

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

«На правах рукопису»

УДК 004.056.5:004.85

«До захисту допущено»

Завідувач кафедри

_____ Сергій ЯКОВЛЄВ

«__» _____ 2026 р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Математичні методи криптографічного захисту інформації»
зі спеціальності: 113 Прикладна математика
на тему: **«Використання протоколів з нульовим розголошенням в системах штучного інтелекту»**

Виконав:

студент IV курсу, групи ФІ-23

Малютіна Марина Сергіївна _____

Керівник:

професор кафедри ММЗІ, д.т.н., с.н.с

Кудін Антон Михайлович _____

Рецензент:

доцент кафедри ММАД, к.ф-м.н.

Терещенко Іван Миколайович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Кафедра математичних методів захисту інформації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»

Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

Рівень вищої освіти — перший (бакалаврський)
Спеціальність — 113 Прикладна математика,
ОПП «Математичні методи криптографічного захисту інформації»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій ЯКОВЛЄВ

«__» _____ 2026 р.

ЗАВДАННЯ
на дипломну роботу

Студент: Малютіна Марина Сергіївна

1. Тема роботи: *«Використання протоколів з нульовим розголошенням в системах штучного інтелекту»*, науковий керівник дипломної роботи: професор кафедри ММЗІ, д.т.н., с.н.с Кудін Антон Михайлович,

затверджені наказом по університету №__ від «__» _____ 2026 р.

2. Термін подання студентом роботи: «__» _____ 2026 р.

3. Об'єкт дослідження: Процеси верифікації обчислень нейронних мереж із забезпеченням конфіденційності даних у системах машинного навчання.

4. Предмет дослідження: Методи побудови доказів з нульовим розголошенням для верифікації інференсу та захисту навчальних даних багат шарового перцептрона.

5. Перелік завдань:

1) Проаналізувати основні загрози конфіденційності та методи захисту даних в системах машинного навчання.

2) Дослідити математичні основи протоколів zk-SNARK та zk-STARK.

3) Розробити архітектуру системи приховування навчальних даних та верифікації інференсу.

4) Реалізувати прототип системи.

5) Провести аналіз безпеки розробленої системи та оцінити її практичні характеристики.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
Презентація доповіді

7. Орієнтовний перелік публікацій: -

8. Дата видачі завдання: 10 вересня 2025 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання	Примітка
1	Узгодження теми роботи із науковим керівником	01-15 вересня 2025 р.	Виконано
2	Огляд опублікованих джерел за тематикою дослідження	Вересень-жовтень 2025 р.	Виконано
3	Аналіз протоколів zk-SNARK та zk-STARK.	Листопад - грудень 2025 р	Виконано
4	Розробка архітектури системи.	Січень - березень 2026 р	Виконано
5	Реалізація прототипу та аналіз безпеки.	Квітень - червень 2026 р	Виконано
6	Оформлення дипломної роботи.	Червень 2026 р	Виконано

Студент

_____ Марина МАЛЮТІНА

Керівник

_____ Антон КУДІН

РЕФЕРАТ

Кваліфікаційна робота містить: 75 стор., 0 рисунки, 6 таблиць, 27 джерел.

Метою роботи є підвищення рівня конфіденційності та верифікованості систем машинного навчання. Об'єктом дослідження є процеси верифікації обчислень нейронних мереж із забезпеченням конфіденційності даних. Предметом дослідження є методи побудови доказів з нульовим розголошенням на основі zk-STARK для верифікації інференсу та захисту навчальних даних багат шарового перцептрона.

У роботі проаналізовано загрози конфіденційності в системах машинного навчання та обґрунтовано вибір протоколів доведення з нульовим розголошенням як основи для їх усунення. Досліджено математичні основи zk-STARK, методи арифметизації обчислень та схеми поліноміальних зобов'язань. Розроблено та реалізовано інтегрований протокол трьох взаємопов'язаних STARK-доказів, який забезпечує одночасну верифікацію інференсу та захист навчального набору даних без розкриття параметрів моделі, вхідних даних і тренувальних прикладів.

ZK-STARK, ПРОТОКОЛИ З НУЛЬОВИМ РОЗГЛОШЕННЯМ,
МАШИННЕ НАВЧАННЯ, ЗАХИСТ НАВЧАЛЬНИХ ДАНИХ,
ВЕРИФІКАЦІЯ ІНФЕРЕНСУ

ABSTRACT

The qualification work contains: 75 pages, 0 figures, 6 tables, 27 references.

The aim of the work is to increase the level of confidentiality and verifiability of machine learning systems. The object of the research is the processes of verifying neural network computations while ensuring data confidentiality. The subject of the research is methods for constructing zero-knowledge proofs based on zk-STARK for verifying inference and protecting the training data of a multilayer perceptron.

The work analyzes privacy threats in machine learning systems and substantiates the choice of zero-knowledge proof protocols as the basis for their elimination. The mathematical foundations of zk-STARK, methods of arithmetization of computations, and schemes of polynomial commitments are studied. An integrated protocol of three interconnected STARK proofs is developed and implemented, which ensures simultaneous verification of inference and protection of the training dataset without disclosing model parameters, input data, and training examples.

ZK-STARK, ZERO-KNOWLEDGE PROOF PROTOCOLS,
MACHINE LEARNING, TRAINING DATA PROTECTION, INFERENCE
VERIFICATION.

ЗМІСТ

Вступ.....	7
1 Аналіз сучасного стану проблеми конфіденційності в системах ШІ ...	9
1.1 Загрози безпеки та приватності	9
1.2 Огляд методів захисту даних	11
1.3 Огляд існуючих zkML-фреймворків та систем	16
Висновки до розділу 1	18
2 Математичні методи побудови протоколів доведення з нульовим розголошенням	19
2.1 Теоретичні основи протоколів з нульовим розголошенням	19
2.2 Протоколи zk-SNARK та zk-STARK	21
2.3 Арифметизація обчислень нейронних мереж	24
2.4 Схеми поліноміальних зобов'язань	28
Висновки до розділу 2	30
3 Програмна реалізація системи на основі zk-STARK	31
3.1 Постановка задачі	31
3.2 Алгоритм побудови доказу	33
3.3 Протокол верифікації	35
3.4 Програмна реалізація	36
Висновки до розділу 3	42
Висновки	43
Перелік посилань	45
Додаток А Тексти програм	48

ВСТУП

Актуальність дослідження. У сучасному світі дедалі ширшого застосування набувають системи штучного інтелекту — для навчання, у професійній діяльності та повсякденному житті. Логічним напрямом для подальшого розвитку є впровадження таких систем у діяльність організацій, котрі оперують чутливими даними, наприклад державних, медичних чи фінансових установ. Наявні методи захисту мають суттєві обмеження і не є повністю стійкими до атак, спрямованих на порушення цілісності системи чи конфіденційності навчальних даних, що унеможлиблює навчання на чутливих даних через загрозу часткової чи повної компрометації.

Протоколи доведення з нульовим розголошенням є перспективним варіантом вирішення проблеми, оскільки ними можна довести коректність виконання обчислень без розголошення будь-якої інформації, що дозволяє одночасно забезпечити верифікацію інференсу та приховування навчальних даних. Попри значну зацікавленість дослідників, сучасні системи орієнтовані на верифікацію інференсу та залишають поза увагою приватність навчальних даних. Тож це зумовлює актуальність розробки системи, котра поєднає в собі обидва завдання в межах єдиного протоколу.

Метою дослідження є підвищення рівня конфіденційності та верифікованості систем машинного навчання. Для досягнення мети необхідно розв'язати **задачу дослідження**, яка полягає у розробці та програмній реалізації протоколу верифікації інференсу нейронної мережі із одночасним захистом навчальних даних на основі zk-STARK. Для розв'язання задачі необхідно вирішити такі завдання:

- 1) Проаналізувати основні загрози конфіденційності та методи захисту даних в системах машинного навчання.
- 2) Дослідити математичні основи протоколів zk-SNARK та

zk-STARK.

3) Розробити архітектуру системи приховування навчальних даних та верифікації інференсу.

4) Реалізувати прототип системи.

5) Провести аналіз безпеки розробленої системи та оцінити її практичні характеристики.

Об'єктом дослідження є процеси верифікації обчислень нейронних мереж із забезпеченням конфіденційності даних у системах машинного навчання.

Предметом дослідження є методи побудови доказів з нульовим розголошенням для верифікації інференсу та захисту навчальних даних багат шарового перцептрона.

При розв'язанні поставлених завдань використовувались такі *методи дослідження*: методи теорії складності алгоритмів, алгебри скінченних полів та арифметизації обчислень, порівняльний аналіз існуючих підходів до захисту даних у системах машинного навчання, методи аналізу безпеки криптографічних протоколів, а також експериментальне оцінювання продуктивності розробленої системи.

Наукова новизна полягає у запропонованій інтегрованій системі трьох взаємопов'язаних STARK-доказів (MLP AIR, Poseidon AIR, Gradient AIR), які спільно забезпечують одночасну верифікацію інференсу та захист навчального датасету без розкриття параметрів моделі, вхідних і тренувальних даних.

Практичне значення результатів полягає у можливості верифікації коректності обчислень нейронної мережі та узгодженості моделі з навчальним датасетом без розголошення параметрів моделі, вхідних і навчальних даних.

1 АНАЛІЗ СУЧАСНОГО СТАНУ ПРОБЛЕМИ КОНФІДЕНЦІЙНОСТІ В СИСТЕМАХ ШІ

Розширення застосування систем машинного навчання супроводжується зростанням загроз конфіденційності тренувальних даних та цілісності самих моделей. У цьому розділі розглянуто основні класи атак, підходи до захисту приватності та існуючі zkML-системи як основу для подальшого дослідження.

1.1 Загрози безпеки та приватності

Системи машинного навчання є вразливими до широкого спектру атак, які здатні компрометувати як конфіденційність даних, так і цілісність самих моделей. Для організацій, що працюють із чутливими даними, наслідки подібних атак можуть бути критичними з погляду приватності користувачів, а також становити пряму загрозу порушення вимог GDPR, HIPAA та інших чинних регуляцій у сфері захисту персональних даних.

Атаки на вивід членства (Membership Inference Attacks, MIA) — клас атак, у яких злоумисник намагається встановити, чи входив конкретний зразок у набір тренувальних даних моделі. У більшості випадків моделі, які мають прийнятну точність на тренувальних даних, виявляють вищу впевненість своїх передбачень для вибірок, які вони бачили під час навчання. Отже злоумисник може використовувати різницю в здатності моделі обробляти знайомі та незнайомі дані задля отримання інформації про тренувальний набір.

Перша значна робота була представлена у 2016 році, де продемонстрували, що аналізуючи поведінку та розподіл впевненості передбачень з високою імовірністю можна визначити чи належить

конкретний зразок до тренувального набору моделі [22]. Зловмисник будує набір тіньових моделей з різними висновками про членство та порівнює їх поведінку з цільовою моделлю. Розроблені у 2018 році, сучасніші підходи, показали, що даний клас атак можна проводити без знання архітектури моделі(модель-незалежні атаки) [20].

Інверсійні атаки на моделі(Model Inversion Attacks, MIA) — клас атак, у яких зловмисник намагається реконструювати вхідні дані чи характеристики тренувального набору, маючи доступ щонайменше до виходів моделі. У більшості випадків моделі зберігають неявну інформацію про тренувальні дані у своїх параметрах, що дозволяє зловмиснику за допомогою деякої кількості правильно сформульованих запитів частково чи повністю відновити дані.

У 2015 році було продемонстровано, що аналізуючи вихідні вектори впевненості моделі можна ітеративно відновлювати вхідні дані [11].Зловмисник формулює задачу реконструкції як оптимізаційну: починаючи з випадкового або усередненого вхідного значення, він поступово мінімізує різницю між виходом моделі та цільовою міткою, отримуючи в результаті синтетичне зображення або вектор ознак, що відображає реальні тренувальні зразки. Розроблені у 2019–2020 роках підходи, зокрема на основі генеративно-змагальних мереж, показали, що даний клас атак можна проводити значно ефективніше навіть у сценарії чорного ящика, де зловмисник має лише обмежений API-доступ до цільової моделі.

Атаки на отруєння даних(Data Poisoning Attacks, DPA) — клас атак, у яких зловмисник цілеспрямовано маніпулює тренувальним набором даних з метою порушення поведінки моделі на етапі навчання. Впровадження шкідливих або модифікованих зразків відбувається до того як починається навчання.

Розрізняють два основні підтипи: *атаки на цілісність*, де метою є змусити модель допускати специфічні помилки на конкретних вхідних даних, зберігаючи загальну точність, та *атаки на доступність*, де мета

— максимально знизити загальну продуктивність моделі на всіх вхідних даних. Окремим підвидом є *бекдор-атаки*, у яких отруєні зразки містять прихований тригер, що активує небажану поведінку лише за умови його присутності у вхідних даних.

Перші систематичні дослідження з’явилися у 2012 році, було продемонстровано вразливість SVM-класифікаторів до цілеспрямованого додавання шкідливих точок у тренувальний набір [7]. Логіка досить проста: отруєні дані підбираються так, аби після навчання модель поводитись конкретним чином. Пізніше, у 2017-2021 роках дослідники з’ясували, що даний клас атак ефективний проти сучасних нейронних мереж.

1.2 Огляд методів захисту даних

У контексті забезпечення приватності та цілісності моделей машинного навчання найбільшого поширення набули три підходи: федеративне навчання, гомоморфне шифрування та багатосторонні обчислення, кожен з яких буде розглянуто далі.

Федеративне навчання — парадигма розподіленого машинного навчання, у якій тренувальні дані залишаються на пристроях користувачів, де і проводилось навчання. Такий підхід дозволяє мінімізувати ризик витоку конфіденційної інформації, оскільки підхід є децентралізованим [18].

Процес складається з таких основних етапів:

1) Центральний сервер розсилає поточні параметри моделі $\theta^{(t)}$ усім клієнтам.

2) Кожен клієнт k локально обчислює градієнти функції втрат на власному датасеті D_k :

$$\Delta_k = \nabla L(D_k; \theta^{(t)}). \quad (1.1)$$

3) Клієнти передають обчислені градієнти або оновлені параметри на

сервер.

4) Сервер агрегує отримані оновлення та формує нові параметри моделі:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \cdot \frac{1}{K} \sum_{k=1}^K \Delta_k, \quad (1.2)$$

де α — швидкість навчання, K — кількість клієнтів.

Федеративне навчання має ряд суттєвих переваг порівняно з централізованими підходами. Насамперед, дані залишаються безпосередньо на пристроях збору і жодного разу не передаються на центральний сервер, що повністю відповідає вимогам конфіденційності та чинним нормативним стандартам захисту персональних даних. Окрім цього, такий підхід природно підходить для розподілених середовищ — включно з IoT-пристроями та мобільними платформами — де передача сирих даних є технічно обмеженою або небажаною. Варто також відзначити стійкість методу до гетерогенності даних: модель здатна навчатися на наборах даних із різними форматами та якістю, що є типовою ситуацією в реальних розподілених системах. Нарешті, архітектура федеративного навчання добре масштабується, забезпечуючи ефективну роботу навіть за умови великої кількості клієнтів-учасників.

Попри зазначені переваги, також є достатньо недоліків, які необхідно враховувати при практичному застосуванні. Одним із ключових є комунікаційне навантаження: передача градієнтів або параметрів моделі між клієнтами та сервером є вузьким місцем, що особливо відчутно для великих архітектур. Додатково, неоднаково та не незалежний розподіл даних між учасниками може суттєво погіршувати збіжність моделі порівняно з централізованим навчанням, що призводить до деградації точності кінцевого результату. З погляду безпеки слід зазначити, що самі градієнти не є нейтральними: через атаки інверсії вони можуть розкривати чутливу інформацію про локальні дані клієнтів, що частково нівелює приватність підходу. Не менш серйозною проблемою є відсутність централізованої верифікації даних — система не має

механізмів гарантування якості або коректності локальних наборів, що відкриває можливість для впровадження шуму або прихованих бекдорів зловмисниками. Необхідність координації великої кількості клієнтів ускладнює роботу системи в умовах нестабільних або низькопропускних мереж, що створює додаткові труднощі синхронізації.

Гомоморфне шифрування — криптосистеми, які дозволяють виконувати обчислення безпосередньо над зашифрованими даними. Усі операції виконуються у зашифрованому просторі, що забезпечує максимальний рівень конфіденційності, а також унеможлиблює доступ до вхідних даних збоку обчислювального сервера.

Формально гомоморфна схема шифрування задає відображення

$$E : \mathbb{Z}_p \times K \rightarrow \mathbb{Z}_N, \quad (1.3)$$

де p — простір можливих відкритих повідомлень, K — простір ключів, N — простір шифротекстів. Схема задовольняє властивості гомоморфності:

Адитивна гомоморфність:

$$E(m_1 + m_2) = E(m_1) \oplus E(m_2), \quad (1.4)$$

Мультиплікативна гомоморфність:

$$E(m_1 \cdot m_2) = E(m_1) \otimes E(m_2), \quad (1.5)$$

де $\oplus$ та $\otimes$ — відповідні операції над шифротекстами. Повна гомоморфна схема дозволяє виконувати обидві операції довільну кількість разів без розшифрування, що робить можливим реалізацію довільних обчислюваних функцій [13].

З переваг можна виділити повну конфіденційність даних, відсутність необхідності довіри до сервера, оскільки він не має доступу до відкритих даних. Також важливо зазначити, що не розкривається жодна інформація про вхідні дані чи проміжні результати і є можливість безпечного делегування обчислень стороннім організаціям.

