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

Навчально-науковий інститут телекомунікаційних систем

Кафедра інформаційних технологій в телекомунікаціях

До захисту допущено:
В.О. завідувача кафедри
_____ Валерій ПРАВИЛО
«____»_____2026 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформаційно-комунікаційні технології»
спеціальності 172 Телекомунікації та радіотехніка

на тему: Модифікований метод підвищення рівня анонімності користувача в мережі Tor

Виконав: здобувач вищої освіти 4 курсу, групи ТІ-21
Юшко Данило Сергійович _____

Науковий керівник: старший викладач кафедри ІТТ,
Курдеча Василь Васильович _____

Рецензент: доцент кафедри ТК НН ІТС
кандидат технічних наук, доцент
Авдєєнко Гліб Леонідович _____

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

Київ – 2026 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра інформаційних технологій в телекомунікаціях

Рівень вищої освіти – перший (бакалаврський)
Освітньо-професійна програма «Інформаційно-комунікаційні технології» спеціальності 172 Телекомунікації та радіотехніка

ЗАТВЕДЖУЮ
В.О. завідувача кафедри
_____ Валерій ПРАВИЛО
«____»_____2026 р.

ЗАВДАННЯ
на дипломну роботу студенту
Юшку Данилу Сергійовичу

1. Тема дипломної роботи: Модифікований метод підвищення рівня анонімності користувача в мережі Tor, науковий керівник роботи : старший викладач кафедри ІТТ Курдеча Василь Васильович -
затверджені наказом по університету від №1912-с від 26.05.2026

2. Строк подання студентом дипломної роботи 06.06.2026 р.

3. Об’єкт дослідження: Процес забезпечення анонімності користувача в мережі із цибулевою маршрутизацією.

4. Вихідні дані до роботи: архітектура та принципи побудови мережі Tor, методи деанонімізації користувачів на основі аналізу відбитків вебсайтів (Website Fingerprinting), алгоритми глибокого навчання для класифікації трафіку, існуючі рішення для генерації змагального шуму та багатошляхової передачі даних.

5. Перелік завдань, які потрібно розробити:

Дослідити сучасні види атак на основі аналізу трафіку у мережі Tor

Проаналізувати існуючі рішення забезпечення анонімності в мережі Tor

Модіифікувати метод підвищення рівня анонімності користувача в мережі Tor, що поєднує багатошляхову передачу та генерацію змагального шуму

Оцінити ефективність запропонованого рішення

6. Перелік ілюстративного матеріалу: 27

7. Дата видачі завдання: 10.10.2025 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Узгодження організаційних питань з науковим керівником	10.10.25-12.11.25	Виконано
2	Узгодження теми роботи та початок роботи над першим розділом	12.11.25-15.02.26	Виконано
3	Робота над першим, другим та третім розділами	15.02.26-01.04.26	Виконано
4	Робота над четвертим розділом. Планування розробки власного рішення	01.04.26-01.05.26	Виконано
5	Розробка власного рішення та його перевірка	01.05.26-13.05.26	Виконано
6	Підготовка п’ятого розділу	13.05.26-17.05.26	Виконано
7	Висновки про ефективність рішення	17.05.26-23.05.26	Виконано
8	Оформлення дипломної роботи	23.05.26-25.05.26	Виконано
9	Підготовка презентації до захисту	25.05.26 - 31.05.26	Виконано

Здобувач вищої освіти _____ Данило ЮШКО

Науковий керівник роботи _____ Василь КУРДЕЧА

РЕФЕРАТ

Дипломна робота «Модифікований метод підвищення рівня анонімності користувача в мережі Tor» містить 126 сторінок тексту, 10 рисунків та 26 таблиць, 1 додаток, а також використовує 35 літературних джерел посилання.

Актуальність теми: Зростання обчислювальних потужностей та розвиток алгоритмів машинного і глибокого навчання дозволяє зловмисникам здійснювати складні атаки типу Website Fingerprinting (аналіз відбитків вебсайтів) на мережі з цибулевою маршрутизацією. Перехоплення та аналіз трафіку локальним пасивним спостерігачем стає серйозною загрозою для приватності користувачів, що робить розробку та вдосконалення модифікованих методів обфускації та захисту анонімності у мережі Tor надзвичайно актуальним завданням кібербезпеки.

Метою роботи є зменшити ймовірність деанонімізації користувачів у мережах з цибулевою маршрутизацією. Це досягається шляхом дослідження сучасних видів атак, аналізу існуючих рішень забезпечення анонімності, проектування архітектури та модифікації методу підвищення рівня анонімності користувача в мережі Tor, що поєднує багатошляхову передачу та генерацію змагального шуму.

Об'єктом дослідження є процес забезпечення анонімності користувача в мережі із цибулевою маршрутизацією.

Предметом дослідження є модифікований метод підвищення рівня анонімності користувача в мережі Tor.

Наукова новизна: Модифіковано метод підвищення рівня анонімності користувачів в мережі Tor, що відрізняється поєднанням багатошляхової передачі і генерації шуму та дозволяє зменшити ймовірність деанонімізації користувачів.

Практична цінність: Запропоноване рішення може бути використане для створення додаткового програмного забезпечення для мережі Tor з метою підвищення рівня анонімності користувачів.

Ключові слова: МЕРЕЖА TOR, WEBSITE FINGERPRINTING, АНАЛІЗ ТРАФІКУ, ДЕАНОНІМІЗАЦІЯ, МАРШРУТИЗАЦІЯ, БАГАТОШЛЯХОВА ПЕРЕДАЧА, ГЕНЕРАЦІЯ ШУМУ, ЗМАГАЛЬНИЙ ШУМ.

ABSTRACT

The master's «Modified method of enhancing the level of user anonymity in the Tor network» contains 126 pages, 10 figures, and 26 tables and references 35 sources.

Relevance of the topic: The growth of computing power and the development of machine learning algorithms allow attackers to perform complex Website Fingerprinting attacks on networks with onion routing. Traffic analysis by a local passive observer becomes a serious threat to user privacy, which makes the development and improvement of modified methods of obfuscation and anonymity protection in the Tor network an extremely urgent cybersecurity task.

