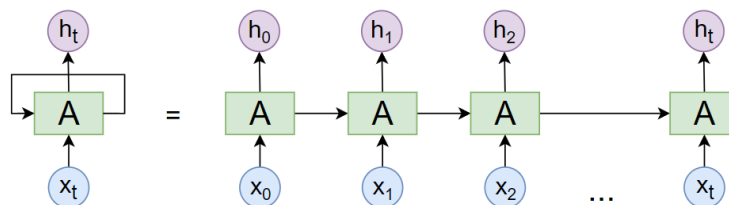


Рисунок 3.1 – Рекурентна нейронна мережа та її розгортання в часі.

Попри значну поширеність і концептуальну простоту, класичні рекурентні нейронні мережі (RNN) на практиці виявляються доволі складними для стабільного

та ефективного навчання, особливо в умовах реальних прикладних задач. Однією з головних причин такої нестійкості є добре відомі явища вибухаючих та зникаючих градієнтів [100]. Ці ефекти виникають у процесі поширення похибки в часі (backpropagation through time), коли градієнти багаторазово перемножуються на матриці ваг при переході через послідовні часові кроки. Унаслідок цього норма градієнта може експоненційно зростати (вибухаючі градієнти) або, навпаки, прямує до нуля (зникаючі градієнти), що суттєво ускладнює налаштування параметрів мережі, обмежує можливість моделювання довгострокових залежностей у послідовності та погіршує збіжність алгоритму навчання. В основі цих проблем лежить саме рекурентна структура архітектури, яка вимагає багаторазового поширення сигналу помилки через часові кроки мережі й, відповідно, ускладнює підтримання достатньо інформативного градієнта на великих часових відстанях.

З погляду організації взаємозв'язку між вхідними та вихідними даними, рекурентні нейронні мережі зазвичай використовують у трьох базових конфігураціях: «послідовність $\rightarrow$ послідовність», «послідовність $\rightarrow$ одиниця» та «один $\rightarrow$ послідовність» [103].

У схемі типу «послідовність $\rightarrow$ послідовність» (sequence-to-sequence) мережа перетворює одну вхідну послідовність на іншу вихідну послідовність. Класичним прикладом є задачі машинного перекладу, коли послідовність токенів вхідного речення однією мовою відображається у послідовність токенів іншою мовою; подібні конфігурації також застосовуються в системах діалогової взаємодії, узагальнення текстів тощо. Конфігурація «послідовність $\rightarrow$ одиниця» (sequence-to-one) орієнтована на ситуації, коли послідовність спостережень необхідно зіставити з одним узагальненим результатом. До цього класу належать задачі класифікації текстів, відео чи тимчасових рядів, у яких на вхід подається послідовність слів, кадрів або числових значень, а на виході формується вектор ймовірностей належності об'єкта до певних класів. Саме така схема дозволяє агрегувати інформацію з усіх часових кроків у компактне представлення, зручне для прийняття рішення. Нарешті, конфігурація «один $\rightarrow$ послідовність» (one-to-sequence) реалізує протилежний підхід: на вхід мережі подається один об'єкт (наприклад, вектор ознак або зображення), а на виході

генерується послідовність елементів. Подібні архітектури застосовують, зокрема, для побудови текстових описів до зображень, генерації тимчасових послідовностей на основі початкового стану системи, а також у задачах посимвольної генерації тексту.

У межах цієї дисертаційної роботи використовується конфігурація типу «послідовність $\rightarrow$ одиниця», оскільки задача виявлення вразливих смарт-контрактів формулюється як задача класифікації. Кожен смарт-контракт можна розглядати як послідовність елементів коду (токенів, інструкцій або операторів), і метою моделі є віднесення цієї послідовності до одного або кількох класів, що відповідають певним типам вразливостей. Таким чином, конфігурація «послідовність $\rightarrow$ одиниця» природно відображає постановку задачі та дозволяє ефективно агрегувати інформацію з усього фрагмента коду для прийняття остаточного класифікаційного рішення.

Зазначені ефекти безпосередньо проявляються в тому, що стандартні комірки RNN виявляються нездатними надійно зберігати інформацію про надто довгі послідовності. На практиці це означає, що модель добре відтворює локальні, короткострокові залежності, але втрачає чутливість до подій, які відбулися значно раніше у вхідному коді, тобто фактично не може оперувати довготривалою пам'яттю. За тривалого поширення сигналу похибки вплив далеких за часом кроків або згасає до нуля, або, навпаки, призводить до чисельної нестабільності, що унеможливорює коректне навчання моделі. З метою подолання цих обмежень були запропоновані модифіковані архітектури рекурентних мереж, зокрема LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit), які вводять механізми керованої пам'яті та додатковий внутрішній стан комірки [104]. Завдяки системі «вхідних», «вихідних» та «забувальних» (або згорткових) вентилів такі моделі здатні вибірково зберігати, оновлювати або відкидати інформацію про попередні кроки, що істотно покращує відтворення довгострокових залежностей у послідовностях, зокрема й у коді смарт-контрактів.

Архітектура GRU схожа на LSTM, але має менше вентилів (вентиль оновлення z_t та вентиль скидання r_t). Разом ці вентиля контролюють, як нова інформація оновлює стан. Вентиль оновлення відповідає за визначення кількості попередньої

інформації, яку потрібно перенести в наступний стан. Вентиль скидання використовується в моделі, щоб вирішити, скільки попередньої інформації слід відкинути (рисунок 3.2). Якщо r_t дорівнює нулю, комірка забуває свій попередній стан [105].

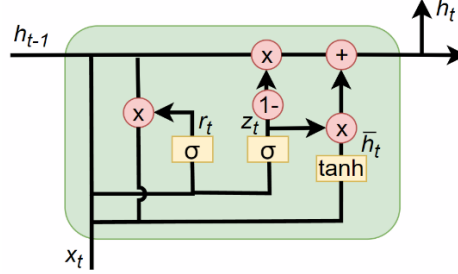


Рисунок 3.2 – Структура комірки GRU.

Усі з'єднання в GRU можна описати за допомогою таких формул:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \quad (3.1)$$

де z_t – гейт оновлення (update gate), σ – сигмоїдна функція активації, W_z та U_z – вагові матриці, x_t – входні дані, h_{t-1} – інформація з попереднього моменту часу t-1, b_z – зміщення (bias).

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r), \quad (3.2)$$

де r_t – гейт скидання (reset gate), σ – сигмоїдна функція активації, W_r та U_r – матриці ваг, x_t – входні дані, h_{t-1} – інформація з попереднього моменту часу t-1, b_r – зміщення (bias).

$$\bar{h}_t = \tanh(W_h x_t + r_t \odot (U_h h_{t-1}) + b_h), \quad (3.3)$$

де $\bar{h}_t$ – поточний вміст пам'яті (candidate hidden state), $\tanh$ – гіперболічна тангенс-функція, W_h та U_h – матриці ваг, r_t – гейт скидання, x_t – входні дані, h_{t-1} – інформація з попереднього моменту часу t-1, b_h – зміщення (bias).

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \bar{h}_t, \quad (3.4)$$

де h_t – фінальний стан пам'яті на поточному часовому кроці, z_t – гейт оновлення, h_{t-1} – інформація з попереднього моменту часу t-1, $\bar{h}_t$ – поточний вміст пам'яті.

Архітектура GRU (Gated Recurrent Unit) була обрана замість LSTM через меншу кількість параметрів: у GRU використовуються лише два гейти – оновлення (update gate) та скидання (reset gate), тоді як у LSTM – три [106]. Це означає, що GRU потребує

меншої кількості вагових коефіцієнтів і забезпечує вищу швидкість навчання, особливо при роботі з невеликими наборами даних. Тому вибір GRU є обґрунтованим з точки зору продуктивності та ефективного використання обчислювальних ресурсів під час навчання.

3.1.2 Двонаправлена обробка (BiGRU) для покращення розуміння коду

У запропонованій моделі застосовано двонапрямну архітектуру GRU (Bidirectional GRU), яка дає змогу для кожного токена в коді смарт-контракту враховувати як попередній, так і наступний контекст. Це є принципово важливим для аналізу програмного коду, оскільки семантика окремої інструкції часто визначається не лише її локальним оточенням, а й тим, які операції виконуються до та після неї; відповідно, доступ до повнішого контексту сприяє підвищенню точності виявлення вразливостей [107].

Для покращення якості моделювання послідовних залежностей архітектура нейронної мережі була розширена за рахунок введення додаткового, зворотного шару на основі GRU-комірок. На відміну від звичайної (однонапрямної) GRU, двонапрямна GRU складається з двох підмереж: одна обробляє послідовність у прямому напрямку (від початку до кінця), а інша – у протилежному напрямку (від кінця до початку). На виході їхні приховані стани поєднуються, завдяки чому модель одночасно використовує інформацію з «лівого» та «правого» контекстів (рисунок 3.3). Така організація дає змогу більш повно відобразити взаємозв'язки між операторами та виразами в обох напрямках послідовності та, як наслідок, підвищити чутливість моделі до складних патернів, характерних для вразливого коду.

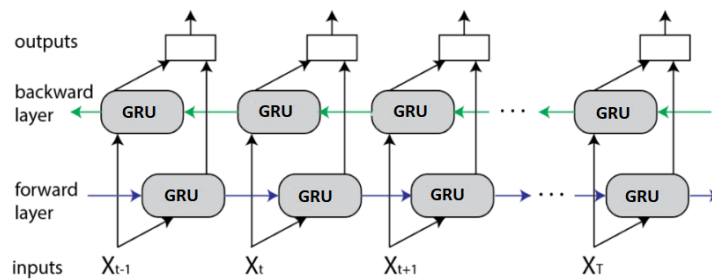


Рисунок 3.3 – Двонаправлена архітектура GRU.

3.1.3 Механізм уваги (Attention) для фокусування на важливих частинах коду

Одним із ключових обмежень базової архітектури GRU є те, що вона не має вбудованого механізму для явного розрізнення важливості різних фрагментів вхідної послідовності: усі часові кроки обробляються однаково, а внесок окремих токенів у фінальне рішення фактично «усереднюється». Щоб подолати цю проблему, у модель було інтегровано механізм уваги (attention), який дає змогу вибірково зосереджуватися на найбільш інформативних частинах коду, потенційно пов'язаних із наявністю вразливостей. Зокрема, attention накладається на вихідні представлення, сформовані двонапрямною GRU (BiGRU), і виконує зважування прихованих станів таким чином, щоб підсилити вплив найбільш значущих токенів під час класифікації. На відміну від стандартної GRU, де внесок кожного часового кроку до підсумкового векторного представлення визначається опосередковано через динаміку рекурентного оновлення, механізм уваги дозволяє явно моделювати відмінності у важливості інформації на різних позиціях послідовності. Він оцінює взаємозв'язки між елементами вхідних та вихідних представлень і призначає кожному прихованому стану відповідний ваговий коефіцієнт [108]. У результаті формується контекстний вектор, який є зваженою сумою виходів BiGRU і в якому більш релевантні фрагменти коду роблять більший внесок у прийняття рішення.

Оскільки в задачі класифікації вразливих смарт-контрактів різні токени (наприклад, виклики зовнішніх функцій, операції з балансами, маніпуляції зі сховищем) мають неоднакову діагностичну цінність, важливіші представлення мають

отримувати підвищену вагу [109]. Шар уваги навчається спільно з усією моделлю й призначає вагові коефіцієнти кожному прихованому стану, сформованому на k -му часовому кроці у шарі BiGRU. Таким чином, модель набуває здатності автоматично фокусуватися на критичних фрагментах коду, що покращує якість виявлення вразливостей і підвищує інтерпретованість отриманих рішень.

Формула для розрахунку ваги уваги α_t для токена t така:

$$\alpha_t = \frac{\exp(u_t^T u_w)}{\sum_t \exp(u_t^T u_w)}, \quad (3.5)$$

де u_w – вектор параметрів уваги, u_t – приховане представлення (hidden representation) для t -го кроку після додаткового перетворення.

Формула для розрахунку u_t така:

$$u_t = \tanh(W_w h_t + b_w), \quad (3.6)$$

де W_w – матриця ваг; b_w – зміщення (bias); h_t – вектор прихованого стану моделі для токена t , який кодує контекстну інформацію про цей токен з урахуванням його оточення в послідовності; $\tanh$ – нелінійна активаційна функція.

Отримавши значення розподілу ймовірностей уваги в кожен момент часу, обчислюється вектор ознак v , що містить текстову інформацію, а формула розрахунку має наступний вигляд:

$$v = \sum_t \alpha_t h_t. \quad (3.7)$$

Отримане зважене представлення також слугує основою для інтерпретації рішень моделі через аналіз розподілу коефіцієнтів уваги.

3.1.4 Векторне представлення коду смарт-контрактів

Смарт-контракти в блокчейні Ethereum зазвичай пишуться мовою високого рівня Solidity. Їх вихідний код є текстовим об'єктом, що складається з послідовності рядків, операторів та лексем. Однак більшість моделей глибокого навчання працюють не безпосередньо з текстом, а з числовими представленнями фіксованої розмірності. Тому виникає необхідність перетворити початковий код смарт-контрактів на

послідовність числових векторів, придатних для подальшої обробки нейронною мережею. Для розв'язання цього завдання в роботі використовуються підходи обробки природної мови (Natural Language Processing, NLP).

Обробка природної мови є міждисциплінарною галуззю, що поєднує методи інформатики, штучного інтелекту та комп'ютерної лінгвістики і зосереджується на автоматичному аналізі, інтерпретації та генерації текстів природною мовою. Сучасні NLP-системи значною мірою ґрунтуються на методах глибокого навчання, які дають змогу моделі самостійно формувати інформативні ознаки з необроблених вхідних даних, мінімізуючи потребу в ручному проектуванні ознак. Традиційний підхід, за якого дослідник самостійно конструює набір характеристик, як правило, є трудомістким, схильним до суб'єктивності й часто призводить до втрати частини суттєвої інформації, що обмежує якість побудованих моделей.

Перехід від текстового представлення коду до числових векторів може здійснюватися різними способами. Класичні методи включають, зокрема, модель Bag-of-Words [110], у якій текст подається у вигляді вектора частот або наявності окремих термів, та підхід N-Gram [111], що враховує локальні послідовності токенів (n-грам) для фіксації простих контекстних залежностей. У більш сучасних архітектурах машинного навчання широкого поширення набули також спеціальні шари векторного відображення (Embedding layers), які навчаються спільно з основною моделлю і забезпечують компактне, семантично насичене відображення токенів (у тому числі елементів програмного коду) у простір низької розмірності, що є зручним вхідним представленням для нейронних мереж.

У даному дослідженні для перетворення вихідного коду смарт-контрактів у вектори була використана модель Word2Vec [112]. Цей алгоритм оперує великим корпусом текстів (у нашому випадку – кодів смарт-контрактів), у межах якого кожне слово з фіксованого словника представляється у вигляді вектора. Алгоритм проходить через кожну позицію t у тексті, визначаючи центральне слово s та контекстне слово o . Далі, на основі подібності векторів слів s та o , обчислюється ймовірність появи o за умови s (або навпаки), і вектори коригуються таким чином, щоб максимізувати цю ймовірність. Наприклад, слова, що позначають типи даних

uint32 та uint128, у векторному просторі будуть розміщені поблизу одне одного, тоді як слова, що позначають модифікатори функцій onlyMinter та onlyOwner, будуть розташовані значно далі (рисунок 3.4).

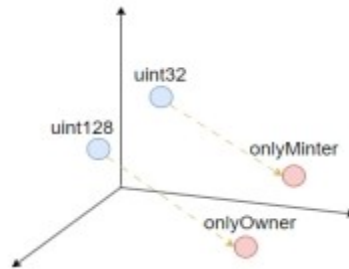


Рисунок 3.4 – У моделі Word2Vec слова з подібним значенням розташовуються ближче у векторному просторі, що відображає їхню семантичну схожість

З метою підвищення ефективності моделі Word2Vec із корпусу попередньо вилучаються стоп-слова – слова, що не несуть суттєвого семантичного навантаження. У випадку смарт-контрактів до таких слів належать ідентифікатори змінних та функцій, які не мають значення для виявлення вразливостей у контрактах.

3.1.5 Загальна архітектура моделі BiGRU-ATT

Побудована модель отримує на вхід вихідний код смарт-контракту, вибирає фрагменти коду, які потенційно можуть мати вразливості, та перетворює їх на масив векторів за допомогою векторних вбудовувань. Кількість токенів у векторному представленні фрагментів контракту встановлюється на 100 (що було визначено як найкраще серед значень [50, 100, 150]), тоді як розмірність векторів встановлюється на 300 (що було визначено як найкраще серед значень [200, 300, 400]). На наступному кроці ці векторні представлення передаються на вхід двонаправленого шару GRU. Цей шар складається з 600 блоків GRU. Після цього застосовується шар з механізмом уваги, вихід якого передається на лінійний шар. Цей шар має два виходи та використовує функцію активації Softmax (рисунок 3.5).

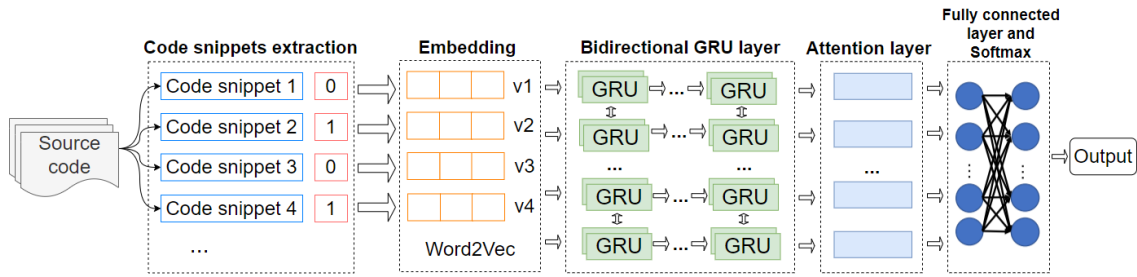


Рисунок 3.5 – Архітектура двонаправленого GRU з механізмом уваги.

Під час навчання моделей як функція втрат використовувалася бінарна крос-ентропія. Як оптимізатор використовувався алгоритм градієнтного спуску Адамса. Значення швидкості навчання було обрано рівним 0.0005 (це значення було визначено як найкраще серед значень [0.0005, 0.0002, 0.001]), а як коефіцієнт відсіву – 0.2 (це значення було визначено як найкраще серед значень [0.2, 0.3, 0.4]). Розмір пакету встановлено на 64. Ця архітектура моделі, заснована на нейронних мережах, була визначена як оптимальна після серії емпіричних досліджень.

3.1.6 Результати моделі BiGRU-ATT

Ефективність кожної моделі оцінюється за допомогою метрик F1-міра, точність (precision) та повнота (recall). Базовими значеннями для обчислення зазначених метрик є: істинно позитивні (TP), істинно негативні (TN), хибнопозитивні (FP) та хибнонегативні (FN) результати [113].

Точність (precision) відображає співвідношення між кількістю істинно позитивних результатів і загальною кількістю позитивних прогнозів. Ця метрика дозволяє оцінити достовірність позитивного прогнозу моделі. Повнота (recall), або чутливість, показує, яка частка дійсно позитивних випадків була правильно класифікована. Формули обчислення точності та повноти наведено нижче:

$$Precision = \frac{TP}{TP + FP}, \quad (3.8)$$

$$Recall = \frac{TP}{TP + FN}. \quad (3.9)$$

Міра F-бета забезпечує більш збалансовану характеристику моделі, ніж використання окремо метрик точності чи повноти. Вона обчислюється як зважене гармонійне середнє значення точності та повноти:

$$F_{\beta} = (1 + \beta^2) \frac{Precision * Recall}{(\beta^2 * Precision) + Recall} \quad (3.10)$$

Найкраще значення міри F-бета – 1, найгірше – 0. Параметр β визначає ступінь важливості повноти відносно точності.

У Таблиці 3.1 наведено результати оцінювання різних моделей згідно з основними метриками. Для порівняння було обрано такі архітектури: стандартна RNN (Simple RNN), LSTM, двонаправлена LSTM (BiLSTM), двонаправлена GRU (BiGRU) та двонаправлена LSTM з механізмом уваги (BiLSTM-ATT).

Таблиця 3.1 – Результати оцінки моделі

Модель	Точність, %	Повнота, %	F-міра ($\beta=2$), %
Simple RNN	66.34	64.85	65.14
LSTM	73.28	76.33	75.70
BiLSTM	86.12	87.97	87.59
BiGRU	86.05	88.10	87.68
BiLSTM-ATT	89.87	90.66	90.50
BiGRU-ATT	90.03	91.76	91.41

Як видно з таблиці 3.1, найгірші результати показала модель на основі Simple RNN, F-міра якої склала 65,14%. Наступною за нею йде модель на основі LSTM, яка продемонструвала дещо кращу ефективність порівняно з попередньою. Значно вищі результати показали моделі, побудовані на двонаправлених архітектурах LSTM та GRU, що обумовлено здатністю таких моделей враховувати контекст як у прямому, так і у зворотному напрямках. Зокрема, для моделі на основі BiGRU значення F-міри становило 87,68%.

Модифікації цих моделей шляхом додавання механізму уваги (attention) також суттєво покращили результати. Отримані метрики для моделей BiLSTM-ATT та BiGRU-ATT виявилися близькими за значеннями, однак модель BiGRU-ATT показала трохи вищий рівень повноти (recall), що дозволило підвищити F-міру до 91,41%.

3.1.7 Обмеження моделі BiGRU-ATT

Попри високі значення показників точності, модель BiGRU-ATT має низку принципових обмежень щодо здатності до глибокого «розуміння» програмного коду. Вона оперує послідовностями токенів і їх векторними представленнями, проте фактично не моделює повноцінну логіку виконання програми та її семантичну структуру. Така архітектура не здатна в достатній мірі виявляти причинно-наслідкові залежності в коді й коректно інтерпретувати складні схеми взаємодії між функціями та контрактами. Це, своєю чергою, може стати критичним обмеженням у разі появи нових, більш витончених або прихованих типів вразливостей, для виявлення яких необхідне глибше семантичне моделювання поведінки смарт-контрактів.

3.2 Багатоміткова модель CodeBERT-BiGRU

Хоча модель BiGRU-ATT продемонструвала високу ефективність у задачі класифікації смарт-контрактів за наявності вразливостей, її архітектура має принципові обмеження. Головним з них є те, що модель орієнтується переважно на синтаксичну послідовність токенів у коді, не враховуючи повноцінно його семантичний зміст. При цьому смарт-контракт є не лише лінійною послідовністю інструкцій, але й логічно зв'язаною структурою, у якій значення мають і типи змінних, і контексти викликів функцій, і модифікатори доступу, і структура викликів.

Вразливості можуть виникати внаслідок взаємодії декількох компонентів коду, віддалених один від одного, що важко виявити лише за допомогою аналізу локальної послідовності токенів. Наприклад, неправильно використаний модифікатор функції

або некоректна логіка в обробці зовнішніх викликів може не супроводжуватись очевидними синтаксичними ознаками. Тому для виявлення таких вразливостей необхідне глибше розуміння структури та семантики коду.

Крім того, моделі на основі GRU не мають знань про мову програмування Solidity, а отже – не можуть враховувати контекст виконання, специфіку конструкцій мови чи типові шаблони програмування. Це обмежує їхню здатність до генералізації та адаптації до нових типів вразливостей, що раніше не зустрічались у навчальній вибірці.

Таким чином, виникла потреба у моделі, яка здатна оперувати не лише токенами, але й розуміти значення цих tokenів у контексті коду, що й стало мотивацією до переходу на CodeBERT.

Традиційно завдання виявлення вразливостей у смарт-контрактах формувалося як бінарна класифікація: модель визначає, чи є в контракті вразливість взагалі, без уточнення типу. Такий підхід обмежений, оскільки не розрізняє різні категорії вразливостей і не може виявити більше ніж одну проблему одночасно. Однак один смарт-контракт може містити кілька вразливостей різних типів одночасно. У реальних умовах часто зустрічаються контракти, які мають комбінацію вразливостей – наприклад, одночасно присутні вразливість повторного входу (reentrancy) та цілочислове переповнення, або часові залежності та помилки порядку транзакцій. У разі застосування звичайної багатокласової класифікації модель була б змушена вибрати лише один клас уразливості ігноруючи інші, що призводить до втрати інформації та помилкової класифікації. Ба більше, більшість попередніх досліджень обмежувалися саме бінарними схемами (“є вразливість/немає”), які лише виявляють факт наявності проблеми і не визначають її тип. Це суттєва вада, адже для забезпечення безпеки важливо ідентифікувати кожен тип вразливості у коді. Багатоміткова (multi-label) класифікація вирішує цю проблему, дозволяючи моделі одночасно присвоювати кілька міток (класів) одному контракту. Таким чином, якщо в коді є декілька вразливостей, модель зможе позначити всі відповідні типи, не обмежуючись одним. Це підвищує повноту виявлення і забезпечує більш точний аналіз безпеки. Крім того, багатомітковий підхід

відображає реальні сценарії експлуатації: атаки можуть комбінувати кілька різних вразливостей, тому система виявлення повинна бути чутливою до всього спектру загроз [114].

3.2.1 CodeBERT для семантичного аналізу програмного коду на основі NLP

Трансформери – це архітектура нейронних мереж, яка вперше була представлена в статті «Attention is All You Need» у 2017 році [115]. Основною інновацією трансформерів є механізм уваги (attention mechanism), який дає змогу моделі динамічно зосереджуватися на різних частинах вхідних даних, що робить цю архітектуру особливо ефективною для завдань обробки природної мови. На відміну від попередніх підходів, таких як рекурентні нейронні мережі (RNN) або згорткові нейронні мережі (CNN), трансформери не потребують послідовної обробки даних і повністю базуються на механізмах уваги. Це суттєво пришвидшує процес навчання та покращує здатність моделі працювати з довгими залежностями в текстах [116].

Однією з найвідоміших реалізацій трансформерної архітектури є BERT (Bidirectional Encoder Representations from Transformers), розроблена компанією Google у 2018 році [117]. Головна відмінність BERT від попередніх моделей трансформерів полягає в її здатності обробляти текст двонаправлено. Традиційні моделі, зокрема GPT (Generative Pre-trained Transformer), працюють або зліва направо, або справа наліво, що обмежує обсяг контексту, доступного для кожного слова у реченні. Натомість BERT аналізує контекст в обох напрямках, що дає змогу краще зрозуміти контекстуальні взаємозв'язки між словами. Рисунок 3.6 демонструє загальну архітектуру моделей класифікації вихідного коду, побудованих на основі CodeBERT.

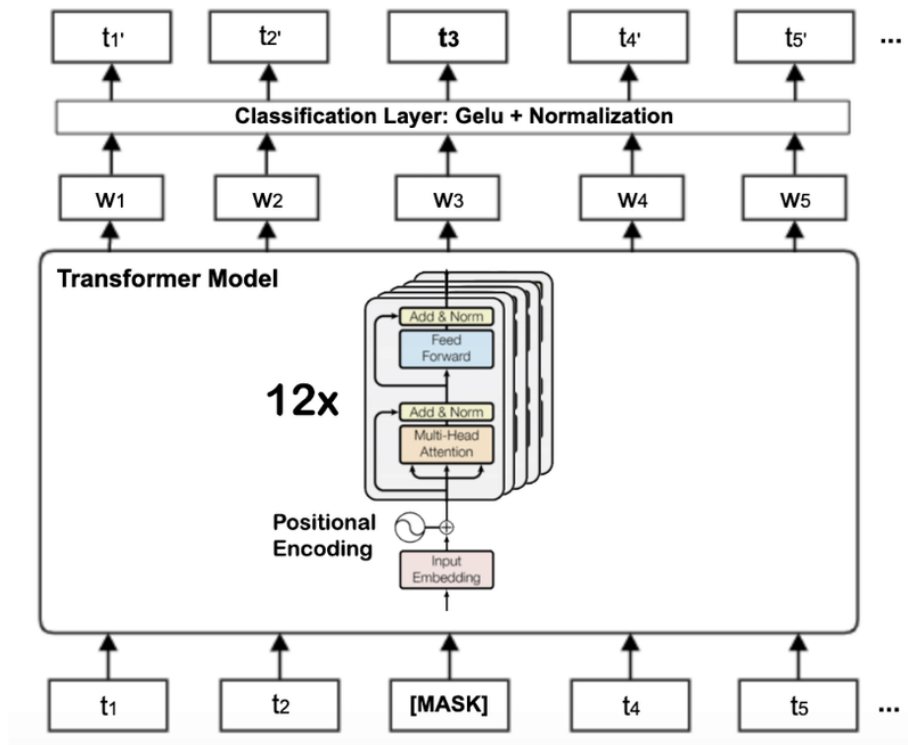


Рисунок 3.6 – Базова архітектура BERT на основі Transformer з дванадцятьма блоками енкодера

У контексті аналізу смарт-контрактів BERT може застосовуватись для вирішення низки завдань, зокрема: класифікації, виявлення вразливостей, автоматичної корекції коду тощо. Завдяки своїй здатності моделювати складні залежності у вхідних даних, BERT ефективно працює як із синтаксичними, так і з семантичними особливостями мов програмування для смарт-контрактів, зокрема Solidity [118]. Це робить модель потужним інструментом для виявлення потенційних вразливостей та помилок у коді, що має критичне значення для забезпечення безпеки та надійності блокчейн-платформ.

CodeBERT – це трансформерна модель, спеціально навчена для роботи з програмним кодом у поєднанні з природною мовою [119]. Вона є результатом передтренування на мільйонах пар *{код – коментар}*, що дозволяє їй розуміти семантику та логіку коду на глибшому рівні, ніж традиційні моделі типу RNN чи GRU. CodeBERT підтримує декілька мов програмування, включно з Solidity, що робить її особливо релевантною для задач безпеки смарт-контрактів.

На відміну від моделей, які працюють лише з послідовностями, CodeBERT використовує архітектуру трансформерів з механізмом самоуваги (self-attention), що

дозволяє їй аналізувати взаємозв'язки між усіма токенами незалежно від їхньої позиції в коді. Завдяки цьому модель здатна виявляти залежності між змінними, умовами та функціями, навіть якщо вони знаходяться на великій відстані один від одного в коді.

Інтеграція CodeBERT у побудову моделі для виявлення вразливостей дозволяє перейти від поверхневого аналізу до семантичного розуміння коду. Модель здатна не просто «бачити» окремі токени, а й інтерпретувати їхню роль у загальній логіці програми, що критично важливо для розпізнавання складних вразливостей, пов'язаних з умовами виконання, залежностями між станами, та специфікою контрактних викликів.

Таким чином, застосування CodeBERT є не лише технічним вдосконаленням, а й якісним переходом до нового рівня аналізу, де машинне навчання поєднується з глибоким розумінням контексту виконання програмного коду.

3.2.2 Архітектура CodeBERT-BiGRU

Для забезпечення коректної роботи моделі глибокого навчання вихідний код кожного смарт-контракту необхідно було перетворити у формат, придатний для обробки нейронною мережею. На першому етапі здійснювалася токенізація тексту контрактів із використанням попередньо навченого токенізатора, пов'язаного з моделлю CodeBERT, а саме `RobertaTokenizer` з бібліотеки `Transformers` [120]. Застосування саме цього токенізатора дає змогу гарантовано узгодити розбиття коду на токени зі словником, використаним під час передтренування CodeBERT, що є критично важливим для коректної інтерпретації вхідних даних моделлю.

Обробка довгих текстових послідовностей традиційно становить окрему проблему в задачах глибинного навчання через обмеження обчислювальних ресурсів і квадратичну залежність складності механізму самоуваги від довжини послідовності. З огляду на це було встановлено обмеження на максимальну довжину вхідної послідовності, яке дорівнює 256 токенам. Таке значення розглядається як компроміс між деталізацією представлення коду та прийнятною вартістю обчислень. У випадку,

коли фрагмент коду містив більше ніж 256 токенів, надлишкові токени відсікалися (truncation); якщо ж довжина послідовності була меншою, вона доповнювалася спеціальними заповнювальними значеннями (padding) до фіксованої довжини.

Запровадження єдиної довжини вхідних послідовностей спрощує формування батчів, відповідає технічним обмеженням моделі CodeBERT та дає змогу зберегти найбільш релевантні фрагменти коду, пов'язані з потенційними вразливостями, при одночасному контролі за обсягом обчислень.

Для оцінки ефективності моделі набір даних D було розподілено на три підмножини: тренувальну, валідаційну та тестову:

$$D = D_{train} \cup D_{val} \cup D_{test}. \quad (3.11)$$

Розподіл вибірки на навчальну, валідаційну та тестову підмножини здійснювався випадковим чином із збереженням початкового співвідношення між вразливими та невразливими смарт-контрактами. Валідаційна підмножина, що становила 20% від загальної кількості прикладів, використовувалася для налаштування гіперпараметрів моделі та моніторингу процесу навчання (контроль перенавчання, вибір найкращої епохи тощо). Тестова підмножина (також 20%) не залучалася до процесу тренування і застосовувалася виключно для підсумкового, об'єктивного оцінювання якості побудованої моделі.

Як базову архітектуру для розв'язання задачі виявлення вразливостей було обрано попередньо навчену модель CodeBERT від Microsoft, що являє собою адаптацію моделі BERT, спеціально орієнтовану на роботу з вихідним програмним кодом. У рамках даного дослідження CodeBERT було додатково донавчено (fine-tuned) на сформованому датасеті смарт-контрактів, що дало змогу пристосувати попередньо набуті нею представлення до специфіки цільової задачі класифікації вразливостей.

На вхід моделі подається текстовий представник смарт-контракту, тобто його вихідний код мовою Solidity. Цей код попередньо перетворюється на послідовність токенів за допомогою токенизатора, пов'язаного з CodeBERT (попередньо натренованого токенизатора моделі). Токенизатор здійснює розбиття коду на лексичні

одиниці та кодування їх у формат, сумісний зі словником моделі [121]. Отримана послідовність токенів подається на вхід CodeBertModel, яка, використовуючи механізм самоуваги та багатошарову трансформерну архітектуру, формує контекстуалізовані векторні представлення токенів, що слугують основою для подальшої класифікації смарт-контрактів за наявністю чи відсутністю вразливостей.