Проте оскільки ведеться робота із зашифрованими даними то необхідні обчислення стають у 10^4 – 10^6 разів повільнішими у порівнянні з аналогічними діями над відкритими. Простежується складність реалізації нелінійних операцій, що в свою чергу погіршує точність та продуктивність. Присутні обмеження глибини обчислень, для складних моделей необхідна окрема процедура, що в свою чергу також сповільнює виконання. Практичні дослідження показують, що навіть для відносно простих нейронних мереж інференс у зашифрованому вигляді може займати десятки секунд замість декількох мілісекунд.

Багатосторонні обчислення — клас криптографічних протоколів, котрі дозволяють декільком сторонам спільно обчислювати функцію над власними вхідними даними без потреби розголошувати їх іншим учасникам. Ґрунтується на методах таємного розділення секретів та механізмах перевірки коректності обчислень.

Одним з базових підходів є схема секретного розділення Шаміра. Нехай секретне значення s розподіляється між n учасниками у вигляді акцій $(s_1, s_2, \dots, s_n)$ таким чином, що знання будь-яких $t - 1$ акцій не дає жодної інформації про секрет s , а знання будь-яких t акцій дозволяє відновити секрет s за допомогою інтерполяції Лагранжа. Подальші обчислення виконуються безпосередньо над акціями секретів із використанням спеціалізованих протоколів додавання та множення, що дозволяє реалізувати довільні обчислювальні функції без розкриття приватних входів [21].

Основною перевагою є гарантована конфіденційність: вхідні дані всіх учасників залишаються приватними, а також деякі протоколи здатні виявляти некоректну поведінку, що допомагає запобігати шахрайству. Для цих протоколів було доведено, що вони можуть реалізувати будь-яку функцію за умови, що більшість учасників чесні.

З недоліків варто зазначити значні комунікаційні втрати і потреби синхронізації, оскільки n учасників повинні одночасно бути доступними одне одному. Висока складність реалізації знижує практичну цінність

переваг, практичні системи потребують ретельного керування криптографічними примітивами та обробки збоїв комунікації. Крім того, протоколи не містять механізмів гарантування якості вхідних даних, тож немає впевненості, що вони відповідають заданим критеріям.

Доведення з нульовим розголошенням (ZKP) — криптографічні протоколи, які дозволяють доказувачу переконати верифікатора в істинності деякого твердження без розкриття будь-якої інформації, окрім самого факту істинності цього твердження. Типовий ZKP-протокол складається з трьох фаз: генерування загальних параметрів, побудова доказу та перевірка доказу.

Протоколи з нульовим розголошенням — універсальні, оскільки їх можна побудувати для будь-якої обчислювальної функції з класу NP [14]. Ключовими перевагами є можливість довести, що навчання моделі виконувалося на певному датасеті без розкриття самих даних, підтвердження відповідності параметрів нейронної мережі заданим вимогам без їх розкриття та можливість масштабувати систему. Деякі сучасні системи не потребують фази довіреного налаштування.

Якщо говорити про недоліки, то перш за все це значні обчислювальні витрати, для складних обчислень розмір та час генерації доказу зростають. Також специфіка реалізації: кожен тип операцій нейронної мережі потребує спеціалізованого відображення.

З огляду на розглянуті підходи, саме протоколи доведення з нульовим розголошенням є найбільш доцільною криптографічною основою для подальшого дослідження. На відміну від федеративного навчання, ZKP не лише зменшують ризик централізованого витоку даних, а й забезпечують перевірку коректності виконаних обчислень. У порівнянні з гомоморфним шифруванням та багатосторонніми обчисленнями, вони краще поєднують властивості конфіденційності, публічної верифікованості та практичної придатності для задач ZKML.

1.3 Огляд існуючих zkML-фреймворків та систем

Розроблено низку фреймворків та систем, призначених для побудови застосувань Машинного навчання з нульовим розголошенням(zkML). Вони відрізняються рівнем абстракції, підтримуваними моделями та орієнтацією на інференс або навчання.

EZKL(Easy Zero-Knowledge Learning) — практична система, розроблена компанією Zkonduit, котра дозволяє конвертувати моделі машинного навчання у форматі ONNX(Open Neural Network Exchange), у ZK-схеми, сумісні з блокчейн-платформами та іншими системами верифікації [27].

Архітектура EZKL передбачає компіляцію ONNX-моделі в арифметичну схему, яка надалі реалізується у вигляді Halo2-схеми. На основі цієї схеми генерується доказ коректності обчислення нейронної мережі для заданого входу та параметрів. Верифікатор може перевірити доказ без доступу до самої моделі, вхідних даних або проміжних результатів.

EZKL підтримує широкий спектр операцій, зокрема згортки, зменшення розміру ознак у нейромережі, нормалізації та механізм attention (засередження моделі на найбільш релевантній частині даних), що робить його придатним для роботи з різними архітектурами нейронних мереж. Крім того, система оптимізує обчислення з використанням GPU-прискорення, зокрема через CUDA та OpenCL, а також надає публічні бенчмарки й інструменти для профайлінгу, які спрощують оцінювання продуктивності та налаштування обчислень.

Основними обмеженнями EZKL є орієнтація переважно на інференс, а не на верифікацію процесу навчання, а також суттєвий час генерування доказів для великих моделей, який може становити від кількох хвилин до кількох годин.

zkDL(Zero-Knowledge Deep Learning) — система, спрямована на

верифікацію процесу навчання глибоких нейронних мереж. На відміну від попереднього, спеціалізується саме на доказі коректності навчання [24].

Ключовими інноваціями zkDL є спеціалізована техніка zkReLU для ефективного доказу прямого та зворотного поширення активаційної функції ReLU, методи компресії градієнтів для зменшення розміру доказу, а також параметризація обчислень із використанням GPU. Перевагою є можливість верифікації обчислень які раніше вважалися непрактичними у ZK-контексті, проте система все ще вимагає багато ресурсів та потребує ретельного налаштування параметрів під конкретну модель.

Окрему групу складають спеціалізовані системи, такі як zkCNN, яка оптимізована для верифікації інференсу згорткових нейронних мереж, та ZEN, що використовує спеціалізовані аргументи, зокрема sum-check і GKR-протоколи, для ефективної верифікації багат шарових моделей [17, 9]. До інших прикладів належать FairProof та OATH, орієнтовані на верифікацію справедливості ML-моделей, а також zkLLM, спрямована на перевірку інференсу великих мовних моделей [26, 1, 25].

Крім спеціалізованих zkML-систем, широко застосовуються універсальні фреймворки для побудови ZK-схем, які можуть бути адаптовані для задач машинного навчання. Circom є мовою опису арифметичних схем і залишається одним із найпоширеніших інструментів у ZK-спільноті. Вона вирізняється простим синтаксисом, наявністю великої кількості готових компонентів та активною спільнотою, однак має обмежені можливості оптимізації складних операцій і потребує зовнішніх інструментів для генерації доказів [15].

Cairo — повнофункціональна мова програмування для ZK-STARK-систем, розроблена компанією StarkWare. Вона не потребує довіреного початкового налаштування і забезпечує високу гнучкість у виборі операцій, проте характеризується вищим порогом входу та меншою кількістю готових бібліотек [23].

Halo2 — бібліотека від Zcash для побудови ZK-схем, яка орієнтована

на гнучкість та ефективність. Вона не потребує довіреного початкового налаштування, однак має складніший API, а реалізація навіть простих схем часто вимагає значного обсягу допоміжного коду [8].

Noir — відносно новий фреймворк від Aztec Network, орієнтований на створення приватних застосувань і смарт-контрактів. Він має сучасний синтаксис і тісну інтеграцію з блокчейн-екосистемами, проте наразі поступається Circom за кількістю готових компонентів і рівнем зрілості екосистеми [2].

Висновки до розділу 1

У цьому розділі було описано основні атаки, спрямовані на порушення конфіденційності даних та цілісності моделей, а також розглянуто підходи, які забезпечують захист, кожен з яких має свої несумнівні переваги, але разом з тим і суттєві практичні обмеження.

Протоколи доведення з нульовим розголошенням активно застосовуються для верифікації інференсу, проте наявні системи не забезпечують одночасного захисту тренувальних даних. Отже, поставлена в цій роботі задача не має повного та універсального розв'язання в наявних підходах, що обґрунтовує подальшу розробку системи для приховування даних навчання та верифікації інференсу.

2 МАТЕМАТИЧНІ МЕТОДИ ПОБУДОВИ ПРОТОКОЛІВ ДОВЕДЕННЯ З НУЛЬОВИМ РОЗГОЛОШЕННЯМ

Метою цього розділу є виклад математичних основ побудови протоколів доведення з нульовим розголошенням, які застосовуються для перевірки обчислень. Зокрема, розглянуто теоретичні засади zk-SNARK та zk-STARK, методи арифметизації обчислень, а також схеми поліноміальних зобов'язань, що забезпечують компактне та ефективне кодування параметрів моделей.

2.1 Теоретичні основи протоколів з нульовим розголошенням

Нехай $R = \{(x, w)\}$ — відношення допустимих пар, де x є публічним твердженням, а w — таємним свідком. Протокол доведення з нульовим розголошенням для відношення R є інтерактивним протоколом між доказувачем $\mathcal{P}$ та верифікатором $\mathcal{V}$, який задовольняє такі властивості [14]:

Повнота: якщо $(x, w) \in R$, то чесний доказувач $\mathcal{P}$ завжди переконує верифікатора $\mathcal{V}$:

$$\Pr [\mathcal{V}(x) \text{ accepts} \mid (x, w) \in R, \text{ honest } \mathcal{P}] = 1. \quad (2.1)$$

Коректність: якщо $(x, w) \notin R$, то жоден зловмисний доказувач не може переконати верифікатора з імовірністю, більшою за незначну функцію $\varepsilon(\lambda)$:

$$\Pr [\mathcal{V}(x) \text{ accepts} \mid (x, w) \notin R] \leq \varepsilon(\lambda). \quad (2.2)$$

Нульове розголошення: існує імітатор $\mathcal{S}$, який може згенерувати стенограму взаємодії, обчислювально невідрізненну від справжнього

протоколу, не маючи доступу до свідка w :

$$\{\mathcal{P}(x, w) \leftrightarrow \mathcal{V}(x)\} \approx_c \{\mathcal{S}(x)\}, \quad (2.3)$$

де $\approx_c$ позначає обчислювальну невідрізненість.

Залежно від типу нерозрізнюваності ансамблів розрізняються два види властивості нульового розголошення: *досконале нульове розголошення* — розподіл виходу симулятора $\mathcal{S}$ є статистично нерозрізнюваним від розподілу реальної взаємодії для будь-якого верифікатора та *обчислювальне нульове розголошення* — розподіли нерозрізнювані лише для поліноміально обмеженого верифікатора [14].

На практиці більшість сучасних протоколів реалізують саме обчислювальне нульове розголошення, оскільки воно забезпечує достатній рівень безпеки при значно менших обчислювальних витратах. У контексті нейронних мереж твердженням, яке необхідно довести, може бути або коректність обчислення виходу моделі для заданого входу (інференс) або ж коректність виконання кроків навчання, при чому структура моделі, її параметри та безпосередньо самі дані залишаються прихованими від верифікатора.

Інтерактивний ZKP — протокол, у якому між доказувачем $\mathcal{P}$ та верифікатором $\mathcal{V}$ відбувається кілька раундів обміну повідомленнями: верифікатор надсилає випадкові виклики, доказувач відповідає, і лише після серії таких обмінів верифікатор ухвалює рішення.

Інтерактивні схеми є непрактичними для більшості реальних застосувань — зокрема для блокчейн-систем, де $\mathcal{P}$ та $\mathcal{V}$ не можуть взаємодіяти в реальному часі.

Неінтерактивний ZKP — протокол у якому доказувач формує єдиний рядок даних — доказ π — без будь-якого діалогу, а верифікатор перевіряє його детерміністично:

$$\text{Verify}(vk, x, \pi) \in \{0, 1\}, \quad (2.4)$$

де vk — ключ верифікації, x — публічний вхід.

Стандартним механізмом перетворення інтерактивного протоколу на неінтерактивний є *евристика Фіата-Шаміра* — метод усунення інтерактивності, за яким випадкові виклики верифікатора замінюються значеннями криптографічної геш-функції від усіх попередніх повідомлень [10]:

$$r_i = H(\text{com}_1 \| \text{com}_2 \| \dots \| \text{com}_{i-1}), \quad (2.5)$$

Оскільки H поводить ся як випадковий оракул, доказувач не може передбачити значення r_i до фіксації зобов'язань. Верифікатор відтворює ті самі геші з отриманого доказу та перевіряє відповідність без жодної взаємодії з $\mathcal{P}$.

2.2 Протоколи zk-SNARK та zk-STARK

zk-SNARK — клас протоколів, що забезпечують короткі, неінтерактивні та швидко верифіковані докази коректності обчислень. Більшість практичних систем базуються на криптографії еліптичних кривих та білінійних спарюваннях [12, 19, 5, 6].

Типова конструкція zk-SNARK поєднує кілька ключових криптографічних компонентів. Спершу цільове обчислення перетворюється на арифметичну схему або систему обмежень типу R1CS чи QAP, що дає змогу звести перевірку коректності виконання до перевірки набору алгебраїчних рівнянь над скінченним полем [12, 6]. Далі використовуються схеми поліноміальних зобов'язань, за допомогою яких комітяться поліноми, що кодують обчислення, а також надаються короткі докази коректності їх значень у вибраних точках. У деяких схемах додатково застосовуються білінійні спарювання на еліптичних кривих, які забезпечують дуже короткі докази та швидку верифікацію, білінійне

спарювання можна подати як відображення

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T, \quad (2.6)$$

що задовольняє властивість білінійності:

$$e(aP, bQ) = e(P, Q)^{ab}, \quad P \in \mathbb{G}_1, Q \in \mathbb{G}_2, a, b \in \mathbb{Z}. \quad (2.7)$$

На практиці для таких схем використовують криві BN254 та BLS12-381. Нарешті, більшість zk-SNARK потребують фази довіреного початкового налаштування, під час якої генеруються структуровані публічні параметри (SRS), залежні від опису обчислення; щоб мінімізувати ризики компрометації, такі церемонії зазвичай проводять у форматі багатостороннього обчислення, де безпека спирається на припущення, що принаймні один учасник є чесним.

Формально протокол складається з трьох алгоритмів: Setup, який генерує SRS, Prove, що будує доказ на основі SRS, публічного твердження та секретного свідка, Verify, який перевіряє доказ за час, поліноміальний від розміру твердження, але незалежний від складності всього обчислення. Розмір доказу — $O(1)$ (≈ 200 байт), час верифікації є сталим відносно розміру обчислення і на практиці є дуже малим [19, 5].

zk-STARK — альтернативний клас протоколів, що робить акцент на прозорості, масштабованості та постквантовій стійкості. Підходи базуються переважно на геш-функціях та кодах Ріда-Соломона [4].

У zk-STARK обчислення подається у вигляді таблиці станів (*execution trace*), де кожен рядок відповідає окремому кроку виконання, а стовпці відображають внутрішні змінні системи. Коректність переходів між послідовними станами задається системою поліноміальних обмежень AIR (Algebraic Intermediate Representation). Таким чином, перевірка того, чи правильно виконано обчислення, зводиться до перевірки того, чи задовольняє побудована таблиця AIR-обмеження [4].

Для доведення того, що значення, задані на великій множині точок, відповідають деякому низькостепеневому поліному, використовується протокол FRI (Fast Reed–Solomon IOP of Proximity). У його основі лежить ітеративне зменшення степеня полінома: початковий поліном $P(x)$ розкладається на парну та непарну частини [3],

$$P(x) = P_{\text{even}}(x^2) + x \cdot P_{\text{odd}}(x^2), \quad (2.8)$$

після чого за допомогою випадкового виклику r будується новий поліном удвічі меншого степеня:

$$P'(x) = P_{\text{even}}(x) + r \cdot P_{\text{odd}}(x). \quad (2.9)$$

Ця процедура повторюється доти, доки не залишається константний поліном. Випадкові виклики генеруються за допомогою евристики Фіата-Шаміра, тобто як $r_i = H(\text{com}_{i-1})$, що дає змогу зробити протокол неінтерактивним [10].

Замість комітментів на основі спарювань у zk-STARK застосовуються геш-базовані комітменти, зазвичай побудовані на деревах Меркла над рядками execution trace. Такий підхід забезпечує прозорість і спирається лише на криптографічну стійкість геш-функцій, а сам комітмент задається коренем дерева Меркла [4].

Таблиця 2.1 – Порівняння zk-SNARK та zk-STARK

Характеристика	zk-SNARK	zk-STARK
Розмір доказу	≈ 200 байт	$\approx 40\text{--}200$ КБ
Складність доказу	$O(n \log n)$	$O(n \log^2 n)$
Складність верифікації	$O(1)$	$O(\log^2 n)$
Довірене початкове налаштування	потрібне	не потрібне
Криптографічна база	еліптичні криві	геш-функції
Квантостійкість	ні	так
Час верифікації	< 10 мс	$10\text{--}100$ мс
Масштабованість	обмежена	краща при великих n
Зрілість екосистеми	висока (2013+)	середня (2018+)

2.3 Арифметизація обчислень нейронних мереж

Щоб застосувати zk-SNARK або zk-STARK до доведення коректності роботи нейронної мережі, її обчислення необхідно подати у вигляді арифметичної схеми над скінченним полем. *Арифметична схема* — орієнтований ациклічний граф, вершини якого реалізують операції додавання та множення, а ребра несуть проміжні значення [12, 6].