The objective of the work is to reduce the probability of users' deanonymization in networks with onion routing. This is achieved by researching modern types of attacks, analyzing existing solutions for ensuring anonymity, designing architecture, and modifying the method for enhancing user anonymity in the Tor network, which combines multipath transmission and the generation of adversarial noise.

The object of the study is the process of ensuring user anonymity in a network with onion routing.

The subject of the study is a modified method of enhancing the level of user anonymity in the Tor network.

Scientific novelty: The method of enhancing the level of user anonymity in the Tor network has been modified, which is distinguished by a combination of multipath transmission and noise generation and allows to reduce the probability of user deanonymization.

Practical value: The proposed solution can be used to create additional software for the Tor network in order to increase the level of user anonymity.

Keywords: TOR NETWORK, WEBSITE FINGERPRINTING, TRAFFIC ANALYSIS, DEANONYMIZATION, ROUTING, MULTIPATH TRANSMISSION, NOISE GENERATION, ADVERSARIAL NOISE

ЗМІСТ

СПИСОК ТЕРМІНІВ І СКОРОЧЕНЬ

OSI Open Systems Interconnection (Базова еталонна модель взаємодії відкритих систем)

TCP Transmission Control Protocol (Протокол управління передачею)

IP Internet Protocol (Міжмережевий протокол)

РОЗДІЛ 1 OR Onion Routers (Вузли-ретранслятори цибулевої маршрутизації)

DA Directory Authorities (Вузли-контролери)

TLS Transport Layer Security (Безпека транспортного рівня)

WF Website Fingerprinting (Аналіз відбитків вебсайтів)

SSH Secure Shell (Безпечна оболонка)

SVM Support Vector Machine (Метод опорних векторів)

VNG++ Variable n-gram attack (Вдосконалений алгоритм атаки на основі змінних n-грам)

k-NN k-Nearest Neighbors (Алгоритм k найближчих сусідів)

TPR True Positive Rate (Частота правильних спрацьовувань)

FPR False Positive Rate (Частота хибних спрацьовувань)

DL Deep Learning (Глибоке навчання)

SDAE Stacked Denoising Autoencoder (Стек шумопоглинаючих автокодувальників)

AWF Automated Website Fingerprinting (Автоматизований аналіз відбитків вебсайтів)

CNN Convolutional Neural Network (Згорткова нейронна мережа)

LSTM Long Short-Term Memory (Довга коротка пам'ять)

DF Deep Fingerprinting (Метод деанонімізації трафіку за допомогою глибокого навчання)

VGG Visual Geometry Group (Архітектура згорткових нейронних мереж)

WTF-PAD Website Traffic Fingerprinting Padding (Метод обфускації з наповненням трафіку фіктивними пакетами)

FRONT FRONT (Метод обфускації та маскування трафіку)

HTTP HyperText Transfer Protocol (Протокол передачі гіпертексту)

IAT Inter-Arrival Time (Часові інтервали між пакетами)

GAN Generative Adversarial Network (Генеративно-змагальна нейромережа)

GPU Graphics Processing Unit (Графічний процесор)

VRAM Video Random Access Memory (Відеопам'ять)

CW Closed World (Модель тестування класифікаторів "закритий світ")

OW Open World (Наближена до реальності модель тестування "відкритий світ")

BWR Bandwidth-Weighted Random (Випадковий вибір вузла з урахуванням пропускної здатності)

CPU Central Processing Unit (Центральний процесор)

RAM Random Access Memory (Оперативна пам'ять)

OOM Out of Memory (Стан нестачі оперативної або відеопам'яті)

DNS Domain Name System (Система доменних імен)

ВСТУП

У сучасному цифровому світі питання захисту приватності та інформаційної безпеки набувають першочергового значення. Мережа Інтернет стала невід'ємною частиною життя суспільства, однак разом із розширенням можливостей для вільного обміну інформацією безперервно зростають і ризики несанкціонованого втручання та стеження. Для збереження конфіденційності та обходу цензури користувачі все частіше звертаються до анонімних мереж, серед яких найпопулярнішою є мережа з цибулевою маршрутизацією – Tor. Її архітектура забезпечує високий рівень анонімності завдяки шифруванню та багаторазовій ретрансляції трафіку через розподілені вузли.

Однак стрімкий розвиток обчислювальних потужностей та алгоритмів штучного інтелекту створив нові серйозні виклики для безпеки анонімних мереж. Зокрема, атаки типу Website Fingerprinting (аналіз відбитків вебсайтів) дозволяють локальному пасивному спостерігачеві з високою точністю деанонімізувати користувача, аналізуючи виключно метадані зашифрованого трафіку – обсяг, напрямок передачі даних та послідовність пакетів. Можливості глибокого навчання (Deep Learning) зводять нанівець більшість базових методів обфускації, що робить розробку нових, більш стійких модифікованих методів захисту трафіку в мережі Tor надзвичайно актуальним завданням сучасної кібербезпеки.

Метою роботи є зменшити ймовірність деанонімізації користувачів у мережах з цибулевою маршрутизацією.

Для досягнення поставленої мети було визначено та виконано такі завдання:

1. дослідити сучасні види атак на основі аналізу трафіку у мережі Tor;
2. проаналізувати існуючі рішення забезпечення анонімності в мережі Tor;
3. виконати проектування архітектури модифікованого методу через аналіз існуючих рішень;
4. модифікувати метод підвищення рівня анонімності користувача в мережі Tor, що поєднує багатошляхову передачу та генерацію змагального шуму;

5. оцінити ефективність запропонованого рішення.

Об'єктом дослідження є процес забезпечення анонімності користувача в мережі із цибулевою маршрутизацією.

Предметом дослідження є модифікований метод підвищення рівня анонімності користувача в мережі Tor.

Методами дослідження є методи системного та порівняльного аналізу (для вивчення існуючих видів атак на основі машинного навчання та методів захисту), методи математичного моделювання (для проектування архітектури багатошляхової передачі і генерації змагального шуму), а також методи імітаційного моделювання і статистичного аналізу (для практичної оцінки ефективності та стійкості запропонованого рішення).

Наукова новизна одержаних результатів полягає у тому, що модифіковано метод підвищення рівня анонімності користувачів в мережі Tor, який відрізняється поєднанням багатошляхової передачі і генерації змагального шуму, що дозволяє суттєво зменшити ймовірність успішної деанонімізації користувачів пасивними спостерігачами.