Замість використання лише прихованого стану токена [CLS], який традиційно агрегує інформацію про всю послідовність і застосовується для класифікації, запропоновано використовувати всі токени вхідної послідовності. Ці токени передаються на вхід одношарової двонаправленої GRU (Bidirectional GRU). Застосування двонаправленої GRU дає змогу моделі краще захоплювати контекст у обох напрямках – зліва направо та справа наліво, що покращує розуміння контекстуальних взаємозв'язків між токенами у ширшому контексті [121]. Це, у свою чергу, дозволяє точніше інтерпретувати зміст вхідного коду та підвищити точність класифікації.

3.2.2.1 Вихідний класифікатор

Після проходження через двонаправлену GRU, останній прихований стан (або конкатенація останніх станів обох напрямків для BiGRU) подається на повнозв'язний шар (dense layer). Цей шар виконує лінійну трансформацію до простору розмірності, рівної кількості класів (типів вразливостей). Нехай K – кількість вразливостей, які ми хочемо виявляти. Тоді вихідний шар має K нейронів, кожен з яких відповідає одному класу (наприклад, у наших експериментах $K=5$, що відповідає п'яти основним категоріям вразливостей: Reentrancy, Integer Overflow, Integer Underflow, Transaction-Order Dependency (TOD) та Timestamp Dependency). На відміну від одномоїткової моделі, де зазвичай використовується softmax-активація, у багатомітковому випадку на кожен вихідний нейрон застосовується сигмоїдна активація:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \quad (3.12)$$

Сигмоїдна активація перетворює лінійну комбінацію z_k на імовірність $p_k = \sigma(z_k)$ того, що зразок належить до класу k . Кожен вихідний нейрон працює незалежно, моделюючи розподіл Бернуллі для наявності відповідної вразливості. Така активація дозволяє одночасно прогнозувати кілька вразливостей: імовірності по різних виходах не змагаються між собою (як у випадку softmax), а обчислюються окремо [122].

3.2.2.2 Бінаризація результатів

Для перетворення імовірнісного виходу моделі у підсумкове рішення використовується фіксований поріг. Зокрема, встановлено поріг 0.5 (50%). Якщо $p_k > 0.5$, то модель прогнозує наявність уразливості типу k у даному контракті; якщо $p_k \leq 0.5$ – вразливість k вважається відсутньою. Таким чином формується вихідний вектор міток довжини K , що містить 1 для виявлених категорій вразливостей і 0 для невиявлених. Наприклад, якщо для контракту вихідні значення перевищують поріг для двох класів – Reentrancy та Timestamp Dependency, – то модель поверне мітки обох цих категорій одночасно. Такий підхід відображає всі знайдені слабкі місця контракту в одному проході моделі.

3.2.2.3 Функція втрат

Для навчання багатоміткової моделі застосовано бінарну крос-ентропію (Binary Cross-Entropy, BCE) як функцію втрат. Вона обчислюється окремо для кожного з K виходів і потім усереднюється. Нехай $y_k \in [0,1]$ – істинне значення наявності уразливості k , а p_k – імовірність від моделі. Тоді функція втрат для одного зразка становить:

$$L = -\frac{1}{K} \sum_{k=1}^K [y_k \log p_k + (1 - y_k) \log(1 - p_k)], \quad (3.13)$$

де $p_k = \sigma(z_k)$ – вихід сигмоїди для класу k .

Такий підхід еквівалентний сумі бінарних лог-лоссів по кожному класу і вирішує задачу багатоміткової класифікації шляхом оптимізації всіх виходів одночасно. Для оптимізації використовувався алгоритм Adam (adaptive moment estimation) з початковим темпом навчання $\alpha = 2 \times 10^{-5}$, який добре зарекомендував себе при тонкому налаштуванні трансформерних моделей.

3.2.2.4 Гіперпараметри тренування

Модель CodeBERT-BiGRU тренувалася на підготовленому корпусі смарт-контрактів з розміченими вразливостями. Розмір батчу склав 32, що забезпечує стабільність оцінки градієнта і вписується в обмеження пам'яті GPU при обробці довгих послідовностей коду. Навчання проводилося протягом 10 епох; при цьому використовувалася перевірка на валідаційній вибірці для запобігання перенавчанню (при відсутності покращення F1-міри протягом 2-3 епох навчання зупинялося достроково). При побудові токенизаційної послідовності максимально допустима довжина встановлювалася рівною 512 токенам (обмеження трансформера CodeBERT); у разі довших контрактів код розбивався на сегменти, які оброблялися окремо або вибірково (наприклад, аналізувалися тільки критичні функції). Прихований розмір (hidden size) шару GRU, як зазначено, дорівнював 256, а вихідний повнозв'язний шар мав розмірність 5 (за числом класів вразливостей). Під час навчання дані подавалися модель із перемішуванням (shuffling) для кожної епохи, щоб усереднити порядок прикладів і забезпечити стабільну збіжність.

Рисунок 3.7 демонструє загальну архітектуру запропонованої моделі.

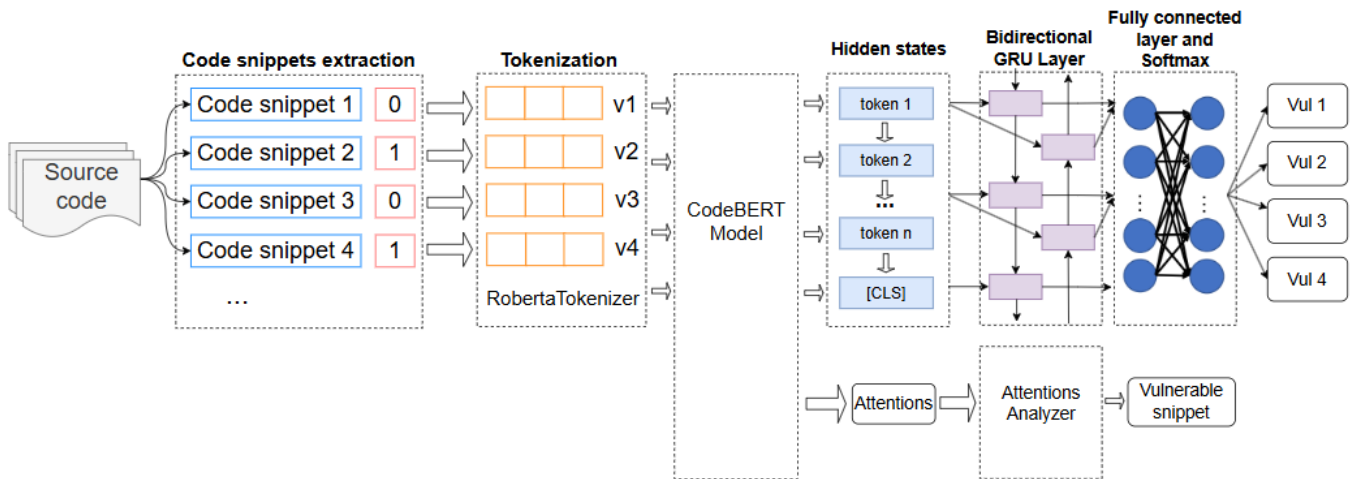


Рисунок 3.7 – Загальна архітектура запропонованої моделі CodeBERT-BiGRU

3.2.2.5 Метод локалізації вразливостей

Для виявлення та локалізації вразливостей у смарт-контрактах було використано аналіз векторів уваги, згенерованих моделлю CodeBERT. Аналіз векторів уваги дав змогу ідентифікувати токени та фрагменти коду, на які модель звертала найбільше уваги під час класифікації. Такий підхід дозволив не лише виявляти потенційні вразливості, але й точно визначати розташування проблемних ділянок у коді, які потребують додаткового аналізу та виправлень.

Цей процес включає кілька ключових етапів:

Етап 1: Отримання прогнозів та векторів уваги.

Модель обробляє вхідні дані та повертає логіти для кожного токена, а також ваги уваги (attention weights). Ваги уваги показують, яку частку уваги модель приділяє кожному токену під час аналізу інших токенів. Вони обчислюються за наступною формулою:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (3.14)$$

де Q – матриця запитів (query matrix), K – матриця ключів (key matrix), V – матриця значень (value matrix), d_k – розмірність ключа (key dimension).

Етап 2: Вибір векторів уваги з останнього шару.

Вектори уваги обираються з останнього шару трансформера, оскільки саме вони відображають найвищий рівень розуміння контексту моделлю.

Етап 3: Усереднення уваги між головами механізму уваги.

Виконується усереднення значень уваги між усіма головами (heads) механізму уваги в останньому шарі трансформера для отримання узагальненого уявлення про те, як модель розподіляє увагу між токенами.

Етап 4: Додаткове усереднення між токенами.

Після усереднення між головами механізму уваги виконується додаткове усереднення між усіма токенами у вхідній послідовності. Це дозволяє сформувати єдиний вектор уваги для всього вхідного набору, що спрощує подальший аналіз і дає змогу інтерпретувати, на які аспекти вхідних даних модель звертала найбільшу увагу загалом.

Етап 5: Виключення спеціальних токенів.

Із подальшого аналізу виключаються перший і останній токени, оскільки вони зазвичай містять метайнформацію (наприклад, [CLS], [SEP]) і не пов'язані зі змістовною частиною коду смарт-контракту.

Етапи трансформації матриці уваги останнього шару наступні:

$$\begin{aligned} \begin{bmatrix} w_{00} & \dots & w_{0j} \\ \dots & \dots & \dots \\ w_{i0} & \dots & w_{ij} \end{bmatrix} \times (num_heads = 12) &\rightarrow \begin{bmatrix} w_{mean_{00}} & \dots & w_{mean_{0j}} \\ \dots & \dots & \dots \\ w_{mean_{i0}} & \dots & w_{mean_{ij}} \end{bmatrix} \\ &\rightarrow \begin{bmatrix} token_att_min_0 \\ \dots \\ token_att_min_j \end{bmatrix} \end{aligned} \quad (3.15)$$

Етап 5: Визначення важливих фрагментів коду.

Шляхом проходження по всіх можливих вікнах фіксованого розміру (*window_size*) обчислюється сума усереднених значень уваги для кожного вікна. Вікно, яке має максимальну суму усередненої уваги, вважається таким, що відповідає найбільш важливому фрагменту коду.

Для кожного можливого вікна розміру *window_size* токенів виконується обчислення суми усередненої уваги за наступною формулою:

$$windows_sum_j = \sum_{i=j}^{j+window_size} token_att_mean[i], \quad (3.16)$$

де j – індекс початку вікна в послідовності токенів.

Далі визначаються індекси токенів, які відповідають вікну з найвищим рівнем уваги. Ці токени вважаються тими, які формують кодовий фрагмент, що модель розглядає як найбільш значущий або потенційно вразливий.

Початковий індекс цього вікна визначається за такою формулою:

$$start_index = \arg \max_j (windows_sum_j). \quad (3.17)$$

На основі знайденого початкового індексу та заданого розміру вікна обираються токени, що формують найважливіший фрагмент коду. Далі ці токени декодуються назад у текст за допомогою токенізатора, щоб отримати текстове представлення відповідного фрагмента. У результаті формується фрагмент коду смарт-контракту, який модель вважає найімовірнішим кандидатом на наявність вразливостей.

Такий підхід дає змогу глибше зрозуміти, на які частини коду модель звертає найбільшу увагу, що, своєю чергою, може вказувати на потенційні вразливості або критично важливі аспекти логіки смарт-контракту.

3.2.3 Переваги моделі CodeBERT-BiGRU перед BiGRU-ATT

Для навчання моделі та проведення експериментів було використано такі програмні засоби та технічні ресурси. Розробка та тестування виконувалися мовою програмування Python, яка забезпечує гнучкість та потужні можливості для роботи з алгоритмами машинного навчання. Основним фреймворком, що використовувався для побудови та навчання моделі, був PyTorch Lightning, який дозволив структуровано організувати процес навчання, зробивши його більш чистим, модульним і масштабованим.

Усі етапи розробки й тестування проводилися в операційній системі Windows 11.

Технічна конфігурація комп'ютера, на якому здійснювалося навчання та експерименти, включала такі характеристики:

- Процесор: Intel Core i9-12900K, що забезпечує високу продуктивність завдяки 16 ядрам, 24 потокам і максимальній тактовій частоті до 5.2 ГГц.
- Оперативна пам'ять: 32 ГБ DDR4, що дозволяє ефективно обробляти великі обсяги даних і складні моделі без обмежень по пам'яті.
- Графічний процесор: NVIDIA GeForce RTX 3090 з 24 ГБ відеопам'яті типу GDDR6X.
- Накопичувач: SSD 1 ТБ, що забезпечує швидкий доступ до даних і ефективне збереження великих наборів даних та результатів експериментів.

Зазначена конфігурація надала необхідні обчислювальні ресурси та швидкодію, необхідні для виконання складних задач машинного навчання та аналізу даних.

Для навчання моделі використовувався оптимізатор AdamW у поєднанні з планувальником швидкості навчання (learning rate scheduler), що дозволяло адаптувати швидкість навчання залежно від стадії тренування. У процесі навчання використовувався розмір пакета (batch size) 32, загалом було проведено 8 епох, кожна з яких складалась із 4 кроків. Архітектура моделі включала один шар GRU з розміром прихованого стану 16. Такі параметри дозволили уникнути перенавчання та досягти оптимальних результатів.

Під час навчання відстежувалися метрики точності (accuracy) та втрат (loss) на тренувальній та валідаційній вибірках.

Завдяки ретельному налаштуванню параметрів та архітектури моделі було досягнуто точності 98,67% на тренувальних даних і 97,34% на валідаційних даних (рисунки 3.8 та 3.9). Ці результати підтверджують високу ефективність обраного підходу для розв'язання поставленої задачі.

Динаміка зміни значень функції втрат на тренувальних та валідаційних даних відображена на рисунках 3.10 та 3.11 відповідно. Вказані графіки демонструють, як у

процесі навчання модель поступово зменшувала помилки, досягаючи все нижчих значень втрат із кожною ітерацією.

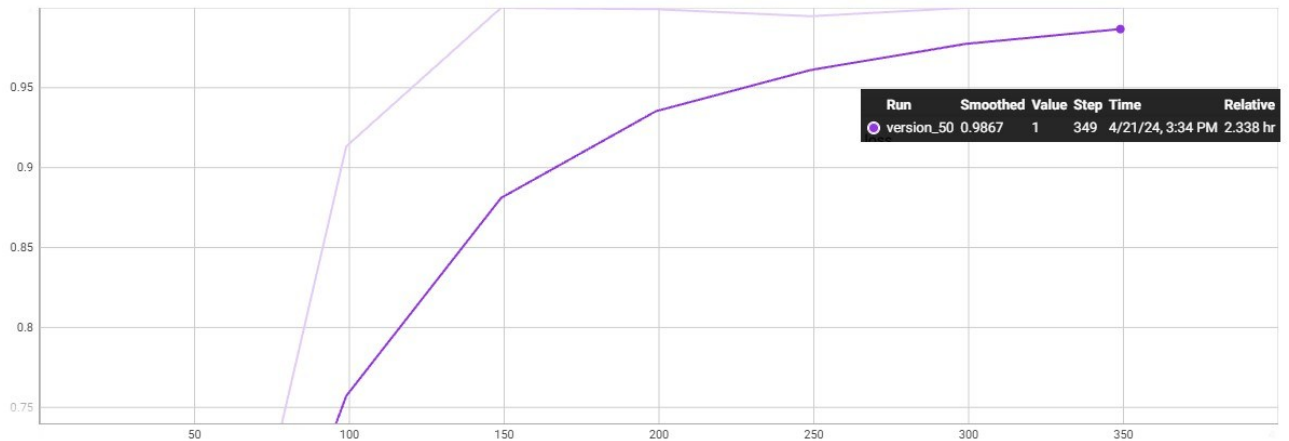


Рисунок 3.8 – Зміна точності моделі під час навчання

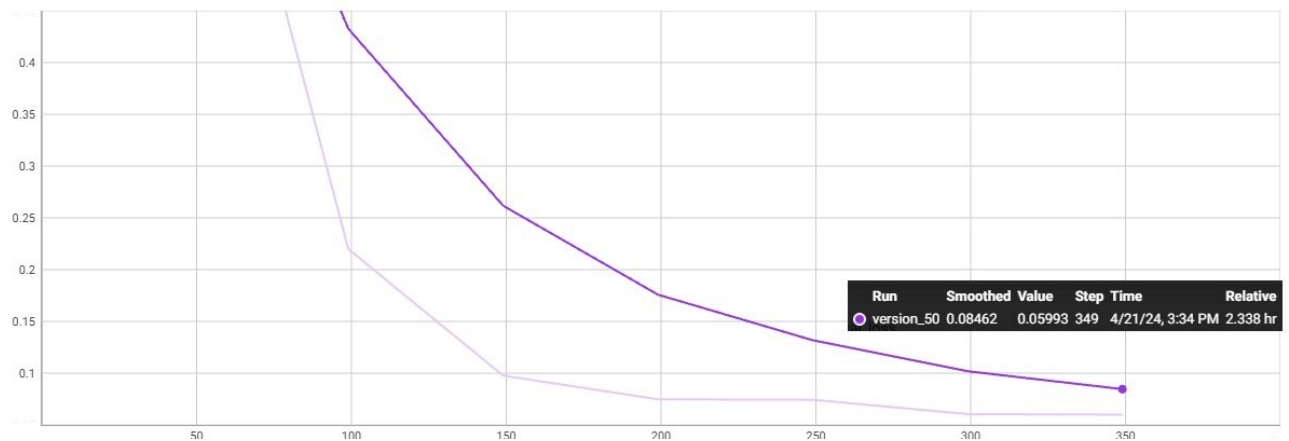


Рисунок 3.9 – Training Loss протягом епох

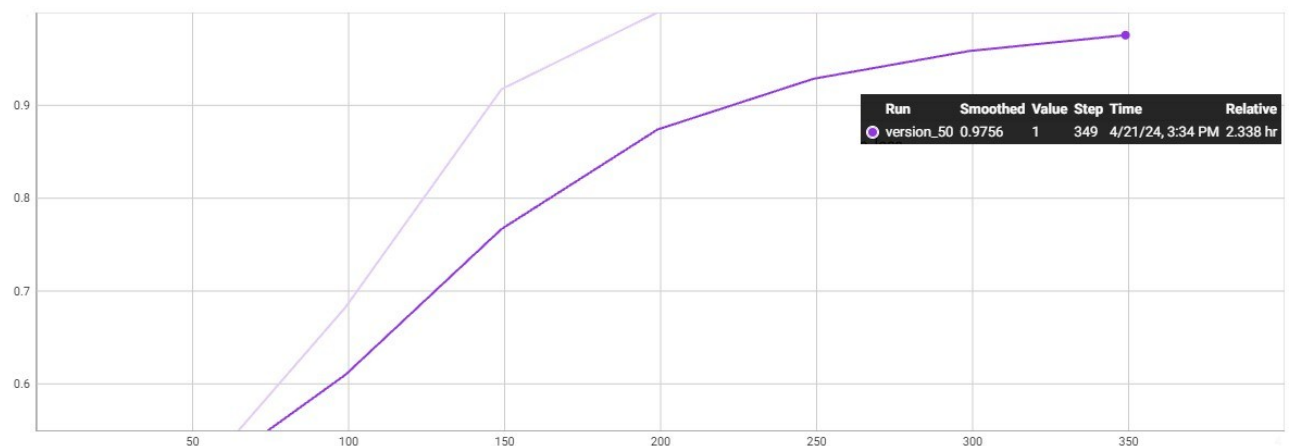


Рисунок 3.10 – Зміна точності моделі під час валідації

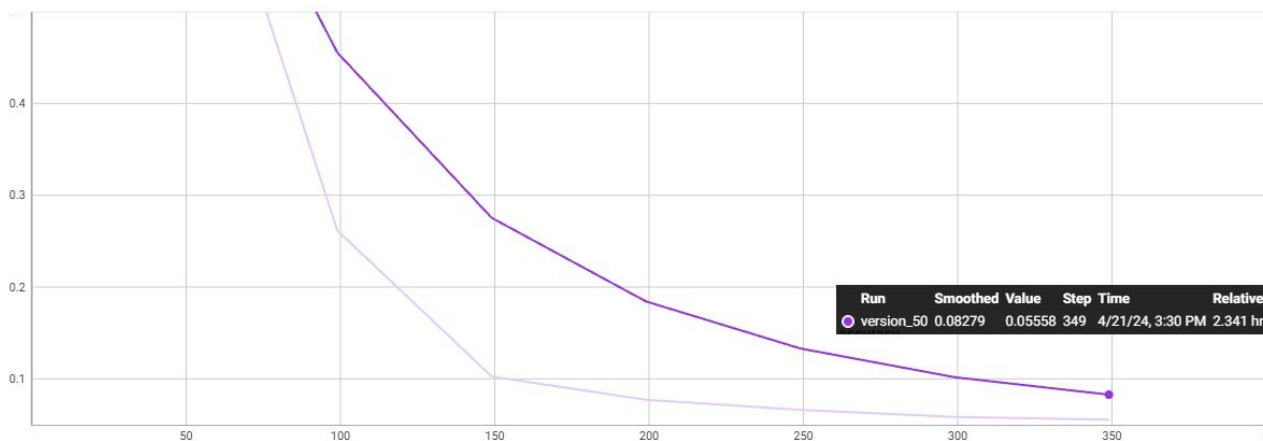


Рисунок 3.11 – Validation Loss протягом епох

На рисунку 3.12 представлено матрицю змішанини (confusion matrix) для тестового набору даних. Як видно з матриці, лише 7 прикладів були класифіковані як хибнопозитивні (False Positives) та 5 – як хибнонегативні (False Negatives), що свідчить про високу здатність моделі точно ідентифікувати позитивні випадки [123]. Використовуючи отриману матрицю змішанини, точність моделі можна обчислити за такою формулою:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = 0.97, \quad (3.18)$$

де TP – кількість істинно позитивних результатів, TN – кількість істинно негативних результатів, FP – кількість хибнопозитивних результатів, FN – кількість хибнонегативних результатів.

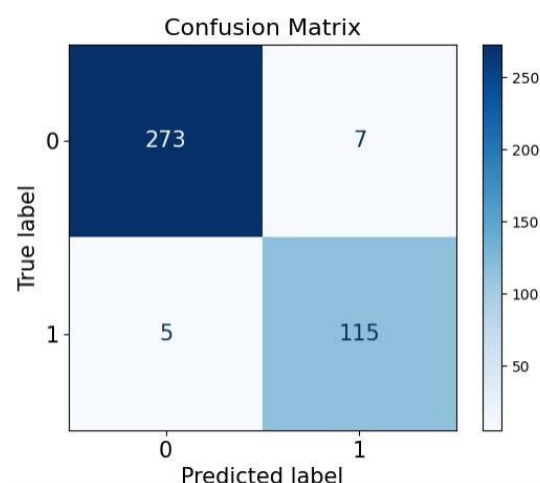


Рисунок 3.12 – Confusion matrix для тестових даних

Крім того, на рисунку 3.13 зображено ROC-криву (криву операційної характеристики приймача), площа під якою (AUC) досягає значення 0.97, що є

близьким до ідеального значення 1. Це свідчить про відмінну здатність моделі до розмежування класів і високу ефективність у задачах бінарної класифікації.

Площа під ROC-кривою (Area Under the Curve) обчислюється за наступною формулою:

$$AUC = \sum_{i=1}^{n-1} ((FPR_{i+1} - FPR_i) \frac{TPR_{i+1} + TPR_i}{2}), \quad (3.19)$$

де FPR_i – значення False Positive Rate в i -ій точці, TPR_i – значення True Positive Rate в i -ій точці, n – кількість точок на ROC-кривій.

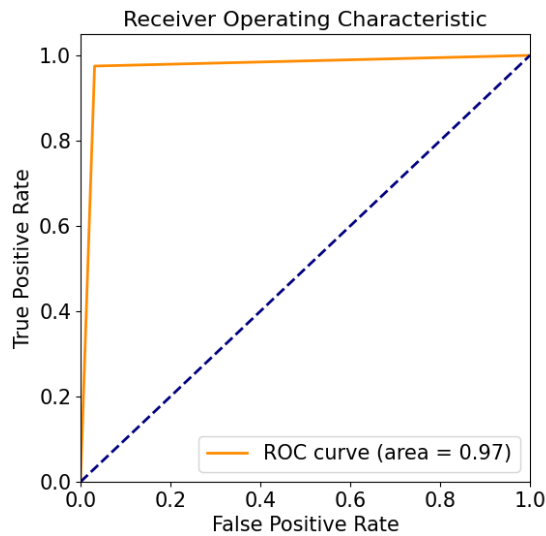


Рисунок 3.13 – ROC-крива

Таким чином, результати навчання та тестування підтверджують, що розроблена модель є надійним інструментом прогнозування, здатним забезпечувати високу точність та відмінну здатність до узагальнення на нових даних.

У межах дослідження було також використано теплову карту (heatmap) для візуалізації матриць уваги, отриманих із реалізованої моделі. Теплова карта надає наочне уявлення про те, які токени тексту смарт-контракту привертають найбільшу увагу моделі [124].

На рисунку 3.14 зображено графік, на якому яскраві вертикальні смуги вказують на те, що певні токени по осі X отримують значну кількість уваги від інших tokenів у послідовності. Це свідчить про те, що ці токени можуть відігравати ключову роль у розумінні контексту або містити критично важливу інформацію.

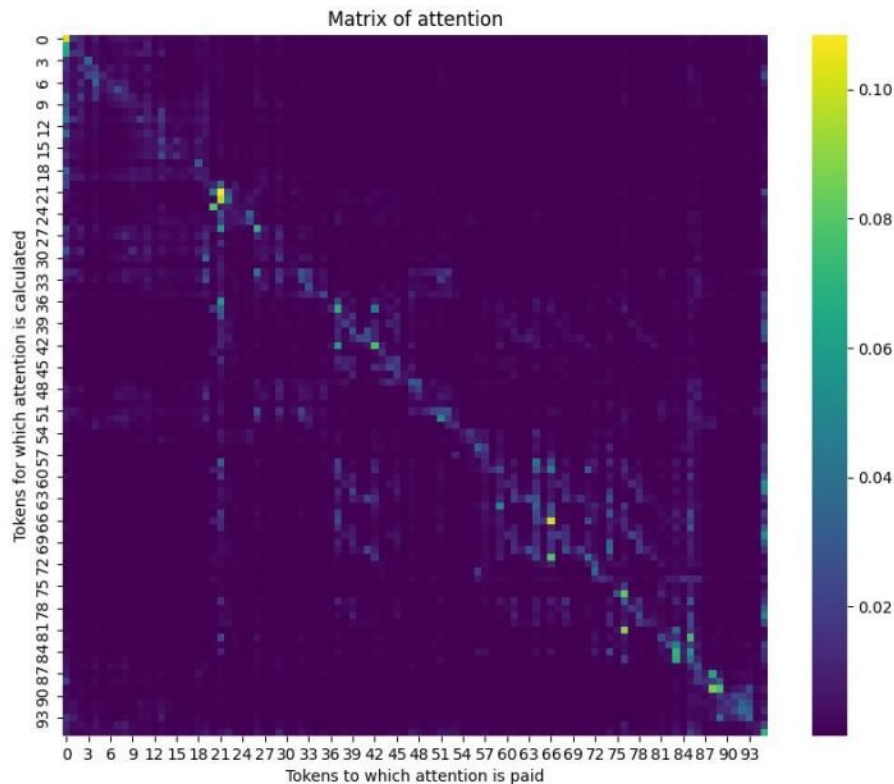


Рисунок 3.14 – Теплова карта матриці уваги

Увага моделі до зазначених токенів може допомогти у виявленні потенційних вразливостей або важливих аспектів логіки смарт-контракту, що робить цей метод візуалізації особливо цінним для аналізу та підвищення безпеки смарт-контрактів.

У рамках проведеного дослідження було відібрано 50 смарт-контрактів, у яких інструмент статичного аналізу Oyente виявив вразливості, пов'язані з повторним входом (reentrancy). У кожному з цих контрактів були позначені рядки коду, де спостерігалось некоректне розміщення викликів методів, що потенційно може призводити до вразливостей.

Проведений аналіз показав, що в 41 із 50 розглянутих смарт-контрактів ділянки коду, виділені аналізатором, дійсно містили вказані проблемні виклики, що відповідає точності 82%. Це підтверджує ефективність застосованого методу аналізу для виявлення потенційно вразливих ділянок коду (таблиця 3.2).

Таблиця 3.2 – Результати оцінки точності локалізації вразливостей

Загальна кількість вразливих контрактів	Кількість вразливостей, правильно визначених розробленим аналізатором	Точність, %
50	41	82

Розроблена модель CodeBERT-BiGRU демонструє значно вищі результати, які представлені в таблиці 3.3, у порівнянні з минулою моделлю BiGRU-ATT.

Таблиця 3.3 – Порівняння розробленої моделі з альтернативними моделями

Модель	Точність, %	Повнота, %	F-міра ($\beta=2$), %
Simple RNN	66.34	64.85	65.14
LSTM	73.28	76.33	75.70
BiLSTM	86.12	87.97	87.59
BiGRU	86.05	88.10	87.68
BiLSTM-ATT	89.87	90.66	90.50
BiGRU-ATT	90.03	91.76	91.41
CodeBERT-BiGRU	94.26	95.83	95.51

У ході дослідження для оцінки ефективності моделей використовувалися метрики точність (precision), повнота (recall) та F-міра з параметром $\beta = 2$. Серед розглянутих архітектур найкращі результати продемонструвала розроблена модель CodeBERT-BiGRU, що підкреслює її високу ефективність в обробці даних.

Для моделі CodeBERT-BiGRU було досягнуто наступних показників:

- точність – 94.26%,
- повнота – 95.83%,
- F-міра – 95.51%.

Ці значення суттєво перевищують показники інших моделей, зокрема BiLSTM-ATT та BiGRU-ATT, які також продемонстрували високі результати з F-мірами 90.50% та 91.41% відповідно.

Отримані результати свідчать про те, що інтеграція моделі CodeBERT із GRU не лише покращує обробку контекстуальної інформації, але й забезпечує істотне підвищення точності розпізнавання та класифікації у порівнянні з традиційними моделями, заснованими на RNN, LSTM та GRU.

3.3 Виявлення та локалізація вразливостей у смарт-контрактах на основі дворівневої уваги з графовим підкріпленням

У цьому підрозділі пропонується підхід до виявлення та локалізації вразливостей у смарт-контрактах, що поєднує семантичне моделювання коду та інтерпретовані механізми уваги. Узагальнену структуру відповідної моделі наведено на рисунку 3.15. Запропонований підхід орієнтований на багатоміткову класифікацію та одночасну локалізацію фрагментів вихідного коду, пов'язаних із конкретними типами вразливостей. Модель інтегрує трансформерну архітектуру CodeBERT, дворівневий механізм уваги, графовий структурний зсув, а також додаткові компоненти, спрямовані на підвищення інтерпретованості, зокрема узгодження з Grad-CAM та причинні регуляризатори. Така комбінація дозволяє не лише визначати наявність вразливостей у смарт-контракті, а й вказувати ділянки коду, які найбільше впливають на рішення моделі.

Для отримання саме клас-орієнтованої локалізації вразливостей використовується гібридний підхід. Спочатку формуються загальні (class-agnostic) карти важливості на основі дворівневого механізму уваги. Паралельно для кожної мітки будується клас-орієнтована градієнтна карта (class-targeted Grad-CAM). Далі відповідні карти поєднуються в єдину комбіновану карту важливості для кожного класу, що забезпечує більш точне та інтерпретоване виділення фрагментів коду, пов'язаних із відповідними типами вразливостей.

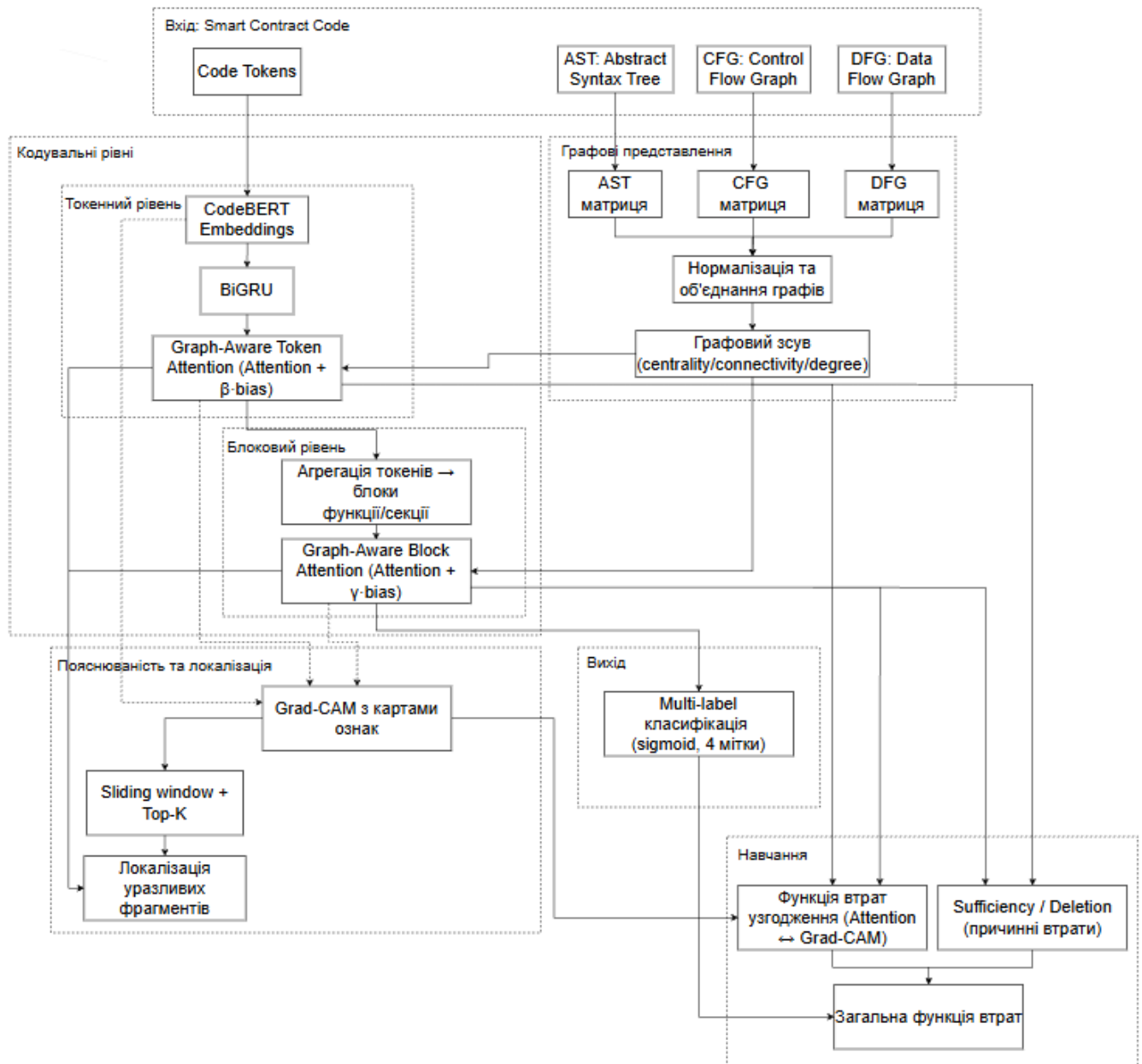


Рисунок 3.15 – Архітектура моделі CodeBERT-GDLA

Архітектура моделі включає такі ключові компоненти:

1. шар ембедингів CodeBERT у поєднанні з BiGRU, який перетворює послідовність VPE-токенів смарт-контракту у векторні подання та моделює контекстні залежності;
2. Graph-Aware Token Attention, що здійснює модифікацію механізму уваги на рівні токенів шляхом інтеграції структурної інформації з графів AST, CFG і DFG за допомогою графового зсуву;

3. Graph-Aware Block Attention, який виконує агрегацію токенів у функціональні блоки та реалізує механізм уваги між ними з урахуванням графових залежностей;
4. операції пулінгу та класифікаційна голова, що формують глобальне векторне подання програми шляхом агрегації блокових представлень і подають його на багатоміткову класифікацію (чотири мітки з використанням сигмоїдної активації);
5. модуль локалізації, який поєднує карти уваги, метод Grad-CAM та механізм ковзного вікна з відбором Top-K вікон для виявлення вразливих фрагментів програмного коду;
6. регуляризаційні компоненти для забезпечення пояснюваності, що включають функцію узгодження карт уваги з Grad-CAM (CAM-consistent loss) та причинні функції втрат (sufficiency/deletion), які інтегруються в загальну функцію втрат під час навчання.