Обчислення типового шару повнозв'язної нейронної мережі має вигляд

$$y = \sigma(Wx + b), \quad (2.10)$$

де x — вхідний вектор, W — матриця ваг, b — вектор зсуву, а σ — нелінійна функція активації. Лінійна частина $Wx + b$ природно розкладається на послідовність додавальних та множильних гейтів у полі. Нелінійна частина потребує спеціального кодування, оскільки ReLU, sigmoid чи tanh не є поліномами над $\mathbb{F}_p$.

У випадку згорткових мереж (CNN) операція згортки для двовимірного ядра $K \in \mathbb{R}^{k \times k}$ та вхідної карти ознак $X \in \mathbb{R}^{H \times W}$:

$$(X * K)_{i,j} = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} X_{i+u, j+v} \cdot K_{u,v} \quad (2.11)$$

зводиться до матричного множення через операцію im2col:

$$z_{\text{conv}} = K_{\text{flat}} \cdot X_{\text{col}}, \quad (2.12)$$

де $K_{\text{flat}} \in \mathbb{R}^{d_{\text{out}} \times (k^2 \cdot C_{\text{in}})}$ — матриця розгорнутих ядер. Після такого перетворення арифметизація зводиться до тієї ж задачі, що і для повнозв'язних шарів.

На практиці арифметичні схеми перетворюють на стандартизовані алгебраїчні представлення — систему обмежень R1CS або програму квадратичних поліномів QAP [12, 6].

R1CS. Система R1CS складається з набору рівнянь вигляду

$$\langle a_i, z \rangle \cdot \langle b_i, z \rangle - \langle c_i, z \rangle = 0, \quad i = 1, \dots, m, \quad (2.13)$$

де z — вектор змінних, а a_i, b_i, c_i — фіксовані вектори коефіцієнтів. Кожне таке рівняння відповідає локальному обмеженню, що перевіряє коректність певної частини обчислення.

QAP. У моделі QAP той самий набір обмежень кодується поліномами $u_j(X)$, $v_j(X)$, $w_j(X)$, які відповідають коефіцієнтам для кожної змінної. Перевірка коректності обчислення зводиться до перевірки того, що різниця між поліномами, які описують ліву та праву частини всіх обмежень, кратна спеціальному поліному $t(X)$, що задає точки оцінки [12]. Таке представлення особливо зручне для zk-SNARK, оскільки природно поєднується з поліноміальними зобов'язаннями та білінійними спарюваннями.

Перетворення обчислювального графа нейромережі у R1CS/QAP є центральним етапом ZKML-пайплайну: від коректності й ефективності цієї арифметизації залежить розмір доказу та час його генерації.

Матричне множення. Лінійні шари нейронної мережі мають вигляд

$$y_i = \sum_j W_{ij} x_j + b_i. \quad (2.14)$$

Кожне додавання та множення може бути безпосередньо представлено в R1CS як окреме обмеження. Кожен елемент $\tilde{z}_j^{(\ell)}$ виражається через $d_{\ell-1}$ множень та додавань над $\mathbb{F}_p$, що дає $d_\ell \cdot d_{\ell-1}$ мультиплікативних обмежень для одного шару. У практичних системах ZKML застосовуються численні оптимізації (агрегація операцій, повторне використання проміжних змінних), щоб зменшити загальну кількість обмежень для великих матриць [17, 9].

Функції активації. Нелінійні активації — найбільш проблемний елемент арифметизації, оскільки ReLU $\sigma(x) = \max(0, x)$ є кусково-лінійною, але не поліноміальною, а Sigmoid та tanh — гладкі

функції з нескінченними рядами Тейлора, які важко точно відобразити поліномами низького степеня над скінченним полем.

Арифметизація ReLU. Функція $\text{ReLU}(\tilde{z}) = \max(0, \tilde{z})$ арифметизується через введення булевої допоміжної змінної $b \in \{0,1\}$:

$$\tilde{a} = \tilde{z} \cdot b, \quad (2.15)$$

$$b(1 - b) = 0, \quad (2.16)$$

$$(\tilde{z} - \tilde{a}) \cdot b = 0. \quad (2.17)$$

Обмеження (3.9) гарантує що b є булевою; обмеження (3.10) — коректність результату. Перевірка знаку потребує додаткових range check обмежень: $0 \leq \tilde{z} < 2^s$ або $2^s \leq \tilde{z} < p$ [24].

Арифметизація Softmax. Функція $\sigma(z)_j = e^{z_j} / \sum_k e^{z_k}$ не може бути точно арифметизована через трансцендентну функцію експоненти. Застосовуються два підходи: поліноміальна апроксимація $\hat{\sigma}$ — заміна softmax наближенням зі збереженням класифікаційних властивостей та верифікація argmax — у схемі верифікується лише $\hat{y} = \text{argmax}(z)$ через порівняння елементів вектора z [25, 9].

У літературі розглядають також кілька загальних підходів до апроксимації нелінійних активацій:

Кусково-поліноміальні апроксимації. Нелінійність апроксимується поліномами невисокого степеня на обмежених інтервалах значень. Це збільшує кількість обмежень, але дозволяє контролювати похибку.

Спеціальні схеми в STARK-системах. Обмеження накладаються безпосередньо на траєкторію виконання (execution trace), що дозволяє опосередковано кодувати поведінку ReLU без явної поліноміальної апроксимації [4].

ZK-friendly активації. Заміна ReLU на поліноміальні функції (x^2 , модифікований softplus) усуває булеві допоміжні змінні, суттєво спрощуючи арифметизацію [9].

Вибір конкретного способу представлення активацій суттєво

впливає на ефективність ZKP: агресивне спрощення зменшує розмір доказу, але може погіршити точність моделі, тоді як точніші схеми призводять до різкого зростання кількості обмежень.

Нейронні мережі в реальних задачах оперують дійсними числами, часто у форматі з плаваючою комою. Натомість протоколи zk-SNARK/zk-STARK зазвичай використовують арифметику в полі $\mathbb{F}_p$, де всі значення є цілими за модулем p . Для узгодження цих двох моделей застосовують квантування [17, 9].

Типова схема квантування:

$$\tilde{w} = \text{round}(w \cdot 2^s) \bmod p, \quad (2.18)$$

де s — масштабний коефіцієнт (scale factor), типово $s \in \{8, 16\}$ біт; p — характеристика поля $\mathbb{F}_p$. Аналогічно квантуються проміжні активації кожного шару ℓ : $\tilde{a}^{(\ell)} = \text{round}(a^{(\ell)} \cdot 2^s) \bmod p$.

Вибір s визначає баланс між точністю моделі та кількістю обмежень у схемі: зниження s з 16 до 8 біт скорочує кількість range check обмежень вдвічі ціною незначної втрати точності [9].

Загальна кількість обмежень арифметичної схеми:

$$C_{\text{total}} = \sum_{\ell=1}^L \left(C_{\text{lin}}^{(\ell)} + C_{\text{act}}^{(\ell)} + C_{\text{range}}^{(\ell)} \right), \quad (2.19)$$

де $C_{\text{lin}}^{(\ell)}$ — обмеження лінійного обчислення, $C_{\text{act}}^{(\ell)}$ — обмеження функції активації, $C_{\text{range}}^{(\ell)}$ — range check обмеження.

Квантування неминуче вносить похибку, але дозволяє застосувати строгі алгебраїчні методи до аналізу коректності навчання. При проєктуванні ZKML-систем необхідно знаходити баланс між точністю (значення s та розрядність) і ефективністю (розмір поля, кількість обмежень).

2.4 Схеми поліноміальних зобов'язань

Поліноміальні зобов'язання (Polynomial Commitments) — криптографічні примітиви, які дозволяють доказувачу закомітити деякий поліном $f(X)$, опублікувавши його коротке представлення (комітмент), а потім багаторазово доводити коректність значення $f(x)$ у різних точках x за допомогою коротких та швидко перевірюваних доказів. Такі схеми є центральним елементом більшості сучасних zk-SNARK та zk-STARK [16, 3].

Основні вимоги:

Зв'язувальність. Неможливо знайти два різні поліноми, які відповідають одному й тому самому комітменту.

Приховувальність (опційно). Комітмент не розкриває інформації про коефіцієнти поліному (для схем з приховуванням). У KZG hiding забезпечується додаванням випадкового маскувального члена до поліному; у FRI — через обмеження на кількість відкритих позицій Merkle-дерева.

Ефективні відкриття. Довести, що певне значення y дорівнює $f(x)$, має бути можливо за допомогою короткого доказу, а перевірка повинна бути швидкою навіть для поліномів великого ступеня.

У задачах ZKML поліноміальні зобов'язання дозволяють зафіксувати структуру та параметри нейронної мережі у вигляді набору поліномів, після чого всі докази коректності навчання чи інференсу посилаються лише на комітмент, не розкриваючи самі ваги.

Схема KZG (Kate–Zaverucha–Goldberg) — одна з найпоширеніших поліноміальних схем зобов'язань, яка базується на білінійних спарюваннях над еліптичними кривими. Її популярність зумовлена сталою довжиною комітментів та короткими доказами, незалежними від ступеня поліному [16].

Основна ідея схеми KZG полягає в тому, що під час довіреного

попереднього налаштування генеруються елементи вигляду g^{τ^i} для $i = 0, \dots, d$, де τ — секретне значення, а d — максимальний ступінь поліному. Нехай задано поліном

$$f(X) = \sum_{i=0}^d f_i X^i. \quad (2.20)$$

Тоді комітмент до f обчислюється як

$$C = g^{f(\tau)} = \prod_{i=0}^d \left(g^{\tau^i}\right)^{f_i}, \quad (2.21)$$

тобто є одним елементом групи. Для відкриття значення $y = f(x)$ доказувач обчислює поліном

$$q(X) = \frac{f(X) - y}{X - x} \quad (2.22)$$

і надає свідоцтво $\pi = g^{q(\tau)}$. Верифікатор перевіряє коректність відкриття за допомогою однієї перевірки на спарюваннях між C , π та публічними параметрами, що еквівалентно перевірці того, що різниця $f(X) - y$ ділиться на $(X - x)$ [16].

KZG-commitment лежить в основі багатьох сучасних SNARK-протоколів (PlonK, Marlin, Sonic тощо) і застосовується у ZKML для компактного комітування великих масивів параметрів нейромереж.

FRI та поліноміальні зобов'язання в zk-STARK

У zk-STARK схеми поліноміальних зобов'язань будуються на протоколі FRI та не потребують білінейних спарювань чи довіреного початкового налаштування. Замість KZG використовуються геш-комітменти до табличних значень поліномів і багаторазові виклики FRI для доведення низького степеня [3].

Загальна структура побудови поліноміальних зобов'язань у zk-STARK полягає в тому, що значення поліному або набору поліномів на великій множині точок кодується у вигляді таблиці, для якої будується

дерево Меркла. Комітментом у такій схемі є корінь дерева Меркла, що забезпечує зв'язувальність і дає змогу виконувати ефективні вибіркові відкриття окремих значень. Далі протокол FRI використовується для доведення того, що закодована таблиця відповідає поліному низького степеня, застосовуючи послідовні перетворення та випадкові запити до оракула [4, 3].

Таблиця 2.2 – Порівняння схем поліноміальних зобов'язань KZG та FRI

Критерій	KZG	FRI
Довірене початкове налаштування	потрібне	не потрібне
Розмір комітменту	$O(1)$	$O(\log n)$
Розмір доказу	$O(1)$	$O(\log^2 n)$
Час верифікації	$O(1)$ (2 спарювання)	$O(\log^2 n)$
Криптографічна база	еліптичні криві	геш-функції
Квантостійкість	ні	так

Висновки до розділу 2

Розглянуті у цьому розділі властивості протоколів доведення з нульовим розголошенням показують, що можливо відділити перевірку коректності обчислень від розкриття даних. За результатами порівняльного аналізу zk-SNARK та zk-STARK було вирішено обрати останній, оскільки STARK забезпечує постквантову стійкість, прозорість та кращу масштабованість для великих обчислень, що є критичним в рамках поставленої задачі.

Арифметизація обчислень і використання поліноміальних зобов'язань формують математичну основу побудови протоколів. Сукупність цих методів створює необхідну математичну основу для розробки протоколів верифікованого навчання та інференсу, що розглядатимуться в наступних розділах.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ НА ОСНОВІ ZK-STARK

У цьому розділі сформульовано задачу, описано архітектуру та програмну реалізацію. Запропонований підхід дозволяє одночасно перевіряти коректність інференсу та зберігати конфіденційність параметрів моделі, вхідних даних та навчального набору.

3.1 Постановка задачі

Нехай задано параметризовану повнозв'язну нейронну мережу, тобто багатошаровий перцептрон (MLP), $\mathcal{M}_\theta : \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}}$, де $\theta = \{W^{(\ell)}, b^{(\ell)}\}_{\ell=1}^L$ — параметри моделі, а L — кількість шарів. Для вхідних даних $x \in \mathbb{R}^{d_{\text{in}}}$ модель обчислює вихід $\hat{y} = \mathcal{M}_\theta(x)$.

Розроблена система розв'язує дві взаємопов'язані задачі: верифікацію інференсу та захист навчальних даних.

Задача 1. Верифікація інференсу. Потрібно побудувати ZKP для твердження

$$\exists (x, \theta) : \mathcal{M}_\theta(x) = \hat{y} \wedge \mathcal{H}(\theta) = h_\theta \wedge \mathcal{H}(x) = h_x, \quad (3.1)$$

де $\mathcal{H}$ — криптографічна геш-функція, а h_θ та h_x — публічно відомі геші параметрів моделі та вхідних даних відповідно. Доказувач має довести, що модель $\mathcal{M}_\theta$ з прихованими параметрами θ справді обчислила результат $\hat{y}$ для прихованого входу x .

Задача 2. Захист навчальних даних. Нехай модель навчена на датасеті $D = \{(x_i, y_i)\}_{i=1}^N$. Необхідно побудувати ZKP для твердження

$$\exists D : \mathcal{H}_{\text{Merkle}}(D) = h_D \wedge \left\| \frac{1}{N} \sum_{i=1}^N \nabla L_i(\theta) \right\| < \varepsilon, \quad (3.2)$$

де h_D — корінь дерева Меркла датасету, L_i — функція втрат на i -му прикладі, а ε — допустимий поріг збіжності. У цьому випадку доказувач доводить, що параметри θ узгоджені з прихованим датасетом D , не розкриваючи самих навчальних прикладів.

Разом твердження (3.1) та (3.2) утворюють єдину систему захисту: верифікатор переконується як у коректності результату класифікації, так і в тому, що модель узгоджується з прихованим навчальним набором даних, не отримуючи доступу ні до вхідних даних, ні до датасету, ні до параметрів моделі.

Система включає двох учасників. Доказувач $\mathcal{P}$, тобто провайдер ML-сервісу, виконує навчання та інференс і формує докази π , використовуючи приватний witness, до якого належать параметри моделі θ , вхідні дані x , навчальний датасет D та проміжні активації. Верифікатор $\mathcal{V}$ перевіряє докази без доступу до моделі, вхідних даних або датасету, спираючись лише на публічні входи протоколу, а саме геші h_θ , h_x , h_D , вихід $\hat{y}$, доказ π та, для частини навчання, фінальний градієнт G_{final} .

Протокол є неінтерактивним: доказувач самостійно формує докази, а верифікатор перевіряє їх без додаткового діалогу. Загальна схема взаємодії має такий вигляд:

1) доказувач навчає модель $\mathcal{M}_\theta$ на датасеті D та публікує геші $h_\theta = \mathcal{H}(\theta)$, $h_x = \mathcal{H}(x)$ і $h_D = \mathcal{H}_{\text{Merkle}}(D)$;

2) доказувач виконує один прохід по датасету D з обчисленням градієнтів, записуючи проміжні значення у таблицю трасування Gradient AIR;

3) доказувач виконує інференс $\hat{y} = \mathcal{M}_\theta(x)$, записуючи проміжні значення у таблицю трасування MLP AIR;

4) доказувач генерує три STARK-докази засобами AIR, Merkle, FRI та Fiat–Shamir і передає їх верифікатору разом з публічними входами;

5) верифікатор перевіряє докази відносно публічних входів $(h_\theta, h_x, h_D, \hat{y}, G_{\text{final}})$ та ухвалює рішення ACCEPT або REJECT.

Для реалізації обрано підхід на основі zk-STARK з кількох причин.

По-перше, STARK не потребує довіреного початкового налаштування, що є важливою перевагою для публічно верифікованих застосувань. По-друге, безпека STARK ґрунтується на стійкості геш-функцій до колізій, що забезпечує постквантову стійкість. По-третє, STARK добре масштабується: складність побудови доказу $O(n \log^2 n)$ та відсутність жорстких обмежень на розмір схеми дають змогу обробляти великі таблиці трасування.

3.2 Алгоритм побудови доказу

ZKP-системи працюють над скінченними полями $\mathbb{F}_p$, тоді як нейронні мережі оперують числами з рухомою комою. Тому першим етапом є квантування всіх параметрів і проміжних значень.

Квантування параметра w визначається як

$$\tilde{w} = \text{round}(w \cdot 2^s) \bmod p, \quad (3.3)$$

де s — масштабний коефіцієнт, а p — характеристика поля $\mathbb{F}_p$.