Практичне значення дослідження полягає в тому, що запропоноване рішення може бути використане як основа для створення додаткового програмного забезпечення для мережі Tor з метою підвищення рівня анонімності кінцевих користувачів.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ СУЧАСНИХ ВИДІВ АТАК НА ОСНОВІ АНАЛІЗУ ТРАФІКУ У МЕРЕЖІ TOR

1.1 Особливості роботи мережі Tor. Алгоритм вибору шляху і основні принципи роботи

Tor мережа (скор. від англ. *The Onion Router*) – це мережа, що була створена для забезпечення анонімонсті та конфіденційності користувачів у мережі Інтернет. Мережа є безкоштовною та являється проєктом з відкритим вихідним кодом, що фінансово підтримується або підтримувалась волонтерами [1]. Технічно мережа Tor є відкритою системою проксі серверів, що працює як оверлейна мережа на прикладному рівні моделі OSI (Application-layer overlay network) поверх стандартної TCP/IP інфраструктури [2].

Основна структура мережі Tor складається з:

- Onion Proxy – клієнтське програмне забезпечення, яке локально запускається користувачем, приймає з'єднання (зазвичай через протокол SOCKS5) та керує побудовою криптографічних ланцюгів [3].
- Onion Routers (OR) – вузли-ретранслятори, які формують маршрут передачі даних. Залежно від позиції у ланцюгу, вони поділяються на вхідні (Guard nodes), проміжні (Middle nodes) та вихідні (Exit nodes). Також часто носять назву Реле – від англ. Relay [3].
- Directory Authorities – вузли-контролери що виступають як довірене джерело істиної інформації щодо поточного стану мережі [3].

Також до інфраструктури мережі належать непублічні ретранслятори – мости (Bridges). Вони виконують роль Guard-вузлів, але з певними обмеженнями – їх IP адреса не публікується у відкритих списках Directory Authorities, що дозволяє приховати факт підключення до мережі Tor від інтернет-провайдерів [2; 3].

Робота Tor починається з клієнта, що забезпечує анонімність свого трафіку створюючи власний ланцюг (circuit) з трьох вузлів. Користувачі

налаштовують ланцюги інкрементно, по одному вузлу за раз. Клієнт спочатку вибирає шлях з вузлів. Після завершення процесу вибору маршруту клієнт встановлює захищений канал зв'язку з кожним із обраних вузлів окремо. Для цього застосовується протокол ефемерного обміну ключами Діффі–Геллмана (Ephemeral Diffie-Hellman), у рамках якого між клієнтом і кожним ретранслятором незалежно узгоджується унікальний симетричний ключ шифрування. Використання ефемерних ключів гарантує властивість досконалої прямої секретності (Perfect Forward Secrecy) – таким чином компрометація одного сеансового ключа не призводить до розкриття інших сеансів чи попередніх з'єднань [2; 3].

Дані про вузли клієнт бере від DA, що створюють актуальній мережевий консенсус, який містить в собі таку інформацію [2; 3; 4]:

- Преамбула, де знаходяться дані самого документу, його строк дії, робочі прапори реле, глобальні параметри мережі.
- Розділ директорних органів, що має в собі дані про самі DA, що брали участь в голосуванні
- Записи про статус реле, що містять детальну інформацію про кожне активне реле, включаючи його нікнейм, IP-адресу, порти, версію програмного забезпечення Tor та набір статусних прапорців (таких як Guard, Exit, Fast, Stable), які визначають функціональну роль та придатність вузла для певних позицій у ланцюгу.
- Футер (Footer), де вказані вагові коефіцієнти пропускної здатності (bandwidth weights), необхідні для зваженого випадкового вибору вузлів та ефективного балансування навантаження в мережі
- Цифрові підписи (Signatures), які розташовані наприкінці документа і є криптографічним доказом того, що більшість довірених директорних органів погодили та завірили цей стан мережі.

Після побудови шляху відбувається безпосередня передача інформації. Клієнт надсилає трафік у вигляді зашифрованих пакетів по 512 Байт – ці пакети називають комірками (Tor Cells)(Рис.1.1) [2; 3; 4].

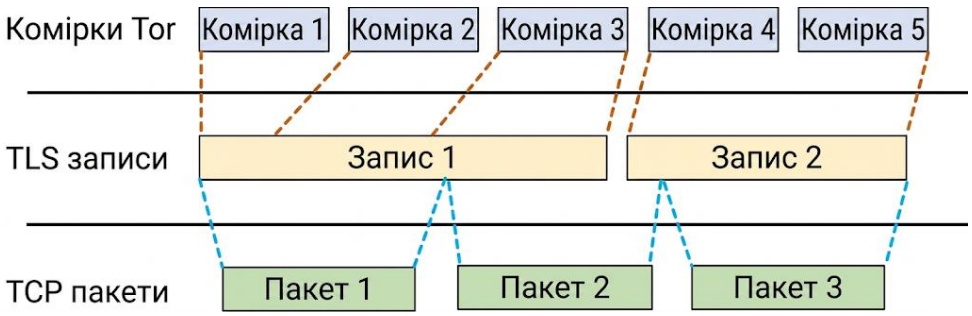

Рисунок 1.1 – Схематичне представлення даних на різних рівнях мережі

Використання фіксованого розміру комірки дозволяє ускладнити пасивний аналіз трафіку: зовнішній спостерігач не може визначити тип контенту (веб-сторінка, текстове повідомлення чи потокове відео) лише за об'ємом переданих пакетів, оскільки весь потік даних розбивається на однакові блоки. Кожна комірка складається із заголовка (містить ідентифікатор ланцюга та тип команди) і корисного навантаження (payload) [2; 3; 4].

На кожному з вузлів відбувається розшифрування верхнього слою криптографічного захисту за допомогою узгодженого симетричного ключа, що був створений під час встановлення зв’язку з кожним із вузлів. Таким чином Вхідний вузол знімає верхній шар шифрування, передаючи пакети далі. Вхідний вузол знає відправника, але не знає отримувача (лише наступний вузол у шляху). Середній вузол знімає другий – середній шар шифрування і надсилає пакети на Вихідний вузол, де знімається останній шар шифрування і оригінальний запит (захищений TLS при умові використання клієнтського додатку із власними засобами захисту на прикладному рівні) відправляється до отримувача (цільового сервера) [2; 3; 4].