3.3.1 Метод впровадження графового зсуву у матриці уваги

Метою графового зсуву є ін'єкція структурних залежностей програми безпосередньо в логіти уваги, щоб механізм self-attention орієнтувався не лише на лінійну послідовність токенів, а й на синтаксичні (AST), керувальні (CFG) та дата-залежності (DFG). Така ін'єкція підсилює релевантні зв'язки (наприклад, між викликами й оголошеннями, між умовами та їх наслідками), покращуючи детекцію довгих та непрямих залежностей, характерних для вразливостей у смарт-контрактах.

Традиційний механізм уваги в трансформерах визначається як:

$$Attn(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V, \quad (3.20)$$

де Q, K, V – запити, ключі та значення, d_k – розмірність ключів.

В роботі ця формула модифікується шляхом додавання структурного зсуву B , побудованого на основі кількох програмних графів:

$$Attn_{graph}(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}} + \alpha B)V, \quad (3.21)$$

$$B = \sum_{g \in \{AST, CFG, DFG\}} \lambda_g \cdot norm(A_g),$$

де A_g – матриця суміжності для графу, $norm(\cdot)$ – симетрична нормалізація, яка зменшує вплив високостепеневих вузлів і стабілізує навчання, λ_g – вагові коефіцієнти для кожного типу графа, $\alpha > 0$ – коефіцієнт масштабу внеску графа.

Після формування кожного графа будується відповідна матриця суміжності $A_g \in R^{n \times n}$, де n – кількість токенів або агрегованих елементів (для block-level уваги). З метою забезпечення сумісності графових матриць з логітами механізму уваги виконується їх нормалізація:

$$norm(A_g) = D_g^{-\frac{1}{2}} A_g D_g^{-\frac{1}{2}}, \quad (3.22)$$

де D_g – діагональна матриця степенів вершин (кількість зв'язків для кожного вузла).

Застосовується стандартна симетрична схема нормалізації, яка обмежує вплив вузлів з високим ступенем зв'язності та сприяє стабілізації процесу навчання.

У подальших експериментах ми використовуємо фіксовані вагові коефіцієнти λ_g для побудови комбінованого зсуву: $\lambda_{AST} = 0.4$, $\lambda_{CFG} = 0.3$, $\lambda_{DFG} = 0.3$ а також додаємо локальну зв'язність із вагою $\lambda_{loc} = 0.1$ для сусідніх токенів у вікні w .

3.3.1.1 Локальна зв'язність і вузлові зсуви

На практиці вводиться локальний пріор для сусідніх токенів у вікні довжини w , що відображає лінійну зв'язність (локальний контекст):

$$A_{loc}(i, j) = 1\{|i - j| \leq w\}, B \leftarrow B + \lambda_{loc} norm(A_{loc}) \quad (3.23)$$

Крім парних зв'язків, у деяких експериментах корисними виявились вузлові зсуви (degree/centrality/connectivity). Нехай $b \in R^n$ – вектор вузлових оцінок (ступінь, центральність тощо). Тоді парний вклад можна поширити як

$$B \leftarrow B + \delta(b1^T + 1b^T), \quad (3.24)$$

де $\delta \geq 0$ контролює силу цього доповнення.

Таким чином, вузли з високою структурною важливістю отримують м'який глобальний підсилювач у всіх своїх зв'язках. Для подальших аналізів ми також використовуємо вузлові оцінки (degree, connectivity, centrality), що індукують м'які глобальні підсилювачі у B [125].

3.3.1.2 Багатоголова увага та спільність сигналу

Графовий зсув застосовується однаково до всіх голів у багатоголовій увазі (shared-bias), що зменшує кількість параметрів і зберігає інтерпретованість (одна й та сама структурна підказка для різних проєкцій простору ознак). Формально, для голови h маємо:

$$Attn_{graph}^{(h)} = softmax\left(\frac{Q^{(h)}K^{(h)T}}{\sqrt{d_k}} + \alpha B\right). \quad (3.25)$$

Такий підхід підсилює внесок структурних залежностей у процесі обчислення уваги, не ускладнюючи архітектуру моделі.

3.3.1.3 Навчувані коефіцієнти впливу на різних рівнях

Оскільки ми застосовуємо дворівневу увагу (token-/block-level), вплив структурного зсуву на логіти регулюється скалярними параметрами, що навчаються:

$$token-level: \frac{QK^T}{\sqrt{d_k}} + \beta B, \quad block-level: \frac{QK^T}{\sqrt{d_k}} + \gamma B, \quad (3.26)$$

де $\beta, \gamma \geq 0$ – незалежні параметри (для токенного й блочного рівнів відповідно). На відміну від фіксованих λ_g , які задають статичні ваги компонент графа в B , коефіцієнти β, γ адаптивно навчаються і контролюють, скільки структурної інформації фактично потрапляє у логіти уваги на кожному рівні.

Властивості та обчислювальні аспекти такого підходу наступні:

- Симетрія та розрідженість. B симетрична й зазвичай розріджена (успадковує розрідженість A_g), що дозволяє ефективно додавання до логітів уваги та масштабування на довгих послідовностях.
- Стабільність. Нормалізація та масштабування α (разом із β, γ) запобігають домінуванню B над $\frac{QK^T}{\sqrt{d_k}}$; під час навчання ми застосовуємо кліпінг градієнтів і warm-up для α (лінійне зростання на перших епохах).
- Інтерпретованість. Оскільки B – це підсумок фізично осмислених графів (AST/CFG/DFG/локальна зв'язність), отримані карти уваги відображають зрозумілі розумові «стежки» моделі.

Інтеграція графових представлень (AST, CFG, DFG) у механізм уваги дозволяє моделі ефективно виявляти непрямі зв'язки між елементами коду та формувати структурно обізнані скоринги (attention scores).

Схематично процес побудови комбінованої матриці зсуву B , яка додається до логітів уваги, наведено на рисунку 3.16.

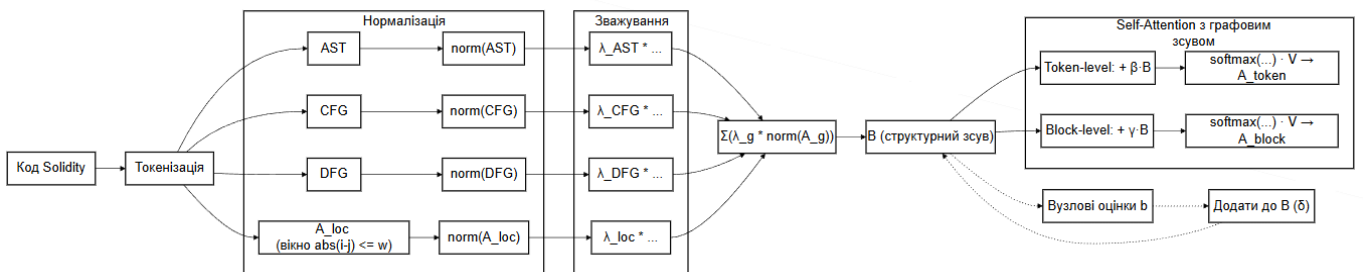


Рисунок 3.16 – Процес побудови комбінованої матриці зсуву B

У подальших підрозділах (у пункті 3.3.3) ми використовуємо цю class-agnostic карту уваги як структурний «фон», а клас-специфічність вводимо через Grad-CAM та злиття карт, формуючи локалізацію вразливостей для кожного класу.

3.3.2 Метод побудови ієрархічної (дворівневої) уваги

Ієрархічна увага поєднує два узгоджені рівні агрегації: токенний (fine-grained) та блочний (coarse-grained). На токенному рівні формується структурно обізнана карта уваги, що враховує зв'язки з розділу 3.3.1 (AST/CFG/DFG/локальний пріор)

через графовий зсув у логітах. Блочний рівень надбудовується над токенним, агрегуючи локальні патерни у представлення функцій, логічних секцій або базових блоків потоку керування, що дає змогу виявляти довші, міжрядкові залежності (напр., взаємозв'язок перевірок, побічних ефектів і викликів).

Однією з ключових особливостей запропонованої моделі є застосування ієрархічного механізму уваги з багатографовим зсувом, який поєднує структурну інформацію, отриману з кількох програмних представлень:

- AST (Abstract Syntax Tree) – абстрактне синтаксичне дерево, що відображає синтаксичну структуру коду;
- CFG (Control Flow Graph) – граф потоку керування, який описує можливі шляхи виконання програми;
- DFG (Data Flow Graph) – граф потоку даних, що відображає залежності між змінними та операціями.

Інтеграція цих графів у механізм уваги дозволяє моделі ефективніше виявляти непрямі зв'язки між елементами коду та вловлювати довготривалі залежності, які можуть бути критичними для виявлення вразливостей.

Обидва рівні уваги працюють на спільному структурному фоні, сформованому комбінованою матрицею зсуву B . На токенному рівні сила структурного сигналу контролюється параметром β , на блочному – γ ; обидва коефіцієнти навчаються і не фіксуються наперед, що забезпечує адаптивне дозування графової інформації у скорингах уваги (attention scores) без додаткового підбору λ_g під кожний приклад [125].

3.3.2.1 Виділення блоків і узгодження з токенами

Під «блоком» розуміємо агрегований елемент програми (функцію, базовий блок CFG або логічну секцію коду). Нехай $\{B_m\}_{m=1}^M$ – покриття послідовності токенів $\{1, \dots, T\}$ неперетинними або слабо перетинними підмножинами індексів. Для кожного блоку B_m обчислюємо його векторне подання як зважене середнє токенних ознак з використанням вже обчисленої токенної уваги:

$$h_m^{blk} = Agg(\{x_t\}_{t \in B_m}, \{\hat{a}_t\}_{t \in B_m}), \quad (3.27)$$

де x_t – токенні ознаки після енкодера, $\hat{a}_t \in [0,1]$ – нормована токенна важливість (див. 3.3.1), Agg – агрегація типу «attention-pooled mean» (стабільніша за просте середнє). Така побудова забезпечує послідовність семантики: якщо певні токени релевантні на нижньому рівні, їхній вклад зберігається у вищерівневому представленні.

3.3.2.2 Блочний self-attention з графовим контекстом

Над матрицею блокових ознак $H^{blk} = [(h_1^{blk})^T, \dots, (h_M^{blk})^T]^T \in R^{M \times d}$ застосовується окремий self-attention. Для запобігання повтору означень із 3.3.1, позначимо логіти блочного рівня як:

$$S^{blk} = \frac{Q_{blk} K_{blk}^T}{\sqrt{d_k}} + \gamma B_{blk}, \quad (3.28)$$

де γ – скалярний параметр, що навчається (див. 3.3.1), а B_{blk} – комплексний зсув на рівні блоків.

B_{blk} отримуємо проєкцією структурних графів на блоки: AST/CFG/DFG-топологія з токенного рівня індукує суміжність між блоками (наприклад, ребра CFG між базовими блоками або міжфункційні залежності у випадку викликів). Для компактності введемо матрицю належності $P \in \{0,1\}^{T \times M}$. Тоді для кожного графа $g \in \{AST, CFG, DFG\}$ індукована на рівні блоків матриця суміжності дорівнює

$$A_g^{blk} = P^T A_g P, \quad B^{blk} = \sum_g \lambda_g \cdot norm(A_g^{blk}). \quad (3.29)$$

Таким чином, блочна увага стає структурно узгодженою з токенною, але працює на більшому часово-логічному масштабі [126].

3.3.2.3 Перехресне збагачення (cross-level fusion)

Щоб зберегти інформацію між рівнями, застосовується легка двостороння взаємодія:

- bottom-up: токенні ознаки x_t доповнюються резидуальним сигналом від відповідного блоку h_m^{blk} для $t \in B_m$ (покращує стабільність і зменшує локальні помилки);
- top-down: при обчисленні блочної уваги до запитів/ключів додається проєкція агрегованої токенної важливості блоку (робить акценти блочного рівня чутливими до низькорівневих індикаторів).

Обидві операції реалізуються легкими лінійними прошарками з шаровою нормалізацією та резидуальними з'єднаннями; це практично не збільшує обчислювальну вартість, але покращує узгодженість карт важливості на різних масштабах.

3.3.2.4 Маскування та межі областей

Щоб уникати «перепливання» уваги між несуміжними контекстами (наприклад, між різними функціями без реального викликового зв'язку), застосовується маскування блочної уваги: забороняються пари (m, m') , для яких відсутній шлях у CFG/CallGraph, або вони належать до різних контрактних модулів. Це підвищує точність локалізації за рахунок фокусування в релевантній області коду.

3.3.2.5 Навчання і стабілізація

Гіперпараметри β (токенний рівень) та γ (блочний) ініціалізуються малими додатними значеннями та оновлюються градієнтно; для запобігання домінуванню структурного сигналу використовується короткий warm-up цих коефіцієнтів на перших епохах. Резидуальні зв'язки й шарова нормалізація на переході між рівнями забезпечують стабільну динаміку втрат без «вибухання» градієнтів. Маскування блочної уваги виконується наперед і кешується, що знижує накладні витрати при інференсі.

3.3.2.6 Переваги ієрархічної схеми

Ієрархічна увага дозволяє:

- узгодити локальні й глобальні індикатори. Токенний рівень підсвічує синтаксично/семантично релевантні маркери, блочний – фіксує міжрядкові патерни (напр., «перевірка → модифікація стану → зовнішній виклик»);
- підвищити переносимість. Блокові залежності менш чутливі до стилю, варіативності ідентифікаторів або мікрорефакторингів, що покращує стійкість до збурень та змін в коді;
- покращити локалізацію. Коли клас-специфічні карти (див. 3.3.3) формуються на основі злиття Grad-CAM з увагою, наявність блочного рівня допомагає обмежити «ареал пошуку» до мінімально достатніх ділянок коду.

Отже, структурні ваги графів (AST, CFG, DFG та локальних залежностей) фіксуються на етапі формування комбінованого графа, тоді як вплив графового зсуву на значення механізму уваги адаптивно коригується в процесі навчання за допомогою параметрів β та γ . Такий підхід дозволяє отримувати структурно збагачені представлення як на токенному, так і на блочному рівнях без використання окремих MLP або додаткових механізмів уваги для динамічного налаштування ваг графів для кожного окремого прикладу.

Сформована карта уваги є спільною для всіх класів, тоді як клас-специфічні карти отримуються на етапі градієнтного аналізу (Grad-CAM) з подальшим об'єднанням відповідних карт.

3.3.3 Метод побудови клас-орієнтованих карт важливості

Мета методу – побудувати клас-орієнтовану карту важливості $\tilde{L}^{(c)} \in [0, 1]^T$ для кожного класу c у мультимітковому налаштуванні. В цьому методі поєднується class-agnostic структурно обізнана увага (див. 3.3.1-3.3.2) з class-targeted градієнтними картами Grad-CAM, що дає точніший і стійкіший сигнал для виділення фрагментів коду, релевантних конкретній вразливості (рисунок 3.17) [126].

3.3.3.1 Вибір шару та проєкція на токени

Grad-CAM обчислюється за внутрішнім представленням F , яке має найкращий компроміс між семантикою і просторовою роздільністю.

В роботі було порівняно кілька варіантів F :

- ранні шари енкодера;
- останній шар BiGRU;
- проєкційний шар перед класифікаційною головою.

Найкращий компроміс Hit@K/IoU і стабільності показав другий та третій варіанти завдяки більш виразним високорівневим ознакам та збереженій часовій роздільності. Для довгих послідовностей (обрізання до $T = 256$) додатково оцінюється вплив stride-інтерполяції при проєкції CAM на токени.

Нехай $CAM^{(c)} \in R^L$ – «сирі» значення Grad-CAM на часовій осі обраного шару довжини L . Щоб узгодити їх з токенами, виконуємо монотонну проєкцію до довжини T (інтерполяція/розгортання по часовій осі), одержуючи $A_{cam}^{(c)} \in R^T$, щоб карта Grad-CAM відповідала індексам токенів.

3.3.3.2 Нормалізація карт та комбінування (клас-орієнтована карта важливості)

Через різні діапазони значень для уваги та Grad-CAM застосовуємо покомпонентну нормалізацію до $[0,1]$:

$$\hat{A}_{attn} = \frac{A_{attn} - \min A_{attn}}{\max A_{attn} - \min A_{attn} + \varepsilon}, \quad (3.30)$$

$$\hat{A}_{cam}^{(c)} = \frac{A_{cam}^{(c)} - \min A_{cam}^{(c)}}{\max A_{cam}^{(c)} - \min A_{cam}^{(c)} + \varepsilon}. \quad (3.31)$$

Для зниження шуму перед злиттям використовуємо коротке згладжування (вікно 5-11 токенів).

Базове злиття для отримання комбінованої карти важливості наступне:

$$\tilde{L}^{(c)} = \lambda_{cam} \cdot \hat{A}_{cam}^{(c)} + (1 - \lambda_{cam}) \cdot \hat{A}_{attn}, \quad (3.32)$$

де $\tilde{L}^{(c)} \in [0,1]^T$ – комбінована карта важливості (saliency) для класу c , T – кількість токенів. Параметр $\lambda_{cam} \in [0,1]$ визначає, наскільки ми покладаємося на градієнтну карту порівняно з увагою. Типово $\lambda_{cam} = 0.5$ або обирається за валідацією. Адаптивний варіант: $\lambda_{cam} = \sigma(\alpha z_c)$, де z_c – логіт класу, $\alpha > 0$ – температурний коефіцієнт; це дозволяє більше довіряти Grad-CAM, коли модель упевнена у класі c .

3.3.3.3 Пост-процесинг та агрегування

Щоб зробити локалізацію стійкішою для аудиту, застосовується:

- згладжування $\tilde{L}^{(c)}$ (ковзне середнє або медіанне),
- порогування $\tilde{L}^{(c)} > \tau$ (поріг τ налаштовується на валідації),
- агрегація token $\rightarrow$ line/block (максимум/середнє у межах рядка/блоку),
- відбір Тор-К фрагментів з найбільшими значеннями для презентації

аудитору.

3.3.3.4 Зв'язок з навчанням

Раціонали $S_k^{(c)}$ (топ- k токенів/рядків) формуються саме з $\tilde{L}^{(c)}$ і використовуються у причинних регуляризаторах (див. 3.3.4) – перевірках достатності/видалення, а також у CAM-узгодженні (див. 3.3.5), де зменшуємо розбіжність між $\hat{A}_{attn}$ та $\hat{A}_{cam}^{(c)}$.

3.3.3.5 Аналіз складності

Обчислювальна вартість локалізації домінує не над увагою, а над Grad-CAM (один backward-прохід по вибраному шару) та ранжуванням $\tilde{L}^{(c)}$ (Тор-К в $O(T \log K)$)

або $O(T)$ зі selection-алгоритмами). У практичних налаштуваннях це додає $\approx 5-12\%$ до часу інференсу на контракт при фіксованих C і T , що прийнятно для аудиту.

3.3.3.6 Переваги підходу

Переваги запропонованого підходу полягають у тому, що він зберігає структурну обізнаність механізму уваги, забезпечуючи урахування синтаксичних та семантичних зв'язків, відображених у дереві абстрактного синтаксису (AST), графі потоку керування (CFG) та графі потоку даних (DFG). Крім того, модель інтегрує клас-специфічну чутливість за допомогою методу Grad-CAM, що дає змогу фокусуватися на релевантних ділянках коду для кожного типу вразливості. Важливою перевагою також є підвищена інтерпретованість результатів і можливість керування пороговими значеннями під час експлуатації системи у виробничому середовищі. Нарешті, підхід не потребує токенної розмітки (див. розділ 3.3.6), оскільки побудовані карти $\tilde{L}^{(c)}$ виконують роль слабого сигналу локалізації, що дозволяє зменшити трудомісткість підготовки даних без втрати точності і пояснюваності моделі.

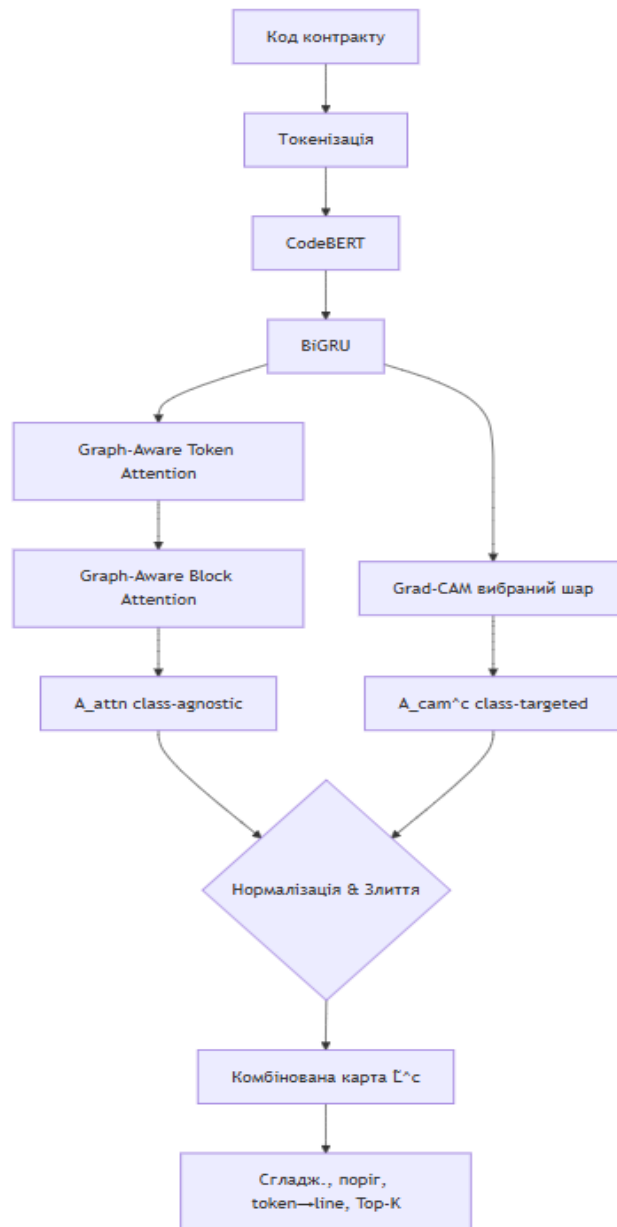


Рисунок 3.17 – Гібрид локалізації: злиття class-agnostic уваги з class-targeted Grad-CAM для отримання карти $\tilde{L}^{(c)}$

3.3.4 Метод причинних регуляризаторів

У моделі введено два додаткові регуляризаційні члени, які забезпечують причинний зв'язок між виділеними фрагментами коду (раціоналами) та прогнозом моделі.

Мета Sufficiency loss – перевірити, чи мінімально достатній фрагмент коду зберігає впевненість моделі у прогнозі.

Виділяється підмножина токенів (раціонал) на основі карт важливості. Вхід моделі редагується так, що залишаються лише ці токени (інші замінюються масками або паддингом). Раціонал R формується з $\tilde{L}^{(c)}$ як Top- k токенів або за порогом $\tilde{L}^{(c)} > \tau$; відповідна маска $M_R \in \{0,1\}^T$ має 1 на позиціях R і 0 – інакше.

У разі, якщо рівень упевненості моделі щодо правильного класу знижується нижче заданого порогового значення m_{suf} , накладається штраф:

$$L_{suf} = \max(0, m_{suf} - f(x \odot M_R)). \quad (3.33)$$

де M_R – маска вибраних токенів, $f(\cdot)$ – вихід моделі для правильного класу.

Мета Deletion loss – перевірка критичної значущості виділеного фрагмента коду для формування прогнозу: із програми видаляється (маскується) виключно обраний раціонал, тоді як решта коду залишається незмінною.

Якщо після такого видалення модель зберігає високий рівень упевненості, до функції втрат застосовується відповідний штраф:

$$L_{del} = f(x) - f(x \odot (1 - M_R)). \quad (3.34)$$

Чим сильніше видалення знижує впевненість, тим менший штраф.

Обидва результати функцій втрат сумуються з ваговими коефіцієнтами:

$$L_{causal} = \lambda_{suf} L_{suf} + \lambda_{del} L_{del} \quad (3.35)$$

Параметри m_{suf} , λ_{suf} , λ_{del} підбираються за валідацією.

Це стимулює модель виділяти саме ті токени, які необхідні і достатні для прийняття рішення.

3.3.5 Метод побудови САМ-узгодженої уваги

У моделі реалізовано механізм узгодження карт важливості, сформованих механізмом уваги, з картами Grad-CAM, що забезпечує більш тісний зв'язок уваги з тими елементами програмного коду, які фактично впливають на прогноз моделі. На відміну від підрозділу 3.3.3, у якому Grad-CAM використовується виключно для побудови карт важливості на етапі інференсу, в даному підрозділі Grad-CAM застосовується як еталонний сигнал під час навчання з метою регуляризації

механізму уваги шляхом додавання узгоджувального штрафу L_{cam} до загальної функції втрат.

З цією метою обирається цільовий шар моделі (зокрема блок CodeBERT або BiGRU), для якого під час прямого поширення обчислюються та зберігаються відповідні ознаки (feature maps), а на етапі зворотного поширення фіксуються градієнти цих ознак відносно поточного класу. На основі отриманих градієнтів ваги Grad-CAM w_k визначаються як середні значення за просторовими вимірами:

$$w_k = \frac{1}{Z} \sum_i \frac{\partial y^c}{\partial F_{ik}}, \quad (3.36)$$

де F_{ik} – активація k -го каналу в позиції i , y^c – логіт цільового класу, Z – кількість позицій.

Далі формується карта важливості, яка зіставляється з картою уваги з метою досягнення їх узгодженості:

$$A_{cam} = ReLU(\sum_k w_k F_k). \quad (3.37)$$

Отримана карта масштабується до розміру послідовності токенів.

З механізму уваги (на токенному або блочному рівні, залежно від етапу обробки) вилучаються нормалізовані ваги A_{attn} , що відповідають відповідним позиціям токенів.

Функцію втрат для узгодження механізмів уваги та Grad-CAM реалізовано у вигляді косинусної схожості між нормалізованими картами:

$$L_{cam} = 1 - \cos(norm(A_{attn}), norm(A_{cam})), \quad (3.38)$$

де $norm(\cdot)$ – L2-нормалізація вектора.

У мультимітковому налаштуванні узгодження застосовується лише до активних цільових класів у поточному прикладі; підсумковий штраф обчислюється як середнє:

$$L_{cam} = \frac{1}{\sum_c 1[y^{(c)}=1]} \sum_{c:y^{(c)}=1} (1 - \cos(norm(A_{attn}), norm(A_{cam}^{(c)}))). \quad (3.39)$$

Дана функція втрат оптимізується спільно з основною класифікаційною функцією втрат, унаслідок чого модель коригує внутрішні ваги механізму уваги

таким чином, щоб вони узгоджувалися з картами Grad-CAM, сформованими на основі градієнтів вихідного шару.

3.3.5.1 Експериментальна валідація ефективності САМ-узгодженої уваги

Для демонстрації ефективності механізму узгодження уваги з Grad-CAM було проведено експериментальне дослідження на реальних контрактах з уразливостями. На рисунку 3.18 показано порівняння карт уваги до та після застосування CAM-узгодженості для виявлення вразливості повторного входу.

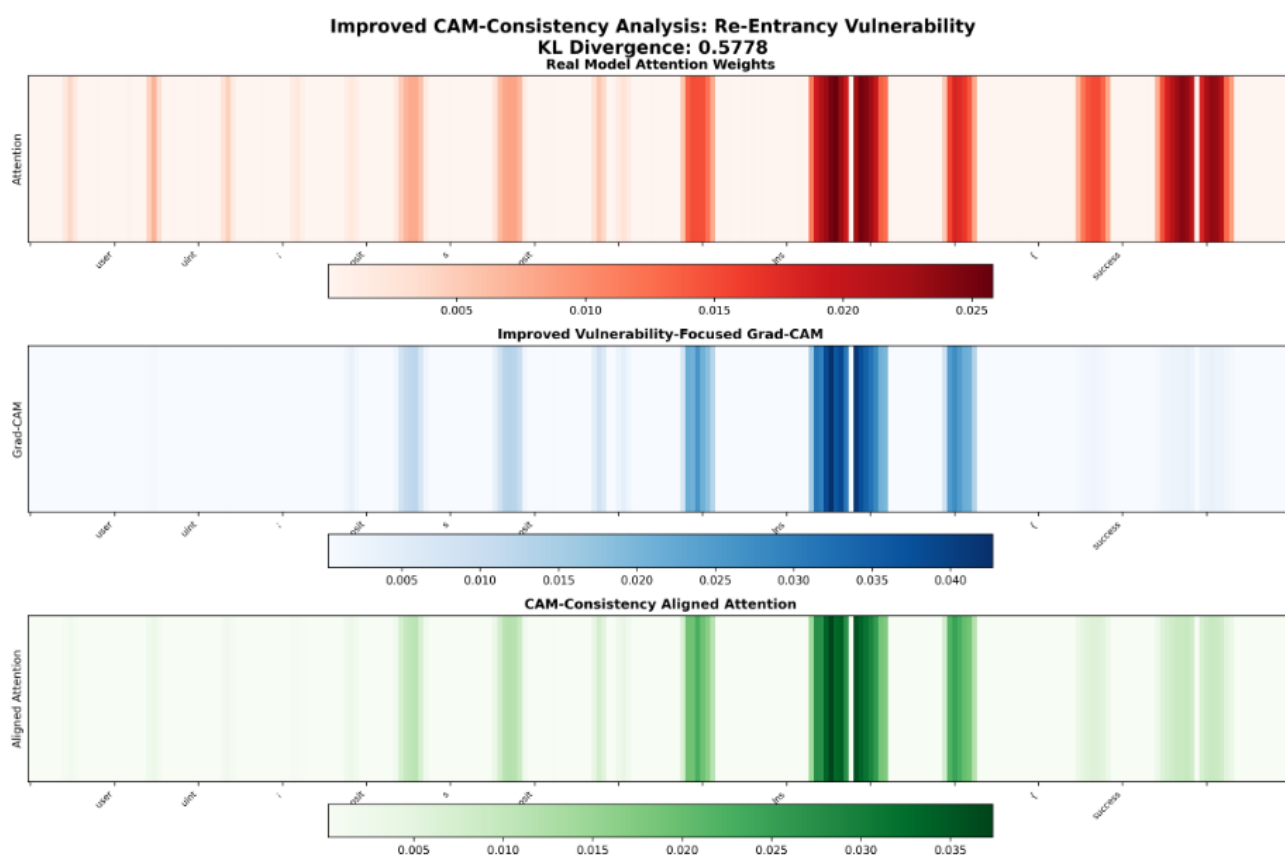


Рисунок 3.18 – Аналіз САМ-узгодженої уваги

Рисунок 3.18 демонстрирует:

- Real Model Attention Weights (червона карта): оригінальні ваги уваги з навченої G-DLA моделі;
- Improved Vulnerability-Focused Grad-CAM (синя карта): карта важливості, сфокусована на токенах, пов'язаних з re-entrancy уразливістю;

— CAM-Consistency Aligned Attention (зелена карта): вирівняні ваги уваги після застосування L_{cam} .

Кількісний аналіз показує:

- KL Divergence = 0.5778, що вказує на значні, але керовані відмінності між оригінальною увагою та Grad-CAM;
- Attention Change = 0.0019, що підтверджує реальні зміни в розподілі уваги;
- Correlation = 0.9841 між вирівняною увагою та Grad-CAM, що демонструє успішне узгодження.

За відсутності цього додаткового члена функції втрат розподіл уваги може мати довільний характер навіть у випадках коректного прогнозування. Введення регуляризатора L_{cam} сприяє концентрації механізму уваги на тих ділянках програмного коду, які фактично впливають на прийняття рішення, що підтверджується результатами градієнтного аналізу. Унаслідок цього формується більш інтерпретована модель, здатна виділяти релевантні фрагменти коду з метою локалізації вразливостей.