Аналогічно квантуються проміжні активації кожного шару:

$$\tilde{a}^{(\ell)} = \text{round}(a^{(\ell)} \cdot 2^s) \bmod p. \quad (3.4)$$

Параметри моделі фіксуються через дерево Меркла:

$$h_\theta = \text{MerkleRoot}\left(\left\{\mathcal{H}\left(\tilde{W}^{(\ell)} \parallel \tilde{b}^{(\ell)}\right)\right\}_{\ell=1}^L\right), \quad h_x = \mathcal{H}(\tilde{x}). \quad (3.5)$$

Для захисту навчальних даних кожен приклад датасету гешується окремо:

$$h_D = \text{MerkleRoot}\left(\left\{\mathcal{H}(x_i \parallel y_i)\right\}_{i=1}^N\right). \quad (3.6)$$

Верифікатор отримує лише корінь h_D , тоді як самі приклади залишаються у witness.

Вибір ZK-friendly геш-функції, зокрема Poseidon, є принципово

важливим, оскільки стандартні SHA-256 та Кессак є надто дорогими для арифметизації над $\mathbb{F}_p$.

Після квантування обчислення повнозв'язного шару ℓ подається як

$$\tilde{z}^{(\ell)} = \tilde{W}^{(\ell)} \tilde{a}^{(\ell-1)} + \tilde{b}^{(\ell)} \pmod{p}. \quad (3.7)$$

Функція активації ReLU арифметизується через булеву допоміжну змінну $\beta \in \{0,1\}$:

$$\tilde{a} = \tilde{z} \cdot \beta, \quad (3.8)$$

$$\beta(1 - \beta) = 0, \quad (3.9)$$

$$(\tilde{z} - \tilde{a}) \cdot \beta = 0. \quad (3.10)$$

Перевірка знаку додатково потребує range-check обмежень: $0 \leq \tilde{z} < 2^s$ або $2^s \leq \tilde{z} < p$.

Для верифікації навчання арифметизується один прохід по датасету з обчисленням градієнтів. Для кожного прикладу $(x_i, y_i) \in D$ градієнт вихідного шару має вигляд

$$\delta_k^{(2)} = \hat{y}_k - y_k, \quad k = 0, 1, \quad (3.11)$$

де y_k — one-hot мітка класу. Градієнт прихованого шару обчислюється як

$$\delta_i^{(1)} = \beta_i \cdot \sum_k W_{ki}^{(2)} \cdot \delta_k^{(2)}, \quad i = 0, \dots, d_1 - 1. \quad (3.12)$$

Акумуляція градієнтів по всіх N прикладах виконується за співвідношеннями

$$G_{W^{(2)}}^{(i+1)} = G_{W^{(2)}}^{(i)} + \delta_k^{(2)} \cdot h_j^{(1)}, \quad (3.13)$$

$$G_{W^{(1)}}^{(i+1)} = G_{W^{(1)}}^{(i)} + \delta_i^{(1)} \cdot x_j. \quad (3.14)$$

Фінальний градієнт $G_{\text{final}} = G^{(N)}$ є публічним входом, а верифікатор перевіряє виконання умови збіжності $\max_{i,j} |G_{\text{final}}[i][j]| < \varepsilon$.

Процедура генерації STARK-доказу складається з п'яти кроків:

- 1) побудова таблиць трасування для MLP AIR та Gradient AIR;
 - 2) арифметизація таблиць трасування у вигляді системи поліноміальних обмежень;
 - 3) формування зобов'язань через дерево Меркла з використанням Poseidon як геш-функції;
 - 4) застосування FRI разом із перетворенням Fiat–Shamir, яке робить протокол неінтерактивним;
 - 5) формування доказу $\pi = (\{\text{com}_i\}, \{\text{query openings}\})$.
- Складність побудови доказу становить $O(\mathcal{C}_{\text{total}} \cdot \log^2 \mathcal{C}_{\text{total}})$ за часом та $O(\mathcal{C}_{\text{total}})$ за пам'яттю.

3.3 Протокол верифікації

Верифікатор $\mathcal{V}$ отримує три докази та публічні входи $(h_\theta, h_x, h_D, \hat{y}, G_{\text{final}})$ і виконує перевірку у кілька етапів:

- 1) перевіряє коректність коренів Меркла та відповідність значень h_θ , h_x , h_D ;
- 2) генерує точки запиту через Fiat–Shamir за правилом $r_i = H(\text{com}_{i-1})$;
- 3) перевіряє FRI-відкриття для кожної вибраної точки;
- 4) перевіряє виконання поліноміальних обмежень у точках запиту;
- 5) перевіряє правильність виходу $\hat{y}$ та умову збіжності $\max_{i,j} |G_{\text{final}}[i][j]| < \varepsilon$;
- 6) перевіряє узгодженість між доказами, тобто збіг значень h_θ і h_x у всіх трьох доказах.

Складність верифікації є сублінійною за розміром моделі:

$$T_{\text{verify}} = O(q \cdot \log \mathcal{C}_{\text{total}}), \quad (3.15)$$

де $q = O(\lambda / \log p)$ — кількість FRI-запитів, а λ — параметр безпеки.

Властивість коректності формалізується таким співвідношенням:

$$\Pr [\text{Verify}(\pi^*, \text{PublicInputs}) = \text{ACCERT} \mid \hat{y} \neq \mathcal{M}_\theta(x)] \leq \varepsilon(\lambda) \leq 2^{-\lambda}. \quad (3.16)$$

Властивість нульового розголошення означає існування ефективного симулятора $\mathcal{S}$, для якого

$$\left\{ \text{View}_{\mathcal{V}^*} \left[\mathcal{P}(\tilde{\theta}, \tilde{x}, D) \leftrightarrow \mathcal{V}^* \right] \right\} \approx_c \left\{ \mathcal{S}(\text{PublicInputs}, 1^\lambda) \right\}. \quad (3.17)$$

Таким чином, протокол забезпечує повноту, коректність, нульове розголошення, неінтерактивність, прозорість та захист датасету. Чесний доказувач завжди проходить перевірку FRI і Меркла, підробити таблицю трасування без зміни кореня Меркла неможливо, параметри θ , x і D не розкриваються, а Fiat–Shamir замінює всі інтерактивні виклики на обчислення $H(\cdot)$.

3.4 Програмна реалізація

Для реалізації системи обрано бібліотеку нативного STARK Winterfell 0.13.1. Вона дає змогу безпосередньо описувати AIR для обчислень нейронної мережі та використовувати Poseidon як єдину геш-функцію системи. Усі обчислення виконуються над полем Goldilocks ($p = 2^{64} - 2^{32} + 1$), яке забезпечує ефективну реалізацію на 64-бітних процесорах та містить великий степінь двійки в $\text{ord}(\mathbb{F}_p^*)$, що є необхідним для NTT-перетворень у FRI-протоколі.

Геш-функція Poseidon реалізована над полем Goldilocks з параметрами, наведеними в таблиці 3.1.

Вибір $\alpha = 7$ є коректним для поля Goldilocks, оскільки $\text{gcd}(7, p - 1) = 1$, тому відображення $x \mapsto x^7$ є бієкцією на $\mathbb{F}_p$. MDS-матриця побудована за конструкцією Коші:

$$M[i][j] = (i + j + 8)^{-1} \bmod p, \quad i, j \in \{0, \dots, 7\}. \quad (3.18)$$

Таблиця 3.1 – Параметри Poseidon у реалізованій системі

Параметр	Позначення	Значення
Ширина стану	t	8
S-box (ступінь)	α	7
Повних раундів (з кожного боку)	$R_f/2$	4
Часткових раундів	R_p	22
Загальна кількість раундів	R	30
Rate (sponge)	r	4
Capacity (sponge)	c	4

Раундові константи визначено детерміновано:

$$rc[r][i] = (r \cdot t + i + 1) \cdot 1\,000\,003 \bmod p. \quad (3.19)$$

Реалізовано структуру `PoseidonHasher`, яка імплементує трейти `Hasher` та `ElementHasher`. Метод `hash_elements` використовує `Felt::from_mont()` для коректної обробки Montgomery representation поля Goldilocks. Після інтеграції всі Merkle commitments і Fiat–Shamir виклики в усіх трьох доказах виконуються через Poseidon.

У системі реалізовано MLP архітектури $4 \rightarrow 8 \rightarrow 2$: вхід має розмірність 4, прихований шар містить 8 нейронів з активацією ReLU, а вихідний шар має розмірність 2. Навчання виконувалося у PyTorch 2.1.0 з використанням оптимізатора Adam ($\text{lr} = 0,01$), функції втрат `CrossEntropyLoss`, 2000 епох та масштабу $\text{SCALE} = 1024$.

Датасет містить 8000 синтетичних прикладів з простору $\{0, \dots, 255\}^4$, що відповідає 2^{32} можливим комбінаціям. Правило класифікації задається співвідношенням

$$y = \begin{cases} 0, & x_0 \cdot x_1 > x_2 \cdot x_3, \\ 1, & \text{інакше.} \end{cases} \quad (3.20)$$

Датасет поділено на 6400 тренувальних і 1600 тестових прикладів. Досягнута точність становить 98.4% для float-подання та 97.8% для

int-подання. Квантування параметрів виконується за формулою

$$\tilde{w} = \text{round}(w \cdot 1024). \quad (3.21)$$

Система складається з трьох STARK-доказів, зв'язаних через спільні публічні входи:

$$h_x = \text{Poseidon}(x) \in \text{MlpPub} \cap \text{PoseidonPub}, \quad (3.22)$$

$$h_\theta = \text{Poseidon}(\theta) \in \text{MlpPub} \cap \text{GradientPub}. \quad (3.23)$$

Таблиця 3.2 – Характеристики трьох STARK-доказів системи

Параметр	MLP AIR	Poseidon AIR	Gradient AIR
Розмір трасування	8×26	32×48	8192×91
Обмежень AIR	26	47	84
Час proving	0.57 с	0.09 с	2.2 с
Час верифікації	0.006 с	0.0001 с	~ 0.01 с
Розмір доказу	7.8 КБ	15.1 КБ	68.2 КБ
Верифікація	Так	Так	Так
Загальний час proving	≈ 2.9 с		
Загальний розмір	≈ 91.1 КБ		

Доказ 1 — MLP AIR: верифікація інференсу. Таблиця MLP AIR містить 8×26 елементів над полем Goldilocks. Рядок 0 містить вхід x . Рядок 1 містить передактивації $z_1[0..7]$, знакові біти $\beta[0..7]$, активації $h_1[0..7]$ та селектор $s_1 = 1$. Рядок 2 містить вихід $\hat{y}[0..1]$ і селектор $s_2 = 1$, а рядки 3–7 є padding.

Для кожного нейрона $i = 0, \dots, 7$ першого шару задаються обмеження

$$s_1 \cdot (z_i^{(1)} - \sum_{j=0}^3 W_1[i][j]x_j - b_1[i]) = 0, \quad (3.24)$$

$$s_1 \cdot \beta_i \cdot (1 - \beta_i) = 0, \quad (3.25)$$

$$s_1 \cdot (h_i - \beta_i \cdot z_i^{(1)}) = 0. \quad (3.26)$$

Для виходів $j = 0, 1$ використовується обмеження

$$s_2 \cdot (\hat{y}_j - \sum_{i=0}^7 W_2[j][i]h_i - b_2[j]) = 0. \quad (3.27)$$

Вхід x залишається приватним і не включається до assertions, а зв'язок між x та публічним h_x забезпечується окремим Poseidon AIR.

Доказ 2 — Poseidon AIR: приватність вхідних даних. Окремий AIR доводить, що $h_x = \text{Poseidon}(x)$ для конкретного входу x , який використовується у трасуванні MLP AIR. Таблиця має розмір 32×48 і охоплює 30 раундів та 2 padding-рядки. Стовпці зберігають проміжні значення кожного раунду: $\text{pre}[i]$, $x^2[i]$, $x^4[i]$, $x^6[i]$, $x^7[i]$ та $\text{state}[i]$ для $i = 0, \dots, 7$. Assertions фіксують початковий стан $[x_0, x_1, x_2, x_3, 0, \dots, 0]$ та кінцевий стан у рядку 29, який дорівнює h_x .

Доказ 3 — Gradient AIR: захист навчальних даних. Замість доведення всього процесу навчання, що для 2000 епох є обчислювально надто дорогим, доводиться умова малої норми фінального градієнта на прихованому датасеті:

$$G_{\text{final}} = \frac{1}{N} \sum_{i=1}^N \nabla L_i(W_{\text{final}}) \approx 0. \quad (3.28)$$

Перевірка $\|G_{\text{final}}\| < \varepsilon$ виконується верифікатором поза STARK-схемою на основі публічного значення G_{final} .

Таблиця Gradient AIR містить 8192×91 елементів, тобто 6400 прикладів і padding до 2^{13} . Кожен рядок відповідає одному тренувальному прикладу та містить стовпці $x[4]$, $y[2]$, $z_1[8]$, $\beta[8]$, $h_1[8]$, $\hat{y}[2]$, $\delta_2[2]$, $\delta_1[8]$, $G_{W_2}[16]$, $G_{W_1}[32]$ і $\text{sel}[1]$.

Gradient AIR містить 84 обмеження трьох типів. До першого типу належать обмеження прямого проходу, які збігаються з MLP AIR і активуються через sel . Другий тип утворюють обмеження зворотного

проходу:

$$\text{sel} \cdot (\delta_{2,k} - \hat{y}_k + y_k) = 0, \quad (3.29)$$

$$\text{sel} \cdot (\delta_{1,i} - \beta_i \cdot \sum_k W_2[k][i] \cdot \delta_{2,k}) = 0. \quad (3.30)$$

Третій тип становлять обмеження акумуляції градієнтів:

$$G_{W_2}^{(\text{next})}[k][i] = G_{W_2}^{(\text{cur})}[k][i] + \text{sel} \cdot \delta_{2,k} \cdot h_{1,i}, \quad (3.31)$$

$$G_{W_1}^{(\text{next})}[i][j] = G_{W_1}^{(\text{cur})}[i][j] + \text{sel} \cdot \delta_{1,i} \cdot x_j. \quad (3.32)$$

Кожен приклад датасету гешується як $\text{leaf}_i = \text{Poseidon}(x_i, y_i)$, після чого будується дерево Меркла з 8192 листів. Корінь `dataset_root` публікується як публічний вхід. Узгодженість між MLP AIR і Gradient AIR забезпечується порівнянням спільного значення h_θ .

Результати. Для тестового прикладу $x = [92, 213, 170, 66]$, для якого виконується $92 \cdot 213 = 19596 > 170 \cdot 66 = 11220$, тобто правильний клас дорівнює 0, отримано результати, наведені в таблиці 3.3.

Таблиця 3.3 – Результати роботи системи трьох STARK-доказів

Параметр	Значення
Архітектура мережі	$4 \rightarrow 8 \rightarrow 2$
Простір входів	$\{0, \dots, 255\}^4$ (2^{32} комбінацій)
Датасет	6400 прикладів (прихований)
Точність float/int	98.4% / 97.8%
Вхід x (приватний)	[92, 213, 170, 66]
Вихід $\hat{y}$ (публічний)	[7 739 399; -7 719 828], клас 0
h_x	6031a438...
h_θ	44ca3a3c...
<code>dataset_root</code>	1fb52f12...
h_x збігається	так
h_θ збігається	так
Загальний час proving	≈ 2.9 с
Загальний розмір доказів	≈ 91.1 КБ
Trusted setup	не потрібен
Постквантова стійкість	так

Аналіз безпеки. Повнота виражається співвідношенням

$$\Pr[\mathcal{V} \text{ приймає} \mid \text{чесний } \mathcal{P}] = 1. \quad (3.33)$$

Для оцінки коректності використано параметри FRI $q = 32$, $\text{blowup} = 8$ та $\text{grinding} = 16$ біт. Тоді верхня оцінка імовірності підробки доказу має вигляд

$$\varepsilon_{\text{sound}} \leq \frac{d \cdot \rho^{-1}}{|\mathbb{F}_p|} + (1 - \rho)^q + 2^{-16}, \quad (3.34)$$

де $d = 3$, $\rho = 1/8$, $|\mathbb{F}_p| \approx 2^{64}$. У практичних налаштуваннях такі параметри забезпечують рівень безпеки близько 128 біт.

Властивість нульового розголошення подається у вигляді

$$\{(\pi_{\text{MLP}}, \pi_{\text{Pos}}, \pi_{\text{Grad}}) \mid \mathcal{P}(x, \theta, D)\} \approx_c \{\mathcal{S}(\hat{y}, h_x, h_\theta, \text{dataset_root})\}. \quad (3.35)$$

Таблиця 3.4 – Аналіз безпеки системи

Тип події	Імовірність	Забезпечується
Підробка доказу	практично неможлива (рівень безпеки ≈ 128 біт)	Soundness, параметри FRI ($q = 32$), grinding (2^{16}).
Хибна відмова	0	Completeness за умови коректного виконання протоколу.
Витік x	практично неможливий	Zero-knowledge властивість протоколу разом із стійкістю Poseidon до колізій та другого прообразу.
Підробка h_θ	практично неможлива (рівень безпеки ≈ 128 біт)	Стійкість до колізій геш-функції Poseidon.
Підробка dataset_root	практично неможлива (рівень безпеки ≈ 128 біт)	Стійкість до колізій геш-функції Poseidon.
Відновлення x перебором	$\leq 2^{-32}$	Скінченний простір вхідних даних $\{0, \dots, 255\}^4$.