Побудований шлях підтримує свою роботу 10хв. Короткий життєвий цикл шляху потрібен для забезпечення додаткового захисту від збору трафіку для подальшого його аналізу [5].

Схема з основним принципом побудови шляху та роботи Tor зображена на діаграмі послідовності нижче на рис. 1.2 [2].

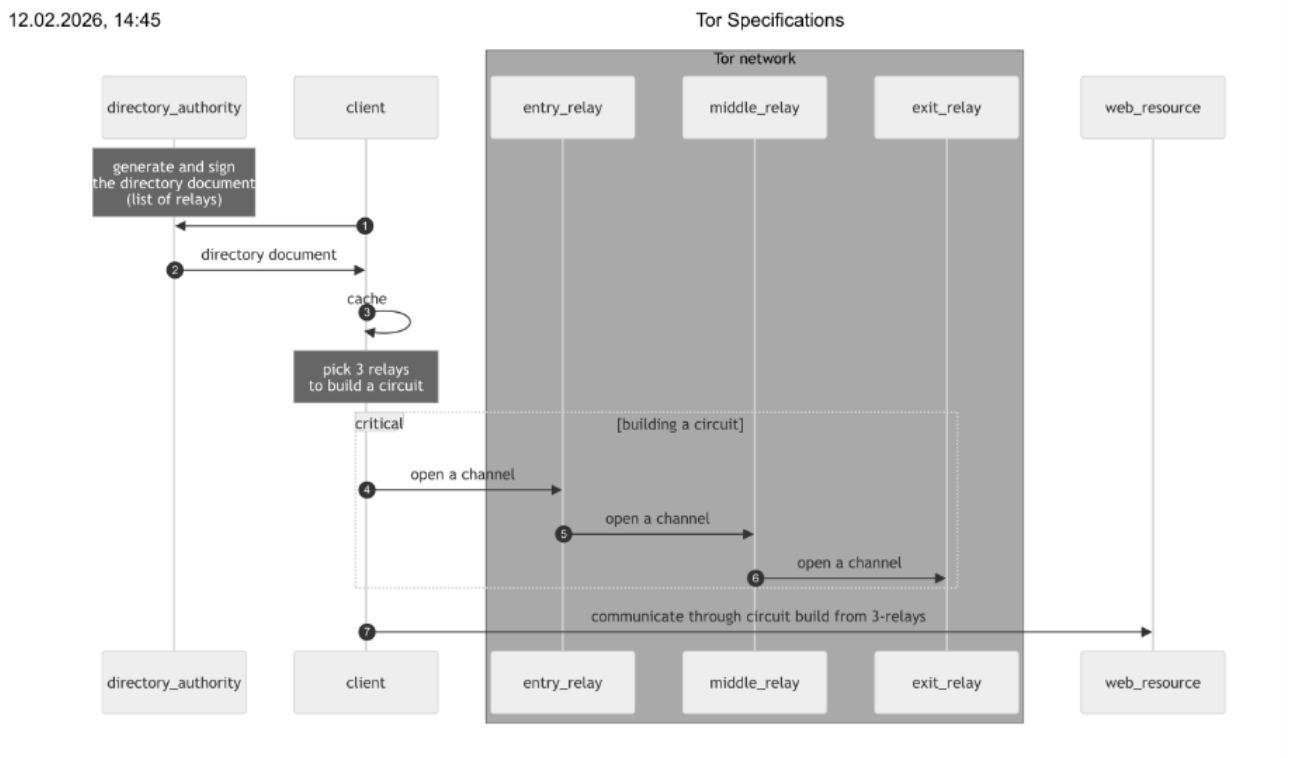

Рисунок 1.2 – Побудова шляху та передача інформації [2]

Процес побудови ланцюга у мережі Tor не є абсолютно випадковим – він базується на алгоритмі зваженого випадкового вибору, який намагається знайти компроміс між максимальним рівнем анонімності та прийнятною затримкою передачі даних. Основною метрикою процесу вибору шляху є пропускна здатність вузлів, дані про яку записані у консенсусі DA у вигляді рядка w (weights) для кожного реле. У консенсус дані попадають з дескрипторів сервера (Server Descriptor), шо описуються у файлі конфігурації (torrc file) на кожному вузлі окремо [2; 4]. Тобто первинним джерелом інформації про пропускну здатність каналу виступає сам сервер-реле, таким чином власник серверу-реле здатен самостійно задавати пропускну здатність яку готовий підтримувати. Також існують незалежні вимірювання проводяться спеціальними генераторами пропускної здатності (bandwidth authorities), які створюють файли з інформацією про поточну пропускну здатність (Bandwidth Files) [2]. У цих файлах для кожного реле вказується параметр bw (виміряна швидкість у

кілобайтах на секунду), а також додаткові діагностичні дані, як-от середнє (bw_mean) та медіанне (bw_median) значення вимірювань [2].

Метою збору цих метрик вузлів є забезпечення балансування навантаження (load balancing). Це дозволяє мережі (та окремим вузлам) не перенаватажуватись, а клієнту обирати оптимальний шлях для передачі даних, зберігаючи при цьому затримку у межах пряйнтних для користувача [2; 3].

Дані про пропускну здатність використовуються у алгоритмі зваженого випадкового вибору. Під час побудови ланцюга клієнт обирає вузли випадковим чином, проте ймовірність обрання конкретного реле прямо пропорційна його пропускній здатності, вказаній у консенсусі. Це означає, що вузол із вищою швидкістю буде частіше з'являтися в ланцюгах користувачів, це є слабким місцем мережі, адже використання зловмисником обладнання з вискоими показниками пропускної здатності – потенційно «маніпулює» алгоритмом зваженого випадкового вибору вузла [2; 3; 4].