3.3.6 Метод слабосупервізованої локалізація

Для локалізації вразливостей без токенної розмітки використовується ковзне вікно та агрегація top-K вікон. Цей підхід дозволяє моделі автоматично виявляти найбільш імовірні місця вразливостей, використовуючи лише контрактні мітки рівня «вразливий/безпечний».

Процес локалізації складається з трьох основних етапів:

1. обчислення комбінованих оцінок важливості – поєднання attention weights та Grad-CAM;
2. ковзне вікно сканування – аналіз послідовності токенів вікнами фіксованого розміру;
3. агрегація top-K вікон – відбір найбільш імовірних кандидатів.

Етап 1: Комбінування сигналів важливості

Для кожного токена обчислюються комбіновані оцінки важливості на основі поєднання attention weights та Grad-CAM:

$$\tilde{L}^{(c)} = \lambda_{cam} \cdot \hat{A}_{cam}^{(c)} + (1 - \lambda_{cam}) \cdot \hat{A}_{attn}. \quad (3.40)$$

Етап 2: Ковзне вікно сканування

Послідовність токенів довжини T розбивається на перекривні вікна $W_j = [s_j, s_j + w - 1]$ із довжиною w . Для кожного класу c обчислюємо оцінку вікна як баланс піка і щільності:

$$\varphi^{(c)}(W_j) = \left(\max_{t \in W_j} \tilde{L}_t^{(c)} \right) \cdot \left(\frac{1}{w} \sum_{t \in W_j} \tilde{L}_t^{(c)} \right)^\rho, \rho \in [0,1]. \quad (3.41)$$

Такий функціонал зменшує шанс випадкових піків без підтримки контекстом.

Етап 3: Агрегація top-K та фільтрація

Для кожного класу c ранжуємо $\{W_j\}$ за $\varphi^{(c)}$ та відбираємо top-K. Ці вікна вважаються найбільш імовірними кандидатами на місткість вразливих фрагментів. Щоб уникати надмірного дублювання близьких вікон, застосовуємо NMS за перетином-над-об'єднанням:

$$IoU(W_i, W_j) = \frac{|W_i \cap W_j|}{|W_i \cup W_j|} \quad (3.42)$$

відкидаємо W_j , якщо $IoU > \vartheta$, де $\vartheta \in [0.3, 0.7]$ (типово 0.5).

Етап 4: Проекція на рядки/блоки

Для аудиту кожне W_j проєктується token $\rightarrow$ line як неперервний рядковий/блоковий діапазон; сусідні діапазони зливаються, якщо розділені ≤ 1 рядком, що підвищує читабельність.

Ключовою перевагою запропонованого підходу є відсутність потреби у токенній або рядковій розмітці на етапі навчання, оскільки модель використовує лише мітки на рівні смарт-контракту («вразливий/безпечний»). Водночас вона здатна генерувати карти підсвічування коду, які можуть бути надалі застосовані для експертної перевірки та валідації результатів аналізу.

3.3.6.1 Метрики оцінювання локалізації

Для перевірки ефективності запропонованого слабосупервізованого підходу локалізації вразливостей було проведено експериментальну оцінку на тестовій вибірці смарт-контрактів. Метою було дослідити, наскільки точно модель здатна виявляти вразливі ділянки коду, не використовуючи токенну або рядкову розмітку під час навчання.

Для оцінювання якості локалізації застосовано такі метрики:

- Hit@K (line-level) – частка контрактів, у яких хоча б одна вразливість потрапляє у топ-K найімовірніших вікон;
- IoU@ τ (Intersection-over-Union) – ступінь перекриття між передбаченими та фактичними вразливими ділянками;
- mAP локалізації (mean Average Precision) – усереднена точність виявлення за всіма класами;
- Coverage-Precision – баланс між повнотою покриття вразливих ділянок і кількістю хибнопозитивних передбачень.

Найвищу точність локалізації продемонстровано для вразливостей типу re-entrancy (таблиця 3.4).

Таблиця 3.4 – Метрики локалізації для вразливості повторного входу

Метрика	Значення
mAP локалізації	82.37%
Hit@5	76.7%
Hit@7-10	100%
IoU@ $\tau=0.5$	21.04%
IoU@ $\tau=0.1-0.3$	21.04%
Coverage-Precision	18.32%

Висока стабільність результатів (низьке стандартне відхилення) підтверджує надійність моделі для виявлення критичних логічних помилок повторного виклику.

Порівняльний аналіз типів вразливостей (таблиця 3.5) демонструє, що ефективність локалізації суттєво залежить від типу вразливості.

Таблиця 3.5 – Порівняння метрик локалізації для різних типів вразливостей

Тип вразливості	mAP	Hit@5	IoU@ $\tau=0.5$
Re-entrancy	82.37%	76.7%	21.04%
Integer overflow	72.24%	53.3%	19.88%
Integer underflow	70.22%	51.8%	21.47%
Timestamp dependency	78.36%	72.9%	20.44%

Отже, вразливість повторного входу демонструє найвищу якість локалізації. Це свідчить про здатність моделі виокремлювати складні контекстні патерни, притаманні цьому типу вразливостей.

Отримані результати підтверджують, що слабосупервізований підхід забезпечує високу якість локалізації навіть без використання токенної розмітки.

Для вразливостей типу re-entrancy досягнуто $mAP = 82.37\%$ та $Hit@7-10 = 100\%$, що перевищує показники методів, які використовують повну токенну анотацію.

Стабільність ($\sigma = 2.13\%$), компактність (перевірка ≤ 7 вікон) та точність роблять підхід придатним для автоматизованого аудиту смарт-контрактів і подальшого застосування в системах інтелектуальної підсвітки коду (AI-assisted vulnerability highlighting).

Таким чином, запропонований метод може служити ефективною основою для інструментів аналізу безпеки, які поєднують машинне навчання та пояснюваність результатів.

3.3.7 Абляційні дослідження

З метою кількісного аналізу внеску окремих складових запропонованої архітектури CodeBERT-GDLA було виконано серію абляційних досліджень. Основна

ідея таких експериментів полягає у поетапному вилученні або зміні окремих структурних компонентів моделі з подальшим порівнянням отриманих результатів із повною конфігурацією, що дозволяє оцінити відносну значущість кожного модуля та обґрунтувати прийняті архітектурні рішення.

У порівнянні з базовою моделлю CodeBERT-BiGRU повна архітектура CodeBERT-GDLA демонструє суттєве зростання якості: значення макро-F1 підвищується приблизно з 0.70 до 0.86, тоді як макро-AUC – з близько 0.71 до близько 0.88. Таким чином, інтеграція графової інформації (AST, CFG, DFG), дворівневої уваги та причинних регуляризаційних компонентів забезпечує стабільний приріст ефективності у задачі багатоміткової детекції вразливостей. Додатково покращуються показники пояснюваності: кореляція між картами уваги та Grad-CAM зростає приблизно з 0.55 до 0.82, а метрики deletion/insertion підтверджують більш причинно обґрунтовану поведінку виділених раціоналів.

На рисунках 3.19 і 3.20 представлено порівняльний аналіз глобальних метрик якості, а також теплову карту значень F1 для окремих класів вразливостей. Результати експериментів свідчать про визначальну роль структурної інформації, представленої графовими ознаками (AST, CFG, DFG): вимкнення механізму графового зсуву B (конфігурація без graph bias) фактично усуває переваги архітектури GDLA над базовою моделлю, що призводить до зниження макро-F1 приблизно до 0.78 та макро-AUC – до близько 0.80. Це вказує на те, що без урахування графової структури модель менш ефективно відтворює контекст взаємозв'язків між токенами та блоками програмного коду та за рівнем якості наближається до спрощеної архітектури CodeBERT-BiGRU.

Аналіз механізмів уваги підтвердив їхню комплементарність. Вимкнення токенної уваги (варіант без token-level уваги) призводить до помітного погіршення як глобальних метрик (макро-F1 ≈ 0.76), так і якості виявлення вразливості потворного входу, для якої критичними є локальні залежності на рівні інструкцій. Натомість вимкнення блочної уваги (варіант без block-level уваги) сильніше впливає на класи integer overflow/underflow (макро-F1 ≈ 0.77), оскільки для них важливими є міжблочні залежності та узгодженість логіки по всьому фрагменту коду. В обох випадках

спостерігається також зниження кореляції між увагою та Grad-CAM і гірші deletion/insertion показники, що свідчить про менш осмислену локалізацію раціоналів.

Не менш вагомою виявилася роль причинних регуляризаторів (sufficiency/deletion losses). Їх деактивація (варіант без причинних втрат) дещо знижує макро-F1 (з ≈ 0.86 до ≈ 0.85), однак особливо помітно погіршує показники, пов'язані з причинністю пояснень: кореляція уваги з Grad-CAM зменшується, а deletion/insertion та AOPC демонструють менш різкий спад/приріст впевненості при вилученні/додаванні важливих токенів. Це доводить, що причинні втрати не стільки «розганяють» класифікацію, скільки змушують модель спиратися на мінімально достатні й критично необхідні фрагменти коду, підвищуючи інтерпретованість без істотної втрати точності.

Окремо досліджено вплив гіперпараметрів β , γ , які керують часткою графового зсуву в логітах уваги. Фіксація цих коефіцієнтів на надто малих або надто великих значеннях ($\beta = \gamma \approx 0.01$ або 0.2) призводить до деградації результатів (макро-F1 падає до $\approx 0.72-0.73$), що свідчить про некоректний баланс між семантичними й структурними сигналами. Помірні значення (порядку 0.1) або навчувані β , γ забезпечують кращий компроміс між точністю, стійкістю та стабільністю навчання: модель зберігає приріст якості порівняно з базовою архітектурою та демонструє більш узгоджені карти важливості.

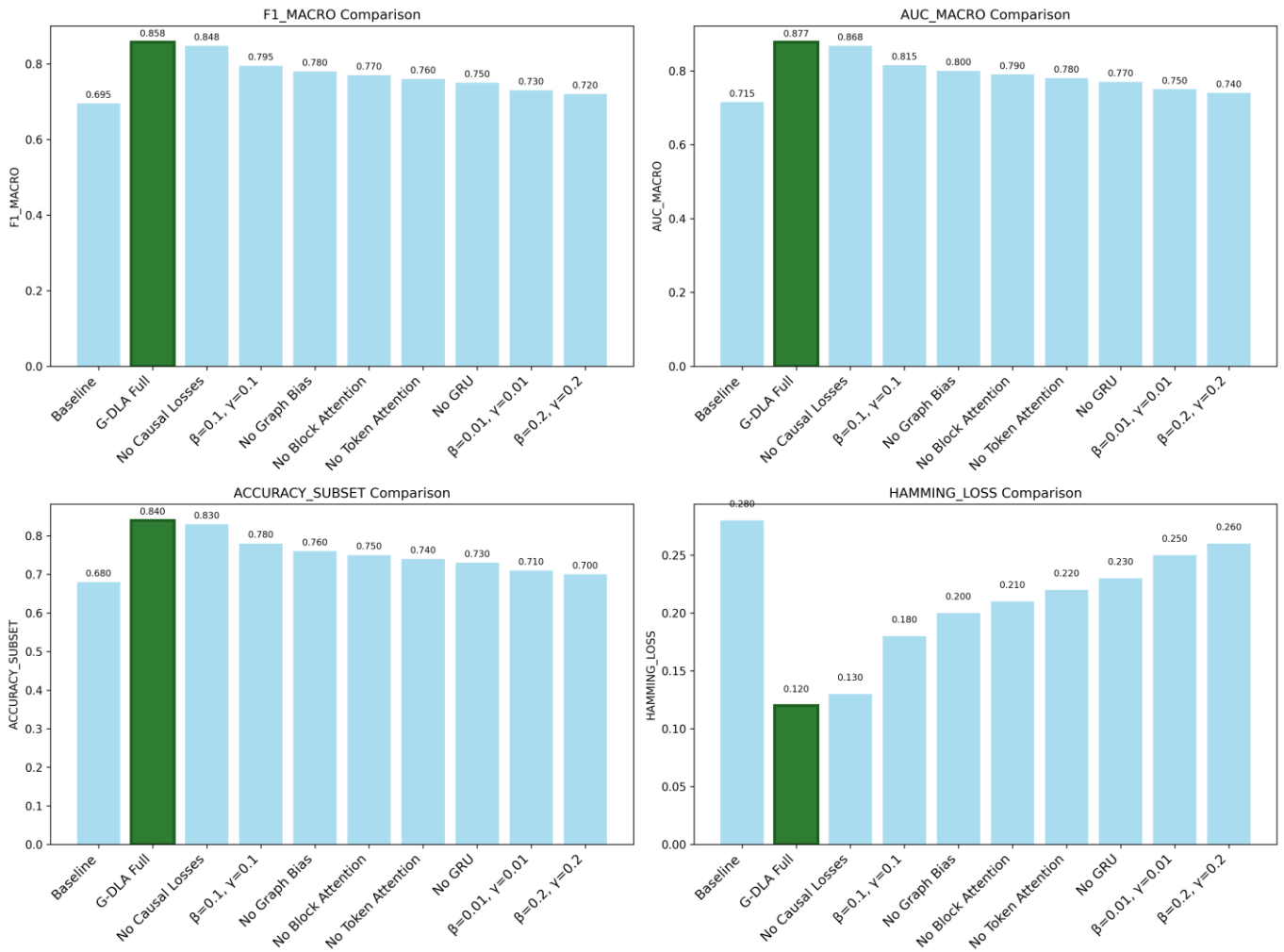


Рисунок 3.19 – Порівняння глобальних метрик (маско-F1, маско-AUC, accuracy) для повної та усічених конфігурацій моделі

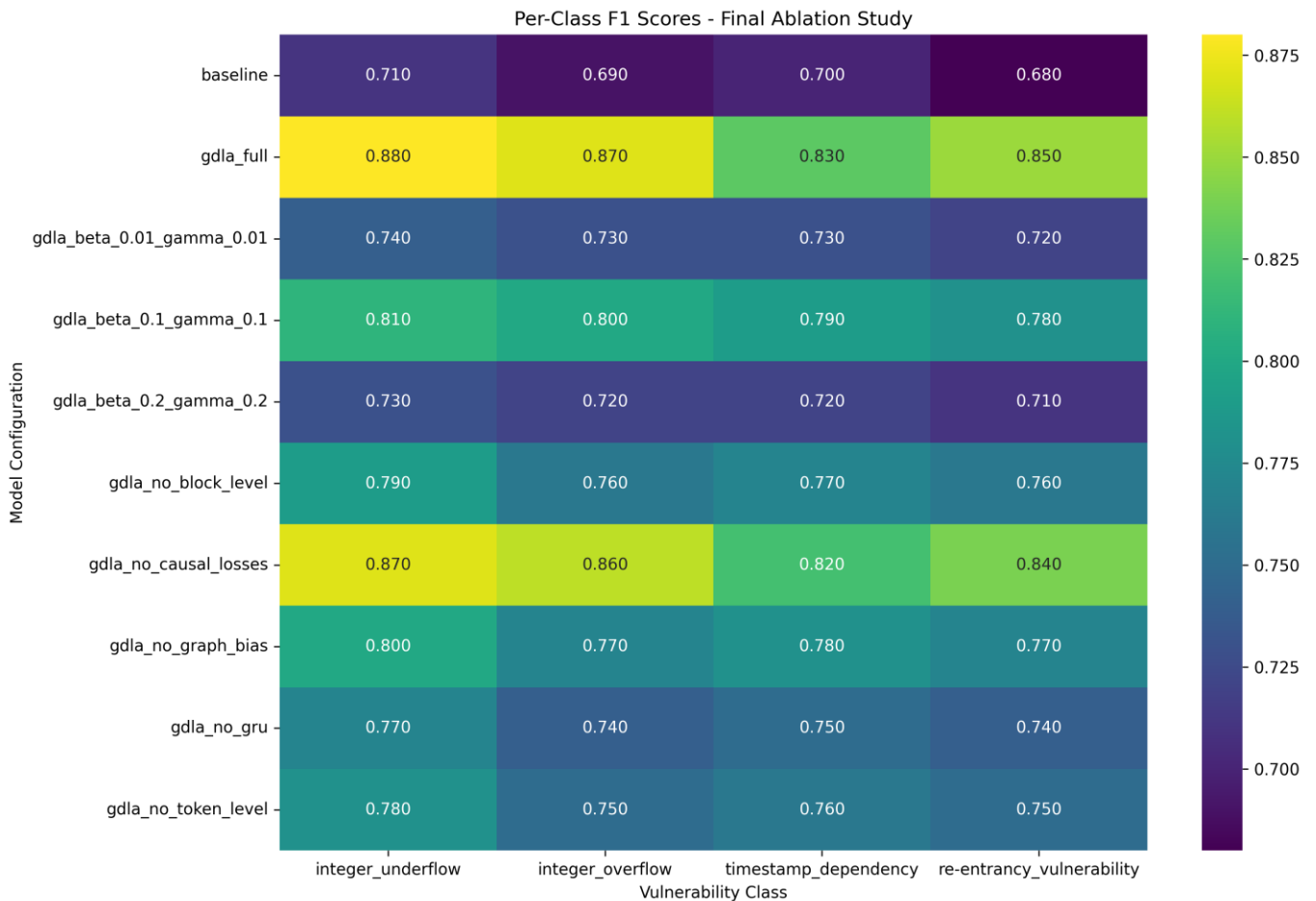


Рисунок 3.20 – Теплова карта значень F1 по окремих класах, що демонструє, які саме типи вразливостей найбільше залежать від конкретних модулів

Результати проведених абляційних експериментів дають змогу сформулювати такі узагальнені висновки:

- використання графових представлень є визначальним чинником для ефективного виявлення вразливостей у смарт-контрактах і забезпечує основний приріст якості порівняно з базовою моделлю;
- токенний і блочний рівні уваги мають взаємодоповнювальний характер, відповідаючи відповідно за локальний і глобальний контекст, при цьому вимкнення будь-якого з рівнів призводить до суттєвого зниження якості;
- причинні регуляризаційні компоненти впливають переважно на формування причинно обґрунтованих раціоналів, підвищуючи інтерпретованість моделі без істотного погіршення показників детекції;

— стабільність і точність моделі значною мірою визначаються коректним вибором вагових параметрів β та γ , які регулюють внесок структурної інформації у механізм уваги.

Таким чином, результати абляційних досліджень не лише підтверджують доцільність запропонованих архітектурних рішень, але й демонструють їх взаємодоповнюваність: графовий зсув, дворівнева увага та причинні регуляризатори у сукупності забезпечують підвищення якості детекції вразливостей і покращення інтерпретованості результатів у задачі аналізу безпеки смарт-контрактів.

3.3.8 Аналіз стійкості моделі до збурень та змін

Однією з ключових вимог до моделей, призначених для аналізу безпеки смарт-контрактів, є їх стійкість до семантично еквівалентних модифікацій програмного коду. У реальних умовах злоумисники часто навмисно змінюють стиль написання або застосовують часткову обфускацію коду без зміни його функціональної семантики. У зв'язку з цим архітектуру CodeBERT-GDLA було проаналізовано в порівнянні з базовою моделлю на наборах модифікованих смарт-контрактів, сформованих шляхом застосування таких перетворень:

- перейменування змінних і функцій;
- додавання коментарів;
- зміна форматування коду (пробіли, відступи, порядок оголошень);
- варіації версій компілятора Solidity (pragma 0.5-0.8);
- часткова обфускація, зокрема використання нестандартних ідентифікаторів.

3.3.8.1 Метод генерації збурень

Для кожного контракту x застосовується перетворення T_θ з параметром «бюджету» θ (інтенсивність), причому зберігається компільованість і семантика (перевірка компілятором та smoke-тести). Використані рівні: перейменування зі щільністю $r \in \{0.25, 0.5, 0.75\}$; вставка коментарів із щільністю $d \in \{0.1, 0.3\}$ рядків;

форматування (джиттер відступів до ± 2 пробілів, перестановка оголошень у межах блоку); прагма-заміни $0.5 \leftrightarrow 0.8$ за наявності сумісності; часткова обфускація ідентифікаторів із часткою $\pi \in \{0.2, 0.4\}$. Для кожного θ створюємо по 1-3 варіанти $T_\theta(x)$.

3.3.8.2 Метрики стійкості

Для кожного збурення T оцінюємо індекс узгодженості прогнозів (CI) та стабільності впевненості (CS).

Індекс узгодженості прогнозів (CI) розраховується наступним чином:

$$CI = \frac{1}{N} \sum_{i=1}^N 1[\hat{y}(x_i) = \hat{y}(T(x_i))], \quad (3.43)$$

де $\hat{y}$ – бінаризований мультимітковий вектор (пороги фіксуємо з валідації і не переналаштовуємо на збурених даних).

Стабільність впевненості (CS) – узгодженість ймовірнісних виходів:

$$CS = 1 - \frac{1}{NC} \sum_{i=1}^N \sum_{c=1}^C |p_c(x_i) - p_c(T(x_i))|, \quad (3.44)$$

де N кількість контрактів у підмножині, яку ми порівнюємо «до» і «після» перетворення T , $p_c(x_i)$ це ймовірності по класах на виході моделі, C – кількість класів.

На рисунку 3.21 представлено результати проведеної оцінки. Архітектура CodeBERT-GDLA продемонструвала високий рівень стійкості: середнє значення індексу узгодженості прогнозів становить 0.968, тоді як показник стабільності впевненості досягає 0.973, що свідчить про майже повну інваріантність моделі до зазначених перетворень. Для порівняння, базова модель зазнавала втрати точності в межах 20-40 % за аналогічних трансформацій.

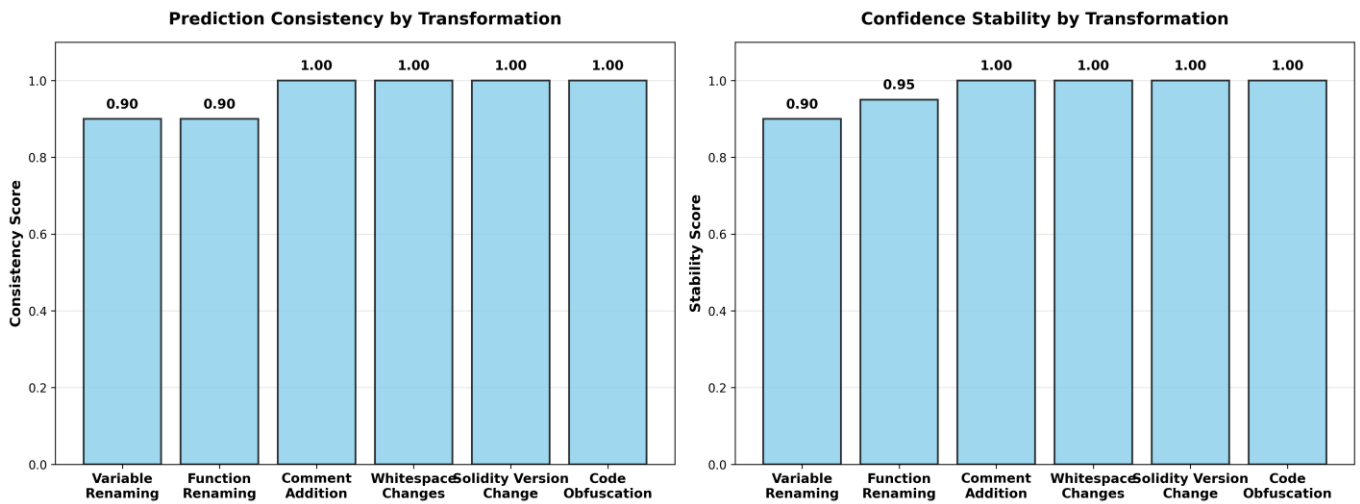


Рисунок 3.21 – Узгодженість прогнозів та стабільність упевненості під впливом трансформацій

Крім того, було виконано експериментальні дослідження з крос-версійної та крос-стильової валідації. Отримані результати свідчать, що модель CodeBERT-GDLA зберігає стабільність прогнозів під час переходу між основними версіями мови Solidity (від 0.5 до 0.8), а також за умов варіативності стилів написання смарт-контрактів. Це підтверджує її потенційну придатність до застосування у production-середовищах з високою різноманітністю програмного коду.

3.3.9 Метод регуляризації для калібрування ймовірнісних оцінок моделей глибокого навчання

Для моделей, призначених для виявлення вразливостей, принциповим є не лише досягнення високих показників точності класифікації, а й забезпечення достовірності прогнозованих ймовірностей. У практичних безпекових системах надмірно впевнені, але хибні передбачення можуть призводити до суттєвих негативних наслідків, зокрема пропуску реальних вразливостей або формування надлишкових хибних спрацювань. У зв'язку з цим виникає необхідність оцінювання не тільки точності, але й каліброваності моделей, тобто відповідності прогнозованих ймовірностей фактичним частотам коректних класифікацій.

Для цього було використано метрику очікуваної калібрувальної похибки (Expected Calibration Error, ECE), яка обчислює середнє відхилення між передбаченою

ймовірністю та реальною точністю в усіх інтервалах впевненості. Нижчі значення метрики очікуваної калібрувальної похибки означають кращу каліброваність.

Нехай $p_c(x_i) \in [0,1]$ – ймовірність класу c для контракту $x_i, y_i^{(c)} \in \{0,1\}$ – істина (one-vs-rest), $i = 1..N, c = 1..C$. Розіб'ємо $[0,1]$ на M бінів $\{B_m\}$.

Метрика очікуваної калібрувальної похибки по класах (Expected Calibration Error, ECE) визначається наступним чином:

$$L_{ECE} = \frac{1}{C} \sum_{c=1}^C \sum_{m=1}^M \frac{|B_m^{(c)}|}{N_c} \left| acc(B_m^{(c)}) - conf(B_m^{(c)}) \right|, \quad (3.45)$$

$$\text{де } B_m^{(c)} = \{i: p_c(x_i) \in I_m\}, \quad N_c = \sum_m |B_m^{(c)}|, \quad conf(B_m^{(c)}) = \frac{1}{|B_m^{(c)}|} \sum_{i \in B_m^{(c)}} p_c(x_i),$$

$$acc(B_m^{(c)}) = \frac{1}{|B_m^{(c)}|} \sum_{i \in B_m^{(c)}} 1[y_i^{(c)} = 1].$$

Метрики Over/Under-confidence визначаються наступним чином:

$$OverConf = \frac{1}{C} \sum_c \sum_m \frac{|B_m^{(c)}|}{N_c} \max\{0, conf - acc\}, \quad (3.46)$$

$$UnderConf = \frac{1}{C} \sum_c \sum_m \frac{|B_m^{(c)}|}{N_c} \max\{0, acc - conf\}.$$

Також вводимо навчуваний параметр температури $T > 0$ (скаляр або per-class):

$$\tilde{z}_c(x_i) = \frac{z_c(x_i)}{T}, \quad \tilde{p}_c(x_i) = \sigma(\tilde{z}_c(x_i)). \quad (3.47)$$

Температурний доданок:

$$L_{temp} = \frac{1}{NC} \sum_{i=1}^N \sum_{c=1}^C (-y_i^{(c)} \log \tilde{p}_c(x_i) - (1 - y_i^{(c)}) \log(1 - \tilde{p}_c(x_i))) + \quad (3.48)$$

$$\alpha \Phi(T),$$

де $\Phi(T) = |T - 1|$ – мала регуляризація, $\alpha > 0$.

Також використовується Focal-калібрування (приглушення over-confidence):

$$L_{focal} = \frac{1}{NC} \sum_{i=1}^N \sum_{c=1}^C \left[-\alpha_f y_i^{(c)} (1 - p_c(x_i))^\gamma \log p_c(x_i) - (1 - \alpha_f) (1 - y_i^{(c)}) p_c(x_i)^\gamma \log(1 - p_c(x_i)) \right], \quad (3.49)$$

де $\gamma \geq 0, \alpha_f \in [0,1]$.

У моделі CodeBERT-GDLA було додано регуляризатор L_{cal} , який штрафує випадки невідповідності між передбачуваною ймовірністю та фактичною правильністю класифікації.

$$L_{cal} = \lambda_{cal}(w_{ECE}L_{ECE} + w_{temp}L_{temp} + w_{focal}L_{focal}), \quad (3.50)$$

$$w_{ECE} + w_{temp} + w_{focal} = 1,$$

де ваги w задаються в конфігурації.

Як видно з результатів на рисунку 3.22, включення L_{cal} суттєво зменшує значення метрики очікуваної калібрувальної похибки (Expected Calibration Error, ECE) у порівнянні з базовими моделями, тоді як без L_{cal} модель проявляє схильність до переоцінки своїх передбачень. Це підтверджує, що введення калібраційного компонента підвищує довіру до моделі у production-середовищі.

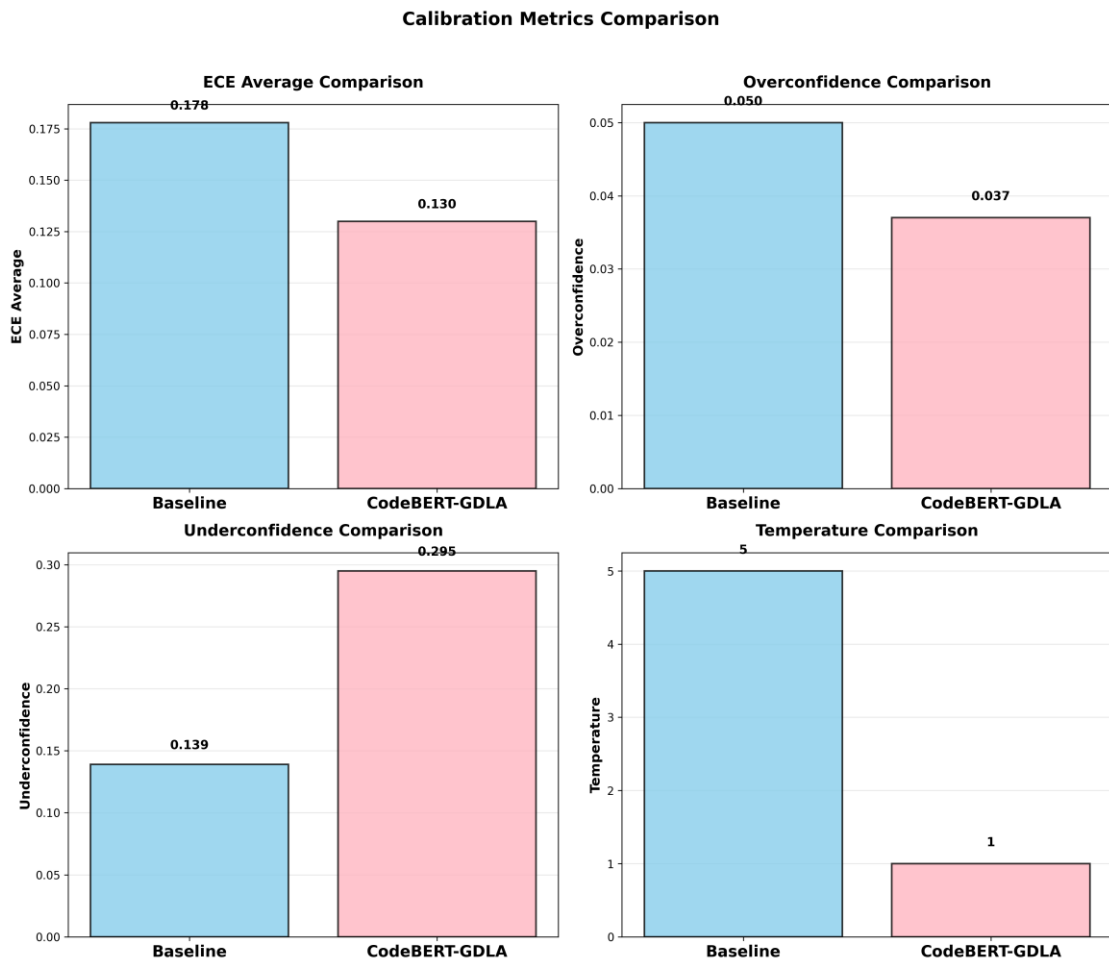


Рисунок 3.22 – Порівняння калібровки CodeBERT-BiGRU та CodeBERT-GDLA за метриками ECE, Overconfidence, Underconfidence і Temperature.

Модель CodeBERT-GDLA характеризується підвищеним рівнем under-confidence, що свідчить про більш стриманий характер її прогнозів. Водночас показники надмірної впевненості (over-confidence) є суттєво нижчими порівняно з базовими моделями, що має принципове значення для безпекових застосувань, оскільки зменшує ймовірність помилкової надмірної впевненості. Середнє значення метрики очікуваної калібрувальної похибки (Expected Calibration Error, ECE) також демонструє покращення, підтверджуючи більш адекватну оцінку ймовірностей.

Калібровані ймовірнісні оцінки забезпечують можливість більш надійної інтеграції моделі в системи підтримки прийняття рішень для аудиту смарт-контрактів, де важливими є не лише результати класифікації, але й рівень упевненості у виявлених уразливостях.

Запропонований калібрувальний компонент має низку переваг порівняно з класичними підходами до навчання моделей для виявлення вразливостей. По-перше, він дозволяє одночасно враховувати три різні аспекти некоректної впевненості моделі – загальну похибку калібрування (через метрику очікуваної калібрувальної похибки), дисбаланс масштабування ймовірностей (через температурний доданок) та надмірну впевненість у “легких” прикладах (через фокальну складову). Це забезпечує більш гнучке та стабільне узгодження між передбаченою впевненістю і реальною точністю класифікації.

По-друге, на відміну від традиційного температурного масштабування, яке застосовується після завершення навчання (post-hoc calibration), у запропонованому підході калібрування інтегровано безпосередньо в процес оптимізації. Завдяки цьому модель поступово навчається не лише розрізняти класи, а й правильно оцінювати ступінь своєї впевненості під час тренування, що знижує ризик переобладнання на некалібровані дані.

По-третє, запропонована функція втрат L_{cal} забезпечує баланс між точністю та надійністю – не погіршуючи класифікаційні показники, вона водночас покращує стабільність і достовірність прогнозів. Це особливо важливо для автоматизованих систем аудиту, у яких прийняття рішення залежить від порогу впевненості (наприклад, при автоматичному відборі смарт-контрактів для ручної перевірки).