Перевірка умови $\|G_{\text{final}}\| < \varepsilon$ виконується верифікатором поза STARK-схемою. Її включення всередину доказу потребувало б range-proof конструкції, що перевищує ліміт у 255 стовпців у Winterfell 0.13.1.

Висновки до розділу 3

У розділі розроблено та описано систему, що поєднує верифікацію інференсу нейронної мережі із захистом тренувальних даних на основі нативного zk-STARK. Узгодженість усіх частин протоколу забезпечується трьома доказами, пов'язаними через значення h_x та h_θ . У межах реалізації використано хеш-функцію Poseidon над полем Goldilocks та виконано її інтеграцію у Winterfell 0.13.1. Експериментальні результати показали, що загальний час побудови доказів становить близько 2.9 с, а сумарний розмір доказів — близько 91.1 КБ. Запропонована система не потребує довіреного початкового налаштування та для обраної конфігурації забезпечує практичний рівень безпеки 128 біт.

ВИСНОВКИ

У ході виконання дипломної роботи розроблено та програмно реалізовано протокол верифікації інференсу із одночасним захистом навчальних даних на основі zk-STARK.

Аналіз основних загроз конфіденційності в системах машинного навчання показав, що атаки на вивід членства, інверсійні атаки та атаки отруєння даних становлять істотну небезпеку для систем, що працюють із чутливою інформацією. Водночас федеративне навчання, гомоморфне шифрування та багатосторонні обчислення вирішують лише окремі аспекти проблеми, тоді як жоден із цих підходів не поєднує конфіденційність даних із публічною верифікованістю обчислень. Огляд існуючих zkML-фреймворків показав їхню орієнтацію переважно на верифікацію інференсу без урахування захисту тренувальних даних, що підтвердило актуальність поставленої задачі.

Порівняльне дослідження протоколів zk-SNARK та zk-STARK обґрунтувало вибір останнього як основи реалізації. Відсутність довіреного початкового налаштування та постквантова стійкість є ключовими перевагами для публічно верифікованих застосувань. Арифметизація лінійних шарів, подання функції активації ReLU через булеву допоміжну змінну та квантування параметрів над скінченним полем сформували математичну основу побудованої системи. Серед схем поліноміальних зобов'язань KZG та FRI обрано FRI як механізм, що не потребує довіреного початкового налаштування і зберігає достатню ефективність верифікації.

На основі бібліотеки Winterfell 0.13.1 розроблено та реалізовано систему верифікації інференсу MLP у вигляді двох взаємопов'язаних доказів. MLP AIR із таблицею трасування 8×26 елементів і 26 поліноміальними обмеженнями перевіряє коректність прямого проходу мережі. Окремий Poseidon AIR із таблицею 32×48 і 47 обмеженнями

підтверджує відповідність публічного гешу h_x приватному входу x , забезпечуючи конфіденційність вхідних даних. Сумарний час генерації цих двох доказів становить 0,66 с, а розмір — 22,9 КБ.

Механізм захисту навчальних даних реалізовано через Gradient AIR із таблицею трасування 8192×91 елемент і 84 обмеженнями. Доказ підтверджує, що фінальний градієнт обчислено на прихованому датасеті з 6400 прикладів, зафіксованому через корінь дерева Меркла h_D . Водночас перевірку норми градієнта винесено поза STARK-схему через обмеження у 255 стовпців Winterfell 0.13.1, тому верифікатор перевіряє умову збіжності безпосередньо на основі публічного значення G_{final} .

Аналіз безпеки підтвердив виконання властивостей повноти, коректності та нульового розголошення. За параметрів FRI $q = 32$, $\text{blowup} = 8$ та $\text{grinding} = 16$ біт імовірність підробки доказу не перевищує 2^{-128} . Загальний час генерації трьох доказів становить приблизно 2,9 с, а сумарний розмір — близько 91,1 КБ, при цьому система не потребує довіреного початкового налаштування та зберігає постквантову стійкість.

Таким чином, розроблено інтегрований протокол трьох взаємопов'язаних STARK-доказів, зв'язаних через спільні публічні входи h_θ та h_x , який забезпечує одночасну верифікацію інференсу та захист навчального датасету без розкриття параметрів моделі, вхідних даних і тренувальних прикладів.

Подальші дослідження доцільно спрямувати на включення перевірки норми градієнта безпосередньо до STARK-схеми через range-proof конструкції, розширення системи на глибші архітектури нейронних мереж, зокрема згорткові, застосування рекурсивної композиції STARK-доказів для верифікації повного процесу навчання, а також інтеграцію з блокчейн-платформами для публічної верифікації ML-рішень у децентралізованих застосуваннях.

ПЕРЕЛІК ПОСИЛАНЬ

- [1] Various Authors. «OATH: Efficient and Flexible Zero-Knowledge Proofs of End-to-End ML Fairness». АНГЛ. В: *arXiv preprint arXiv:2410.02777* (2024).
- [2] Aztec Network. *Noir: A Domain-Specific Language for Zero-Knowledge Applications*. АНГЛ. <https://noir-lang.org/docs/>. Online documentation. 2023.
- [3] Eli Ben-Sasson, Iddo Bentov, Lior Goldberg та ін. «Fast Reed–Solomon Interactive Oracle Proofs of Proximity». АНГЛ. В: *Electronic Colloquium on Computational Complexity (ECCC)* (2017). ECCC TR17-134.
- [4] Eli Ben-Sasson та ін. «Scalable, Transparent, and Post-Quantum Secure Computational Integrity». АНГЛ. В: *Proceedings of the 4th Workshop on Bitcoin and Blockchain Research*. 2018.
- [5] Eli Ben-Sasson та ін. «SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge». АНГЛ. В: *23rd USENIX Security Symposium*. 2014, с. 67–82.
- [6] Eli Ben-Sasson та ін. «Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture». АНГЛ. В: *23rd USENIX Security Symposium*. 2014, с. 781–796.
- [7] Battista Biggio, Blaine Nelson та Pavel Laskov. «Poisoning Attacks against Support Vector Machines». АНГЛ. В: *Proceedings of the 29th International Conference on Machine Learning (ICML)*. 2012, с. 1467–1474.
- [8] Electric Coin Company. *Technical Explainer: Halo 2 on Zcash*. АНГЛ. <https://electriccoin.co/blog/technical-explainer-halo-on-zcash/>. 2021.
- [9] Runyu Feng, Zhichao Qin, Yupeng Zhang та ін. «ZEN: Efficient Zero-Knowledge Proofs for Neural Networks». АНГЛ. В: *arXiv preprint arXiv:2109.06890* (2021).

- [10] Amos Fiat та Adi Shamir. «How to Prove Yourself: Practical Solutions to Identification and Signature Problems». АНГЛ. В: *Advances in Cryptology – CRYPTO’86*. Т. 263. Lecture Notes in Computer Science. 1987, с. 186–194.
- [11] Matt Fredrikson, Somesh Jha та Thomas Ristenpart. «Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures». АНГЛ. В: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015, с. 1322–1333.
- [12] Rosario Gennaro та ін. «Quadratic Span Programs and Succinct NIZKs without PCPs». АНГЛ. В: *Advances in Cryptology – EUROCRYPT 2013*. Т. 7881. Lecture Notes in Computer Science. 2013, с. 626–645.
- [13] Craig Gentry. «Fully Homomorphic Encryption Using Ideal Lattices». АНГЛ. В: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*. 2009, с. 169–178.
- [14] Shafi Goldwasser, Silvio Micali та Charles Rackoff. «The Knowledge Complexity of Interactive Proof Systems». АНГЛ. В: *SIAM Journal on Computing* 18.1 (1989), с. 186–208.
- [15] Iden3. *Circom: A Circuit Description Language for Zero-Knowledge Proofs*. АНГЛ. <https://docs.circom.io/>. Online documentation. 2023.
- [16] Aniket Kate, Gregory Zaverucha та Ian Goldberg. «Constant-size Commitments to Polynomials and Their Applications». АНГЛ. В: *Advances in Cryptology – ASIACRYPT 2010*. Т. 6477. Lecture Notes in Computer Science. 2010, с. 177–194.
- [17] Tianyi Liu, Xiang Xie, Yupeng Zhang та ін. «zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy». АНГЛ. В: *arXiv preprint arXiv:2109.06532* (2021).
- [18] H. Brendan McMahan та ін. «Communication-Efficient Learning of Deep Networks from Decentralized Data». АНГЛ. В: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2017, с. 1273–1282.

- [19] Bryan Parno та ін. «Pinocchio: Nearly Practical Verifiable Computation». АНГЛ. В: *2013 IEEE Symposium on Security and Privacy (SP)*. 2013, с. 238—252.
- [20] Ahmed Salmed та ін. «ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models». АНГЛ. В: *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. 2018.
- [21] Adi Shamir. «How to Share a Secret». АНГЛ. В: *Communications of the ACM* 22.11 (1979), с. 612—613.
- [22] Reza Shokri та ін. «Membership Inference Attacks against Machine Learning Models». АНГЛ. В: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, с. 3—18.
- [23] StarkWare. *The Cairo Programming Language*. АНГЛ. <https://www.starknet.io/cairo-book/>. Online documentation. 2023.
- [24] Haochen Sun та ін. «zkDL: Efficient Zero-Knowledge Proofs of Deep Learning Training». АНГЛ. В: *arXiv preprint arXiv:2307.16273* (2023).
- [25] Haochen Sun та ін. «zkLLM: Zero Knowledge Proofs for Large Language Models». АНГЛ. В: *arXiv preprint arXiv:2404.16109* (2024).
- [26] Chhavi Yadav та ін. «FairProof: Confidential and Certifiable Fairness for Neural Networks». АНГЛ. В: *arXiv preprint arXiv:2402.12572* (2024).
- [27] Zkonduit. *EZKL: Easy Zero-Knowledge Learning*. АНГЛ. <https://github.com/zkonduit/ezkl>. Library and CLI for zkML over Halo2. 2022.

ДОДАТОК А ТЕКСТИ ПРОГРАМ

Лістинг 1: src/main.rs

```

1  mod poseidon_air;
2  mod gradient_air;
3
4  use std::time::Instant;
5
6  use poseidon_air::{
7      bytes32_to_felt4, build_trace as poseidon_build_trace, validate_trace as
          poseidon_validate,
8      Felt, PoseidonAir, PoseidonDigest, PoseidonHasher, PoseidonProver, PoseidonPub,
9      PoseidonProofHasher, PoseidonProofVC, PoseidonProofCoin,
10 };
11
12 use gradient_air::{
13     GradientAir, GradientProver, GradientPub,
14     build_gradient_trace, dataset_merkle_root, load_dataset,
15     GRAD_TW, GRAD_NC,
16 };
17 use winter_crypto::{DefaultRandomCoin, Digest, ElementHasher, MerkleTree};
18 use winterfell::{
19     math::{FieldElement, StarkField, ToElements},
20     matrix::ColMatrix,
21     Air, AirContext, Assertion, AuxRandElements, BatchingMethod, CompositionPoly,
22     CompositionPolyTrace, ConstraintCompositionCoefficients, DefaultConstraintCommitment,
23     DefaultConstraintEvaluator, DefaultTraceLde, EvaluationFrame, FieldExtension,
24     AcceptableOptions, PartitionOptions, ProofOptions, Prover, StarkDomain, TraceInfo,
25     TracePolyTable, TraceTable, TransitionConstraintDegree,
26 };
27
28 pub const W1: [[i64; 4]; 8] = [
29     [ 8, -10,  7,  7],
30     [ 10,  9, -15, 15],
31     [ 0,  0,  0,  0],
32     [ 9,  9,  7, -16],
33     [ 1,  0, 13, -15],
34     [-12, -9, 14, 14],
35     [-12, 17,  2,  4],
36     [ 15, -13,  1,  1],
37 ];
38 pub const B1: [i64; 8] = [-326, 106, -1, 42, -26, 196, 13, -36];
39 pub const W2: [[i64; 8]; 2] = [
40     [-2991, 4494,  0, 3862, 3590, -4437, -3908, -3499],

```

```

41     [ 3008, -4523,    0, -3841, -3598,  4455,  3908,  3506],
42 ];
43 pub const B2: [i64; 2] = [-450, 453];
44
45 const TW: usize = 26;
46 const TN: usize = 8;
47 const NC: usize = 26;
48 const NA: usize = 4;
49
50 pub fn f(x: i64) -> Felt {
51     if x >= 0 { Felt::new(x as u64) }
52     else      { Felt::new(0) - Felt::new((-x) as u64) }
53 }
54
55 pub fn compute_h_theta() -> PoseidonDigest {
56     let mut elems: Vec<Felt> = Vec::new();
57     for row in &W1 { for &w in row { elems.push(f(w)); } }
58     for &b in &B1 { elems.push(f(b)); }
59     for row in &W2 { for &w in row { elems.push(f(w)); } }
60     for &b in &B2 { elems.push(f(b)); }
61     PoseidonHasher::hash_elements(&elems)
62 }
63
64 pub fn felt_to_i64(v: Felt) -> i64 {
65     let u = v.as_int();
66     if u <= Felt::MODULUS / 2 { u as i64 } else { -(Felt::MODULUS - u) as i64 }
67 }
68
69 struct Mlp { z1: [Felt; 8], bs1: [Felt; 8], h1: [Felt; 8], out: [Felt; 2] }
70
71 fn forward(x: [Felt; 4]) -> Mlp {
72     let xi: [i64; 4] = std::array::from_fn(|j| x[j].as_int() as i64);
73     let z1i: [i64; 8] = std::array::from_fn(|i|
74         (0..4).map(|j| W1[i][j] * xi[j]).sum::<i64>() + B1[i]);
75     let bs1: [Felt; 8] = std::array::from_fn(|i|
76         if z1i[i] >= 0 { Felt::ONE } else { Felt::ZERO });
77     let z1: [Felt; 8] = std::array::from_fn(|i| f(z1i[i]));
78     let h1: [Felt; 8] = std::array::from_fn(|i| bs1[i] * z1[i]);
79     let hi: [i64; 8] = std::array::from_fn(|i| felt_to_i64(h1[i]));
80     let oi: [i64; 2] = std::array::from_fn(|j|
81         (0..8).map(|i| W2[j][i] * hi[i]).sum::<i64>() + B2[j]);
82     let out: [Felt; 2] = std::array::from_fn(|j| f(oi[j]));
83     Mlp { z1, bs1, h1, out }
84 }
85

```

```

86 fn build_mlp_trace(x: [Felt; 4]) -> TraceTable<Felt> {
87     let r = forward(x);
88     let mut t = TraceTable::new(TW, TN);
89     t.fill(
90         |row| { for i in 0..4 { row[i] = x[i]; } },
91         |step, row| {
92             for i in 0..TW { row[i] = Felt::ZERO; }
93             match step {
94                 0 => {
95                     for i in 0..8 {
96                         row[i] = r.h1[i];
97                         row[8 + i] = r.z1[i];
98                         row[16 + i] = r.bs1[i];
99                     }
100                     row[24] = Felt::ONE;
101                 }
102                 1 => {
103                     row[0] = r.out[0];
104                     row[1] = r.out[1];
105                     row[25] = Felt::ONE;
106                 }
107                 _ => {}
108             }
109         },
110     );
111     t
112 }
113
114 #[derive(Clone)]
115 pub struct MlpPub {
116     out: [Felt; 2],
117     hx: PoseidonDigest,
118     h_theta: PoseidonDigest,
119 }
120
121 impl ToElements<Felt> for MlpPub {
122     fn to_elements(&self) -> Vec<Felt> {
123         let mut v: Vec<Felt> = self.out.iter().copied().collect();
124         for digest in [&self.hx, &self.h_theta] {
125             for chunk in digest.as_bytes().chunks(8) {
126                 let mut b = [0u8; 8]; b[..chunk.len()].copy_from_slice(chunk);
127                 v.push(Felt::new(u64::from_le_bytes(b)));
128             }
129         }
130         v

```

```

131     }
132 }
133
134 pub struct MlpAir { ctx: AirContext<Felt>, inp: MlpPub }
135
136 impl Air for MlpAir {
137     type BaseField      = Felt;
138     type PublicInputs    = MlpPub;
139
140     fn new(info: TraceInfo, p: MlpPub, opts: ProofOptions) -> Self {
141         let mut deg: Vec<TransitionConstraintDegree> = Vec::with_capacity(NC);
142         for _ in 0..8 {
143             deg.push(TransitionConstraintDegree::new(2));
144             deg.push(TransitionConstraintDegree::new(3));
145             deg.push(TransitionConstraintDegree::new(3));
146         }
147         for _ in 0..2 { deg.push(TransitionConstraintDegree::new(2)); }
148         debug_assert_eq!(deg.len(), NC);
149         MlpAir {
150             ctx: AirContext::new(info, deg, NA, opts),
151             inp: p,
152         }
153     }
154
155     fn context(&self) -> &AirContext<Felt> { &self.ctx }
156
157     fn evaluate_transition<E: FieldElement + From<Felt>>(&self, frame: &EvaluationFrame<E>, _: &[E], res: &mut [E],
158 ) {
159         let (cur, next) = (frame.current(), frame.next());
160         let (s11, s12) = (next[24], next[25]);
161         let mut c = 0;
162
163         for i in 0..8 {
164             let (nz, nbs, nh) = (next[8 + i], next[16 + i], next[i]);
165             let ez: E = (0..4)
166                 .map(|j| E::from(f(W1[i][j]))) * cur[j])
167                 .fold(E::ZERO, |a, b| a + b) + E::from(f(B1[i]));
168             res[c] = s11 * (nz - ez);          c += 1;
169             res[c] = s11 * (nbs * (E::ONE - nbs)); c += 1;
170             res[c] = s11 * (nh - nbs * nz);    c += 1;
171         }
172         for j in 0..2 {
173             let eo: E = (0..8)
174                 .map(|i| E::from(f(W2[j][i]))) * cur[i])