З урахуванням короткого життєвого циклу шляху, перебудови шляху при зміні кінцевого хоста та основної метрики при побудові шляху можна розрахувати математичну модель, що буде показувати ймовірність захоплення трафіку на вузлах:

$$P = pG \times [1 - (1 - pE)^N] \qquad (1.1)$$

де P – загальна ймовірність того, що хоча б один з ваших шляхів буде скомпрометований;

pG – частка контрольованих зловмисником Вхідних вузлів;

pE – частка контрольованих зловмисником вихідних вузлів;

N – кількість використаних шляхів (ланцюжків) за певний час.

При змінних pG, pE та N модель покаже наступне (рис. 1.3):

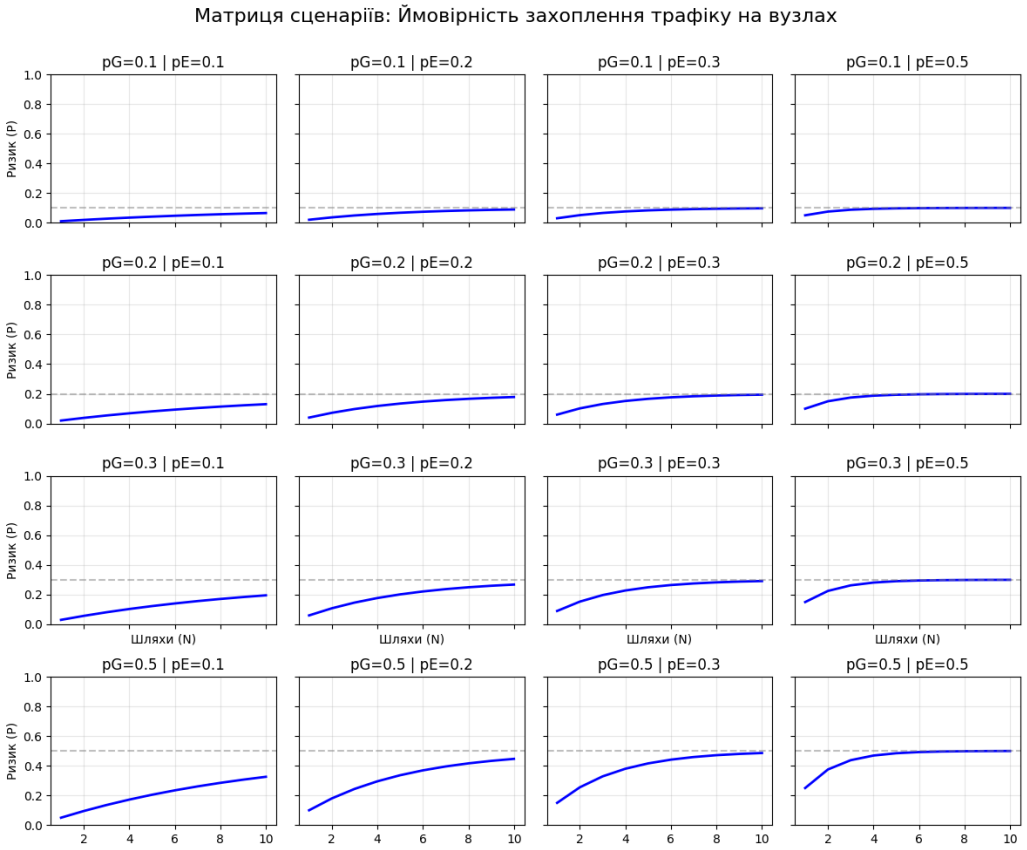

Рисунок 1.3 – Ймовірність захоплення трафіку на вузлах

На графіках (рис. 1.3) бачимо криву логарифмічного насичення – вона швидко зростає на малих N і згодом уповільнюється впираючись у стелю ризику, що визначається pG – часткою контрольованих зловмисником Вхідних вузлів. Тобто навіть якщо зловмисник контролює 100% вихідних вузлів (pE = 1), максимальний досяжний ризик обмежений часткою його вхідних вузлів. В свою чергу частка контрольованих Вихідних вузлів впливає на швидкість насичення кривої, тобто стрімкість збільшення ризику зростає пропорційно до збільшення частки контрольованих Вихідних вузлів. Критичним фактором безпеки мережі є можливість збору даних, адже при високих значеннях контролю обох типів вузлів стрімко зростає ризик деанонімізації шляхом аналізу зібраного трафіку.

Приводом для активного початку обговорення проблеми у спільноті користувачів мережі Tor стали багато випадків, що відбувались з 2020 року і актуальні дотепер. Наприклад атака Сивілли (Sybil-атака) на вузли Tor – починаючи з 2014 з найактивнішою фазою у 2021 році були отримані

повідомлення про тривалу атаку Сивілли на Tor, коли хтось запустив тисячі нових ретрансляторів – можливо, з метою деанонімізації користувачів. Припускалося, що це могли бути правоохоронні органи або дослідницька організація [6].

Починаючи щонайменше з 2017 року і досягнувши свого оперативного піку в період 2020-2021 років, невідомий, але надзвичайно високоресурсне кібер-угрупування, відоме серед дослідників безпеки під кодовою назвою KAX17, розгорнуло колосальну за масштабами тіньову інфраструктуру всередині самої мережі Tor [7; 8]. За своєю природою це була класична, але доведена до індустріальних масштабів атака Сивілли, метою якої є перехоплення контролю над значною часткою вузлів для впливу на алгоритми маршрутизації. Подібні захоплення великих часток мережі досягались завдяки алгоритмам зваженого випадкового вибору – нападник мав більшу пропускну здатність.

1.2 Теоретичні засади та еволюція класичних методів машинного навчання в атаках Website Fingerprinting

Найбільшу увагу серед усіх можливих атак на багатошарові анонімні мережі, такі як Tor, варто приділити атаці класу Website Fingerprinting (WF). Ця атака надає можливість локальному пасивному спостерігачу man-in-the-middle з високою точністю встановлювати який саме веб-сайт відвідує користувач. це може відбуватись навіть з урахуванням того, що атакуючий не має доступу до розшифрованого вмісту захоплених пакетів. Успішне застосування WF залишається можливим навіть за умови захисту з'єднання передовими методами шифрування або використання багатошарових анонімних мереж, таких як Tor.