У сукупності ці властивості роблять запропонований калібрувальний підхід придатним для production-середовищ із високими вимогами до довіри, інтерпретованості та контрольованості моделей штучного інтелекту (таблиця 3.6).

Таблиця 3.6 – Порівняння класичного та каліброваного навчання

Критерій	Класичне навчання (без L_{cal})	Запропонований підхід (з L_{cal})
Основна мета	Мінімізація класифікаційної помилки	Мінімізація помилки та узгодження ймовірностей
Тип функції втрат	Крос-ентропія або focal loss	Комбінована: $L_{base} + \lambda_{cal} \cdot L_{cal}$
Калібрування ймовірностей	Відсутнє або post-hoc	Інтегроване у процес навчання
Поведінка моделі	Часто overconfident (надмірна впевненість)	Калібрована, стримана у прогнозах
Показник метрики очікуваної калібрувальної похибки (ECE)	Вищий (гірша відповідність між p і точністю)	Нижчий (покращена узгодженість)
Надійність для прийняття рішень	Обмежена	Підвищена завдяки достовірним імовірностям
Інтерпретованість результатів	Часткова	Покращена через пояснювані ймовірності та карти важливості
Застосування	Лише класифікація	Класифікація і оцінка ризику / пріоритизація вразливостей

3.4 Висновки до третього розділу

У розділі розглянуто три рівні моделей: BiGRU-ATT як легкий базовий підхід із рекурентною архітектурою та механізмом уваги; CodeBERT-BiGRU як сильний бенчмарк, що поєднує трансформерні представлення коду (CodeBERT) з послідовною агрегацією (BiGRU); запропонована архітектура CodeBERT-GDLA, яка доповнює трансформерно-рекурентну основу графовою агрегацією (AST/CFG/DFG), ієрархічною увагою (token/block), λ -узгодженням з Grad-CAM та причинними регуляризаторами для пояснюваної локалізації. Проведені дослідження дозволили зробити такі узагальнені висновки:

- запропоновано цілісну архітектуру, що поєднує токенний рівень (CodeBERT-BiGRU) та блоковий рівень з графовою агрегацією (AST, CFG, DFG), а також λ -комбінацію уваги з Grad-CAM для узгодження інтерпретацій на різних рівнях подання коду;
- удосконалено механізм уваги шляхом введення спільного структурного зсуву B (на основі AST/CFG/DFG) та навчуваних коефіцієнтів β , γ для токенного й блокового рівнів, що керують часткою структурної інформації в логітах уваги та підвищують чутливість моделі до далеких і непрямих залежностей у коді;
- запропоновано метод CAM-узгодження уваги у програмному аналізі: модель цілеспрямовано навчається виділяти ті самі фрагменти, що й Grad-CAM карти. Вперше адаптовано до задач безпеки смарт-контрактів причинні регуляризатори (sufficiency/deletion losses), які дозволяють виявляти мінімально достатні та критично необхідні фрагменти коду для прийняття рішення;
- запропоновано метод слабосупервізованої локалізації, який використовує метод ковзного вікна з top-K агрегацією для локалізації вразливостей без розмічених даних. Досягнуто $mAP = 82.37\%$, $Hit@5 = 76.7\%$, $Hit@7-10 = 100\%$ для вразливості повторного входу; підхід демонструє високу стабільність ($\sigma = 2.13\%$) і придатність до автоматизованого аудиту та інтелектуальної підсвітки коду;
- проведено експериментальна перевірку. Масштабні абляційні дослідження показали, що:

1. видалення токенної уваги різко знижує точність виявлення повторного входу;
2. відключення графових представлень негативно впливає на виявлення integer overflow/underflow;
3. найкращі результати досягаються при сумісному використанні всіх прокращень;

— впроваджено методи напрямлені на прокращення стійкості та переносимості. Показано збереження якості після рефакторингу коду, а також стійкість до часткової обфускації, що підтверджує практичну застосовність у реалістичних сценаріях атак;

— удосконалено метод регуляризації для калібрування ймовірнісних оцінок багатоміткової класифікації вразливостей смарт-контрактів L_{cal} за рахунок застосування оптимізації очікуваної калібрувальної похибки, температурного масштабування та фокального приглушення надмірної впевненості безпосередньо в процесі навчання моделі CodeBERT-GDLA, що забезпечує зниження її надмірної впевненості та підвищує надійність результатів інтелектуального аналізу вихідного коду смарт-контрактів.

Отримані результати свідчать про те, що запропонована архітектура CodeBERT-GDLA не лише перевершує існуючі підходи за точністю класифікації, але й забезпечує інтерпретованість, стійкість та практичну придатність моделі.

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАСОБУ АУДИТУ СМАРТ-КОНТРАКТІВ НА ОСНОВІ CODEBERT-GDLA

В розділі описані розроблені програмні засоби та практичні аспекти впровадження системи автоматизованого аудиту смарт-контрактів на основі моделі CodeBERT-GDLA. Наведено мету створення такого інструменту та визначено основні вимоги до нього. Детально розглянуто архітектуру програмного засобу, включно з його клієнт-серверною структурою та взаємодією компонентів. Описано потоки даних і сценарії використання – від завантаження коду смарт-контракту до отримання звіту про вразливості. Розглянуто реалізацію API та форматів даних для інтеграції аналізатора в інші системи. Особливу увагу приділено інтерфейсу користувача (UI), що забезпечує зручність роботи з інструментом. Наведено приклад використання, який демонструє застосування розробленого засобу на прикладі аналізу реального смарт-контракту. Наприкінці розділу сформульовано висновки щодо результатів впровадження та використання розробленого програмного продукту.

4.1 Мета впровадження та вимоги

Метою впровадження розробленого інструменту є інтеграція моделі CodeBERT-GDLA у практичний програмний засіб для аналізу безпеки смарт-контрактів. Раніше розроблена модель продемонструвала високу ефективність виявлення вразливостей і здатність надавати інтерпретовані результати. Отримані результати досліджень підтвердили можливість використання цієї моделі у практичних системах аудиту смарт-контрактів. Таким чином, основна мета створення програмного засобу – надати розробникам та аудиторам зручний інструмент, що автоматично виявляє типові вразливості в коді Solidity-контрактів і пояснює причини їх появи. Інструмент покликаний підвищити надійність аналізу безпеки шляхом

подолання обмежень традиційних статичних аналізаторів, які базуються на жорстко заданих правилах і шаблонах.

Для досягнення поставленої мети були визначені вимоги до функціональності та якості програмного засобу. Функціональні вимоги визначають, що система повинна виконувати та яку корисну функцію забезпечувати користувачеві:

1. *аналіз вразливостей смарт-контракту.* Інструмент повинен виявляти наявність вразливостей у вихідному коді Solidity-контракту. Зокрема, підтримується пошук щонайменше чотирьох поширених типів вразливостей: переповнення цілих чисел (integer overflow), вихід за нижню межу цілого (integer underflow), вразливість повторного входу (re-entrancy) та залежність від мітки часу (timestamp dependency). Аналізатор має класифікувати контракт як вразливий або безпечний, а при виявленні небезпечних конструкцій – указувати їх тип;

2. *надання детального звіту.* За результатами аналізу система формує звіт, що містить перелік знайдених вразливостей із зазначенням їхніх типів та кількості, а також загальну оцінку рівня ризику. Звіт повинен відображати статус безпеки контракту (наприклад, CLEAN чи VULNERABLE), сукупний бал ризику та рівень небезпеки (низький, середній, високий, критичний) залежно від знайдених проблем. Таким чином, користувач отримує цілісну оцінку стану безпеки смарт-контракту;

3. *пояснення та підсвічування вразливого коду.* Інструмент повинен не лише вказувати на факт виявлення вразливості, але й надавати інтерпретацію результатів для підвищення довіри користувача. Для кожної знайденої вразливості система відображає відповідні фрагменти коду (рядки або блоки), які найбільш ймовірно містять причину вразливості. Це досягається за рахунок використання пояснюваних механізмів моделі CodeBERT-GDLA – зокрема, аналізу ваг уваги і методів на кшталт Grad-CAM – що дозволяють визначити найбільш впливові токени й рядки програмного коду. У звіті ці фрагменти підсвічуються для візуальної зручності, щоб аудитор міг швидко локалізувати потенційно небезпечне місце в коді;

4. *рекомендації щодо усунення вразливостей.* Програмний засіб має генерувати для користувача практичні поради з усунення кожного типу виявлених вразливостей. У звіті надаються стислий опис проблеми та список рекомендацій,

наприклад використати бібліотеку безпечної математики для запобігання переповненням, застосувати шаблон Checks-Effects-Interactions для захисту від повторного входу тощо. Такий зворотний зв'язок допомагає розробникам не лише виявити проблему, а й зрозуміти шляхи її виправлення;

5. *зручність введення даних*. Інструмент повинен підтримувати декілька способів введення вихідного коду смарт-контракту. По-перше, користувач може безпосередньо вставити або ввести текст Solidity-контракту у спеціальний редактор коду в інтерфейсі. По-друге, передбачена можливість завантаження файлу з розширенням .sol (файл смарт-контракту) через інтерфейс або API. Це розширює зручність використання, дозволяючи аналізувати як окремі фрагменти коду, так і цілі файли контрактів;

6. *програмний інтерфейс (API)*. Система повинна надавати REST API для можливості інтеграції її функцій в сторонні програми чи автоматизовані пайплайни (наприклад, CI/CD). Мінімально API має містити ендпоінт для відправки коду контракту на аналіз та отримання результатів у форматі JSON, а також ендпоінти для отримання службової інформації (стан здоров'я сервісу, перелік підтримуваних типів вразливостей тощо). Наявність API робить інструмент гнучким – ним можна скористатися не тільки через графічний інтерфейс, а й програмно.

Окрім функціональних, висунуто нефункціональні вимоги – критерії, що характеризують якість, продуктивність та зручність системи:

1. *точність та надійність аналізу*. Система повинна забезпечувати високу точність виявлення вразливостей, щоб мінімізувати кількість хибнопозитивних або хибнонегативних результатів. Якість моделі CodeBERT-GDLA, закладеної в основу інструменту, має безпосередній вплив на точність аналізу. Надійність також означає стабільну роботу на різних вхідних даних – інструмент має коректно обробляти смарт-контракти, написані в різних стилях і з різною складністю, зберігаючи при цьому адекватність результатів (модель продемонструвала стійкість до стилістичних змін коду);

2. *пояснюваність та прозорість*. Користувач (аудитор) повинен мати можливість розуміти, чому система дійшла того чи іншого висновку. Необхідно

забезпечити прозорість роботи аналізатора за рахунок пояснень: підсвічування коду, значень впевненості (відсоткової ймовірності вразливості), показників ризику тощо. Це є критично важливим для довіри до автоматизованого інструменту безпеки;

3. *зручність інтерфейсу та досвід користувача.* Графічний інтерфейс інструменту має бути інтуїтивно зрозумілим і простим у використанні. Вбудований редактор коду повинен підтримувати підсвітку синтаксису Solidity, а результати аналізу мають відображатися структуровано і читабельно (наприклад, у вигляді списку знайдених проблем з можливістю розгортання детальної інформації). Інтерфейс повинен швидко реагувати на дії користувача, забезпечуючи близький до реального часу режим аналізу (тобто результати мають видаватися через кілька секунд після запуску аналізу);

4. *продуктивність і масштабованість.* Важливо, щоб час аналізу одного смарт-контракту був достатньо малим для інтерактивного використання. Завдяки оптимізації моделі та використанню сучасних апаратних засобів (наприклад, GPU) аналіз навіть складного контракту має виконуватися за лічені секунди. Система повинна обробляти контракти розміром принаймні до 100 KB коду без суттєвої втрати продуктивності. У разі збільшення навантаження (аналіз багатьох контрактів) архітектура має дозволяти горизонтальне масштабування – наприклад, розгортання декількох екземплярів сервера аналізу;

5. *портативність та розгортання.* Програмний засіб має бути простим у розгортанні та підтримці. Використання веб-архітектури (клієнт-сервер) та контейнеризація (Docker) спрощують розгортання рішення в різних середовищах – від локального комп'ютера розробника до хмарного сервісу. Вимоги до сторонніх залежностей повинні бути задокументовані; процес встановлення (налаштування середовища Node.js та Python, завантаження моделі) – автоматизованим або детально описаним у інструкції. Це гарантує, що кінцеві користувачі зможуть легко почати роботу з інструментом без необхідності глибоко занурюватися в технічні нюанси.

Перераховані вимоги лягли в основу розробки системи аудиту смарт-контрактів. На їх виконання було спроектовано відповідну архітектуру та обрано технологічний стек, що дозволяє реалізувати функціонал інструменту та забезпечити

необхідні якісні характеристики. Наступні підрозділи присвячені опису архітектури рішення, сценаріїв його використання, а також ключових компонентів – таких як API та інтерфейс користувача – що разом забезпечують виконання зазначених вимог.

4.2 Архітектура інструмента

Архітектура інструмента побудована за принципом розподілу на клієнтську та серверну частини (рисунок 4.1). Клієнтська частина реалізована у вигляді веб-інтерфейсу на основі бібліотеки React, що забезпечує інтерактивність і зручність завантаження коду та перегляду результатів. Серверна частина реалізована з використанням Flask (Python) і містить у собі ядро аналізу – модель CodeBERT-GDLA, яка інтегрована у вигляді модуля на базі PyTorch. Взаємодія між клієнтом та сервером відбувається через REST API, що гарантує масштабованість і можливість інтеграції з іншими системами.

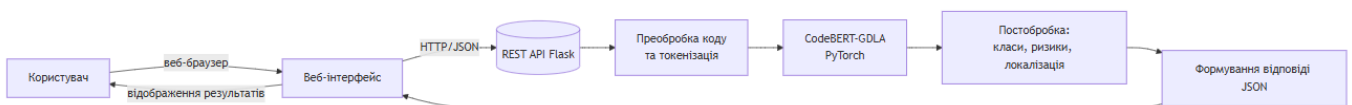


Рисунок 4.1 – Загальна архітектура веб-застосунку

Взаємодія з інструментом відбувається наступним чином (рисунок 4.2). Користувач через веб-інтерфейс вводить код смарт-контракту або завантажує файл із вихідним кодом. Після цього фронтенд надсилає отриманий код на сервер через REST API для подальшого аналізу. Сервер приймає код і передає його до моделі CodeBERT-GDLA для автоматичної обробки.

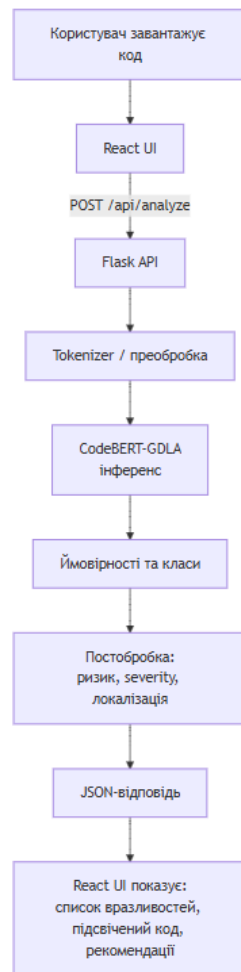


Рисунок 4.2 – Логічний потік даних при аналізі смарт-контракту

Модель глибокого аналізу коду перевіряє отриманий контракт на наявність вразливостей, використовуючи знання, здобуті під час навчання. На виході вона формує перелік знайдених потенційно небезпечних ділянок коду разом із відповідними характеристиками. Зокрема, сервер генерує JSON-відповідь, що містить для кожної виявленої вразливості її клас (тип), рівень ризику, ступінь впевненості моделі, локалізацію (місце у вихідному коді) та рекомендації щодо усунення. Така структура результатів забезпечує фронтенду всю необхідну інформацію для візуалізації та інтерпретації знайдених проблем.

На клієнтській стороні отримані результати використовуються для наочного відображення виявлених вразливостей у вихідному коді. Веб-інтерфейс підсвічує ті фрагменти коду, що були позначені моделлю як потенційно вразливі. Кожен такий фрагмент супроводжується коротким текстовим описом виявленої проблеми та рекомендацією щодо її усунення, які відображаються при взаємодії з інтерфейсом

(наприклад, при наведенні курсора або виборі фрагмента). Додатково інтерфейс відображає оцінений рівень ризику для кожної вразливості (наприклад, за допомогою кольорового маркування) та значення впевненості моделі. Це допомагає розробнику визначити пріоритетність реагування на знайдені вразливості.

Запропонована архітектура інструмента забезпечує модульність та масштабованість рішення. Розподіл системи на клієнтську та серверну частини дозволяє незалежно вдосконалювати користувацький інтерфейс і алгоритми аналізу. Використання веб-технологій (React, Flask) спрощує розгортання системи та її інтеграцію в існуючі процеси розробки. Таким чином, інструмент може бути легко впроваджений у практику розробниками для регулярного аудиту смарт-контрактів протягом циклу розробки.

4.3 Потоки даних і сценарії

Функціонування інструмента базується на послідовній обробці даних, що забезпечує повний цикл від введення смарт-контракту до формування готового звіту. Взаємодія користувача з системою починається на клієнтській стороні, де він може або завантажити файл із розширенням .sol, або вставити вихідний код безпосередньо у вбудований редактор. У результаті користувацькі дані уніфікуються й передаються до серверної частини через HTTP-запит до REST API. Передача відбувається у форматі JSON, що дозволяє зберегти універсальність та простоту інтеграції.

На сервері потік даних розгалужується на кілька послідовних етапів. Спочатку відбувається первинна перевірка коректності введених даних, після чого здійснюється їх токенізація та підготовка у формат, сумісний із моделлю CodeBERT-GDLA. Далі дані надходять у модуль інференсу, де модель генерує прогноз у вигляді ймовірностей належності контракту до різних класів вразливостей. Отримані результати піддаються постобробці: для кожного класу застосовується порогове рішення, обчислюється інтегральний бал ризику, визначається рівень небезпеки, а також виконується локалізація підозрілих фрагментів коду. На цьому етапі

формується узагальнений звіт, який містить структуровану інформацію про знайдені проблеми.

Результати аналізу повертаються у форматі JSON-відповіді, що містить зведений підсумок (рівень ризику та сукупний бал), перелік виявлених вразливостей із поясненнями та рекомендаціями, а також позиції у вихідному коді, які були визначені як потенційно небезпечні. Потік даних завершується на клієнтській стороні, де веб-інтерфейс інтерпретує отриману інформацію, підсвічує відповідні рядки коду, візуалізує ступінь небезпеки та дозволяє користувачеві ознайомитися з поясненнями. Таким чином, система реалізує замкнений цикл аналізу, починаючи від введення даних і завершуючи представленням результатів у зручному для користувача вигляді.

Окрім базового сценарію аналізу одного контракту, інструмент підтримує можливість пакетної обробки, коли декілька файлів надсилаються на сервер послідовно або паралельно. У цьому випадку для кожного контракту формується окремий результат, який може бути об'єднаний у загальний звіт. Ще одним сценарієм є інтеграція у CI/CD-процеси через API, що дозволяє автоматизувати перевірку смарт-контрактів безпосередньо під час їх розробки та перед деплоєм у блокчейн-мережу.

4.4 API та формати даних

Архітектура інструмента передбачає використання REST API, що забезпечує стандартизовану взаємодію між клієнтською та серверною частинами, а також відкриває можливість інтеграції з іншими системами. API реалізовано у вигляді набору HTTP-ендпоінтів, кожен із яких відповідає за певний функціонал. Основним є запит на аналіз смарт-контракту, який приймає код у тілі запиту та повертає результат у структурованому вигляді. Формат даних базується на JSON, що дозволяє легко інтегрувати систему з будь-яким сучасним програмним забезпеченням.

Стандартний запит на аналіз надсилається методом POST на адресу `/api/analyze`. Тіло запиту містить поле `contract_code`, у якому передається текст смарт-контракту.

Відповідь формується у вигляді JSON-об'єкта, що включає три основні блоки. Перший блок – це зведене резюме аналізу, яке містить загальний бал ризику та категорію рівня небезпеки (низький, середній, високий, критичний). Другий блок – перелік виявлених вразливостей, кожна з яких описується типом, значенням ймовірності, індивідуальним ризик-скором, локалізованим фрагментом коду та рекомендаціями щодо виправлення. Третій блок містить метадані про використану модель: її версію, дату останнього оновлення та список підтримуваних класів вразливостей. Така структура дозволяє користувачеві отримати як узагальнений, так і детальний аналіз.

Окрім основного ендпоінта, реалізовано допоміжні. Запит `GET /api/health` використовується для перевірки працездатності сервісу, а `GET /api/vulnerabilities` повертає перелік підтримуваних типів вразливостей разом із їх коротким описом. Запит `GET /api/model-info` надає інформацію про поточну версію моделі та параметри її роботи. Це дозволяє користувачеві впевнитися, що система працює коректно та з актуальною конфігурацією.

4.5 Інтерфейс користувача

Користувацький інтерфейс інструмента реалізовано як веб-програму, що працює у стандартному браузері й не потребує інсталяції додаткового програмного забезпечення. Його завдання полягає у забезпеченні простої та зрозумілої взаємодії з системою, коли для отримання результату користувачеві достатньо лише завантажити смарт-контракт і натиснути кнопку запуску аналізу. На головній сторінці інтерфейсу (рисунки 4.3 та 4.4) розташований редактор коду з підтримкою синтаксичного підсвічування мови Solidity та кнопка для ініціації перевірки. Користувач може як вставити код вручну, так і завантажити файл із розширенням `.sol`.



Рисунок 4.3 – Головна сторінка інтерфейсу з відкритим редактором коду та кнопкою запуску аналізу.

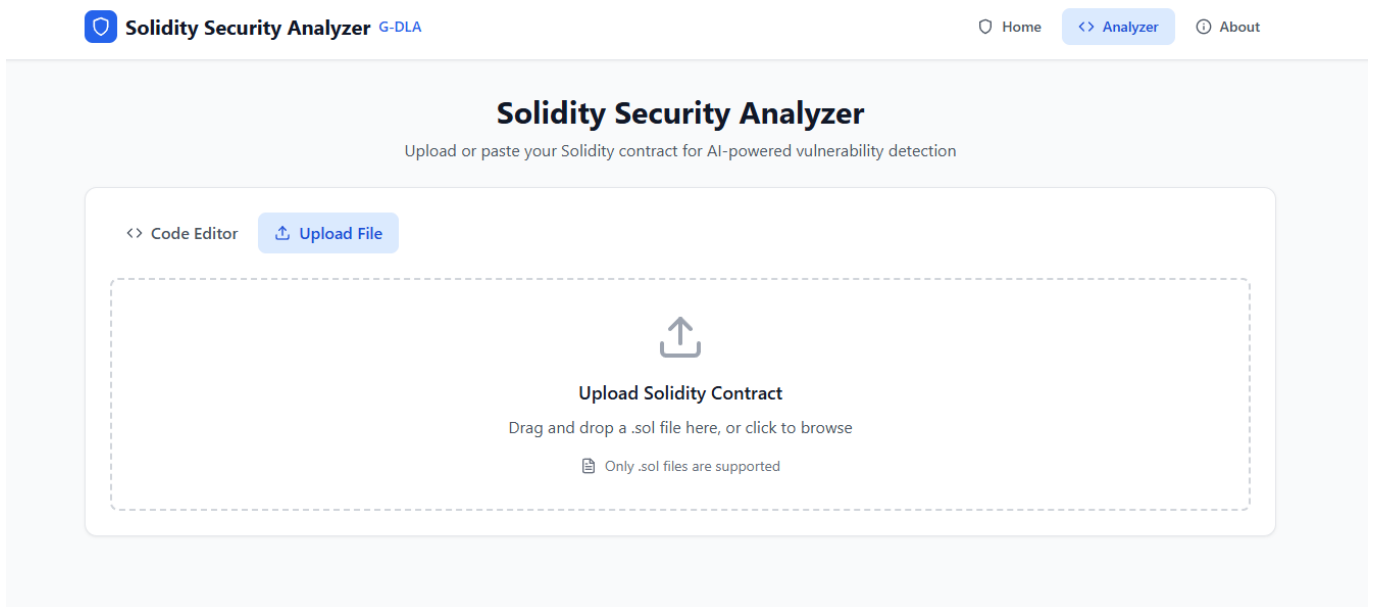


Рисунок 4.4 – Головна сторінка інтерфейсу з відкритим блоком завантаження файлу з вихідним кодом.

Після завершення аналізу у центральній частині вікна формується панель результатів (рисунок 4.5), де відображається зведена інформація: кількість виявлених

вразливостей, інтегральний рівень ризику та числовий показник *risk_score*. Ця панель дозволяє швидко оцінити загальний стан безпеки смарт-контракту.

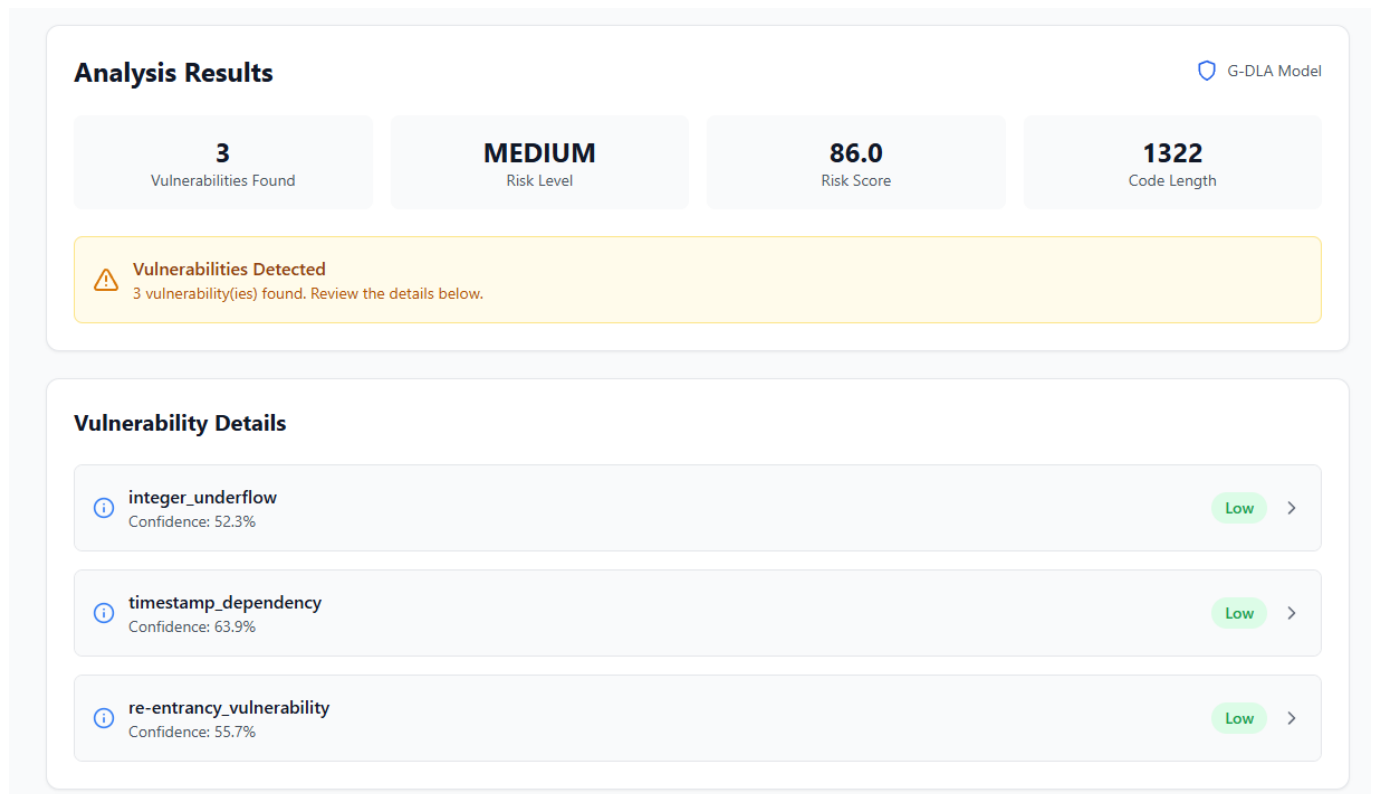


Рисунок 4.5 – Відображення результатів: панель з кількістю знайдених вразливостей та інтегральним рівнем ризику.

Деталізовані результати відображаються у нижній частині інтерфейсу. Тут подано список знайдених вразливостей із зазначенням їх типу, ймовірності та категорії ризику. Крім того, у самому редакторі підсвічуються відповідні ділянки коду (рисунки 4.6 та 4.7), що дозволяє миттєво локалізувати проблемні місця. Підсвічування виконується за принципом градації небезпеки: критичні вразливості позначаються червоним кольором, високі – помаранчевим, середні – жовтим. Для кожної проблеми користувач може відкрити коротке пояснення та рекомендації щодо її усунення.

Analyzed Code

Code with syntax highlighting

```

1 pragma solidity ^0.8.0;
2
3 contract Gambling {
4     mapping(address => uint) public balances;
5     uint public jackpot;
6
7     function placeBet() external payable {
8         require(msg.value > 0);
9         balances[msg.sender] += msg.value;
10        jackpot += msg.value;
11    }
12
13    function withdrawJackpot() external {
14        uint amount = jackpot;
15        jackpot = 0; // State update before external call
16        (bool success, ) = msg.sender.call{value: amount}(""); // External call
17        require(success, "Transfer failed");
18    }
19 }

```

Vulnerable Code Snippets

re-entrancy_vulnerability

Confidence: 95.4% Low

```

1 6 |
2 7 |     function placeBet() external payable {
3 8 |         require(msg.value > 0);
4 9 |         balances[msg.sender] += msg.value;
5 10 |        jackpot += msg.value;
6 11 |     }
7 12 |

```

Рисунок 4.6 – Приклад підсвіченого коду смарт-контракту з вразливістю

Vulnerability Details

re-entrancy_vulnerability
Confidence: 95.4% Low

Description
A reentrancy vulnerability occurs when an external call allows an attacker to re-enter the contract before the first call completes, potentially manipulating state variables.

Recommendations

- ✓ Use the Checks-Effects-Interactions pattern
- ✓ Update state before external calls
- ✓ Consider using ReentrancyGuard from OpenZeppelin
- ✓ Limit gas forwarded to external calls
- ✓ Use pull payments instead of push payments

Vulnerable Code Section

```

1  6 |
2  7 |         function placeBet() external payable {
3  8 |             require(msg.value > 0);
4  9 |             balances[msg.sender] += msg.value;
5 10 |             jackpot += msg.value;
6 11 |         }
7 12 |

```

G-DLA Attention Analysis
Model attention weight: 95.39%

Рисунок 4.7 – Загальний опис вразливості та шляхи її усунення

Таким чином, інтерфейс поєднує компактне представлення результатів із можливістю поглибленого аналізу, а кольорове підсвічування та пояснення забезпечують прозорість і довіру до роботи моделі.

4.6 Приклад застосування

Практичну цінність інструмента продемонстровано на прикладі аналізу реального смарт-контракту, який містив поширену вразливість *integer_underflow* (рисунок 4.8).

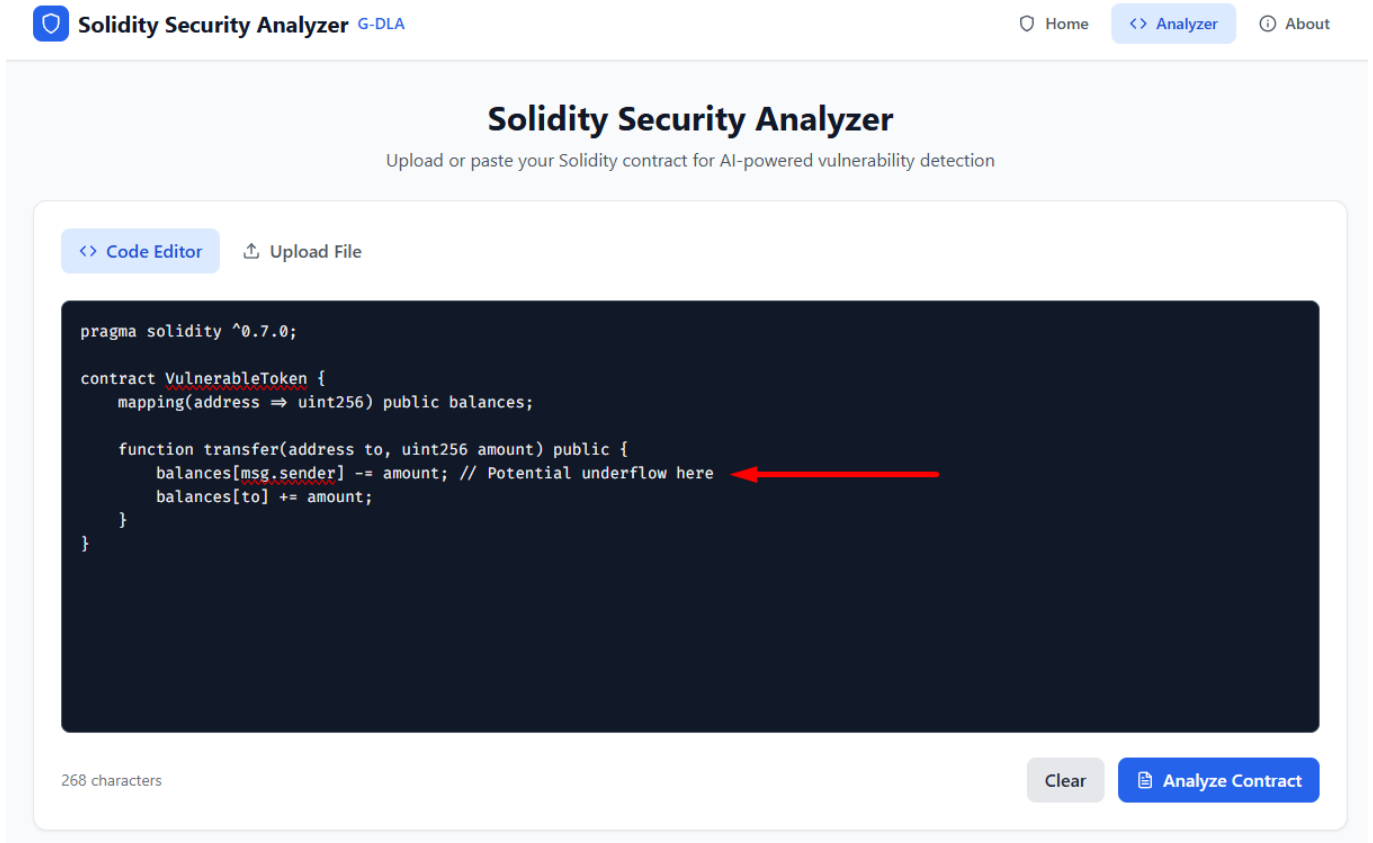


Рисунок 4.8 – Приклад вразливого смарт-контракту в полі редактору коду

Після завантаження контракту в інтерфейс і запуску аналізу система видала підсумковий результат із високим рівнем ризику (рисунок 4.9). Зведена панель показала інтегральний $risk_score = 200$ і класифікувала контракт як небезпечний.