```

```

176         .fold(E::ZERO, |a, b| a + b) + E::from(f(B2[j]));
177         res[c] = s12 * (next[j] - eo);          c += 1;
178     }
179 }
180
181 fn get_assertions(&self) -> Vec<Assertion<Felt>> {
182     vec![
183         Assertion::single(0, 2, self.inp.out[0]),
184         Assertion::single(1, 2, self.inp.out[1]),
185         Assertion::single(24, 1, Felt::ONE),
186         Assertion::single(25, 2, Felt::ONE),
187     ]
188 }
189 }
190
191 struct MlpProver { opts: ProofOptions }
192 impl MlpProver { fn new(o: ProofOptions) -> Self { Self { opts: o } } }
193
194 impl Prover for MlpProver {
195     type BaseField = Felt;
196     type Air = MlpAir;
197     type Trace = TraceTable<Felt>;
198     type HashFn = PoseidonHasher;
199     type VC = MerkleTree<PoseidonHasher>;
200     type RandomCoin = DefaultRandomCoin<PoseidonHasher>;
201     type TraceLde<E: FieldElement<BaseField = Felt>> =
202         DefaultTraceLde<E, PoseidonHasher, MerkleTree<PoseidonHasher>>;
203     type ConstraintCommitment<E: FieldElement<BaseField = Felt>> =
204         DefaultConstraintCommitment<E, PoseidonHasher, MerkleTree<PoseidonHasher>>;
205     type ConstraintEvaluator<'a, E: FieldElement<BaseField = Felt>> =
206         DefaultConstraintEvaluator<'a, Self::Air, E>;
207
208     fn get_pub_inputs(&self, t: &Self::Trace) -> MlpPub {
209         let x: [Felt; 4] = std::array::from_fn(|i| t.get(i, 0));
210         let out: [Felt; 2] = std::array::from_fn(|j| t.get(j, 2));
211         let hx = PoseidonHasher::hash_elements(&x);
212         let h_theta = compute_h_theta();
213         MlpPub { out, hx, h_theta }
214     }
215
216     fn options(&self) -> &ProofOptions { &self.opts }
217
218     fn new_trace_lde<E: FieldElement<BaseField = Felt>>(
219         &self, i: &TraceInfo, m: &ColMatrix<Felt>,
220         d: &StarkDomain<Felt>, p: PartitionOptions,

```

```

221     ) -> (Self::TraceLde<E>, TracePolyTable<E>) {
222         DefaultTraceLde::new(i, m, d, p)
223     }
224
225     fn build_constraint_commitment<E: FieldElement<BaseField = Felt>>(
226         &self, p: CompositionPolyTrace<E>, n: usize,
227         d: &StarkDomain<Felt>, po: PartitionOptions,
228     ) -> (Self::ConstraintCommitment<E>, CompositionPoly<E>) {
229         DefaultConstraintCommitment::new(p, n, d, po)
230     }
231
232     fn new_evaluator<'a, E: FieldElement<BaseField = Felt>>(
233         &self, a: &'a Self::Air,
234         aux: Option<AuxRandElements<E>>,
235         c: ConstraintCompositionCoefficients<E>,
236     ) -> Self::ConstraintEvaluator<'a, E> {
237         DefaultConstraintEvaluator::new(a, aux, c)
238     }
239 }
240
241 fn main() {
242     let x: [Felt; 4] = [Felt::new(92), Felt::new(213), Felt::new(170), Felt::new(66)];
243     let mlp = forward(x);
244     let pred = if felt_to_i64(mlp.out[0]) > felt_to_i64(mlp.out[1]) { 0 } else { 1 };
245     let hx_display: PoseidonDigest = PoseidonHasher::hash_elements(&x);
246
247     let opts = ProofOptions::new(
248         32, 8, 16, FieldExtension::None, 4, 127,
249         BatchingMethod::Linear, BatchingMethod::Linear,
250     );
251
252     let mlp_trace = build_mlp_trace(x);
253     let mlp_hx: PoseidonDigest = PoseidonHasher::hash_elements(&x);
254     let mlp_h_theta = compute_h_theta();
255     let mlp_pub = MlpPub { out: mlp.out, hx: mlp_hx, h_theta: mlp_h_theta };
256     let t1 = Instant::now();
257     let mlp_proof = MlpProver::new(opts.clone()).prove(mlp_trace)
258         .expect("MLP_prover_failed");
259     let mlp_pt = t1.elapsed();
260     let mlp_proof_size = mlp_proof.to_bytes().len();
261     let t1v = Instant::now();
262     winterfell::verify::<MlpAir, PoseidonHasher, DefaultRandomCoin<PoseidonHasher>,
263         MerkleTree<PoseidonHasher>>(
264         mlp_proof, mlp_pub, &AcceptableOptions::OptionSet(vec![opts.clone()]),
265     ).expect("MLP_verification_failed");

```

```

265     let mlp_vt = t1v.elapsed();
266
267     let pos_trace = poseidon_build_trace(x);
268     poseidon_validate(&pos_trace);
269     let pos_pub = PoseidonPub {
270         initial: std::array::from_fn(|i| pos_trace.get(poseidon_air::COL_STATE + i, 0)),
271         output: std::array::from_fn(|i| pos_trace.get(poseidon_air::COL_STATE + i,
272             poseidon_air::N_ROUNDS)),
273     };
274     let hx_from_trace: [Felt; 4] = std::array::from_fn(|i| pos_pub.output[i]);
275     let t2 = Instant::now();
276     let pos_proof = PoseidonProver::new(opts.clone()).prove(pos_trace)
277         .expect("Poseidon_prover_failed");
278     let pos_pt = t2.elapsed();
279     let pos_proof_size = pos_proof.to_bytes().len();
280     let t3 = Instant::now();
281     winterfell::verify::<PoseidonAir, PoseidonProofHasher, PoseidonProofCoin,
282         PoseidonProofVC>(
283         pos_proof, pos_pub, &AcceptableOptions::OptionSet(vec![opts.clone()]),
284     ).expect("Poseidon_verification_failed");
285     let pos_vt = t3.elapsed();
286
287     let mlp_hx_felts: [Felt; 4] = bytes32_to_felt4(mlp_hx.as_bytes());
288     let hx_matches = mlp_hx_felts == hx_from_trace;
289
290     let dataset_path = "dataset.csv";
291     let examples = load_dataset(dataset_path);
292     let n_examples = examples.len();
293     let dataset_root = dataset_merkle_root(&examples);
294     let (grad_trace, g_w1_final, g_w2_final) = build_gradient_trace(&examples);
295     let grad_h_theta = compute_h_theta();
296     let grad_pub = GradientPub {
297         dataset_root,
298         h_theta: grad_h_theta,
299         g_w1_final,
300         g_w2_final,
301         n_examples,
302     };
303     let t4 = Instant::now();
304     let grad_proof = GradientProver::new(opts.clone(), n_examples, dataset_root)
305         .prove(grad_trace)
306         .expect("Gradient_prover_failed");
307     let grad_pt = t4.elapsed();
308     let grad_proof_size = grad_proof.to_bytes().len();
309     let t5 = Instant::now();

```

```

308     winterfell::verify::<GradientAir, PoseidonHasher, DefaultRandomCoin<PoseidonHasher>,
        MerkleTree<PoseidonHasher>>>(
309         grad_proof, grad_pub, &AcceptableOptions::OptionSet(vec![opts]),
310     ).expect("Gradient_verification_failed");
311     let grad_vt = t5.elapsed();
312
313     let max_gw1 = g_w1_final.iter().flat_map(|r| r.iter()).map(|&v| v.abs()).max().unwrap_or
        (0);
314     let max_gw2 = g_w2_final.iter().flat_map(|r| r.iter()).map(|&v| v.abs()).max().unwrap_or
        (0);
315     let theta_ok = mlp_h_theta.as_bytes() == grad_h_theta.as_bytes();
316
317     if !hx_matches || !theta_ok { std::process::exit(1); }
318 }

```

ЛІСТИНГ 2: src/gradient_air.rs

```

1  use winter_crypto::{Digest, ElementHasher, MerkleTree};
2  use winterfell::{
3      math::{FieldElement, ToElements},
4      matrix::ColMatrix,
5      Air, AirContext, Assertion, AuxRandElements, BatchingMethod, CompositionPoly,
6      CompositionPolyTrace, ConstraintCompositionCoefficients, DefaultConstraintCommitment,
7      DefaultConstraintEvaluator, DefaultTraceLde, EvaluationFrame, FieldExtension,
8      AcceptableOptions, PartitionOptions, ProofOptions, Prover, StarkDomain, TraceInfo,
9      TracePolyTable, TraceTable, TransitionConstraintDegree,
10 };
11 use winter_crypto::DefaultRandomCoin;
12
13 use crate::poseidon_air::{PoseidonDigest, PoseidonHasher};
14 use crate::{compute_h_theta, f, felt_to_i64, Felt, W1, B1, W2, B2};
15
16 pub const COL_X:      usize = 0;
17 pub const COL_Y:      usize = 4;
18 pub const COL_Z1:     usize = 6;
19 pub const COL_BS1:    usize = 14;
20 pub const COL_H1:     usize = 22;
21 pub const COL_OUT:    usize = 30;
22 pub const COL_DELTA2: usize = 32;
23 pub const COL_DELTA1: usize = 34;
24 pub const COL_GW2:    usize = 42;
25 pub const COL_GW1:    usize = 58;
26 pub const COL_SEL:    usize = 90;
27
28 pub const GRAD_TW:    usize = 91;
29 pub const GRAD_NC:    usize = 84;

```

```

30 pub const GRAD_NA: usize = 96;
31
32 pub fn load_dataset(path: &str) -> Vec<([i64; 4], i64)> {
33     let content = std::fs::read_to_string(path)
34         .unwrap_or_else(|e| panic!("Cannot read {path}: {e}"));
35     content.lines()
36         .filter(|l| !l.trim().is_empty())
37         .map(|line| {
38             let v: Vec<i64> = line.split(',').
39                 .map(|s| s.trim().parse::<i64>()).expect("bad CSV value")
40                 .collect();
41             assert_eq!(v.len(), 5, "CSV row must have 5 values: {x0,x1,x2,x3,y}");
42             ([v[0], v[1], v[2], v[3]], v[4])
43         })
44         .collect()
45 }
46
47 pub fn dataset_merkle_root(examples: &([i64; 4], i64)) -> PoseidonDigest {
48     let n_leaves = examples.len().next_power_of_two();
49     let zero_leaf = PoseidonHasher::hash_elements(&[Felt::ZERO; 5]);
50     let mut leaves: Vec<PoseidonDigest> = examples.iter()
51         .map(|(x, y)| {
52             let elems = [f(x[0]), f(x[1]), f(x[2]), f(x[3]), f(*y)];
53             PoseidonHasher::hash_elements(&elems)
54         })
55         .collect();
56     leaves.resize(n_leaves, zero_leaf);
57     let tree = MerkleTree::<PoseidonHasher>::new(leaves)
58         .expect("MerkleTree build failed");
59     *tree.root()
60 }
61
62 pub fn build_gradient_trace(
63     examples: &([i64; 4], i64),
64 ) -> (TraceTable<Felt>, [[i64; 4]; 8], [[i64; 8]; 2]) {
65     let n = examples.len();
66     let trace_len = (n + 1).next_power_of_two().max(8);
67
68     let mut rows: Vec<[Felt; GRAD_TW]> = Vec::with_capacity(trace_len);
69
70     let mut g_w2 = [[0i64; 8]; 2];
71     let mut g_w1 = [[0i64; 4]; 8];
72
73     for (x, y) in examples.iter() {
74         let row = compute_row(x, *y, &g_w2, &g_w1);

```

```

75
76     let (z1i, bs1i, h1i) = forward_ints(x);
77     let outi: [i64; 2] = std::array::from_fn(|j|
78         (0..8).map(|i| W2[j][i] * h1i[i]).sum::<i64>() + B2[j]);
79     let y_oh = [if *y == 0 { 1i64 } else { 0 }, if *y == 1 { 1i64 } else { 0 }];
80     let delta2: [i64; 2] = [outi[0] - y_oh[0], outi[1] - y_oh[1]];
81     let delta1: [i64; 8] = std::array::from_fn(|i|
82         (0..2).map(|k| W2[k][i] * delta2[k]).sum::<i64>() * bs1i[i]);
83
84     for k in 0..2 { for i in 0..8 { g_w2[k][i] += delta2[k] * h1i[i]; } }
85     for i in 0..8 { for j in 0..4 { g_w1[i][j] += delta1[i] * x[j]; } }
86
87     let _ = (z1i, outi);
88     rows.push(row);
89 }
90
91 let padding_row = make_grad_row(&g_w2, &g_w1, false);
92 while rows.len() < trace_len {
93     rows.push(padding_row);
94 }
95
96 let g_w1_final = g_w1;
97 let g_w2_final = g_w2;
98
99 let mut t = TraceTable::new(GRAD_TW, trace_len);
100 {
101     let row0 = rows[0];
102     t.fill(
103         move |row| { row.copy_from_slice(&row0); },
104         |step, row| { row.copy_from_slice(&rows[step + 1]); },
105     );
106 }
107
108 (t, g_w1_final, g_w2_final)
109 }
110
111 fn forward_ints(x: &[i64; 4]) -> ([i64; 8], [i64; 8], [i64; 8]) {
112     let z1i: [i64; 8] = std::array::from_fn(|i|
113         (0..4).map(|j| W1[i][j] * x[j]).sum::<i64>() + B1[i]);
114     let bs1i: [i64; 8] = std::array::from_fn(|i| if z1i[i] >= 0 { 1 } else { 0 });
115     let h1i: [i64; 8] = std::array::from_fn(|i| bs1i[i] * z1i[i]);
116     (z1i, bs1i, h1i)
117 }
118
119 fn compute_row(x: &[i64; 4], y: i64, g_w2: &[[i64; 8]; 2], g_w1: &[[i64; 4]; 8]) -> [Felt;

```

```

GRAD_TW] {
120     let (z1i, bs1i, h1i) = forward_ints(x);
121     let outi: [i64; 2] = std::array::from_fn(|j|
122         (0..8).map(|i| W2[j][i] * h1i[i]).sum::<i64>() + B2[j]);
123     let y_oh = [if y == 0 { 1i64 } else { 0 }, if y == 1 { 1i64 } else { 0 }];
124     let delta2: [i64; 2] = [outi[0] - y_oh[0], outi[1] - y_oh[1]];
125     let delta1: [i64; 8] = std::array::from_fn(|i|
126         (0..2).map(|k| W2[k][i] * delta2[k]).sum::<i64>() * bs1i[i]);
127
128     let mut row = [Felt::ZERO; GRAD_TW];
129     for j in 0..4 { row[COL_X + j] = f(x[j]); }
130     row[COL_Y] = f(y_oh[0]);
131     row[COL_Y + 1] = f(y_oh[1]);
132     for i in 0..8 { row[COL_Z1 + i] = f(z1i[i]); }
133     for i in 0..8 { row[COL_BS1 + i] = Felt::new(bs1i[i] as u64); }
134     for i in 0..8 { row[COL_H1 + i] = f(h1i[i]); }
135     for j in 0..2 { row[COL_OUT + j] = f(outi[j]); }
136     for k in 0..2 { row[COL_DELTA2 + k] = f(delta2[k]); }
137     for i in 0..8 { row[COL_DELTA1 + i] = f(delta1[i]); }
138     for k in 0..2 { for i in 0..8 { row[COL_GW2 + k*8 + i] = f(g_w2[k][i]); } }
139     for i in 0..8 { for j in 0..4 { row[COL_GW1 + i*4 + j] = f(g_w1[i][j]); } }
140     row[COL_SEL] = Felt::ONE;
141     row
142 }
143
144 fn make_grad_row(g_w2: &[[i64; 8]; 2], g_w1: &[[i64; 4]; 8], sel: bool) -> [Felt; GRAD_TW] {
145     let mut row = [Felt::ZERO; GRAD_TW];
146     for k in 0..2 { for i in 0..8 { row[COL_GW2 + k*8 + i] = f(g_w2[k][i]); } }
147     for i in 0..8 { for j in 0..4 { row[COL_GW1 + i*4 + j] = f(g_w1[i][j]); } }
148     if sel { row[COL_SEL] = Felt::ONE; }
149     row
150 }
151
152 #[derive(Clone)]
153 pub struct GradientPub {
154     pub dataset_root: PoseidonDigest,
155     pub h_theta: PoseidonDigest,
156     pub g_w1_final: [[i64; 4]; 8],
157     pub g_w2_final: [[i64; 8]; 2],
158     pub n_examples: usize,
159 }
160
161 impl ToElements<Felt> for GradientPub {
162     fn to_elements(&self) -> Vec<Felt> {
163         let mut v: Vec<Felt> = Vec::new();