Фундаментальний принцип атаки WF ґрунтується на тому, що кожен веб ресурс має характерні ознаки – набір об’єктів, що згодом стають складовою трафіку (зображення, скрипти, таблиці стилів тощо). У цій моделі зловмисник фокусується виключно на метаданих мережевого обміну: часі прибуття пакетів, їхньому напрямку (вхідний/вихідний) та об'ємі переданої інформації. Оскільки мережа Tor розбиває дані на комірки фіксованого розміру (512 байт), класичні

методи аналізу розмірів пакетів є менш ефективними, проте послідовність цих комірок, їхня щільність у часі та тривалість сплесків активності залишаються вразливими для статистичного аналізу.

Можливість такого глибинного аналізу полягає у використанні одного шляху для передачі данних від клієнт-сервер для кожної сесії. Це означає, що весь потік даних проходить через єдиний комунікаційний канал і маючи доступ до Вхідного вузла зловмисник може перехопити повний трафік .

Процес реалізації атаки концептуально поділяється на два послідовні етапи:

- Етап навчання (профілювання): На цій стадії агресор автономно генерує трафік, здійснюючи багаторазові автоматизовані запити до великої вибірки цільових веб-сайтів, та записує отримані мережеві траси. Зібраний масив статистичних даних та ознак трафіку використовується для тренування класифікаторів на основі алгоритмів машинного навчання (Machine Learning) [9].

- Етап імплементації (класифікації): Під час цього етапу агресор перехоплює трафік цільового користувача. Отриманий зашифрований трафік аналізується та порівнюється з раніше створеною базою шаблонів. Якщо алгоритм виявляє статистично значущу схожість між перехопленим трафіком та одним із відомих відбитків, анонімність користувача вважається скомпрометованою, оскільки факт відвідування конкретного ресурсу стає достовірно відомим спостерігачу [9].

Для оцінки ефективності методів деанонімізації на основі атак WF прийнято використовувати дві базові моделі тестування, які визначають спроможність моделі визначати «відбитки» за різних умов – це сценарії «закритого світу» (closed-world) та «відкритого світу» (open-world) [9].

Сценарій «закритого світу» є виключно лабораторним процесом, виконаним в ізольованих умовах. Цей сценарій припускає, що цільовий користувач може відвідувати інтернет-ресурси виключно з завчасно прописаного списку. Це може бути вибірка з 500 вебсайтів на основі частини якої відбувався процес навчання моделі. Хоча ця модель є базовим тестом для початкового

порівняльного аналізу математичних алгоритмів та архітектур нейромереж – вона суттєво переоцінює реальну загрозу деанонімізації, оскільки повністю ігнорує реальний масштаб мережі Інтернет та невідомий трафік, що буде заважати моделі у практичному використанні, зничуючи точність роботи класифікатора.

Зі свого боку, сценарій «відкритого світу» максимально наближений до реальних умов виконання атак WF. У цій парадигмі зловмисник прагне виявити відвідування лише невеликої групи специфічних цільових ресурсів (наприклад, заблокованих у певній країні новинних порталів або опозиційних форумів). Водночас клієнт генерує колосальний обсяг фонового трафіку, відвідуючи мільйони інших, невідомих класифікатору сторінок. Головним викликом для WF в умовах відкритого світу стає проблема хибних спрацьовувань (False Positives) та так звана проблема «базової ставки» (Base Rate Fallacy)[9]. Навіть за умови високої загальної точності класифікатора, більшість невідомого трафіку призводитиме до значної кількості хибних спрацювань, що суттєво ускладнює застосування WF для систем масового автоматизованого спостереження, залишаючи його ефективним переважно для цільового стеження.

Сьогоднішні досягнення у сфері деанонімізації за допомогою Deep Learning моделей стали можливими завдяки довгій еволюції класичних методів машинного навчання. Перше покоління успішних атак базувалось на класичному машинному навчанні з ручним вибором ознак трафіку. Дослідники розробляли алгоритми, що аналізували загальну кількість переданих байтів, розміри сплесків пакетів (bursts) та кількість змін напрямку передачі даних (рис. 1.4)[9].

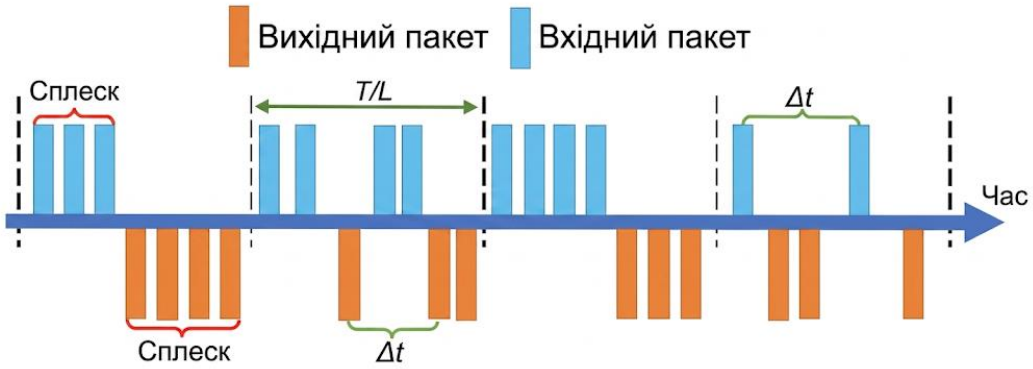

Рисунок 1.4 – Графік динаміки трафіку з урахуванням напрямку, сплесків та послідовності передачі пакетів

Розглядати еволюцію класичних методів машинного навчання можна у декілька етапів (розглянуті основні методи, фактичних досліджень набагато більше):

Ранні дослідження: статистично-частотний аналіз та Наївний Баєс (Naïve Bayes)(2006–2009)

Перші спроби деанонімізації зашифрованого трафіку спиралися на відносно прості статистичні моделі. У 2006 році Liberatore та Levine застосували алгоритм Наївного Баєса та метод марковських ланцюгів для ідентифікації SSH-тунелів, використовуючи як метрику розміри пакетів [10]. В таких умовах точність класифікатора досягала 96.65% [9]. Згодом, у 2009 році, Herrmann та команда адаптували цей підхід для мережі Tor, застосувавши мультиноміальний Наївний Баєс (Multinomial Naïve Bayes)[11]. Їхня модель аналізувала частотний розподіл розмірів пакетів. Проте цей метод виявився малоефективним проти Tor через фіксований розмір комірок (512 байт), що змусило дослідників шукати нові метрики, ознаки трафіку [9; 11].