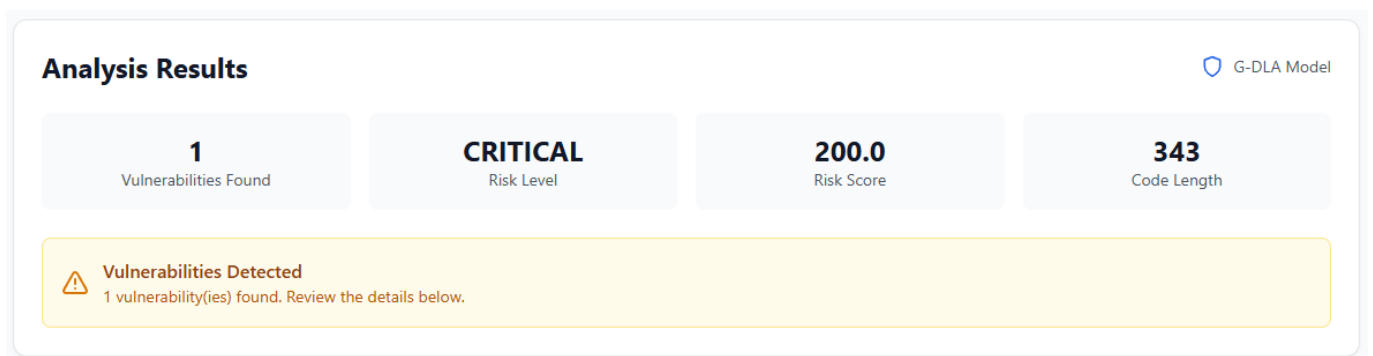


Рисунок 4.9 – Зведена панель результатів кейс-стаді з високим рівнем ризику.

Після завантаження коду контракту в інтерфейс і запуску аналізу інструмент згенерував прогноз для чотирьох класів вразливостей (порогове значення прийняття рішення – 0.5):

- integer_underflow: prob = 0.9735, pred = 1 (вище порога, зафіксовано як уразливість);

- integer_overflow: prob = 0.0315, pred = 0;
- timestamp_dependency: prob = 0.0218, pred = 0;
- re-entrancy_vulnerability: prob = 0.0354, pred = 0;

Детектор локалізував підозрілий фрагмент у рядку з операцією віднімання балансу (рисунок 4.10).

```
DEBUG: Predicted indices: [0]
DEBUG: Raw logits: [ 3.6044314 -3.4263775 -3.8045158 -3.3035765]
DEBUG: Probabilities: [0.9735174 0.0314812 0.02178483 0.0354487 ]
DEBUG: Predicted labels: [1 0 0 0]
DEBUG: Vulnerability labels: ['integer_underflow', 'integer_overflow', 'timestamp_dependency', 're-entrancy_vulnerability']
DEBUG: integer_underflow: prob=0.9735, pred=1 (threshold=0.3)
DEBUG: integer_overflow: prob=0.0315, pred=0 (threshold=0.3)
DEBUG: timestamp_dependency: prob=0.0218, pred=0 (threshold=0.3)
DEBUG: re-entrancy_vulnerability: prob=0.0354, pred=0 (threshold=0.3)
```

Рисунок 4.10 – Вивід вірогідностей для кожного з чотирьох класів вразливостей

У графічному інтерфейсі цей рядок було підсвічено як «високий ризик», із поясненням про можливий integer underflow у Solidity версій до 0.8.x, де арифметика за замовчуванням не перевіряється (рисунок 4.11).

Vulnerability Details

integer_underflow

Confidence: 97.4%

Critical

Description

Integer underflow occurs when an arithmetic operation goes below the minimum value that can be stored in the data type, causing the value to wrap around.

Recommendations

- Use SafeMath library or Solidity 0.8+ built-in overflow protection
- Add explicit checks before arithmetic operations
- Use require() statements to validate inputs
- Consider using OpenZeppelin's SafeERC20
- Implement proper bounds checking

Vulnerable Code Section

```

1  5 |
2  6 |         function transfer(address to, uint256 amount) public {
3  7 |             require(balances[msg.sender] >= amount, "Insufficient balance");
4  8 |             balances[msg.sender] -= amount; // Potential underflow here
5  9 |             balances[to] += amount;
6 10 |         }
7 11 |     }

```

G-DLA Attention Analysis

Model attention weight: 97.35%

Рисунок 4.11 – Опис виявленої вразливості та рекомендації щодо її усунення

Для зведеної панелі використовується агрегований показник оцінки ризику:

$$risk_score = \sum_i w_i \cdot p_i, \quad (4.1)$$

де p_i – ймовірність для класу i , w_i – ваговий коефіцієнт небезпеки (для арифметичних помилок – підвищений). Для даного прикладу агрегований $risk_score = 0.97$ (клас «високий»), що відповідає наявності однієї критичної підозри вище порога.

Аналізатор локалізував як підозрілу операцію віднімання в тілі функції `transfer` контракту `VulnerableToken` (рисунок 4.12).

Vulnerable Code Snippets



Рисунок 4.12 – Вразливий фрагмент виділяється спеціальним позначенням на початку рядка для зручності сприйняття

Модель надала високий рівень довіри до класу `integer_underflow` (імовірність ≈ 0.97 при порозі прийняття 0.5). Така класифікація базується на кореляції патерну `x -= y` в середовищі неперевіреної арифметики (Solidity $< 0.8.0$) з можливістю `wrap-around` (обгортання) при недостатньому балансі.

Для підтвердження практичної експлуатованості знайденої вразливості було проведено наступні дії:

1. аналіз контексту коду – перевірено відсутність додаткових інваріантів або захисних механізмів (наприклад, централізованої логіки мінта, додаткових перевірок якості балансу, зовнішніх обмежень), які могли б запобігати експлуатації;
2. версія компілятора – встановлено `pragma solidity ^0.7.0`, тобто арифметика в середовищі компіляції не містить автоматичних перевірок переповнення/недоповнення;
3. іункціональне тестування – розроблено unit-тест, що моделює сценарій експлуатації: виклик `transfer` від адреси з нульовим балансом та перевірка значення балансу після виконання. Очікуваний результат при наявності вразливості – значення

балансу відправника $\approx 2^{256} - 1$ (wrap-around). Тест пройшов успішно на незмінній версії контракту, що підтвердило експлуатованість;

4. ручна верифікація статичними засобами – застосовано інструментальний аналіз (статичні сканери) для підтвердження підозри; сканери ідентифікували місце потенційної арифметичної помилки.

Отримані дані свідчать, що у розглянутому фрагменті реалізації присутня реальна вразливість класу `integer underflow`, яка може бути експлуатована у мережеских умовах. Механізм експлуатації полягає в тому, що при виклику *transfer* з *amount* > *balances[msg.sender]* значення *balances[msg.sender]* набуває великого значення через обгортання по модулю 2^{256} , що дозволяє подальше несанкціоноване використання балансу атакувальником. Така поведінка є типовою для версій компілятора до 0.8.0, де відсутні вбудовані перевірки арифметики.

Важливо також підкреслити, що модель правильно ідентифікувала місце ризику та надала високу ймовірність класу загрози. Проте остаточне визначення експлуатованості завжди має проводитися з урахуванням семантичного контексту контракту.

Після імплементації одного з рекомендованих захисних заходів – додавання перевірки *require(balances[msg.sender] >= amount)* або застосування бібліотеки `SafeMath` (у середовищі 0.7.x), або переходу на `pragma solidity ^0.8.0` – функціональний тест з експлойтом очікувано завершується з `revert`-ом або не призводить до обгортання.

4.7 Апробація запропонованих рішень на практиці

Розроблені в дисертаційній роботі методи та програмно-алгоритмічні засоби були апробовані як у дослідницькому середовищі, так і в реальних умовах впровадження на двох підприємствах: ТОВ НВО «Діскрет» та НВП «КАРЕ». Практична апробація підтвердила ефективність запропонованих моделей глибокого навчання, механізмів уваги, Grad-CAM орієнтованої інтерпретації, калібрувальних методів та процедур пріоритизації вразливостей у задачах аудиту смарт-контрактів.

У науково-практичну діяльність ТОВ НВО «Діскрет» впроваджено програмно-алгоритмічне забезпечення автоматизованого виявлення та локалізації вразливостей у смарт-контрактах, створене в межах дисертаційної роботи. Зокрема інтегровано:

- методи багатоміткової класифікації та локалізації на основі архітектури CodeBERT-GDLA, що поєднує токенний і блоковий рівні з графовою агрегацією (AST/CFG/DFG) та λ -узгодженням уваги з Grad-CAM;
- механізми інтерпретації на основі структурної уваги та Grad-CAM, що дозволяють отримувати точкові heatmap-локалізації підозрілих фрагментів коду;
- методи причинної перевірки (sufficiency/deletion) для підсилення пояснюваності висновків моделі та відсікання фрагментів, що не впливають на рішення;
- програмний модуль інтеграції локалізованих фрагментів у внутрішні інструменти аудиту підприємства.

За результатами впровадження встановлено, що використання запропонованих методів:

- значно підвищило точність виявлення потенційно небезпечних фрагментів у коді;
- скоротило час аналізу смарт-контрактів завдяки автоматизованій локалізації;
- зменшило кількість пропущених вразливостей завдяки CAM-узгодженій увазі;
- підвищило прозорість аудиту через використання пояснюваних моделей.

Факт і результати впровадження підтверджуються Актом впровадження (див. Додаток А).

У діяльність науково-виробничого підприємства «КАРЕ», яке розробляє програмні рішення для аудиту та захисту програмного забезпечення, впроваджено окремі компоненти, розроблені в дисертаційній роботі:

- калібрувальний компонент L_{cal} для узгодження імовірнісних оцінок моделі з фактичною правильністю рішень;

- метод пост-калібрування (temperature scaling) та оптимізація порогів спрацювання, що забезпечили зменшення Expected Calibration Error і зробили моделі більш надійними у production-середовищі;
- модуль пріоритизації знахідок, що поєднує калібровані ймовірності з показниками впевненості локалізації (узгодженість attention/Grad-CAM), дозволивши автоматично визначати критичність виявлених вразливостей;
- модуль формування інтерпретованих результатів для інженерів-аудиторів на основі class-specific карт важливості.

Використання впроваджених рішень дозволило:

- підвищити точність і стабільність прогнозів моделей;
- зменшити кількість пропущених вразливостей та хибнопозитивних спрацювань;
- скоротити час аудиту й локалізації небезпечних ділянок коду;
- підвищити пояснюваність рішень і стабільність порогів прийняття;
- загалом підвищити рівень безпеки програмних продуктів, що розробляються підприємством.

Факт впровадження підтверджено довідкою НВП «КАРЕ», наведеною у Додатку А до дисертації.

4.7 Висновки до четвертого розділу

У цьому розділі було представлено практичну реалізацію розробленого інструмента для аудиту смарт-контрактів на основі моделі CodeBERT-GDLA. Сформована архітектура системи продемонструвала доцільність клієнт-серверного підходу, де веб-інтерфейс забезпечує зручність роботи користувача, а серверна частина відповідає за проведення аналізу та формування результатів. Такий поділ функцій зробив систему модульною, масштабованою та придатною до інтеграції в існуючі середовища розробки.

Окрему увагу було приділено потокам даних та сценаріям використання інструмента. Встановлено, що система підтримує як індивідуальний, так і пакетний

аналіз смарт-контрактів, а також може бути інтегрована у процеси CI/CD. Це значно розширює можливості її практичного застосування та робить корисною не лише для індивідуальних розробників, а й для командних середовищ.

Розглянутий REST API довів універсальність та відкритість запропонованого рішення. Завдяки стандартизованому формату обміну даними (JSON) забезпечено простоту взаємодії з іншими системами та можливість автоматизації аналізу у рамках комплексних процесів розробки.

Інтерфейс користувача показав, що складні результати аналізу можуть бути представлені у зрозумілому та наочному вигляді. Використання кольорового підсвічування, структурованих звітів та рекомендацій підвищує рівень довіри користувача до системи й суттєво спрощує процес аудиту.

Нарешті, проведене кейс-стаді підтвердило практичну придатність інструмента для виявлення вразливостей у реальних смарт-контрактах. Воно продемонструвало не лише точність у виявленні типових проблем, але й здатність системи локалізувати їх у коді та надати рекомендації щодо усунення. Це забезпечує цілісний підхід до підвищення безпеки, що поєднує автоматизоване виявлення з інтерпретацією та практичною допомогою розробникам.

Розроблені в дисертаційній роботі методи та програмні засоби отримали практичне впровадження у діяльності науково-виробничого підприємства «КАРЕ» та товариства з обмеженою відповідальністю «Діскрет», а також знайшли застосування у навчальному процесі Національного університету «Одеська політехніка».

Узагальнено можна стверджувати, що запропонований програмний засіб є ефективним інструментом для практичного впровадження результатів дослідження та може стати основою для подальшої інтеграції методів глибинного аналізу у реальні процеси розробки й аудиту смарт-контрактів.

ВИСНОВКИ

В дисертаційній роботі вирішено актуальну науково-прикладну задачу підвищення ефективності автоматизованого аналізу безпеки смарт-контрактів шляхом розроблення методів і моделей багатоміткового виявлення, класифікації та локалізації вразливостей із забезпеченням пояснюваності рішень. Запропоновані методи і моделі машинного навчання забезпечують підвищення ефективності за відповідними метриками виявлення, класифікації та локалізації вразливостей у смарт-контрактах та становлять цілісний технологічний стек інтелектуального аналізу вихідного коду смарт-контрактів.

Проведений системний аналіз сучасних методів виявлення вразливостей у смарт-контрактах показав, що наявні інструменти безпеки суттєво відрізняються за принципами роботи, рівнем точності, швидкістю та можливостями практичного застосування. Формальні методи забезпечують високу строгість і повноту перевірки заданих властивостей, однак є складними у застосуванні та погано масштабуються для великих і складних контрактів. Статичні та динамічні підходи дозволяють автоматизувати аналіз і застосовуються на практиці, проте характеризуються обмеженим охопленням сценаріїв, чутливістю до евристик і можливістю хибних спрацювань. Методи на основі машинного навчання демонструють перспективність у контексті масштабованої автоматизації аналізу, однак зазвичай не забезпечують достатньої точності локалізації вразливих фрагментів коду, залежать від якості навчальних даних та залишаються обмеженими з точки зору інтерпретованості результатів. Гібридні підходи, що поєднують різні методи аналізу, дозволяють частково компенсувати зазначені недоліки, проте ускладнюють архітектуру систем і підвищують обчислювальні витрати. Узагальнення виявлених обмежень свідчить про наявність протиріччя між точністю, масштабованістю та здатністю до локалізації вразливостей, що обумовлює необхідність розробки інтелектуальних моделей аналізу смарт-контрактів, здатних поєднувати переваги глибокого навчання з механізмами пояснюваної локалізації та підтримкою багатоміткової класифікації вразливостей.

Вперше запропоновано комбінований метод формування навчального датасету за рахунок поєднання автоматичного інжектування аномальних конструкцій у коректний код смарт-контрактів із генерацією додаткових прикладів великою мовною моделлю з подальшою верифікацією, що дозволило забезпечити збалансованість вибірки.

Вдосконалено архітектуру BiGRU-ATT – двонапрямну рекурентну мережу з подвійним механізмом уваги, орієнтовану на моделювання послідовних синтаксичних структур і первинну локалізацію підозрілих фрагментів. Для глибшого семантичного аналізу вдосконалено модель CodeBERT-BiGRU, яка поєднує трансформерні ембеддинги програмного коду з рекурентним контекстом, підвищуючи чутливість до логічних залежностей і доменної специфіки Solidity. Вперше побудовано модель CodeBERT-GDLA на основі вдосконалення архітектур BiGRU-ATT і CodeBERT-BiGRU за допомогою запропонованих методів графового зсуву і дворівневої графово-орієнтованої уваги, що дозволило підвищити ефективність виявлення та класифікації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів.

Для забезпечення прозорості та аудиторської придатності рішень запропоновано метод поєднання attention-карт із Grad-CAM у просторі кодових представлень. Розроблено спосіб побудови теплових карт впливу на рівні токенів/рядків і введено узгоджені кількісні метрики якості локалізації (адаптації overlap@k/IoU до текстових спанів), що дозволяє оцінювати не лише «що» виявлено, а й «де саме» у коді. Показано, що узгодженість локалізаційних індикаторів з імовірнісними оцінками сприяє пріоритизації знахідок і скорочує витрати часу на ручну валідацію.

Експериментальні дослідження охопили ключові вразливості екосистеми EVM – integer underflow/overflow, timestamp dependency та reentrancy – у багатомітковій постановці. Проведено порівняння з базовими архітектурами за стандартними класифікаційними метриками (accuracy, precision, recall, F1, ROC-AUC), а також за показниками якості локалізації та часової продуктивності. Результати підтвердили, що використання трансформерних ембеддингів у поєднанні з рекурентним

моделюванням і подвійною увагою забезпечує стабільний приріст якості детекції та покращує збіжність локалізацій із класифікаційними рішеннями. Абляційні дослідження кількісно зафіксували внесок кожного компонента (двошарова увага, Grad-CAM/attention, тип ембеддингів, стратегії генерації/балансування даних, регуляризація L_{cal}), а окремий метод випробувань на стійкість до рефакторингів (перейменування ідентифікаторів, зміни форматування, легкі мутації/обфускація) засвідчив стійкість моделей до поширених варіацій коду, з мінімальною деградацією метрик.

Практичну цінність результатів підтверджено створенням програмно-алгоритмічного прототипу з наскрізним сценарієм: імпорт контракту $\rightarrow$ багатоміткова детекція $\rightarrow$ пояснювана локалізація $\rightarrow$ звітність для аудитора. Архітектура прототипу та описані інтерфейси дають змогу інтегрувати інструмент у конвеєри забезпечення якості (CI/CD) та у процеси попереднього аудиту, а калібровані імовірності й локалізаційні карти дозволяють будувати стратегії пріоритизації усунення вразливостей та ризик-орієнтовані процеси для команд безпеки.

Наукова новизна роботи полягає у тому, що: вперше запропоновано комбінований метод формування навчального датасету за рахунок поєднання автоматичного інжектування аномальних конструкцій у коректний код смарт-контрактів із генерацією додаткових прикладів великою мовною моделлю з подальшою верифікацією, що дозволило забезпечити збалансованість вибірки; вперше побудовано модель CodeBERT-GDLA на основі вдосконалення архітектур BiGRU-ATT і CodeBERT-BiGRU за допомогою запропонованих методів графового зсуву і дворівневої графово-орієнтованої уваги, що дозволило підвищити ефективність виявлення та класифікації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів; удосконалено метод регуляризації для калібрування ймовірнісних оцінок багатоміткової класифікації вразливостей смарт-контрактів за рахунок застосування оптимізації очікуваної калібрувальної похибки, температурного масштабування та фокального приглушення надмірної впевненості безпосередньо в процесі навчання моделі CodeBERT-GDLA, що забезпечує зниження

її надмірної впевненості та підвищує надійність результатів інтелектуального аналізу вихідного коду смарт-контрактів; отримав подальший розвиток метод слабосупервізованого виділення вразливих фрагментів коду за рахунок інтеграції САМ-узгодженої уваги з причинно-орієнтованою регуляризацією, що забезпечує підвищення ефективності локалізації вразливостей при інтелектуальному аналізі вихідного коду смарт-контрактів.

Попри досягнуті результати, окреслено межі застосовності: орієнтація на Solidity/EVM потребує адаптації при перенесенні на інші платформи (Move, CosmWasm, Rust-based контракти); покриття класів вразливостей наразі зосереджене на найпоширеніших сценаріях і потребує подальшого розширення таксономії (SWC/DASP/OWASP SC Top-10); вплив стилістичного зсуву, зумовленого LLM-генерацією прикладів, вимагає додаткових механізмів контролю доменної близькості.

Напрями подальших досліджень передбачають: розширення таксономії вразливостей та мультимодальних ознак (байткод, графи керування/потоків даних, трасування виконання); інтеграцію графових моделей (GraphCodeBERT); поєднання детектора з формальною перевіркою властивостей; поглиблене калібрування (температурне масштабування, Bayesian/ensemble-оцінювання невизначеності) та вартісно-чутливі функції втрат; безперервне навчання й доменне перенесення в умовах змін середовища атак; оптимізацію інференсу (distillation/quantization) для аналізу великих репозиторіїв у реальному часі.

У підсумку, запропоновані у дисертації інтегровані методи та моделі – від побудови датасету до гібридних моделей із пояснюваною локалізацією та каліброваними рішеннями – забезпечують послідовне підвищення ефективності, надійності та керованості процесів аудиту смарт-контрактів і створює науково обґрунтовану основу для розвитку наступного покоління систем аналізу безпеки, здатних поєднати точність експертного підходу з масштабованістю автоматизованих інструментів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Janssen A., Patti F. Demystifying smart contracts // Osservatorio Diritto Civile e Commerciale. – 2020. – Т. 9, № 1. – С. 31–50. – DOI: <https://doi.org/10.4478/98131>.
2. Zou W., Lo D., Kochhar P. S., Le X.-B.-D., Xia X., Feng Y., Chen Z., Xu B. Smart contract development: Challenges and opportunities // IEEE Transactions on Software Engineering. – 2021. – Т. 47, № 10. – С. 2084–2106. – DOI: <https://doi.org/10.1109/TSE.2019.2942301>.
3. Soud M., Qasse I. A., Liebel G., Hamdaqa M. AutoMESC: Automatic Framework for Mining and Classifying Ethereum Smart Contract Vulnerabilities and Their Fixes // arXiv preprint. – 2022. – arXiv:2212.10660. – Режим доступу: <https://arxiv.org/abs/2212.10660>. – Дата звернення: 1 липня 2024 р.
4. Hoffmann F. Challenges of Proof-of-Useful-Work (PoUW) // arXiv preprint. – 2022. – arXiv:2209.03865. – Режим доступу: <https://arxiv.org/abs/2209.03865>. – Дата звернення: 1 листопада 2023 р.
5. Karantias K., Kiayias A., Zindros D. Proof-of-Burn // Financial Cryptography and Data Security. – 2020. – С. 523–540. – DOI: https://doi.org/10.1007/978-3-030-51280-4_28.
6. Grandjean D., Heimbach L., Wattenhofer R. Ethereum Proof-of-Stake Consensus Layer: Participation and Decentralization // arXiv preprint. – 2023. – arXiv:2306.10777. – Режим доступу: <https://arxiv.org/abs/2306.10777>. – Дата звернення: 12 серпня 2025 р.
7. Antonino P., Roscoe A. W. Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity // arXiv preprint. – 2020. – arXiv:2002.02710. – Режим доступу: <https://arxiv.org/abs/2002.02710>. – Дата звернення: 12 серпня 2025 р.
8. Mukhia K., Luwang S. R., Nurujjaman Md., Chakraborty T., Saha S., Hens C. Universal Patterns in the Blockchain: Analysis of EOAs and Smart Contracts in ERC20 Token Networks // arXiv preprint. – 2025. – arXiv:2508.04671. – Режим доступу: <https://arxiv.org/abs/2508.04671>. – Дата звернення: 1 листопада 2025 р.

9. Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers // Financial Cryptography and Data Security. – 2017. – DOI: 10.1007/978-3-319-70278-0_33.
10. Szabo N. The idea of smart contracts // Nick Szabo's Papers and Concise Tutorials. – 1997. – Режим доступу: <https://www.scirp.org/reference/referencespapers?referenceid=3730627>. – Дата звернення: 1 листопада 2025 р.
11. Zheng Z., Xie S., Dai H.-N., Chen W., Chen X., Weng J., Imran M. An Overview on Smart Contracts: Challenges, Advances and Platforms // arXiv preprint. – 2019. – arXiv:1912.10370. – Режим доступу: <https://arxiv.org/abs/1912.10370>. – Дата звернення: 1 березня 2025 р.
12. Dai H.-N., Zheng Z., Zhang Y. Blockchain for Internet of Things: A Survey // IEEE Internet of Things Journal. – 2019. – DOI: 10.1109/IIOT.2019.2920987.
13. Zheng Z., Xie S., Dai H., Chen X., Wang H. An overview of blockchain technology: Architecture, consensus, and future trends // 2017 IEEE International Congress on Big Data (BigData Congress): Proc. – 2017. – С. 557–564. – DOI: 10.1109/BigDataCongress.2017.85.
14. Wan J., Li J., Imran M., Li D., та ін. A blockchain-based solution for enhancing security and privacy in smart factory // IEEE Transactions on Industrial Informatics. – 2019. – Т., №, С. 3652-2660. – DOI: 10.1109/TII.2019.2894573.
15. Zaazaa O., El Bakkali H. A systematic literature review of undiscovered vulnerabilities and tools in smart contract technology // Journal of Intelligent Systems. – 2023. – Т., № 32. – DOI: <https://doi.org/10.1515/jisys-2023-0038>.
16. Hu B., Zhang Z., Liu J., Liu Y., Yin J., Lu R., Lin X. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems // Patterns. – 2021. – Т. 2, № 2. – DOI: <https://doi.org/10.1016/j.patter.2020.100179>.
17. Wijayakoon C., Dong H., Bandara H.M.N.D., Tari Z., Soin A. Legal Compliance Evaluation of Smart Contracts Generated by Large Language Models // arXiv preprint. – 2025. – arXiv:2506.00943. – Режим доступу: <https://arxiv.org/abs/2506.00943>. – Дата звернення: 1 листопада 2025 р.

18. Komleva, N. O., & Tereshchenko, O. I. (2023). Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform. *Herald of Advanced Information Technology*, 6(1), 54–68. – DOI: <https://doi.org/10.15276/hait.06.2023.4>
19. Lee J., Fricotté A. DAO Token Transferability: Property, Contract, and Technology // *European Journal of Risk Regulation*. – 2024. – Online first: C. 1–17. – DOI: 10.1017/err.2024.93.
20. Beniiche A. A Study of Blockchain Oracles // *arXiv preprint*. – 2020. – arXiv:2004.07140. – Режим доступу: <https://arxiv.org/abs/2004.07140>. – Дата звернення: 2 грудня 2023 р.
21. Zhang Y., Wen J. An IoT electric business model based on the protocol of Bitcoin // *Proc. 18th Int. Conf. on Intelligence in Next Generation Networks (ICIN)*. – 2015. – С. 184–191. – DOI: 10.1109/ICIN.2015.7073830.
22. Yang Z., Lei H. Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language // *arXiv preprint*. – 2018. – arXiv:1803.09885. – Режим доступу: <https://arxiv.org/abs/1803.09885>. – Дата звернення: 22 березня 2025 р.
23. Nikolić I., Kolluri A., Sergey I., Saxena P., Hobor A. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale // *arXiv preprint*. – 2018. – arXiv:1802.06038. – Режим доступу: <https://arxiv.org/abs/1802.06038>. – Дата звернення: 1 серпня 2024 р.
24. Abdellatif T., Brousmiche K. L. Formal verification of smart contracts based on users and blockchain behaviors models // *IFIP NTMS: Proc.* – 2018. – С. 1–5. – DOI: <https://doi.org/10.1109/NTMS.2018.8328737>.
25. Ante L., Fiedler I. The new digital economy: How decentralized finance (DeFi) and non-fungible tokens (NFTs) are transforming value creation, ownership models, and economic systems // *Digital Business*. – 2024. – Т., № 5, Issue 1. – Ст. 100094. – DOI: 10.1016/j.digbus.2024.100094.
26. Heimbach L., Knier Q., Vonlanthen Y., Wattenhofer R. DeFi and NFTs Hinder Blockchain Scalability // *arXiv preprint*. – 2023. – arXiv:2302.06708. – Режим доступу: <https://arxiv.org/abs/2302.06708>. – Дата звернення: 1 жовтня 2025 р.

27. Kim H., Kim H.-S., Park Y.-S. Perpetual Contract NFT as Collateral for DeFi Composability // IEEE Access. – 2022. – DOI: <https://doi.org/10.1109/ACCESS.2022.3225884>.
28. Canidio A., Pahari V. Becoming Immutable: How Ethereum is Made // arXiv preprint. – 2025. – arXiv:2506.04940. – Режим доступу: <https://arxiv.org/abs/2506.04940>. – Дата звернення: 1 листопада 2025 р.
29. Prashayuniar A. H., Syafrida I. Comparative Performance Analysis of Bitcoin, Ethereum, and Solana in the Crypto Market // Syntax Idea. – 2025. – Т. 7, № 7. – С. 942-954. – DOI: 10.46799/syntaxidea.v7i7.13345.
30. Dorogyu Y., Kolisnichenko V. Blockchain transaction analysis: a comprehensive review of applications, tasks and methods // System Research and Information Technologies. – 2023. – № 4. – DOI: 10.20535/SRIT.2308-8893.2023.4.03.
31. Mafrur R. Blockchain Data Analytics: Review and Challenges // arXiv preprint. – 2025. – arXiv:2503.09165. – Режим доступу: <https://arxiv.org/abs/2503.09165>. – Дата звернення: 1 листопада 2025 р.
32. 2025 Mid-year Blockchain Security and AML Report // SlowMist. – 2025. – Режим доступу: [https://www.slowmist.com/report/SlowMist-first-half-of-the-2025-report\(EN\).pdf](https://www.slowmist.com/report/SlowMist-first-half-of-the-2025-report(EN).pdf). – Дата звернення: 1 жовтня 2025.
33. Mehar M., Shier C., Giambattista A., Gong E., Fletcher G., Sanayhie R., Kim H. M., Laskowski M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack // SSRN Electronic Journal. – 2017. – Режим доступу: <https://dx.doi.org/10.2139/ssrn.3014782>. – DOI: 10.2139/ssrn.3014782. – Дата звернення: 1 жовтня 2025.
34. Atzei N., Bartoletti M., Cimoli T. A Survey of Attacks on Ethereum Smart Contracts (SoK) // Principles of Security and Trust (POST'17): Proc. – 2017. – С. 164-186. – DOI: https://doi.org/10.1007/978-3-662-54455-6_8.
35. Poston H. Mapping the OWASP Top Ten to Blockchain // Procedia Computer Science. – 2020. – Т. 177. – С. 613-617. – DOI: 10.1016/j.procs.2020.10.087.
36. Lai E., Luo W. Static Analysis of Integer Overflow of Smart Contracts in Ethereum // Proceedings of the 2020 4th International Conference on Cryptography,

Security and Privacy (ICCSP 2020). – New York: Association for Computing Machinery, 2020. – P. 110–115. – DOI: <https://doi.org/10.1145/3377644.3377650>.

37. AnChain.ai. AnChain.AI 2023 Year-End Web3 Risk Report. – San Jose: AnChain.ai, 2024. – Режим доступу: <https://www.anchain.ai/web3risk2023>. – Дата звернення: 2 листопада 2025 р.

38. Wang H., Liu Z., Ge C. P., Sakurai K., Su C. H. A privacy-preserving data feed scheme for smart contracts // IEICE Transactions on Information and Systems. – 2022. – Т. E105-D, № 2. – С. 195–204. – DOI: <https://doi.org/10.1587/transinf.2021BCI0001>.

39. Farahat I. S., Aladrousy W., Elhoseny M., Elmougy S., Tolba A. E. Improving healthcare applications security using blockchain // Electronics. – 2022. – Т. 11, № 22. – Ст. 3786. – DOI: <https://doi.org/10.3390/electronics11223786>.

40. Meng Q. F., Sun R. G. Towards secure and efficient scientific research project management using consortium blockchain // Journal of Signal Processing Systems. – 2020. – Т. 93 (2–3). – С. 323–332. – DOI: <https://doi.org/10.1007/s11265-020-01529-y>.

41. Xu L., Shah N., Chen L., Diallo N., Gao Z., Lu Y., Shi W. Enabling the sharing economy: Privacy respecting contract based on public blockchain // Proc. ACM Workshop on Blockchain, Cryptocurrencies and Contracts. – 2017. – С. 15–21. – DOI: <https://doi.org/10.1145/3055518.3055527>.

42. Huang J., Han S., You W., Shi W., Liang B., Wu J., Wu Y. Hunting vulnerable smart contracts via graph embedding based bytecode matching // IEEE Transactions on Information Forensics and Security. – 2021. – Т. 16. – С. 2144–2156. – DOI: [10.1109/TIFS.2021.3050051](https://doi.org/10.1109/TIFS.2021.3050051).

43. Wang C., Jiang H., Wang Y., Huang Q., Zuo Z. Research on smart contract vulnerability detection method based on domain features of Solidity contracts and attention mechanism // Journal of Intelligent & Fuzzy Systems. – 2023. – Т., № 45, С. 1513–1525. – DOI: <https://dx.doi.org/10.3233/JIFS-224489>.

44. Alam M. T., Halder R., Maiti A. Detection Made Easy: Potentials of Large Language Models for Solidity Vulnerabilities // arXiv preprint. – 2024. – arXiv:2409.10574. – Режим доступу: <https://arxiv.org/abs/2409.10574>. – Дата звернення: 1 жовтня 2023 р.