```

```

164         for digest in [&self.dataset_root, &self.h_theta] {
165             for chunk in digest.as_bytes().chunks(8) {
166                 let mut b = [0u8; 8]; b[..chunk.len()].copy_from_slice(chunk);
167                 v.push(Felt::new(u64::from_le_bytes(b)));
168             }
169         }
170         for row in &self.g_w1_final { for &val in row { v.push(f(val)); } }
171         for row in &self.g_w2_final { for &val in row { v.push(f(val)); } }
172         v.push(Felt::new(self.n_examples as u64));
173         v
174     }
175 }
176
177 pub struct GradientAir { ctx: AirContext<Felt>, inp: GradientPub }
178
179 impl Air for GradientAir {
180     type BaseField = Felt;
181     type PublicInputs = GradientPub;
182
183     fn new(info: TraceInfo, p: GradientPub, opts: ProofOptions) -> Self {
184         let mut deg: Vec<TransitionConstraintDegree> = Vec::with_capacity(GRAD_NC);
185         for _ in 0..8 { deg.push(TransitionConstraintDegree::new(2)); }
186         for _ in 0..8 { deg.push(TransitionConstraintDegree::new(3)); }
187         for _ in 0..8 { deg.push(TransitionConstraintDegree::new(3)); }
188         for _ in 0..2 { deg.push(TransitionConstraintDegree::new(2)); }
189         for _ in 0..2 { deg.push(TransitionConstraintDegree::new(2)); }
190         for _ in 0..8 { deg.push(TransitionConstraintDegree::new(3)); }
191         for _ in 0..16 { deg.push(TransitionConstraintDegree::new(3)); }
192         for _ in 0..32 { deg.push(TransitionConstraintDegree::new(3)); }
193         debug_assert_eq!(deg.len(), GRAD_NC);
194         GradientAir { ctx: AirContext::new(info, deg, GRAD_NA, opts), inp: p }
195     }
196
197     fn context(&self) -> &AirContext<Felt> { &self.ctx }
198
199     fn evaluate_transition<E: FieldElement + From<Felt>>(&self, frame: &EvaluationFrame<E>, _ : &[E], res: &mut [E],
200 ) {
201         let (cur, next) = (frame.current(), frame.next());
202         let sel = cur[COL_SEL];
203         let mut c = 0;
204
205         for i in 0..8 {
206             let ez: E = (0..4)
207                 .map(|j| E::from(f(W1[i][j]))) * cur[COL_X + j])

```

```

209         .fold(E::ZERO, |a, b| a + b) + E::from(f(B1[i]));
210     res[c] = sel * (cur[COL_Z1 + i] - ez);
211     c += 1;
212 }
213 for i in 0..8 {
214     let bs = cur[COL_BS1 + i];
215     res[c] = sel * bs * (E::ONE - bs);
216     c += 1;
217 }
218 for i in 0..8 {
219     res[c] = sel * (cur[COL_H1 + i] - cur[COL_BS1 + i] * cur[COL_Z1 + i]);
220     c += 1;
221 }
222 for j in 0..2 {
223     let eo: E = (0..8)
224         .map(|i| E::from(f(W2[j][i]))) * cur[COL_H1 + i])
225         .fold(E::ZERO, |a, b| a + b) + E::from(f(B2[j]));
226     res[c] = sel * (cur[COL_OUT + j] - eo);
227     c += 1;
228 }
229 for k in 0..2 {
230     res[c] = sel * (cur[COL_DELTA2 + k] - cur[COL_OUT + k] + cur[COL_Y + k]);
231     c += 1;
232 }
233 for i in 0..8 {
234     let ed: E = (0..2)
235         .map(|k| E::from(f(W2[k][i]))) * cur[COL_DELTA2 + k])
236         .fold(E::ZERO, |a, b| a + b);
237     res[c] = sel * (cur[COL_DELTA1 + i] - cur[COL_BS1 + i] * ed);
238     c += 1;
239 }
240 for k in 0..2 {
241     for ii in 0..8 {
242         res[c] = next[COL_GW2 + k*8 + ii]
243             - cur[COL_GW2 + k*8 + ii]
244             - sel * cur[COL_DELTA2 + k] * cur[COL_H1 + ii];
245         c += 1;
246     }
247 }
248 for i in 0..8 {
249     for j in 0..4 {
250         res[c] = next[COL_GW1 + i*4 + j]
251             - cur[COL_GW1 + i*4 + j]
252             - sel * cur[COL_DELTA1 + i] * cur[COL_X + j];
253         c += 1;

```

```

254         }
255     }
256     debug_assert_eq!(c, GRAD_NC);
257 }
258
259 fn get_assertions(&self) -> Vec<Assertion<Felt>> {
260     let mut v: Vec<Assertion<Felt>> = Vec::with_capacity(GRAD_NA);
261     let n = self.inp.n_examples;
262
263     for col in COL_GW2..(COL_GW2 + 16) { v.push(Assertion::single(col, 0, Felt::ZERO));
264         }
265     for col in COL_GW1..(COL_GW1 + 32) { v.push(Assertion::single(col, 0, Felt::ZERO));
266         }
267
268     for k in 0..2 {
269         for i in 0..8 {
270             v.push(Assertion::single(COL_GW2 + k*8 + i, n, f(self.inp.g_w2_final[k][i]))
271                 );
272         }
273     }
274     for i in 0..8 {
275         for j in 0..4 {
276             v.push(Assertion::single(COL_GW1 + i*4 + j, n, f(self.inp.g_w1_final[i][j]))
277                 );
278         }
279     }
280
281     debug_assert_eq!(v.len(), GRAD_NA);
282     v
283 }
284
285 pub struct GradientProver {
286     opts: ProofOptions,
287     n_examples: usize,
288     dataset_root: PoseidonDigest,
289 }
290
291 impl GradientProver {
292     pub fn new(opts: ProofOptions, n_examples: usize, dataset_root: PoseidonDigest) -> Self
293     {
294         Self { opts, n_examples, dataset_root }
295     }
296 }

```

```

294 impl Prover for GradientProver {
295     type BaseField = Felt;
296     type Air        = GradientAir;
297     type Trace      = TraceTable<Felt>;
298     type HashFn     = PoseidonHasher;
299     type VC         = MerkleTree<PoseidonHasher>;
300     type RandomCoin = DefaultRandomCoin<PoseidonHasher>;
301     type TraceLde<E: FieldElement<BaseField = Felt>> =
302         DefaultTraceLde<E, PoseidonHasher, MerkleTree<PoseidonHasher>>;
303     type ConstraintCommitment<E: FieldElement<BaseField = Felt>> =
304         DefaultConstraintCommitment<E, PoseidonHasher, MerkleTree<PoseidonHasher>>;
305     type ConstraintEvaluator<'a, E: FieldElement<BaseField = Felt>> =
306         DefaultConstraintEvaluator<'a, Self::Air, E>;
307
308     fn get_pub_inputs(&self, t: &Self::Trace) -> GradientPub {
309         let n = self.n_examples;
310         let g_w2_final: [[i64; 8]; 2] = std::array::from_fn(|k|
311             std::array::from_fn(|i| felt_to_i64(t.get(COL_GW2 + k*8 + i, n))));
312         let g_w1_final: [[i64; 4]; 8] = std::array::from_fn(|i|
313             std::array::from_fn(|j| felt_to_i64(t.get(COL_GW1 + i*4 + j, n))));
314         GradientPub {
315             dataset_root: self.dataset_root,
316             h_theta:      compute_h_theta(),
317             g_w1_final,
318             g_w2_final,
319             n_examples: n,
320         }
321     }
322
323     fn options(&self) -> &ProofOptions { &self.opts }
324
325     fn new_trace_lde<E: FieldElement<BaseField = Felt>>(
326         &self, i: &TraceInfo, m: &ColMatrix<Felt>,
327         d: &StarkDomain<Felt>, p: PartitionOptions,
328     ) -> (Self::TraceLde<E>, TracePolyTable<E>) {
329         DefaultTraceLde::new(i, m, d, p)
330     }
331
332     fn build_constraint_commitment<E: FieldElement<BaseField = Felt>>(
333         &self, p: CompositionPolyTrace<E>, n: usize,
334         d: &StarkDomain<Felt>, po: PartitionOptions,
335     ) -> (Self::ConstraintCommitment<E>, CompositionPoly<E>) {
336         DefaultConstraintCommitment::new(p, n, d, po)
337     }
338

```

```

339     fn new_evaluator<'a, E: FieldElement<BaseField = Felt>>(<
340         &self, a: &'a Self::Air,
341         aux: Option<AuxRandElements<E>>,
342         c: ConstraintCompositionCoefficients<E>,
343     ) -> Self::ConstraintEvaluator<'a, E> {
344         DefaultConstraintEvaluator::new(a, aux, c)
345     }
346 }

```

ЛІСТИНГ 3: src/poseidon_air.rs

```

1  use winter_crypto::{
2      hashers::Blake3_256, DefaultRandomCoin as BlakeRandomCoin,
3      Digest, ElementHasher, Hasher,
4      MerkleTree as BlakeMerkle,
5  };
6  use winter_utils::{ByteReader, ByteWriter, Deserializable, DeserializationError,
7      Serializable};
8  use winterfell::{
9      math::{fields::f64::BaseElement, FieldElement, StarkField, ToElements},
10     matrix::ColMatrix,
11     Air, AirContext, Assertion, AuxRandElements, CompositionPoly,
12     CompositionPolyTrace, ConstraintCompositionCoefficients, DefaultConstraintCommitment,
13     DefaultConstraintEvaluator, DefaultTraceLde, EvaluationFrame,
14     PartitionOptions, ProofOptions, Prover, StarkDomain, TraceInfo, TracePolyTable,
15     TraceTable, TransitionConstraintDegree,
16 };
17 pub type Felt = BaseElement;
18
19 pub const T: usize = 8;
20 const RF_HALF: usize = 4;
21 const RP: usize = 22;
22 pub const N_ROUNDS: usize = 2 * RF_HALF + RP;
23 const TRACE_LEN: usize = 32;
24
25 pub const COL_STATE: usize = 0;
26 const COL_H1: usize = T;
27 const COL_H2: usize = 2 * T;
28 const COL_H3: usize = 3 * T;
29 const COL_SBOX: usize = 4 * T;
30 const WIDTH: usize = 5 * T;
31
32 pub const COL_X2: usize = COL_H1;
33 pub const COL_X4: usize = COL_H2;
34 pub const COL_X6: usize = COL_H3;

```

```

35 pub const COL_X7: usize = COL_SBOX;
36
37 pub const N_CONSTRAINTS: usize = 4 + 5 * (T - 1) + T;
38 const N_ASSERTIONS: usize = 2 * T;
39
40 pub fn round_constants() -> [[u64; T]; N_ROUNDS] {
41     let mut rc = [[0u64; T]; N_ROUNDS];
42     for r in 0..N_ROUNDS {
43         for i in 0..T { rc[r][i] = ((r * T + i + 1) as u64) * 1_000_003; }
44     }
45     rc
46 }
47
48 pub fn mds_matrix() -> [[Felt; T]; T] {
49     let mut m = [[Felt::ZERO; T]; T];
50     for i in 0..T { for j in 0..T { m[i][j] = Felt::new((i + j + 8) as u64).inv(); } }
51     m
52 }
53
54 pub fn is_full_round(r: usize) -> bool { r < RF_HALF || r >= N_ROUNDS - RF_HALF }
55
56 #[inline(always)]
57 fn exp7(x: Felt) -> Felt { let x2 = x * x; let x4 = x2 * x2; x4 * x2 * x }
58
59 pub fn permute(state: &mut [Felt; T]) {
60     let rc = round_constants();
61     let mds = mds_matrix();
62     for r in 0..N_ROUNDS {
63         let sbox: [Felt; T] = std::array::from_fn(|i| {
64             let pre = state[i] + Felt::new(rc[r][i]);
65             if is_full_round(r) || i == 0 { exp7(pre) } else { pre }
66         });
67         *state = std::array::from_fn(|j|
68             (0..T).fold(Felt::ZERO, |acc, i| acc + mds[j][i] * sbox[i])
69         );
70     }
71 }
72
73 fn poseidon_sponge(inputs: &[Felt]) -> [Felt; 4] {
74     let mut state = [Felt::ZERO; T];
75     let mut idx = 0;
76     while idx < inputs.len() {
77         for j in 0..4 { if idx + j < inputs.len() { state[j] += inputs[idx + j]; } }
78         permute(&mut state);
79         idx += 4;

```

```

80     }
81     [state[0], state[1], state[2], state[3]]
82 }
83
84 pub fn felt4_to_bytes(arr: [Felt; 4]) -> [u8; 32] {
85     let mut out = [0u8; 32];
86     for (i, v) in arr.iter().enumerate() {
87         out[i * 8..(i + 1) * 8].copy_from_slice(&v.as_int().to_le_bytes());
88     }
89     out
90 }
91
92 pub fn bytes32_to_felt4(b: [u8; 32]) -> [Felt; 4] {
93     std::array::from_fn(|i| {
94         let mut c = [0u8; 8];
95         c.copy_from_slice(&b[i * 8..(i + 1) * 8]);
96         Felt::new(u64::from_le_bytes(c))
97     })
98 }
99
100 fn bytes_to_felts(bytes: &[u8]) -> Vec<Felt> {
101     bytes.chunks(8).map(|c| {
102         let mut b = [0u8; 8];
103         b[..c.len()].copy_from_slice(c);
104         Felt::new(u64::from_le_bytes(b) % Felt::MODULUS)
105     }).collect()
106 }
107
108 #[derive(Debug, Default, Copy, Clone, PartialEq, Eq)]
109 pub struct PoseidonDigest(pub [u8; 32]);
110
111 impl Digest for PoseidonDigest { fn as_bytes(&self) -> [u8; 32] { self.0 } }
112 impl Serializable for PoseidonDigest {
113     fn write_into<W: ByteWriter>(&self, w: &mut W) { w.write_bytes(&self.0); }
114 }
115 impl Deserializable for PoseidonDigest {
116     fn read_from<R: ByteReader>(r: &mut R) -> Result<Self, DeserializationError> {
117         Ok(PoseidonDigest(r.read_array::<32>()?))
118     }
119 }
120
121 unsafe impl Send for PoseidonDigest {}
122 unsafe impl Sync for PoseidonDigest {}
123
124 pub struct PoseidonHasher;
125 impl Hasher for PoseidonHasher {

```

```

125     type Digest = PoseidonDigest;
126     const COLLISION_RESISTANCE: u32 = 128;
127     fn hash(bytes: &[u8]) -> Self::Digest {
128         PoseidonDigest(felt4_to_bytes(poseidon_sponge(&bytes_to_felts(bytes))))
129     }
130     fn merge(v: &[Self::Digest; 2]) -> Self::Digest {
131         let mut inp = [Felt::ZERO; 8];
132         let (l, r) = (bytes32_to_felt4(v[0].as_bytes()), bytes32_to_felt4(v[1].as_bytes()));
133         for i in 0..4 { inp[i] = l[i]; inp[4 + i] = r[i]; }
134         let mut st = inp; permute(&mut st);
135         PoseidonDigest(felt4_to_bytes([st[0], st[1], st[2], st[3]]))
136     }
137     fn merge_many(vals: &[Self::Digest]) -> Self::Digest {
138         let elems: Vec<Felt> = vals.iter().flat_map(|d| bytes32_to_felt4(d.as_bytes())).
139             collect();
140         PoseidonDigest(felt4_to_bytes(poseidon_sponge(&elems)))
141     }
142     fn merge_with_int(seed: Self::Digest, value: u64) -> Self::Digest {
143         let mut inp: Vec<Felt> = bytes32_to_felt4(seed.as_bytes()).to_vec();
144         inp.push(Felt::new(value));
145         PoseidonDigest(felt4_to_bytes(poseidon_sponge(&inp)))
146     }
147
148     impl ElementHasher for PoseidonHasher {
149         type BaseField = Felt;
150         fn hash_elements<E: FieldElement<BaseField = Felt>>(elements: &[E]) -> Self::Digest {
151             let raw = E::elements_as_bytes(elements);
152             let felts: Vec<Felt> = raw.chunks(core::mem::size_of::<u64>())
153                 .map(|c| {
154                     let mut b = [0u8; 8];
155                     b[..c.len()].copy_from_slice(c);
156                     Felt::from_mont(u64::from_le_bytes(b))
157                 })
158                 .collect();
159             PoseidonDigest(felt4_to_bytes(poseidon_sponge(&felts)))
160         }
161     }
162
163     fn fill_helpers(row: &mut [Felt], r: usize, rc: &[[u64; T]; N_ROUNDS]) {
164         for i in 0..T {
165             let pre = row[COL_STATE + i] + Felt::new(rc[r][i]);
166             if is_full_round(r) || i == 0 {
167                 let h1 = pre * pre;
168                 let h2 = h1 * h1;