Алгоритм називається Наївним через простоту своєї дії, за основу бралось припущення що кожна ознака є сама по собі незалаженою від інших, наприклад: Він «наївно» вважає, що розмір одного пакету ніяк не пов'язаний із часом прибуття наступного [9; 11]. У реальному житті це майже ніколи не так, проте саме завдяки цій «наївності» алгоритм працює дуже швидко – навчання може тривати від лічених секунд до годин, що було достатньо хорошим показником на той час.

Перехід до аналізу послідовностей та сплесків: метод опорних векторів (Support Vector Machine) (2011–2014)

У 2011 році Андрій Панченко та команда спеціалістів довели, що аналіз не лише розміру, а й напрямку та послідовності пакетів значно підвищує точність. Вони застосували Метод опорних векторів (Support Vector Machine, SVM), використовуючи такі ознаки, як відсоток вхідних/вихідних пакетів та обсяг даних, переданих у неперервних "сплесках" (bursts)[12]. Метод опорних векторів найкраще застосовується до WF атак через свою здатність до роботи з маленькими об’ємами даних та багатовимірними просторами. Фактично принцип роботи методу заключається у розділені кількох ознак між собою, наприклад: якщо взяти за основні метрики розмір пакетів та їх кількість – алгоритм буде здатний розділити цей об’єм даних, розташований на площині, на окремі простори, класи певним вектором [9; 12].

Пізніше з'явилися вдосконалені атаки, такі як VNG++ (2014), які враховували час між пакетами (inter-arrival times). Також особливо відзначився метод запропонований Вангом та командою де використовувався SVM з представленням трафіку мережі у вигляді одновимірного масиву розміром [1*L] де L – це елемент у вигляді пакету зі знаком + або – що відображає напрямок передачі комірки. З такими метриками точність класифікатора сягнула 90% у закритому світі. Це представлення трафіку стало класичним і для більшості сучасних моделей [9].

Однак ці підходи мали спільний недолік – вони потребували обчислення сотень складних часових ознак, що робило їх повільними на етапі навчання, також точність подібних моделей сильно залежала від обраних метрик, адже при виборі неправильних метрик, наприклад занадто “шумних”, модель просто не могла відрізнити певні закономірності і відповідно ставала недієздатною. Також дослідження показали, що часові інтервали в Tor є вкрай нестабільними через високий мережевий джитер (jitter), тому подібні алгоритми, базовані на часі, погано працювали в реальних, не лабораторних умовах.

Етап ускладнення: Атаки базовані на визначенні «найближчого сусіда» (2015)

Сам по собі алгоритм найближчих сусідів (k-NearestNeighbour) з'явився ще в середині XX століття, а в контексті деанонімізації Tor він здобув найбільшу популярність саме як складова частина гібридної атаки k-FP [9]. Механізм роботи методу k-NN полягає у створенні простору великої кількості ознак, що зазвичай відбирались вручну. Ознакою могла бути будь яка метрика, наприклад: кількість вхідних/вихідних пакетів, статистика сплесків, розподіл пакетів у перших 30 пакетах тощо. Далі у простір цих ознак вставляють досліджуваний об’єкт і обирається k кількість сусідніх ознак у просторі, що мають найменшу відстань до досліджуваного об’єкту. Далі відбувається голосування, де алгоритм перевіряє, до яких класів належать ці k найближчих сусідів. Нова точка відноситься до того класу, який є мажоритарним.

Саме на основі алгоритму найближчого сусіда у 2015 році був запропонований метод гібридної атаки k-FP (k-Fingerprinting)[13]. У поєднанні з алгоритмом Випадкового лісу (random forest) точність досягала 91% в умовах закритиого світу і 85% в умовах відкритого світу[9; 13].

Механізм роботи k-FP полягає у створенні нового простору ознак за допомогою алгоритму Random forest. Метрики відбирались вручну і велика кількість метрик дозволяла алгоритму досягати такої високої точності (класично 175 ознак для кожної сесії [13]). Алгоритм Random forest сторював унікальні вектори для кожної сесії, ці вектори були набором кінцевих листків кожного дерева. Таким чином створювався відбиток сесії, і після цього використовувався алгоритм k-NN для того щоб визначити найближчих сусідів досліджуваного об’єкту у просторі стоврених відбитків [9; 13].

Але алгоритм мав недолік, він потребував значних ресурсів для обчислення великої кількості метрик, також через особливість передачі даних по мережі Tor, а саме через його джитер, нівелювалась користь від складних часових метрик.

Оптимізація векторного представлення та стійкість до затримок: алгоритм CUMUL (2016)

CUMUL – це метод ідентифікації вебсайтів (Website Fingerprinting), запропонований Андрієм Панченко та іншими у 2016 році [14]. Замість того, щоб вимірювати складні часові інтервали між пакетами і брати до уваги велику кількість ознак, CUMUL дивиться на «форму» всього процесу завантаження сайту. Це відбувається завдяки створенню загальної картини у вигляді кумулятивної суми, звідки і пішла назва алгоритму (від слова "cumulative" – кумулятивний) [14].

Нехай ми маємо перехоплений трафік Tor, в цьому випадку ми оберемо за основну метрику напрямок передачі комірок. CUMUL повністю відкидає мілісекунди прибуття цих комірок. Замість цього він працює з їхнім порядковим номером (1-й пакет, 2-й пакет ... n-пакет) і будує кумулятивну (накопичувальну) суму на основі напрямку та ваги комірки. Нехай перехоплена мережева сесія (траса) складається з послідовності N пакетів (або комірок Tor). Кожен і-й пакет у цій послідовності має певне значення s_i, яке відображає його напрямок та розмір. Тоді при $s_i > 0$ (наприклад, +512 байт або просто +1), якщо і-й пакет є вхідним (завантажується від сервера до клієнта); $s_i < 0$ (наприклад, -512 байт або -1), якщо і-й пакет є вихідним (відправляється від клієнта до сервера). На основі цього формується унікальна крива – відбиток, що згодом інтерполюється для підготовки до SVM (оскільки класифікатор SVM вимагає, щоб усі вхідні вектори мали однакову довжину). CUMUL інтерполює отримані дані так, щоб зчитати значення у визначеній кількості рівновіддалених точок (класично використовується 100 точок). Ці точки стають математичним вектором (де кожен елемент – це значення кумулятивної суми на певному відрізку завантаження сторінки), який, разом із кількома глобальними характеристиками траси, використовується для навчання моделі або розпізнавання цільового ресурсу (класично вектор складється зі 100 точок та 4 глобальних ознак) [9; 14]. Порівняння методів атак можна побачити у таблиці 1.1, де TPR (True Positive Rate/Recall) – частота правильних спрацювань; FPR (False Positive Rate) –