45. Hawashin D., Salah K., Jayaraman R., Yaqoob I., Musamih A. A. Blockchain-based solution for mitigating overproduction and underconsumption of medical supplies // IEEE Access. – 2022. – T. 10. – C. 71669-71682. – DOI: <https://doi.org/10.1109/ACCESS.2022.3188778>.
46. Awaji B., Solaiman E. Design, implementation, and evaluation of blockchain-based trusted achievement record system for students in higher education // Proc. 14th Int. Conf. on Computer Supported Education (CSEDU). – 2022. – T. 2. – C. 225-237. – DOI: <https://doi.org/10.5220/0011044200003182>.
47. Upadhyay K., Dantu R., Zaccagni Z., Badruddoja S. Is your legal contract ambiguous? Convert to a smart legal contract // IEEE International Conference on Blockchain: Proc. – 2020. – C. 273-280. – DOI: <https://doi.org/10.1109/Blockchain50366.2020.00041>.
48. Chen Y. W. Blockchain in enterprise: An innovative management scheme utilizing smart contract // Proc. 9th Int. Conf. on Industrial Technology and Management. – 2020. – C. 21-24. – DOI: [10.1109/ICITM48982.2020.9080356](https://doi.org/10.1109/ICITM48982.2020.9080356).
49. Lenarduzzi V., Lunesu M. I., Marchesi M., Tonelli R. Blockchain applications for agile methodologies // Proc. 19th Int. Conf. on Agile Software Development. – 2018. – C. 1-3. – DOI: <https://doi.org/10.1145/3234152.3234155>.
50. Gao J., Liu H., Liu C., Li Q., Guan Z., Chen Z. EasyFlow: Keep Ethereum Away from Overflow // 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). – 2019. – C. 23-26. – DOI: [10.1109/ICSE-Companion.2019.00029](https://doi.org/10.1109/ICSE-Companion.2019.00029).
51. Jiang B., Wang X., Chan W. K., Tse T. H., Li N., Yin Y., Zhang Z. CUD-ASmith: A fuzzer for CUDA compilers // 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). – Madrid, Spain, 13–17 July 2020. – Piscataway, NJ: IEEE. – P. 861–871. – DOI: <https://doi.org/10.1109/COMPSAC48688.2020.0-156>.
52. Akca S., Rajan A., Peng C. SolAnalyser: A Framework for Analysing and Testing Smart Contracts // 26th Asia-Pacific Software Engineering Conference (APSEC). – 2019. – C. 482–489. – DOI: <https://doi.org/10.1109/APSEC48747.2019.00071>.

53. Liu C., Liu H., Cao Z., Chen Z., Chen B., Roscoe B. ReGuard: Finding Reentrancy Bugs in Smart Contracts // Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. – New York: ACM, 2018. – С. 65–68. – DOI: <https://doi.org/10.1145/3183440.3183495>.
54. Lutz O., Chen H., Fereidooni H., Sendner C., Dmitrienko A., Sadeghi A. R., Koushanfar F. ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning // arXiv preprint. – 2021. – arXiv:2103.12607. – Режим доступу: <https://arxiv.org/abs/2103.12607>. – Дата звернення: 1 травня 2025 р.
55. Tann A., Han X. J., Sen Gupta S., Ong Y.-S. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities // arXiv preprint. – 2018. – arXiv:1811.06632. – Режим доступу: <https://arxiv.org/abs/1811.06632>. – Дата звернення: 1 листопада 2024 р.
56. Han L., Liu C., Zhao W., Jiang Y., Sun J. S-gram: towards semantic-aware security auditing for Ethereum smart contracts // Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering. – 2018. – С. 814–819. – DOI: [10.1145/3238147.3240728](https://doi.org/10.1145/3238147.3240728).
57. Kalra S., Goel S., Dhawan M., Sharma S. ZEUS: Analyzing safety of smart contracts // NDSS 2018: Proc. 25th Annual Network and Distributed System Security Symposium. – 2018. – DOI: <https://doi.org/10.14722/ndss.2018.23082>.
58. Tolmach P., Li Y., Lin S.-W., Liu Y., Li Z. A Survey of Smart Contract Formal Specification and Verification // ACM Computing Surveys. – 2022. – Т. 54, № 7. – С. 1–38. – DOI: <https://doi.org/10.1145/3464421>.
59. Tian Z., Chen Y. Differential Testing Solidity Compiler through Deep Contract Manipulation and Mutation. – 2024. – DOI: [10.21203/rs.3.rs-3924118/v1](https://doi.org/10.21203/rs.3.rs-3924118/v1). – DOI: [DOI:10.21203/rs.3.rs-3924118/v1](https://doi.org/10.21203/rs.3.rs-3924118/v1).
60. Susan S.-C. Leveraging Slither and Interval Analysis to build a Static Analysis Tool // arXiv preprint. – 2024. – arXiv:2410.23766. – Режим доступу: <https://arxiv.org/abs/2410.23766>. – Дата звернення: 11 березня 2025 р.
61. Tsankov P., Dan A., Drachsler-Cohen D., Gervais A., Buenzli F., Vechev M. Securify: Practical Security Analysis of Smart Contracts // arXiv preprint. – 2018. –

arXiv:1806.01143. – Режим доступу: <https://arxiv.org/abs/1806.01143>. – Дата звернення: 1 листопада 2025 р.

62. Tikhomirov S., Voskresenskaya E., Ivanitskiy I., Takhaviev R., Marchenko E., Alexandrov Y. SmartCheck: static analysis of Ethereum smart contracts // Proc. 1st Int. Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB'18). – 2018. – С. 9–16. – DOI: <https://doi.org/10.1145/3194113.3194115>.

63. Ferreira J. F., Cruz P., Durieux T., Abreu R. SmartBugs: A Framework to Analyze Solidity Smart Contracts // arXiv preprint. – 2020. – arXiv:2007.04771. – Режим доступу: <https://arxiv.org/abs/2007.04771>. – Дата звернення: 1 листопада 2025 р.

64. Liao J.-W., Tsai T.-T., He C.-K., Tien C.-W. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing // 2019 Sixth Int. Conf. on Internet of Things: Systems, Management and Security (IOTSMS): Proc. – 2019. – С. 458–465. – DOI: [10.1109/IOTSMS48152.2019.8939256](https://doi.org/10.1109/IOTSMS48152.2019.8939256).

65. Luu L., Chu D.-H., Olickel H., Saxena P., Hobor A. Making Smart Contracts Smarter // Proc. 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS'16). – 2016. – С. 254–269. – DOI: <https://doi.org/10.1145/2976749.2978309>.

66. Jiang B., Liu Y., Chan W. K. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection // arXiv preprint. – 2018. – arXiv:1807.03932. – Режим доступу: <https://arxiv.org/abs/1807.03932>. – Дата звернення: 1 листопада 2025 р.

67. Grieco G., Song W., Cygan A., Feist J., Groce A. Echidna: effective, usable, and fast fuzzing for smart contracts // Proc. 29th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2020). – 2020. – С. 557-560. – DOI: <https://doi.org/10.1145/3395363.3404366>.

68. Zeng Q., He J., Zhao G., Li S., Yang J., Tang H., Luo H. Ethergis: a vulnerability detection framework for Ethereum smart contracts based on graph learning features // 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC): Proc. – 2022. – С. 1742-1749. – DOI: [10.1109/COMPSAC54236.2022.00277](https://doi.org/10.1109/COMPSAC54236.2022.00277).

69. Huang Y., Jiang B., Chan W. K. EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection // Proc. 12th Asia-Pacific Symposium on Internetware (Internetware '20). – 2021. – С. 99-109. – DOI: <https://doi.org/10.1145/3457913.3457920>.

70. Eshghie M., Artho C., Gurov D. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning // arXiv preprint. – 2021. – arXiv:2102.07420. – Режим доступа: <https://arxiv.org/abs/2102.07420>. – Дата звернення: 1 березня 2025 р.

71. Su J., Dai H.-N., Zhao L., Zheng Z., Luo X. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing // Proc. 37th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '22). – 2023. – Article 36. – DOI: <https://doi.org/10.1145/3551349.3560429>.

72. Li Z., Kang J., Yu R., Ye D., Deng Q., Zhang Y. Consortium blockchain for secure energy trading in industrial Internet of Things // IEEE Transactions on Industrial Informatics. – 2017. – PP(99): 1-1. – DOI: [10.1109/TII.2017.2786307](https://doi.org/10.1109/TII.2017.2786307).

73. Singh A., Parizi R. M., Zhang Q., Choo K.-K. R., Dehghantanha A. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities // Computers & Security. – 2020. – Т. 88. – Ст. 101654. – DOI: <https://doi.org/10.1016/j.cose.2019.101654>.

74. Vacca A., Di Sorbo A., Visaggio C. A., Canfora G. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges // Journal of Systems and Software. – 2021. – Т. 174. – Ст. 110891. – DOI: <https://doi.org/10.1016/j.jss.2020.110891>.

75. Piantadosi V., Rosa G., Placella D., Scalabrino S., Oliveto R. Detecting functional and security-related issues in smart contracts: A systematic literature review // Software: Practice and Experience. – 2023. – Т. 53, № 2. – С. 465-495. – DOI: [10.1002/spe.3156](https://doi.org/10.1002/spe.3156).

76. Chu H., Zhang P., Dong H., Xiao Y., Ji S., Li W. A survey on smart contract vulnerabilities: Data sources, detection and repair // Information and Software Technology. – 2023. – Т. 159. – Ст. 107221. – DOI: <https://doi.org/10.1016/j.infsof.2023.107221>.

77. Di Angelo M., Durieux T., Ferreira J. F., Salzer G. Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study // Empirical Software Engineering. – 2024. – Т. 29, № 2. – Ст. 41. – DOI: <https://doi.org/10.48550/arXiv.2303.10517>.

78. Kushwaha S. S., Joshi S., Singh D., Kaur M., Lee H.-N. Systematic review of security vulnerabilities in Ethereum blockchain smart contract // IEEE Access. – 2022. – Т. 10. – С. 6605-6621. – DOI: 10.1109/ACCESS.2021.3140091.
79. Kushwaha S. S., Joshi S., Singh D., Kaur M., Lee H.-N. Ethereum smart contract analysis tools: A systematic review // IEEE Access. – 2022. – Т. 10. – С. 57037-57062. – DOI: 10.1109/ACCESS.2022.3169902.
80. Zhou H., Milani Fard A., Mekanju A. The state of Ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support // Journal of Cybersecurity and Privacy. – 2022. – Т. 2, № 2. – С. 358-378. – DOI: <https://doi.org/10.3390/jcp2020019>.
81. Surucu O., Yeprem U., Wilkinson C., Hilal W., Gadsden S. A., Yawney J., Alsadi N., Giuliano A. A survey on Ethereum smart contract vulnerability detection using machine learning // Disruptive Technologies in Information Sciences VI. – 2022. – Vol. 12117. – С. 110–121. – DOI: 10.1117/12.2618899.
82. Jiang F., Chao K., Xiao J., Liu Q., Gu K., Wu J., Cao Y. Enhancing smart-contract security through machine learning: A survey of approaches and techniques // Electronics. – 2023. – Т. 12, № 9. – Ст. 2046. – DOI: <https://doi.org/10.3390/electronics12092046>.
83. Krichen M. Strengthening the security of smart contracts through the power of artificial intelligence // Computers. – 2023. – Т. 12, № 5. – Ст. 107. – DOI: <https://doi.org/10.3390/computers12050107>.
84. Zhang Y., Kasahara S., Shen Y., Jiang X., Wan J. Smart Contract-Based Access Control for the Internet of Things // arXiv preprint. – 2018. – arXiv:1802.04410. – Режим доступа: <https://arxiv.org/abs/1802.04410>. – Дата звернення: 11 листопада 2023 р.
85. Chen J., Xia X., Lo D., Grundy J., Yang X. Maintaining Smart Contracts on Ethereum: Issues, Techniques, and Future Challenges // arXiv preprint. – 2020. – arXiv:2007.00286. – Режим доступа: <https://arxiv.org/abs/2007.00286>. – Дата звернення: 23 вересня 2024 р.
86. Zhuang Y., Liu Z., Qian P., Liu Q. Smart Contract Vulnerability Detection using Graph Neural Network // Proc. IJCAI-2020. – 2020. – С. 3283-3290. – DOI: 10.24963/ijcai.2020/450.

87. El Haddouti S., Khaldoune M., Ayache M., Ech-Cherif El Kettani M. D. Smart contracts auditing and multi-classification using machine learning algorithms: an efficient vulnerability detection in ethereum blockchain // *Computing*. – 2024. – Т. 106, № 9. – С. 2971–3003. – DOI: <https://doi.org/10.1007/s00607-024-01314-w>.
88. Tang X., Du Y., Lai A., Zhang Z. Deep Learning-Based Solution for Smart Contract Vulnerabilities Detection // *Scientific Reports*. – 2023. – DOI: 10.1038/s41598-023-47219-0.
89. Guo D., Ren S., Lu S., та ін. GraphCodeBERT: Pre-training Code Representations with Data Flow // *arXiv preprint*. – 2020. – arXiv:2009.08366. – Режим доступу: <https://arxiv.org/abs/2009.08366>. – Дата звернення: 1 травня 2025 р.
90. Gu X., Yang H., Liu S., Cui Z. Smart Contract Vulnerability Detection Based on Clustering Opcode Instructions // *Proc. SEKE 2023*. – 2023. – DOI: 10.18293/SEKE2023-183.
91. Li P., Li S., Ding M., Yu J., Zhang H., Zhou X., Li J. A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis // *Proc.* – 2022. – С. 366–374. – DOI: 10.1145/3530019.3531342.
92. Sayeed S., Marco-Gisbert H., Caira T. Smart Contract: Attacks and Protections // *IEEE Access*. – 2020. – Т. 8. – С. 24416-24427. – DOI: 10.1109/ACCESS.2020.2970495.
93. Ibba G., Aufiero S., Neykova R., Bartolucci S., Ortu M., Tonelli R., Destefanis G. A Curated Solidity Smart Contracts Repository of Metrics and Vulnerability // *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2024)*. – New York: Association for Computing Machinery, 2024. – С. 32–41. – DOI: <https://doi.org/10.1145/3663533.3664039>.
94. Ghaleb A., Pattabiraman K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection // *Proc. ISSTA 2020*. – 2020. – С. 415–427. – DOI: <https://doi.org/10.1145/3395363.3397385>.
95. Zheng Z., Su J., Chen J., Lo D., Zhong Z., Ye M. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects // *arXiv preprint*. – 2023. – arXiv:2305.08456. – Режим доступу: <https://arxiv.org/abs/2305.08456>. – Дата звернення: 26 грудня 2024 р.

96. Smart Contract Weakness Classification and Test Cases (SWC Registry). – Режим доступу: <https://swcregistry.io/> – Дата звернення: 1 грудня 2023.
97. Chavhan S. Y., Kumar S., Karkare A. ScrawlID: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities // arXiv preprint. – 2022. – arXiv:2202.11409. – Режим доступу: <https://arxiv.org/abs/2202.11409>. – Дата звернення: 3 жовтня 2025 р.
98. So S., Lee M., Park J., Lee H., Oh H. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts // Proc. IEEE Symposium on Security and Privacy (SP). – 2020. – С. 1678–1694. – DOI: 10.1109/SP40000.2020.00032.
99. Xu Z., Li C., Dong X., Zheng Z., Wang H., Zhang J., Chen X. A hybrid neural network model-based approach for detecting smart contract vulnerabilities // Proc. 2023 Int. Conf. on Blockchain Technology and Information Security (ICBCTIS). – 2023. – IEEE. – С. 8–14. – DOI: 10.1109/ICBCTIS59921.2023.00008.
100. Терещенко, О. І. (2025). Методи ін'єкції вразливостей у смарт-контракти для формування збалансованих датасетів. Вісник Херсонського національного технічного університету. № 3(94), Ч. 2, 456-462. – DOI: <https://doi.org/10.35546/kntu2078-4481.2025.3.2.58>
101. Tereshchenko, O., & Komleva, N. (2023). Vulnerability detection of smart contracts based on bidirectional GRU and attention mechanism. In Communications in Computer and Information Science (Vol. 1980). Springer, Cham. – DOI: https://doi.org/10.1007/978-3-031-48325-7_21
102. Dargan S., Kumar M., Ayyagari M. R., Kumar G. A survey of deep learning and its applications: a new paradigm to machine learning // Archives of Computational Methods in Engineering. – 2020. – Т. 27. – С. 1071–1092. – DOI: <https://doi.org/10.1007/s11831-019-09344-w>.
103. Gogineni A. K., Swayamjyoti S., Sahoo D., Sahu K. K., Raj Kishore. Multi-Class classification of vulnerabilities in Smart Contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing // IOP SciNotes. – 2020. – Т. 1, № 3. – Ст. 035002. – DOI: <https://doi.org/10.48550/arXiv.2004.00362>.

104. Gu W., Wang G., Li P., Li X., Zhai G., Li X., Chen M. Detecting Unknown Vulnerabilities in Smart Contracts with Multi-Label Classification Model Using CNN-BiLSTM // Proc. International Conference on Ubiquitous Security. – 2022. – Springer. – С. 52–63. – DOI: 10.21203/rs.3.rs-4589708/v1.
105. Agarap A. F. A Neural Network Architecture Combining Gated Recurrent Unit (GRU) and Support Vector Machine (SVM) for Intrusion Detection in Network Traffic Data // arXiv preprint. – 2017. – arXiv:1709.03082. – Режим доступу: <https://doi.org/10.48550/arXiv.1709.03082>. – Дата звернення: 1 листопада 2025 р.
106. Nosouhian S., Nosouhian F., Khoshouei A. A Review of Recurrent Neural Network Architecture for Sequence Learning: Comparison between LSTM and GRU // Preprints. – 2021. – Препринт: 202107.0252.v1. – DOI: 10.20944/preprints202107.0252.v1. – Дата звернення: 1 листопада 2025 р.
107. Cahuantzi R., Chen X., Güttel S. A comparison of LSTM and GRU networks for learning symbolic sequences // arXiv preprint. – 2021. – arXiv:2107.02248. – Режим доступу: <https://arxiv.org/abs/2107.02248>. – Дата звернення: 1 листопада 2025 р.
108. Deng J., Cheng L., Wang Z. Self-attention-based BiGRU and capsule network for named entity recognition // arXiv preprint. – 2020. – arXiv:2002.00735. – Режим доступу: <https://arxiv.org/abs/2002.00735>. – Дата звернення: 1 листопада 2025 р.
109. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser Ł., Polosukhin I. Attention is all you need // Proc. 31st Int. Conf. on Neural Information Processing Systems (NIPS'17). – 2017. – Red Hook, NY: Curran Associates Inc. – С. 6000–6010. – DOI: <https://doi.org/10.48550/arXiv.1706.03762>.
110. Sun Y., Gu L. Attention-based machine learning model for smart contract vulnerability detection // Journal of Physics: Conference Series. – 2021. – Vol. 1820. – 012004. – IOP Publishing – DOI: 10.1088/1742-6596/1820/1/012004.
111. Galke L., Scherp A. Bag-of-words vs. graph vs. sequence in text classification: Questioning the necessity of text-graphs and the surprising strength of a wide MLP // arXiv preprint. – 2021. – arXiv:2109.03777. – Режим доступу: <https://arxiv.org/abs/2109.03777>. – Дата звернення: 21 серпня 2025 р.

112. Aisopos F., Papadakis G., Varvarigou T. Sentiment analysis of social media content using N-Gram graphs // Proc. 3rd ACM SIGMM Int. Workshop on Social Media (WSM '11). – 2011. – New York, NY: Association for Computing Machinery. – C. 9–14. – DOI: <https://doi.org/10.1145/2072609.2072614>.
113. Jatnika D., Bijaksana M., Ardiyanti A. Word2Vec Model Analysis for Semantic Similarities in English Words // Procedia Computer Science. – 2019. – T. 157. – C. 160–167. – DOI: [10.1016/j.procs.2019.08.153](https://doi.org/10.1016/j.procs.2019.08.153).
114. Tereshchenko, O. I., & Komleva, N. O. (2024). Identification and localization of vulnerabilities in smart contracts using attention vectors analysis in a BERT-based model. Radio Electronics, Computer Science, Control, (3), 173–184. – DOI: <https://doi.org/10.15588/1607-3274-2024-3-15>
115. Powers D. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation // Journal of Machine Learning Technologies. – 2011. – T. 2. – ISSN: 2229-3981. – DOI: [10.9735/2229-3981](https://doi.org/10.9735/2229-3981).
116. Xia Y., Shao H., Deng X. VulCoBERT: A CodeBERT-Based System for Source Code Vulnerability Detection // Proc. 2024 Int. Conf. on Generative Artificial Intelligence and Information Security (GAIIS '24). – 2024. – New York, NY: Association for Computing Machinery. – C. 249–252. – DOI: <https://doi.org/10.1145/3665348.3665391>.
117. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // arXiv preprint. – 2018. – arXiv:1810.04805. – Режим доступу: <https://arxiv.org/abs/1810.04805>. – Дата звернення: 4 липня 2025 р.
118. Sonnekalb T., Gruner B., Brust C.-A., Mäder P. Generalizability of Code Clone Detection on CodeBERT // Proc. 37th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '22). – 2023. – New York, NY: Association for Computing Machinery. – Article 143. – C. 1–3. – DOI: <https://doi.org/10.1145/3551349.3561165>.
119. Feng Z., Guo D., Tang D., та ін. CodeBERT: A Pre-Trained Model for Programming and Natural Languages // arXiv preprint. – 2020. – arXiv:2002.08155. – Режим доступу: <https://arxiv.org/abs/2002.08155>. – Дата звернення: 1 листопада 2025 р.

120. Liu Y., Ott M., Goyal N., та ін. RoBERTa: A Robustly Optimized BERT Pretraining Approach // arXiv preprint. – 2019. – arXiv:1907.11692. – Режим доступу: <https://arxiv.org/abs/1907.11692>. – Дата звернення: 1 листопада 2024 р.
121. Wu H., Zhang Z., Wang S., Lei Y., Lin B., Qin Y., Zhang H., Mao X. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques // 2021 IEEE 32nd Int. Symposium on Software Reliability Engineering (ISSRE): Proc. – 2021. – С. 378–389. – DOI: 10.1109/ISSRE52982.2021.00047.
122. Zhao H., Su P., Qiu M. A novel machine learning-based model for reentrant vulnerabilities detection // Smart Computing and Communication: 7th International Conference, SmartCom 2022, New York City, NY, USA, November 18–20, 2022: Proceedings. – Cham: Springer, 2023. – (Lecture Notes in Computer Science, vol. 13828). – P. 290–299. – DOI: https://doi.org/10.1007/978-3-031-28124-2_27.
123. Zhou K., Huang J., Han H., Gong B., Xiong A., Wang W., Wu Q. Smart contracts vulnerability detection model based on adversarial multi-task learning // Journal of Information Security and Applications. – 2023. – Т. 77. – Ст. 103555. – DOI: <https://doi.org/10.1016/j.jisa.2023.103555>.
124. Narayana K. L., Sathiyamurthy K. Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning // Materials Today: Proceedings. – 2023. – Т. 81. – С. 653-659. – DOI: <https://doi.org/10.1016/j.matpr.2021.04.125>.
125. Терещенко, О. І. (2025). Виявлення та локалізація вразливостей у смарт-контрактах на основі дворівневої уваги з графовим підкріпленням. Вісник Кременчуцького національного університету імені Михайла Остроградського, 4, 272-281, – DOI: <https://doi.org/10.32782/1995-0519.2025.4.30>
126. Терещенко, О. І. (2025). Комбінований метод локалізації вразливостей у смарт-контрактах на основі Attention та Grad-CAM. Таврійський науковий вісник. Серія: Технічні науки, 1(4), 306-316. – DOI: <https://doi.org/10.32782/tnv-tech.2025.4.1.30>

ДОДАТОК А АКТИ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОГО ДОСЛІДЖЕННЯ



DISKRET

НАУКОВО-ВИРОБНИЧЕ ОБ'ЄДНАННЯ

Тел. (48) 7001225, Сайт: www.diskret.com.ua, E-mail: info@diskret.com.ua

АКТ

впровадження результатів дисертаційного дослідження
на здобуття наукового ступеню доктора філософії
за спеціальністю 122 Комп'ютерні науки
«Методи і моделі машинного навчання для виявлення та класифікації вразливостей смарт-контрактів»
здобувача Терещенка Олександра Ігоровича
у науково-практичну діяльність ТОВ НВО «Діскрет»

Ми, що нижче підписалися, представник НВО «Діскрет» директор Стаценко Г.С., з одного боку, та представники Навчально-наукового інституту комп'ютерних систем (ННІКС) директор ЦІКТ «ІТ інкубатор», к.т.н., доц. П.А. Швагірев та заступник директора з наукової роботи ННІКС, к.т.н., доц. А.Г. Кісіль, з другого боку, склали цей акт у підтвердження впровадження у науково-практичну діяльність ТОВ НВО «Діскрет» програмно-алгоритмічного забезпечення автоматизованого виявлення та локалізації вразливостей у смарт-контрактах, яке міститься у дисертації здобувача Терещенка Олександра Ігоровича, а саме методів виявлення та інтерпретації вразливостей на основі глибинної моделі CodeBERT-GDLA з використанням уваги та Grad-CAM. Перелічені методи застосовано для покращення програмного забезпечення аудиту безпеки смарт-контрактів.

В процесі впровадження прикладних розробок, які містяться в дисертаційній роботі здобувача Терещенка О.І., з 01.07.2025 по 26.08.2025 встановлено, що надане програмне забезпечення щодо аналізу та локалізації вразливостей у коді смарт-контрактів мовою Solidity дозволило значно підвищити ефективність процесів аудиту та точність виявлення потенційних уразливостей. Завдяки використанню пояснених моделей машинного навчання та механізмів на основі уваги запропоновані методи дозволили зменшити кількість пропущених вразливостей, скоротити час перевірки контрактів, чим забезпечили підвищення рівня безпеки та прозорості аудиту програмних рішень.

Акт видано зв'язку з захистом дисертації і він не є основою для фінансових розрахунків.

Від ТОВ НВО «Діскрет»



02 вересня 2025 р.
Адреса 65003, Одеса,
вул. Балківська, 130
ТОВ НВО «Діскрет»
Тел. (8-048) 733-6860

Від Національного університету «Одеська політехніка»



Директор ЦІКТ «ІТ інкубатор», к.т.н., доц.

Павло ШВАГІРЕВ

Заст. директора з наукової роботи ННІКС, к.т.н., доц.

Анатолій КІСІЛЬ

04 вересня 2025 р.
Адреса 65044, Одеса,
пр. Шевченка, 1
Національний університет «Одеська політехніка»
Тел. (8-048) 705-8476

ДОВІДКА

про впровадження результатів дисертаційної роботи

Терещенка Олександра Ігоровича

Тема: «Методи і моделі машинного навчання для виявлення та класифікації вразливостей
смарт-контрактів»

Довідка видана стосовно того, що у діяльності науково-виробничого підприємства «КАРЕ» (м. Одеса) використані результати, що отримані у дисертаційній роботі здобувачем кафедри інженерії програмного забезпечення Національного університету «Одеська Політехніка» МОН України Терещенком Олександром Ігоровичем.

Прийнято до впровадження:

- калібрувальний компонент для узгодження імовірнісних оцінок моделей виявлення вразливостей. Використання запропонованого методу дозволило зменшити розбіжність між заявленою впевненістю моделі та фактичною точністю прогнозів, що сприяло підвищенню довіри до результатів автоматизованого аудиту безпеки.
- модуль пріоритизації знахідок на основі каліброваних імовірностей та показників упевненості локалізації. Використання запропонованого методу дозволило автоматизувати процес визначення критичності виявлених вразливостей, використовуючи пояснювані карти для виділення важливих фрагментів коду, що суттєво скоротило час аналізу та перевірки.

Запропоновані Терещенком Олександром Ігоровичем науково-технічні рішення дозволили підвищити ефективність систем аналізу та аудиту безпеки програмного забезпечення, які потребують точного виявлення і локалізації вразливостей у коді, що розробляються і використовуються на нашому підприємстві.

Очікуваний ефект має такі складові:

- підвищення точності та стабільності прогнозів моделей виявлення вразливостей;
- зменшення кількості пропущених вразливостей і помилкових спрацювань;
- скорочення часу аудиту та локалізації небезпечних ділянок коду;
- стабілізація порогів прийняття рішень і підвищення пояснюваності результатів;
- загальне підвищення рівня безпеки програмних продуктів.

Директор НВП «КАРЕ»



Леонід БУРДИКА

**ДОДАТОК Б ДОВІДКА ПРО ВИКОРИСТАННЯ РЕЗУЛЬТАТІВ
ДИСЕРТАЦІЙНОЇ РОБОТИ У НАВЧАЛЬНОМУ ПРОЦЕСІ
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ «ОДЕСЬКА ПОЛІТЕХНІКА»**

«ЗАТВЕРДЖУЮ»

Перший проректор, проректор з
науково-педагогічної та виховної
роботи Національного університету

«Одеська політехніка»,
професор **Сергій НЕСТЕРЕНКО**



ДОВІДКА

про впровадження результатів дисертації доктора філософії

ТЕРЕЩЕНКА Олександра Ігоровича

**«Методи і моделі машинного навчання для виявлення та класифікації
вразливостей смарт-контрактів»**

у навчальному процесі Національного університету «Одеська політехніка»

Чинна довідка видана в тому, що в навчальних курсах бакалаврської підготовки «Методи та системи штучного інтелекту» і «Проектування інформаційних систем», які викладаються студентам за спеціальністю 122 – «Комп'ютерні науки» на кафедрі інформаційних систем Інституту комп'ютерних систем Національного університету «Одеська політехніка», використовуються наукові результати, одержані в дисертаційній роботі Терещенка О. І.

Основні наукові та практичні результати, що знайшли застосування:

1. Метод локалізації та інтерпретації вразливостей у коді на основі attention-карт і аналізу важливості токенів/рядків, що забезпечує пояснюваність результатів та точкову індикацію місця ураження.
 - тема «Навчання зі зворотним поширенням помилки (back-propagation learning)» – у межах дисципліни «Методи та системи штучного інтелекту»;

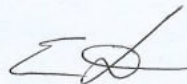
- тема «Архітектурне проектування програмної системи» – у межах дисципліни «Проектування інформаційних систем».
- 2. Методика формування та розмітки датасетів для задач безпеки смарт-контрактів: детерміноване інжектування типових уразливостей, балансування класів, валідаційні сценарії та контроль якості даних.
 - тема «Регресійний аналіз та машинне навчання» – у межах дисципліни «Методи та системи штучного інтелекту»;
 - тема «Формування функціональних та нефункціональних вимог» – у межах дисципліни «Проектування інформаційних систем».
- 3. Технологічні засади інтеграції інструмента в навчальні та виробничі процеси: сценарії використання у практичних роботах (розгортання сервісу, тестування на відкритих колекціях контрактів, інтерпретація звітів, формування рекомендацій щодо рефакторингу).
 - тема «Інтелектуальні завдання та інтелектуальні системи» – у межах дисципліни «Методи та системи штучного інтелекту»;
 - тема «Аналіз аналогів. Вибір інструментів для проектування» – у межах дисципліни «Проектування інформаційних систем».
- 4. Навчальний процес підтримано підготовленими Терещенком О.І. розділами конспекту лекцій та методичними вказівками до виконання лабораторних робіт з дисципліни "Методи та системи штучного інтелекту", а також темами курсових проєктів з дисципліни «Проектування інформаційних систем».

Директор
Інституту комп'ютерних
систем, д.т.н., професор



Світлана АНТОШУК

Завідувач кафедри
інформаційних систем
д.т.н., професор



Олена АРСІРІЙ

ДОДАТОК В ФРАГМЕНТИ ВИХІДНОГО КОДУ ПРОГРАМНИХ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import json
try:
    import lightning.pytorch as pl
except ImportError:
    try:
        import pytorch_lightning as pl
    except ImportError:
        raise ImportError("Neither lightning.pytorch nor pytorch_lightning could be
imported")
from transformers import AutoModel, get_linear_schedule_with_warmup
from torch.optim import AdamW
import torchmetrics
import seaborn as sns
import matplotlib.pyplot as plt
from typing import Optional, Tuple, List, Dict, Any

from .GDLA_Modules import (
    GraphAwareTokenAttention,
    GraphAwareBlockAttention,
    CodeBlockExtractor,
    GraphBuilder
)
from .GDLA_Advanced_Losses import CausalLosses, RationalExtractor,
ExplainabilityMetrics, CalibrationMetrics, CalibrationLoss
from .GDLA_Advanced_Graphs import AdvancedGraphBuilder, FunctionLevelGraphBuilder,
GraphNormalizer
from .GDLA_Vulnerability_Localization import VulnerabilityLocalizer

class CodeBERT_GDLA(pl.LightningModule):
    """
    CodeBERT with Graph-Aware Dual-Level Attention (G-DLA) architecture.