```

```

169         let h3 = h2 * h1;
170         row[COL_H1 + i] = h1;
171         row[COL_H2 + i] = h2;
172         row[COL_H3 + i] = h3;
173         row[COL_SBOX + i] = h3 * pre;
174     } else {
175         row[COL_H1 + i] = Felt::ZERO;
176         row[COL_H2 + i] = Felt::ZERO;
177         row[COL_H3 + i] = Felt::ZERO;
178         row[COL_SBOX + i] = pre;
179     }
180 }
181 }
182
183 pub fn build_trace(x: [Felt; 4]) -> TraceTable<Felt> {
184     let rc = round_constants();
185     let mds = mds_matrix();
186     let init: [Felt; T] = std::array::from_fn(|i| if i < 4 { x[i] } else { Felt::ZERO });
187
188     let mut trace = TraceTable::new(WIDTH, TRACE_LEN);
189     trace.fill(
190         |row| {
191             for i in 0..T { row[COL_STATE + i] = init[i]; }
192             fill_helpers(row, 0, &rc);
193         },
194         |step, row| {
195             let sbox: [Felt; T] = std::array::from_fn(|i| row[COL_SBOX + i]);
196             let ns: [Felt; T] = std::array::from_fn(|j|
197                 (0..T).fold(Felt::ZERO, |acc, i| acc + mds[j][i] * sbox[i])
198             );
199             for i in 0..T { row[COL_STATE + i] = ns[i]; }
200
201             if step + 1 < N_ROUNDS {
202                 fill_helpers(row, step + 1, &rc);
203             } else {
204                 let h1_0 = ns[0] * ns[0];
205                 let h2_0 = h1_0 * h1_0;
206                 let h3_0 = h2_0 * h1_0;
207                 row[COL_H1 + 0] = h1_0;
208                 row[COL_H2 + 0] = h2_0;
209                 row[COL_H3 + 0] = h3_0;
210                 row[COL_SBOX + 0] = h3_0 * ns[0];
211                 for i in 1..T {
212                     row[COL_H1 + i] = Felt::ZERO;
213                     row[COL_H2 + i] = Felt::ZERO;

```

```

214             row[COL_H3 + i] = Felt::ZERO;
215             row[COL_SBOX + i] = ns[i];
216         }
217     }
218 },
219 );
220 trace
221 }
222
223 #[derive(Clone)]
224 pub struct PoseidonPub {
225     pub initial: [Felt; T],
226     pub output: [Felt; T],
227 }
228
229 impl PoseidonPub {
230     pub fn from_parts(x: [Felt; 4], hx: [Felt; 4], output_cap: [Felt; 4]) -> Self {
231         let initial = std::array::from_fn(|i| if i < 4 { x[i] } else { Felt::ZERO });
232         let output = std::array::from_fn(|i| if i < 4 { hx[i] } else { output_cap[i - 4] });
233         ;
234         Self { initial, output }
235     }
236     pub fn x(&self) -> [Felt; 4] { std::array::from_fn(|i| self.initial[i]) }
237     pub fn hx(&self) -> [Felt; 4] { std::array::from_fn(|i| self.output[i]) }
238 }
239
240 impl ToElements<Felt> for PoseidonPub {
241     fn to_elements(&self) -> Vec<Felt> {
242         self.initial.iter().chain(self.output.iter()).copied().collect()
243     }
244 }
245
246 pub struct PoseidonAir {
247     ctx: AirContext<Felt>,
248     inputs: PoseidonPub,
249     mds: [[Felt; T]; T],
250 }
251
252 impl Air for PoseidonAir {
253     type BaseField = Felt;
254     type PublicInputs = PoseidonPub;
255
256     fn new(info: TraceInfo, pub_in: PoseidonPub, opts: ProofOptions) -> Self {
257         assert_eq!(WIDTH, info.width(), "Trace width must be {WIDTH}");
258         let mut degrees: Vec<TransitionConstraintDegree> = Vec::with_capacity(N_CONSTRAINTS)

```

```

    ;
258     for _ in 0..4 { degrees.push(TransitionConstraintDegree::new(2)); }
259     for _ in 1..T {
260         for _ in 0..4 { degrees.push(TransitionConstraintDegree::new(3)); }
261         degrees.push(    TransitionConstraintDegree::new(2));
262     }
263     for _ in 0..T { degrees.push(TransitionConstraintDegree::new(1)); }
264     debug_assert_eq!(degrees.len(), N_CONSTRAINTS);
265     PoseidonAir {
266         ctx:    AirContext::new(info, degrees, N_ASSERTIONS, opts),
267         inputs: pub_in,
268         mds:    mds_matrix(),
269     }
270 }
271
272 fn context(&self) -> &AirContext<Felt> { &self.ctx }
273
274 fn get_periodic_column_values(&self) -> Vec<Vec<Felt>> {
275     let rc = round_constants();
276     let mut cols: Vec<Vec<Felt>> = Vec::new();
277     let mut is_full = vec![Felt::ZERO; TRACE_LEN];
278     for r in 0..N_ROUNDS { if is_full_round(r) { is_full[r] = Felt::ONE; } }
279     cols.push(is_full);
280     for i in 0..T {
281         let mut col = vec![Felt::ZERO; TRACE_LEN];
282         for r in 0..N_ROUNDS { col[r] = Felt::new(rc[r][i]); }
283         cols.push(col);
284     }
285     cols
286 }
287
288 fn evaluate_transition<E: FieldElement + From<Felt>>(&self,
289     frame: &EvaluationFrame<E>,
290     periodic: &[E],
291     result: &mut [E],
292 ) {
293     let cur = frame.current();
294     let next = frame.next();
295     let is_full = periodic[0];
296     let is_part = E::ONE - is_full;
297
298     let s: Vec<E> = (0..T).map(|i| cur[COL_STATE + i]).collect();
299     let h1: Vec<E> = (0..T).map(|i| cur[COL_H1 + i]).collect();
300     let h2: Vec<E> = (0..T).map(|i| cur[COL_H2 + i]).collect();

```

```

302     let h3: Vec<E> = (0..T).map(|i| cur[COL_H3 + i]).collect();
303     let sbox: Vec<E> = (0..T).map(|i| cur[COL_SBOX + i]).collect();
304     let rc: Vec<E> = (0..T).map(|i| periodic[1 + i]).collect();
305     let pre: Vec<E> = (0..T).map(|i| s[i] + rc[i]).collect();
306     let mut c = 0usize;
307
308     result[c] = h1[0] - pre[0] * pre[0]; c += 1;
309     result[c] = h2[0] - h1[0] * h1[0]; c += 1;
310     result[c] = h3[0] - h2[0] * h1[0]; c += 1;
311     result[c] = sbox[0] - h3[0] * pre[0]; c += 1;
312
313     for i in 1..T {
314         result[c] = is_full * (h1[i] - pre[i] * pre[i]); c += 1;
315         result[c] = is_full * (h2[i] - h1[i] * h1[i]); c += 1;
316         result[c] = is_full * (h3[i] - h2[i] * h1[i]); c += 1;
317         result[c] = is_full * (sbox[i] - h3[i] * pre[i]); c += 1;
318         result[c] = is_part * (sbox[i] - pre[i]); c += 1;
319     }
320
321     for j in 0..T {
322         let expected: E = (0..T)
323             .map(|i| E::from(self.mds[j][i]) * sbox[i])
324             .fold(E::ZERO, |acc, v| acc + v);
325         result[c] = next[COL_STATE + j] - expected;
326         c += 1;
327     }
328     debug_assert_eq!(c, N_CONSTRAINTS);
329 }
330
331 fn get_assertions(&self) -> Vec<Assertion<Felt>> {
332     let mut a: Vec<Assertion<Felt>> = Vec::new();
333     for i in 0..T {
334         a.push(Assertion::single(COL_STATE + i, 0, self.inputs.initial[i]));
335         a.push(Assertion::single(COL_STATE + i, N_ROUNDS, self.inputs.output[i]));
336     }
337     debug_assert_eq!(a.len(), N_ASSERTIONS);
338     a
339 }
340 }
341
342 pub fn compute_output(x: [Felt; 4]) -> [Felt; T] {
343     let mut state: [Felt; T] = std::array::from_fn(|i| if i < 4 { x[i] } else { Felt::ZERO });
344     permute(&mut state);
345     state

```

```

346 }
347
348 pub fn validate_trace(trace: &TraceTable<Felt>) -> bool {
349     let mds = mds_matrix();
350     let rc = round_constants();
351     let mut is_full_col = vec![Felt::ZERO; TRACE_LEN];
352     for r in 0..N_ROUNDS { if is_full_round(r) { is_full_col[r] = Felt::ONE; } }
353     let mut rc_cols: Vec<Vec<Felt>> = (0..T).map(|_| vec![Felt::ZERO; TRACE_LEN]).collect();
354     for i in 0..T { for r in 0..N_ROUNDS { rc_cols[i][r] = Felt::new(rc[r][i]); } }
355
356     let mut ok = true;
357     for step in 0..TRACE_LEN - 1 {
358         let cur: Vec<Felt> = (0..WIDTH).map(|i| trace.get(i, step)).collect();
359         let next: Vec<Felt> = (0..WIDTH).map(|i| trace.get(i, step + 1)).collect();
360         let is_full = is_full_col[step];
361         let is_part = Felt::ONE - is_full;
362         let s: Vec<Felt> = (0..T).map(|i| cur[COL_STATE + i]).collect();
363         let h1: Vec<Felt> = (0..T).map(|i| cur[COL_H1 + i]).collect();
364         let h2: Vec<Felt> = (0..T).map(|i| cur[COL_H2 + i]).collect();
365         let h3: Vec<Felt> = (0..T).map(|i| cur[COL_H3 + i]).collect();
366         let sbbox: Vec<Felt> = (0..T).map(|i| cur[COL_SBOX + i]).collect();
367         let rc_p: Vec<Felt> = (0..T).map(|i| rc_cols[i][step]).collect();
368         let pre: Vec<Felt> = (0..T).map(|i| s[i] + rc_p[i]).collect();
369         let mut c = 0usize;
370         macro_rules! chk { ($v:expr) => {{ let v: Felt = $v;
371             if v != Felt::ZERO { eprintln!("{}", "C{c}@step{step}={}", v.as_int()); ok=false; }
372             c += 1; }}; }
373         chk!(h1[0] - pre[0] * pre[0]);
374         chk!(h2[0] - h1[0] * h1[0]);
375         chk!(h3[0] - h2[0] * h1[0]);
376         chk!(sbox[0] - h3[0] * pre[0]);
377         for i in 1..T {
378             chk!(is_full * (h1[i] - pre[i] * pre[i]));
379             chk!(is_full * (h2[i] - h1[i] * h1[i]));
380             chk!(is_full * (h3[i] - h2[i] * h1[i]));
381             chk!(is_full * (sbox[i] - h3[i] * pre[i]));
382             chk!(is_part * (sbox[i] - pre[i]));
383         }
384         for j in 0..T {
385             let exp: Felt = (0..T).map(|i| mds[j][i]*sbox[i]).fold(Felt::ZERO, |a,b|a+b);
386             chk!(next[COL_STATE + j] - exp);
387         }
388     }
389     ok
390 }

```

```

391
392 pub struct PoseidonProver { opts: ProofOptions }
393 impl PoseidonProver { pub fn new(opts: ProofOptions) -> Self { Self { opts } } }
394
395 pub type PoseidonProofHasher = Blake3_256<Felt>;
396 pub type PoseidonProofVC      = BlakeMerkle<Blake3_256<Felt>>;
397 pub type PoseidonProofCoin    = BlakeRandomCoin<Blake3_256<Felt>>;
398
399 impl Prover for PoseidonProver {
400     type BaseField = Felt;
401     type Air        = PoseidonAir;
402     type Trace      = TraceTable<Felt>;
403     type HashFn     = Blake3_256<Felt>;
404     type VC         = BlakeMerkle<Blake3_256<Felt>>;
405     type RandomCoin = BlakeRandomCoin<Blake3_256<Felt>>;
406     type TraceLde<E: FieldElement<BaseField = Felt>> =
407         DefaultTraceLde<E, Blake3_256<Felt>, BlakeMerkle<Blake3_256<Felt>>>;
408     type ConstraintCommitment<E: FieldElement<BaseField = Felt>> =
409         DefaultConstraintCommitment<E, Blake3_256<Felt>, BlakeMerkle<Blake3_256<Felt>>>;
410     type ConstraintEvaluator<'a, E: FieldElement<BaseField = Felt>> =
411         DefaultConstraintEvaluator<'a, Self::Air, E>;
412
413     fn get_pub_inputs(&self, trace: &Self::Trace) -> PoseidonPub {
414         PoseidonPub {
415             initial: std::array::from_fn(|i| trace.get(COL_STATE + i, 0)),
416             output:  std::array::from_fn(|i| trace.get(COL_STATE + i, N_ROUNDS)),
417         }
418     }
419     fn options(&self) -> &ProofOptions { &self.opts }
420
421     fn new_trace_lde<E: FieldElement<BaseField = Felt>>(
422         &self, info: &TraceInfo, main: &ColMatrix<Felt>,
423         domain: &StarkDomain<Felt>, part: PartitionOptions,
424     ) -> (Self::TraceLde<E>, TracePolyTable<E>) {
425         DefaultTraceLde::new(info, main, domain, part)
426     }
427     fn build_constraint_commitment<E: FieldElement<BaseField = Felt>>(
428         &self, poly: CompositionPolyTrace<E>, n: usize,
429         domain: &StarkDomain<Felt>, part: PartitionOptions,
430     ) -> (Self::ConstraintCommitment<E>, CompositionPoly<E>) {
431         DefaultConstraintCommitment::new(poly, n, domain, part)
432     }
433     fn new_evaluator<'a, E: FieldElement<BaseField = Felt>>(
434         &self, air: &'a Self::Air,
435         aux: Option<AuxRandElements<E>>,

```

```

436         coeff: ConstraintCompositionCoefficients<E>,
437     ) -> Self::ConstraintEvaluator<'a, E> {
438         DefaultConstraintEvaluator::new(air, aux, coeff)
439     }
440 }

```

Лістинг 4: src/poseidon_standalone.rs

```

1  mod poseidon_air;
2  use poseidon_air::{
3      build_trace, compute_output, validate_trace,
4      Felt, PoseidonAir, PoseidonProver, PoseidonPub,
5      PoseidonProofHasher, PoseidonProofVC, PoseidonProofCoin,
6      N_ROUNDS,
7  };
8  use winterfell::{
9      math::{FieldElement, StarkField},
10     AcceptableOptions, BatchingMethod, FieldExtension, ProofOptions, Prover,
11 };
12
13 fn main() {
14     let x: [Felt; 4] = [Felt::new(1), Felt::new(2), Felt::new(3), Felt::new(4)];
15     let full_output = compute_output(x);
16     println!("Output:␣{:?}", full_output.map(|v| v.as_int()));
17
18     let opts = ProofOptions::new(
19         32, 8, 16, FieldExtension::None, 4, 127,
20         BatchingMethod::Linear, BatchingMethod::Linear,
21     );
22
23     let trace = build_trace(x);
24     let trace_ok = validate_trace(&trace);
25     if !trace_ok { panic!("Trace␣invalid!"); }
26     println!("Trace:␣all␣constraints␣=␣0");
27
28     let pub_in = PoseidonPub {
29         initial: std::array::from_fn(|i| trace.get(i, 0)),
30         output: std::array::from_fn(|i| trace.get(i, N_ROUNDS)),
31     };
32
33     let proof = PoseidonProver::new(opts.clone()).prove(trace).expect("Prover␣failed");
34
35     winterfell::verify::<PoseidonAir, PoseidonProofHasher, PoseidonProofCoin,
36         PoseidonProofVC>(
37         proof, pub_in, &AcceptableOptions::OptionSet(vec![opts]),

```

```

38
39     println!("Success");
40 }

```

Лістинг 5: train_mlp_synthetic.py

```

1  mod poseidon_air;
2  use poseidon_air::{
3      build_trace, compute_output, validate_trace,
4      Felt, PoseidonAir, PoseidonProver, PoseidonPub,
5      PoseidonProofHasher, PoseidonProofVC, PoseidonProofCoin,
6      N_ROUNDS,
7  };
8  use winterfell::{
9      math::{FieldElement, StarkField},
10     AcceptableOptions, BatchingMethod, FieldExtension, ProofOptions, Prover,
11 };
12
13 fn main() {
14     let x: [Felt; 4] = [Felt::new(1), Felt::new(2), Felt::new(3), Felt::new(4)];
15     let full_output = compute_output(x);
16     println!("Output:␣{:?}", full_output.map(|v| v.as_int()));
17
18     let opts = ProofOptions::new(
19         32, 8, 16, FieldExtension::None, 4, 127,
20         BatchingMethod::Linear, BatchingMethod::Linear,
21     );
22
23     let trace = build_trace(x);
24     let trace_ok = validate_trace(&trace);
25     if !trace_ok { panic!("Trace␣invalid!"); }
26     println!("Trace:␣all␣constraints␣=␣0");
27
28     let pub_in = PoseidonPub {
29         initial: std::array::from_fn(|i| trace.get(i, 0)),
30         output: std::array::from_fn(|i| trace.get(i, N_ROUNDS)),
31     };
32
33     let proof = PoseidonProver::new(opts.clone()).prove(trace).expect("Prover␣failed");
34
35     winterfell::verify::<PoseidonAir, PoseidonProofHasher, PoseidonProofCoin,
36         PoseidonProofVC>(
37         proof, pub_in, &AcceptableOptions::OptionSet(vec![opts]),
38     ).expect("Verification␣failed");
39
40     println!("Success");

```

```
40 }
```

Лістинг 6: Cargo.toml

```
1  [package]
2  name      = "mlp_poseidon"
3  version   = "0.1.0"
4  edition   = "2021"
5
6  [[bin]]
7  name      = "mlp_poseidon"
8  path      = "src/main.rs"
9
10 [[bin]]
11 name      = "poseidon_test"
12 path      = "poseidon_winterfell.rs"
13
14 [[bin]]
15 name      = "poseidon_standalone"
16 path      = "src/poseidon_standalone.rs"
17
18 [dependencies]
19 winterfell      = { version = "0.13.1", features = ["concurrent"] }
20 winter-crypto   = "0.13"
21 winter-utils    = "0.13"
22
23 [profile.release]
24 opt-level = 3
25
26 [profile.dev]
27 opt-level = 1
```