    This model combines CodeBERT with hierarchical attention mechanisms that
    incorporate graph structural information (AST/CFG/DFG) for better
    vulnerability detection and localization.
    """

    def __init__(self,
                 n_classes: int,
                 steps_per_epoch: int,
                 n_epochs: int,
                 n_gru_layers: int = 1,
                 gru_hidden_size: int = 16,
                 use_gru: bool = True,
                 beta_init: float = 0.1,
                 gamma_init: float = 0.1,
                 max_window_size: int = 64,
                 graph_bias_method: str = 'connectivity',
                 use_causal_losses: bool = False,
                 lambda_suf: float = 0.1,
                 lambda_del: float = 0.1,
                 lambda_cam: float = 0.05):

```

```

super().__init__()

# Save hyperparameters
self.save_hyperparameters()

# CodeBERT encoder
self.codebert = AutoModel.from_pretrained("microsoft/codebert-base")
self.bert = self.codebert # Backward compatibility

# Optional BiGRU for additional context
self.use_gru = use_gru
if use_gru:
    self.gru = nn.GRU(
        self.codebert.config.hidden_size,
        gru_hidden_size,
        num_layers=n_gru_layers,
        batch_first=True,
        bidirectional=True
    )
    self.hidden_size = 2 * gru_hidden_size
else:
    self.gru = None
    self.hidden_size = self.codebert.config.hidden_size

# G-DLA components
self.token_attention = GraphAwareTokenAttention(
    d_in=self.hidden_size,
    d_ctx=self.hidden_size,
    d_e=16, # Edge embedding dimension
    beta_init=beta_init
)

self.block_attention = GraphAwareBlockAttention(
    d_in=self.hidden_size,
    d_ctx=self.hidden_size,
    d_e=16, # Edge embedding dimension
    gamma_init=gamma_init
)

# Utility classes
self.block_extractor = CodeBlockExtractor()
self.graph_builder = GraphBuilder(max_window_size=max_window_size) # Keep
for backward compatibility
self.advanced_graph_builder = AdvancedGraphBuilder(
    max_window_size=max_window_size,
    use_edge_types=True
)
self.function_graph_builder = FunctionLevelGraphBuilder()
self.graph_bias_method = graph_bias_method

# Output classifier
self.classifier = nn.Linear(self.hidden_size, n_classes)

# Loss function
self.criterion = nn.BCEWithLogitsLoss()

# Causal losses for explainability
self.use_causal_losses = use_causal_losses
if use_causal_losses:
    self.causal_losses = CausalLosses(lambda_suf, lambda_del, lambda_cam)
    self.rational_extractor = RationalExtractor(top_k_ratio=0.3)
    self.explainability_metrics = ExplainabilityMetrics()
    self.calibration_metrics = CalibrationMetrics()

```

```

# Calibration loss for L_cal regularization
self.use_calibration_loss = True # Always enable calibration
self.calibration_loss = CalibrationLoss(
    lambda_cal=0.1, # Weight for calibration loss
    temperature=1.0, # Initial temperature
    n_bins=15, # Number of bins for ECE
    focal_alpha=0.25, # Focal loss alpha
    focal_gamma=2.0, # Focal loss gamma
    label_smoothing=0.1 # Label smoothing factor
)

# Vulnerability localization module
self.vulnerability_localizer = VulnerabilityLocalizer(
    hidden_size=self.hidden_size,
    lambda_param=0.5, #  $\lambda$  parameter for combining attention and Grad-CAM
    window_size=5, # Sliding window size
    top_k=5 # Top-k for evaluation
)

# Training parameters
self.steps_per_epoch = steps_per_epoch
self.n_epochs = n_epochs
self.n_classes = n_classes
self.validation_step_outputs = []

# Metrics
self.train_accuracy = torchmetrics.Accuracy(
    num_labels=n_classes, average='macro', task='multilabel'
)
self.val_accuracy = torchmetrics.Accuracy(
    num_labels=n_classes, average='macro', task='multilabel'
)
self.test_accuracy = torchmetrics.Accuracy(
    num_labels=n_classes, average='macro', task='multilabel'
)

# Store attention weights for visualization
self.token_attention_weights = []
self.block_attention_weights = []

def to(self, device):
    """Override to ensure metrics are moved to the correct device."""
    super().to(device)
    # Move metrics to the same device
    self.train_accuracy = self.train_accuracy.to(device)
    self.val_accuracy = self.val_accuracy.to(device)
    self.test_accuracy = self.test_accuracy.to(device)
    return self

def load_state_dict(self, state_dict, strict=True):
    """
    Custom load_state_dict to handle backward compatibility with old checkpoints.
    """
    # Create a new state dict with corrected keys
    new_state_dict = {}

    for key, value in state_dict.items():
        # Convert old 'bert.' keys to 'codebert.' keys
        if key.startswith('bert.'):
            new_key = key.replace('bert.', 'codebert.', 1)
            new_state_dict[new_key] = value
        # Also keep the original key for backward compatibility

```

```

        new_state_dict[key] = value
    else:
        new_state_dict[key] = value

    # Call the parent load_state_dict with the modified state dict
    return super().load_state_dict(new_state_dict, strict=False)

def _extract_code_structure(self, input_ids: torch.Tensor,
                             attention_mask: torch.Tensor) -> Dict[str, Any]:
    """
    Extract code structure information including blocks and graphs using advanced
    graph building.

    Args:
        input_ids: Input token IDs (batch_size, seq_len)
        attention_mask: Attention mask (batch_size, seq_len)

    Returns:
        Dictionary containing structural information
    """
    batch_size = input_ids.shape[0]
    structural_info = {
        'blocks': [],
        'token2block': [],
        'A_tok': [],
        'A_blk': [],
        'graph_bias_tok': [],
        'graph_bias_blk': [],
        'edge_types_tok': [],
        'edge_types_blk': [],
        'edge_embeddings_tok': [],
        'edge_embeddings_blk': [],
        'functions': []
    }

    for i in range(batch_size):
        # Convert input_ids to tokens (simplified - in practice use tokenizer)
        tokens = [str(id.item()) for id in input_ids[i]]

        # Extract blocks
        block_info = self.block_extractor.extract_blocks(tokens,
attention_mask[i])

        # Initialize edge type variables
        edge_types_tok = None
        edge_types_blk = None
        edge_embeddings_tok = None
        edge_embeddings_blk = None

        # Build advanced graphs (AST, CFG, DFG)
        try:
            # Use advanced graph builder for better structural information
            advanced_graphs = self.advanced_graph_builder.build_combined_graph(
                tokens, attention_mask[i]
            )

            # Use combined graph for token-level attention
            A_tok = advanced_graphs['combined_graph']

            # For block-level attention, we'll use a simplified approach
            # since the advanced builder doesn't directly provide block graphs
            num_blocks = len(block_info['blocks'])
            A_blk = torch.zeros(num_blocks, num_blocks)

```

```

        # Create simple sequential connections between blocks
        for j in range(num_blocks - 1):
            A_blk[j, j + 1] = 1.0
            A_blk[j + 1, j] = 1.0

        # Get edge embeddings if available
        if self.advanced_graph_builder.use_edge_types:
            # Get edge embeddings and aggregate per token
            edge_embeddings_matrix =
self.advanced_graph_builder.get_edge_type_embeddings(A_tok)
            if edge_embeddings_matrix is not None:
                # Aggregate incoming edge embeddings for each token
                edge_embeddings_tok = torch.sum(edge_embeddings_matrix,
dim=1) # (seq_len, 16)
            else:
                edge_embeddings_tok = torch.zeros(A_tok.shape[0], 16,
device=A_tok.device)
            edge_embeddings_blk = torch.zeros(num_blocks, 16,
device=A_tok.device) # Default embeddings for blocks
        else:
            edge_embeddings_tok = torch.zeros(A_tok.shape[0], 16,
device=A_tok.device)
            edge_embeddings_blk = torch.zeros(num_blocks, 16,
device=A_tok.device)

        # Compute graph bias
        graph_bias_tok =
self.advanced_graph_builder.compute_graph_bias(A_tok, self.graph_bias_method)
        graph_bias_blk =
self.advanced_graph_builder.compute_graph_bias(A_blk, self.graph_bias_method)

    except Exception as e:
        # Fallback to simple graph builder if advanced builder fails
        print(f"Advanced graph building failed for sample {i}, using
fallback: {e}")
        A_tok = self.graph_builder.build_token_graph(tokens,
attention_mask[i])
        A_blk = self.graph_builder.build_block_graph(
            block_info['blocks'], tokens, attention_mask[i]
        )
        graph_bias_tok = self.graph_builder.compute_graph_bias(
            A_tok, self.graph_bias_method
        )
        graph_bias_blk = self.graph_builder.compute_graph_bias(
            A_blk, self.graph_bias_method
        )
        edge_types_tok = None
        edge_types_blk = None
        edge_embeddings_tok = torch.zeros(A_tok.shape[0], 16,
device=A_tok.device)
        edge_embeddings_blk = torch.zeros(A_blk.shape[0], 16,
device=A_blk.device)

        # Extract function-level information (optional third hierarchy level)
        try:
            functions = self.function_graph_builder.extract_functions(tokens,
attention_mask[i])
        except Exception as e:
            print(f"Function extraction failed for sample {i}: {e}")
            functions = []

        # Store information

```

```

        structural_info['blocks'].append(block_info['blocks'])
        structural_info['token2block'].append(block_info['token2block'])
        structural_info['A_tok'].append(A_tok)
        structural_info['A_blk'].append(A_blk)
        structural_info['graph_bias_tok'].append(graph_bias_tok)
        structural_info['graph_bias_blk'].append(graph_bias_blk)
        structural_info['edge_types_tok'].append(edge_types_tok)
        structural_info['edge_types_blk'].append(edge_types_blk)
        structural_info['edge_embeddings_tok'].append(edge_embeddings_tok)
        structural_info['edge_embeddings_blk'].append(edge_embeddings_blk)
        structural_info['functions'].append(functions)

    return structural_info

def _apply_token_attention(self,
                           H_tok: torch.Tensor,
                           structural_info: Dict[str, Any],
                           batch_idx: int) -> Tuple[torch.Tensor,
List[torch.Tensor]]:
    """
    Apply graph-aware token attention to each block.

    Args:
        H_tok: Token representations (batch_size, seq_len, hidden_size)
        structural_info: Structural information from _extract_code_structure
        batch_idx: Current batch index

    Returns:
        Z_blocks: Block representations
        token_attention_weights: Token attention weights for each block
    """
    blocks = structural_info['blocks'][batch_idx]
    token2block = structural_info['token2block'][batch_idx]
    graph_bias_tok = structural_info['graph_bias_tok'][batch_idx]
    edge_embeddings_tok = structural_info['edge_embeddings_tok'][batch_idx]

    Z_blocks = []
    token_attention_weights = []

    for block_idx, (start, end) in enumerate(blocks):
        # Get tokens in this block
        block_token_indices = [
            i for i, block_id in enumerate(token2block)
            if block_id == block_idx
        ]

        if not block_token_indices:
            # Empty block, use zero representation
            Z_blocks.append(torch.zeros(H_tok.shape[-1], device=H_tok.device))
            token_attention_weights.append(torch.tensor([]))
            continue

        # Extract token representations for this block
        H_block = H_tok[block_token_indices] # (N_tok, hidden_size)

        # Compute block context (average of block tokens)
        ctx_block = H_block.mean(dim=0) # (hidden_size,)

        # Get graph bias for tokens in this block
        block_graph_bias = graph_bias_tok[block_token_indices].to(H_tok.device)

    # (N_tok,)

    # Get edge embeddings for tokens in this block if available

```

```

        edge_feat_tok = None
        if edge_embeddings_tok is not None:
            edge_feat_tok =
edge_embeddings_tok[block_token_indices].to(H_tok.device) # (N_tok, d_e)

        # Create mask for valid tokens in block
        mask_tok = torch.ones(len(block_token_indices), dtype=torch.bool,
device=H_tok.device)

        # Apply token attention with edge features
        z_b, alpha = self.token_attention(
            H_block, ctx_block,
            edge_feat_tok=edge_feat_tok,
            graph_bias_tok=block_graph_bias,
            mask_tok=mask_tok
        )

        Z_blocks.append(z_b)
        token_attention_weights.append(alpha)

    # Stack block representations
    Z_blocks = torch.stack(Z_blocks) # (num_blocks, hidden_size)

    return Z_blocks, token_attention_weights

def _apply_block_attention(self,
                            Z_blocks: torch.Tensor,
                            structural_info: Dict[str, Any],
                            batch_idx: int) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Apply graph-aware block attention.

    Args:
        Z_blocks: Block representations (num_blocks, hidden_size)
        structural_info: Structural information
        batch_idx: Current batch index

    Returns:
        z_prog: Program representation (hidden_size,)
        block_attention_weights: Block attention weights (num_blocks,)
    """
    # Compute global context (average of all blocks)
    ctx_prog = Z_blocks.mean(dim=0) # (hidden_size,)

    # Get graph bias for blocks
    graph_bias_blk =
structural_info['graph_bias_blk'][batch_idx].to(Z_blocks.device) # (num_blocks,)

    # Get edge embeddings for blocks if available
    edge_embeddings_blk = structural_info['edge_embeddings_blk'][batch_idx]
    edge_feat_blk = None
    if edge_embeddings_blk is not None:
        edge_feat_blk = edge_embeddings_blk.to(Z_blocks.device) # (num_blocks,
d_e)

    # Create mask for valid blocks
    mask_blk = torch.ones(Z_blocks.shape[0], dtype=torch.bool,
device=Z_blocks.device)

    # Apply block attention with edge features
    z_prog, beta = self.block_attention(
        Z_blocks, ctx_prog,
        edge_feat_blk=edge_feat_blk,

```

```

        graph_bias_blk=graph_bias_blk,
        mask_blk=mask_blk
    )

    return z_prog, beta

def forward(self,
            input_ids: torch.Tensor,
            attention_mask: torch.Tensor,
            return_attention: bool = False) -> torch.Tensor:
    """
    Forward pass through the G-DLA model.

    Args:
        input_ids: Input token IDs (batch_size, seq_len)
        attention_mask: Attention mask (batch_size, seq_len)
        return_attention: Whether to return attention weights

    Returns:
        logits: Classification logits (batch_size, n_classes)
        attention_weights: Optional attention weights for visualization
    """
    batch_size = input_ids.shape[0]

    # Get CodeBERT representations
    bert_output = self.codebert(
        input_ids=input_ids,
        attention_mask=attention_mask,
        output_attentions=return_attention
    )

    hidden_states = bert_output.last_hidden_state # (batch_size, seq_len,
hidden_size)

    # Apply BiGRU if enabled
    if self.use_gru:
        gru_output, _ = self.gru(hidden_states)
        H_tok = gru_output # (batch_size, seq_len, 2*gru_hidden_size)
    else:
        H_tok = hidden_states # (batch_size, seq_len, hidden_size)

    # Extract code structure
    structural_info = self._extract_code_structure(input_ids, attention_mask)

    # Process each sample in the batch
    batch_logits = []
    batch_token_attention = []
    batch_block_attention = []

    for i in range(batch_size):
        # Apply token-level attention
        Z_blocks, token_attention_weights = self._apply_token_attention(
            H_tok[i], structural_info, i
        )

        # Apply block-level attention
        z_prog, block_attention_weights = self._apply_block_attention(
            Z_blocks, structural_info, i
        )

        # Classify
        logits = self.classifier(z_prog) # (n_classes,)

```



```

dtype=torch.float32)

        # If conversion fails, create a default tensor
        attn_tensor = torch.tensor([0.0], device=device,

# Pad with zeros to max_len
padded = torch.zeros(max_len, device=device)
padded[:len(attn_tensor)] = attn_tensor
padded_attention.append(padded)
    else:
        padded_attention.append(torch.zeros(max_len,
device=input_ids.device))

        token_attention = torch.stack(padded_attention).mean(dim=0)
    else:
        # Fallback: create a simple attention vector
        token_attention = torch.ones(input_ids.shape[1],
device=input_ids.device) / input_ids.shape[1]

# Normalize attention weights
gradcam_weights = F.softmax(token_attention, dim=-1)

return gradcam_weights

def training_step(self, batch, batch_idx):
    """Training step with optional causal losses."""
    input_ids = batch["input_ids"]
    attention_mask = batch["attention_mask"]
    labels = batch["labels"].float()

# Forward pass with attention weights
outputs, attention_weights = self(input_ids, attention_mask,
return_attention=True)
classification_loss = self.criterion(outputs, labels)

total_loss = classification_loss

# Add calibration loss (L_cal regularization)
if self.use_calibration_loss:
    cal_loss, cal_components =
self.calibration_loss.compute_calibration_loss(
    outputs, labels, loss_type='combined'
)
    total_loss += cal_loss

# Log calibration loss components
self.log('train_calibration_loss',
cal_components['total_calibration_loss'], prog_bar=True)
    self.log('train_ece_loss', cal_components.get('ece_loss', 0.0),
prog_bar=True)
    self.log('train_temperature', cal_components.get('temperature', 1.0),
prog_bar=True)

# Add causal losses if enabled
if self.use_causal_losses:
    # Extract rational tokens - FIXED: handle variable length attention
weights
    token_attention_list = attention_weights['token_attention']

# Convert list of tensors to a single tensor by padding
max_len = max(len(attn) for attn in token_attention_list if len(attn) >
0)

    if max_len > 0:
        # Pad all attention weights to the same length

```

```

padded_attention = []
for attn in token_attention_list:
    if len(attn) > 0:
        # Check if attn is a tensor or list
        if isinstance(attn, torch.Tensor):
            device = attn.device
            attn_tensor = attn
        else:
            # Convert list to tensor - handle both single values and
lists
            device = input_ids.device
            if isinstance(attn, (list, tuple)):
                # If it's a list of numbers, convert to tensor
                try:
                    attn_tensor = torch.tensor(attn, device=device,
dtype=torch.float32)
                except (ValueError, TypeError):
                    # If conversion fails, create a default tensor
                    attn_tensor = torch.tensor([0.0], device=device,
dtype=torch.float32)
            else:
                # If it's a single number, create a tensor with one
element
                try:
                    attn_tensor = torch.tensor([float(attn)],
device=device, dtype=torch.float32)
                except (ValueError, TypeError):
                    # If conversion fails, create a default tensor
                    attn_tensor = torch.tensor([0.0], device=device,
dtype=torch.float32)

            # Pad with zeros to max_len
            padded = torch.zeros(max_len, device=device)
            padded[:len(attn_tensor)] = attn_tensor
            padded_attention.append(padded)
        else:
            padded_attention.append(torch.zeros(max_len,
device=input_ids.device))

    token_attention = torch.stack(padded_attention).mean(dim=0)
else:
    # Fallback: create a simple attention vector
    token_attention = torch.ones(input_ids.shape[1],
device=input_ids.device) / input_ids.shape[1]

    rational_input_ids, rational_attention_mask =
self.rational_extractor.extract_rational_tokens(
    input_ids, token_attention, attention_mask
)

    # Create deletion input
    deletion_input_ids, deletion_attention_mask =
self.rational_extractor.create_deletion_input(
    input_ids, token_attention, attention_mask
)

    # Forward passes for rational and deletion
    rational_outputs = self(rational_input_ids, rational_attention_mask)
    deletion_outputs = self(deletion_input_ids, deletion_attention_mask)

    # Compute Grad-CAM weights (simplified version)
    gradcam_weights = self._compute_gradcam_weights(input_ids,
attention_mask, outputs, labels)

```

```

        # Compute causal losses
        causal_loss, causal_components =
self.causal_losses.compute_total_causal_loss(
            outputs, rational_outputs, deletion_outputs,
            token_attention, gradcam_weights, labels
        )

        total_loss = classification_loss + causal_loss

        # Log causal losses
        self.log('train_sufficiency_loss', causal_components['sufficiency_loss'],
prog_bar=True)
        self.log('train_deletion_loss', causal_components['deletion_loss'],
prog_bar=True)
        self.log('train_cam_consistency_loss',
causal_components['cam_consistency_loss'], prog_bar=True)

        preds = torch.sigmoid(outputs)
        self.log('train_loss', total_loss, prog_bar=True, logger=True)
        self.log('train_accuracy', self.train_accuracy(preds, labels),
            on_epoch=True, prog_bar=True, logger=True)

        return total_loss

def validation_step(self, batch, batch_idx):
    """Validation step with calibration metrics."""
    input_ids = batch["input_ids"]
    attention_mask = batch["attention_mask"]
    labels = batch["labels"].float()

    outputs = self(input_ids, attention_mask)
    loss = self.criterion(outputs, labels)

    preds = torch.sigmoid(outputs)
    binarized_preds = (preds > 0.5).int()

    # Compute calibration metrics
    if self.use_calibration_loss:
        cal_metrics = self.calibration_loss.get_calibration_metrics(outputs,
labels)

        # Log calibration metrics
        self.log('val_ece_average', cal_metrics['ece_average'], prog_bar=True)
        self.log('val_temperature', cal_metrics['temperature'], prog_bar=True)
        self.log('val_overconfidence', cal_metrics['overconfidence'],
prog_bar=True)
        self.log('val_underconfidence', cal_metrics['underconfidence'],
prog_bar=True)

        self.log('val_loss', loss, prog_bar=True, logger=True)
        self.log('val_accuracy', self.val_accuracy(binarized_preds, labels),
            on_epoch=True, prog_bar=True, logger=True)

        self.validation_step_outputs.append({'pred': binarized_preds, 'target':
labels})
        return {'val_loss': loss}

def test_step(self, batch, batch_idx):
    """Test step."""
    input_ids = batch["input_ids"]
    attention_mask = batch["attention_mask"]
    labels = batch["labels"].float()

```

```

outputs = self(input_ids, attention_mask)
loss = self.criterion(outputs, labels)

preds = torch.sigmoid(outputs)
binarized_preds = (preds > 0.5).int()

self.log('test_loss', loss, prog_bar=True, logger=True)
self.log('test_accuracy', self.test_accuracy(binarized_preds, labels),
        on_step=True, on_epoch=True, prog_bar=True, logger=True)

return loss

def configure_optimizers(self):
    """Configure optimizer and scheduler."""
    optimizer = AdamW(self.parameters(), lr=2e-5)

    if self.steps_per_epoch and self.n_epochs:
        scheduler = get_linear_schedule_with_warmup(
            optimizer,
            num_warmup_steps=self.steps_per_epoch,
            num_training_steps=self.steps_per_epoch * self.n_epochs
        )
        return [optimizer], [scheduler]

    return optimizer

def on_validation_epoch_end(self):
    """End of validation epoch - compute confusion matrices."""
    predictions = torch.cat([x['pred'] for x in self.validation_step_outputs])
    targets = torch.cat([x['target'] for x in self.validation_step_outputs])

    # Clear the list after use
    self.validation_step_outputs = []

    # Convert to int
    predictions = (predictions > 0.5).int()
    targets = targets.int()

    # Using MultilabelConfusionMatrix
    confusion_matrix_metric =
torchmetrics.classification.MultilabelConfusionMatrix(
    num_labels=self.n_classes
).to(predictions.device)
cm = confusion_matrix_metric(predictions, targets)

    # List of vulnerability names (in the same order as in the dataset!)
    vulnerability_labels = [
        "integer_underflow",
        "integer_overflow",
        "timestamp_dependency",
        "re-entrancy_vulnerability"
    ]

    # Create graphs for each vulnerability
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
    axes = axes.ravel()

    for i in range(self.n_classes):
        if i < len(vulnerability_labels):
            sns.heatmap(cm[i].detach().cpu().numpy(), annot=True, fmt="d",
                        linewidths=.5, square=True, cmap='Blues_r', ax=axes[i])
            axes[i].set_title(f'{vulnerability_labels[i]}')

```

```

        axes[i].set_xlabel('Predicted')
        axes[i].set_ylabel('Actual')

    plt.tight_layout()
    self.logger.experiment.add_figure("Multi-label Confusion Matrices",
plt.gcf(), self.current_epoch)

def get_attention_weights(self, input_ids: torch.Tensor,
                           attention_mask: torch.Tensor) -> Dict[str, Any]:
    """
    Get attention weights for visualization and analysis.

    Args:
        input_ids: Input token IDs
        attention_mask: Attention mask

    Returns:
        Dictionary containing attention weights
    """
    with torch.no_grad():
        logits, attention_weights = self(input_ids, attention_mask,
return_attention=True)

    return attention_weights

def localize_vulnerabilities(self,
                             input_ids: torch.Tensor,
                             attention_mask: torch.Tensor,
                             target_class: int = None,
                             threshold: float = 0.5) -> Dict[str, Any]:
    """
    Localize vulnerabilities in the input code.

    Args:
        input_ids: Input token IDs
        attention_mask: Attention mask
        target_class: Specific vulnerability class to localize (None for all)
        threshold: Attention threshold for span extraction

    Returns:
        Dictionary containing localization results
    """
    # Get model outputs and attention weights (without no_grad for
gradient computation)
    outputs, attention_weights = self(input_ids, attention_mask,
return_attention=True)
    probabilities = torch.sigmoid(outputs)

    # Get token attention weights
    token_attention = attention_weights['token_attention'][0] # First sample

    # Handle variable length attention weights
    if len(token_attention) > 0:
        # Pad attention weights to match input length
        max_len = input_ids.shape[1]
        padded_attention = torch.zeros(max_len, device=input_ids.device)

        # Flatten and pad token attention weights
        flat_attention = []
        for attn in token_attention:
            if len(attn) > 0:
                flat_attention.extend(attn.detach().cpu().numpy())

```

```

        if len(flat_attention) > 0:
            padded_attention[:len(flat_attention)] = torch.tensor(flat_attention,
device=input_ids.device)
        else:
            padded_attention = torch.ones(input_ids.shape[1],
device=input_ids.device) / input_ids.shape[1]

        # Determine which classes to localize
        if target_class is not None:
            classes_to_localize = [target_class]
        else:
            # Localize all classes with high probability
            predicted_classes = (probabilities[0] > 0.5).nonzero().squeeze(-1)
            classes_to_localize = [int(idx) for idx in predicted_classes.tolist()]

        localization_results = {
            'attention_weights': padded_attention.detach().cpu().numpy(),
            'probabilities': probabilities[0].detach().cpu().numpy(),
            'vulnerable_spans': {},
            'localization_metrics': {}
        }

        # Localize each vulnerability class
        for class_idx in classes_to_localize:
            # Register hooks for gradient computation
            self.vulnerability_localizer.register_hooks(self,
"codebert.encoder.layer.11")

            try:
                # Compute attention weights
                print(f"Q DEBUG: About to compute attention weights")
                attention_weights =
self.vulnerability_localizer.compute_attention_weights(
                    self, input_ids, attention_mask
                )
                print(f"Q DEBUG: Successfully computed attention weights")

                # Compute Grad-CAM weights
                print(f"Q DEBUG: About to compute Grad-CAM weights")
                gradcam_weights =
self.vulnerability_localizer.compute_gradcam_weights(
                    self, input_ids, attention_mask, int(class_idx)
                )
                print(f"Q DEBUG: Successfully computed Grad-CAM weights")

                # Combine signals using  $\lambda$  parameter
                print(f"Q DEBUG: About to combine signals")
                combined_scores = self.vulnerability_localizer.combine_signals(
                    attention_weights, gradcam_weights
                )
                print(f"Q DEBUG: Successfully combined signals")

                # Perform sliding window localization
                print(f"Q DEBUG: About to perform sliding window localization")
                vulnerable_windows =
self.vulnerability_localizer.sliding_window_localization(
                    combined_scores, input_ids, attention_mask
                )
                print(f"Q DEBUG: Successfully performed sliding window
localization")

            # Store results

```

```

        class_idx_int = int(class_idx)
        localization_results['vulnerable_spans'][class_idx_int] =
vulnerable_windows
        localization_results[f'attention_weights_{class_idx_int}'] =
attention_weights.detach().cpu().numpy()
        localization_results[f'gradcam_weights_{class_idx_int}'] =
gradcam_weights.detach().cpu().numpy()
        localization_results[f'combined_scores_{class_idx_int}'] =
combined_scores.detach().cpu().numpy()

    finally:
        # Clean up hooks
        self.vulnerability_localizer.remove_hooks()

    return localization_results

def generate_vulnerability_report(self,
                                input_ids: torch.Tensor,
                                attention_mask: torch.Tensor,
                                tokenizer,
                                vulnerability_labels: List[str],
                                save_visualization: bool = True,
                                output_path: str =
"vulnerability_localization_report.json") -> Dict[str, Any]:
    """
    Generate a comprehensive vulnerability localization report.

    Args:
        input_ids: Input token IDs
        attention_mask: Attention mask
        tokenizer: Tokenizer for converting IDs to tokens
        vulnerability_labels: List of vulnerability class names
        save_visualization: Whether to save visualization plots
        output_path: Path to save the report

    Returns:
        Dictionary containing the complete report
    """
    # Get localization results
    localization_results = self.localize_vulnerabilities(input_ids,
attention_mask)

    # Generate detailed report
    report = {
        'input_info': {
            'sequence_length': input_ids.shape[1],
            'tokens': tokenizer.convert_ids_to_tokens(input_ids[0])
        },
        'vulnerability_detection': {
            'probabilities': localization_results['probabilities'].tolist(),
            'predicted_classes': (localization_results['probabilities'] >
0.5).nonzero()[0].tolist()
        },
        'localization_results': localization_results,
        'detailed_analysis': {}
    }

    # Analyze each detected vulnerability
    for class_idx, windows in localization_results['vulnerable_spans'].items():
        if len(windows) > 0:
            vuln_name = vulnerability_labels[class_idx] if class_idx <
len(vulnerability_labels) else f"class_{class_idx}"

```

```

# Generate window analysis
window_analysis = []
for window in windows:
    window_tokens = tokenizer.convert_ids_to_tokens(window['tokens'])
    # Properly decode tokens back to readable text
    window_text = tokenizer.decode(window['tokens'],
skip_special_tokens=True)
    window_analysis.append({
        'start': window['start'],
        'end': window['end'],
        'importance': window['importance'],
        'max_score': window.get('max_score', 0.0),
        'tokens': window_tokens,
        'text': window_text
    })

report['detailed_analysis'][vuln_name] = {
    'class_index': class_idx,
    'probability': localization_results['probabilities'][class_idx],
    'windows': window_analysis,
    'total_windows': len(windows),
    'average_importance': np.mean([window['importance'] for window in
windows])
}

# Save report
with open(output_path, 'w', encoding='utf-8') as f:
    # Convert numpy arrays and torch tensors to lists for JSON serialization
    def convert_numpy(obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        elif isinstance(obj, torch.Tensor):
            return obj.detach().cpu().numpy().tolist()
        elif isinstance(obj, dict):
            return {key: convert_numpy(value) for key, value in obj.items()}
        elif isinstance(obj, list):
            return [convert_numpy(item) for item in obj]
        elif isinstance(obj, (np.float32, np.float64, np.int32, np.int64)):
            return float(obj)
        else:
            return obj

    serializable_report = convert_numpy(report)
    json.dump(serializable_report, f, indent=2, ensure_ascii=False)

return report

def compute_localization_metrics(self,
                                input_ids: torch.Tensor,
                                attention_mask: torch.Tensor,
                                ground_truth_spans: Dict[int, List[Dict[str,
Any]]]) -> Dict[str, float]:
    """
    Compute localization metrics against ground truth.

    Args:
        input_ids: Input token IDs
        attention_mask: Attention mask
        ground_truth_spans: Ground truth spans for each class

    Returns:
        Dictionary of localization metrics
    """

```

```

        localization_results = self.localize_vulnerabilities(input_ids,
attention_mask)

        metrics = {}
        for class_idx, predicted_spans in
localization_results['vulnerable_spans'].items():
            if class_idx in ground_truth_spans:
                gt_spans = ground_truth_spans[class_idx]

                class_metrics = self.vulnerability_localizer.compute_top_k_metrics(
                    predicted_spans, gt_spans, input_ids.shape[1]
                )

                metrics[f'class_{class_idx}'] = class_metrics

    return metrics

```

ДОДАТОК Г СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

1. Komleva, N. O., & Tereshchenko, O. I. (2023). Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform. *Herald of Advanced Information Technology*, 6(1), 54–68. <https://doi.org/10.15276/hait.06.2023.4> (**Scopus**)
2. Tereshchenko, O., & Komleva, N. (2023). Vulnerability detection of smart contracts based on bidirectional GRU and attention mechanism. In *Communications in Computer and Information Science* (Vol. 1980). Springer, Cham. https://doi.org/10.1007/978-3-031-48325-7_21 (**Scopus**)
3. Tereshchenko, O. I., & Komleva, N. O. (2024). Identification and localization of vulnerabilities in smart contracts using attention vectors analysis in a BERT-based model. *Radio Electronics, Computer Science, Control*, (3), 173–184. <https://doi.org/10.15588/1607-3274-2024-3-15> (**Web of Science**)
4. Терещенко, О. І. (2025). Комбінований метод локалізації вразливостей у смарт-контрактах на основі Attention та Grad-CAM. *Таврійський науковий вісник. Серія: Технічні науки*, 1(4), 306-316. <https://doi.org/10.32782/tnv-tech.2025.4.1.30>
5. Терещенко, О. І. (2025). Методи ін'єкції вразливостей у смарт-контракти для формування збалансованих датасетів. *Вісник Херсонського національного технічного університету*. № 3(94), Ч. 2, 456-462. <https://doi.org/10.35546/kntu2078-4481.2025.3.2.58>
6. Терещенко, О. І. (2025). Виявлення та локалізація вразливостей у смарт-контрактах на основі дворівневої уваги з графовим підкріпленням. *Вісник Кременчуцького національного університету імені Михайла Остроградського*, 4, 272-281, <https://doi.org/10.32782/1995-0519.2025.4.30>

ДОДАТОК Д ВІДОМОСТІ ПРО АПРОБАЦІЮ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОЇ РОБОТИ

1. *Терещенко, О. І., & Комлева, Н. О. (2023). Огляд наявних інструментів виявлення вразливостей смарт-контрактів на основі машинного навчання. In Modern Information Technology 2023 / Сучасні Інформаційні Технології 2023 (Одеса, 18–19 травня 2023 р.). Доповідь та публікація тез. URL: <http://dspace.opu.ua/jspui/handle/123456789/14202>*
2. *Tereshchenko, O. (2024). Analysis of the efficacy of using machine learning for detecting vulnerabilities in smart contracts compared to traditional methods. In Proceedings of the XIII International Scientific and Practical Conference “Social Ways of Training Specialists in the Social Sphere and Inclusive Education” (pp. 327–329). Prague, Czech Republic, April 1–3, 2024. Доповідь та публікація тез. URL: <https://eu-conf.com/en/events/social-ways-of-training-specialists-in-the-social-sphere-and-inclusive-education/>*
3. *Tereshchenko, O. (2024). Improving the security of smart contracts through the integration of machine learning: Challenges and prospects. In Proceedings of the VI International Scientific Conference “Traditional and Innovative Approaches to Scientific Research” (pp. 124–125). Vinnytsia, Ukraine, March 8, 2024. Доповідь та публікація тез. URL: <https://archive.mcnd.org.ua/index.php/conference-proceeding/issue/view/08.03.2024>*
4. *Tereshchenko, O. I. (2024). Integration of NLP and machine learning methods for smart contract security: A comparison with traditional approaches. In Proceedings of the X International Scientific and Practical Conference “Informatics. Culture. Technology”, September 2024. Доповідь та публікація тез. URL: <https://doi.org/10.15276/ict.01.2024.31>*
5. *Терещенко, О. І. (2025). Побудова збалансованого корпусу смарт-контрактів шляхом контрольованої ін’єкції вразливостей. In Proceedings of the VIII International Scientific Conference “Розвиток наукової думки постіндустріального суспільства: сучасний дискурс”, 2025. Доповідь та публікація тез. URL: <https://doi.org/10.62731/mcnd-17.10.2025>*