частота помилкових спрацювань; Accuracy – загальна частка правильно класифікованих випадків

Таблиця 1.1 – Порівняння методів ідентифікації вебсайтів

Метод (Рік)	Закритий світ)	Відкритий світ	Механізм роботи	Слабкості (коротко)	Ознаки (Features)
Naïve Bayes (2006/2009)[10; 11]	~96% (SSH), ~3% (Tor)	Дуже низька	Базується на теоремі Байєса; припускає повну незалежність ознак.	Нереалістичне припущення про незалежність ознак; неефективний проти Tor через фіксований розмір комірок.	Гістограми частоти розмірів пакетів, розподіл довжини послідовностей та напрямок трафіку.
SVM (2011/2012)[12]	~55% (2011), ~90% (2012)	TPR ~73%	Пошук оптимальної гіперплощини для розділення класів.	Висока чутливість до вибору ознак; складність масштабування на великі набори даних.	Обсяг трафіку, час та напрямок
VNG++ (2012)[9]	~93.9%	Не була пріоритетною	Класифікатор на основі змінних n-грам, що використовує Наївний Байєс як базу.	Вразливість до методів наповнення трафіку (padding) та обфускації.	"Грубі" ознаки: загальний час передачі, ширина каналу на напрямок, маркери сплесків (bursts).
CUMUL (2016)14	90–96%	TPR ~96.6%, FPR ~9.6%	SVM з відображення траси через кумулятивну суму довжин пакетів.	Залежність від статистичних патернів обсягу, що легко руйнується активним захистом (padding).	100 кумулятивної суми розмірів пакетів + загальна кількість пакетів та байтів.
k-Fingerprinting (k-FP) (2016)[13]	91–96%	TPR 85%, FPR 0.02% (на 100k сайтів)	Ансамбль Випадкових лісів для генерації відбитків та k-NN (відстань Хеммінга) для класифікації.	Велика потреба в пам'яті; необхідність глибоких знань для ручного проектування ознак.	175 ознак: кількість пакетів, їхня частка за напрямком, статистика впорядкування, інтервали часу та пакети за секунду.

Але як і у всіх перечислених раніше методів – CUMUL має слабкість, вона полягає в його надмірній чутливості до засобів захисту, які додають у трафік фіктивні пакети (наповнення або padding). Оскільки алгоритм базується на підрахунку кумулятивної суми довжин пакетів, то будь-яка обфускація, що

змінює кількість або порядок пакетів, легко руйнує статистичні патерни, на які він покладається.

Але всі згадані методи (від Наївного Баєса до CUMUL) мали одну фундаментальну вразливість – жорстку залежність від математичних ознак, які дослідник обирав вручну, і будь-яке втручання у трафік змушували переписувати математичну модель.

1.3 Сучасні методи деанонімізації з використанням глибокого навчання (Deep Learning)

Складнощі у вигляді залежності від математичних ознак змусили дослідників змістити фокус наукових досліджень у бік глибокого навчання (Deep Learning). Це ознаменувало перехід від ручного проектування до автоматизованого визначення складних патернів із зашифрованого трафіку. Замість того, щоб подавати на вхід класифікатора розраховані людиною метрики, глибокі нейронні мережі здатні приймати "сирі" (raw) послідовності метаданих трафіку та самостійно виявляти складні, нелінійні та неочевидні просторово-часові закономірності.

Знову ж таки DL методи можна розглядати як і класичні методи машинного навчання, у вигляді еволюції:

Автоматизація видобутку ознак (2016-2017)

У 2016 році Ейб та команда вперше застосували методи глибокого навчання до атак Website Fingerprinting (WF) та запропонували атаку, засновану на механізмі стеку шумопоглинаючих автокодувальників (Stacked Denoising Autoencoder, SDAE) [9; 14]. У роботі було продемонстровано значний потенціал методів глибокого навчання для підвищення точності WF-атак: на наборі даних closed-world було досягнуто точності класифікації на рівні 88% [9].

Згодом Ріммер та команда запропонували підхід Automated Website Fingerprinting (AWF) [9;15], у якому ручний вибір ознак було замінено автоматичним вилученням ознак за допомогою SDAE (що допомагало у подальшій роботі вилучити шум) із подальшим використанням згорткових

нейронних мереж (CNN) та рекурентних мереж із довгою короткочасною пам’яттю (LSTM). Такий підхід забезпечив кращу адаптивність атаки до методів обфускації трафіку та змін мережевих шаблонів із часом. Крім того, AWF підвищив практичність і прихованість WF-атак, особливо в умовах динамічного вебконтенту та великомасштабних open-world середовищ [15].

Прорив у точності та створення еталонів (2018–2019)

Хоча згорткові нейронні мережі (CNN) були розроблені для задач комп’ютерного бачення (Computer vision), вони знайшли своє місце і у полі аналізу трафіку. Здатність CNN до автоматичного видобутку корисних ознак з сирих даних робить ці нейромережі дуже ефективними у вивченні складних закономірностей зашифрованого трафіку (зокрема трафіку Tor) [9; 16].

Модель Deep Fingerprinting (DF) розроблена дослідником Payap Sirinam та командою, вона базується на складній архітектурі згорткової нейронної мережі (CNN), натхненній відомою моделлю VGG для розпізнавання зображень [16]. На вхід DF отримує той самий вектор у вигляді одвимірної послідовності напрямків пакетів розглянутої раніше [9; 16].

DF складається з базового блоку на 8 шарів, що повторюється ітеративно 4 рази, та повнозв’язного шару, що повторюється ітеративно 2 рази (рис.1.5) [16]:

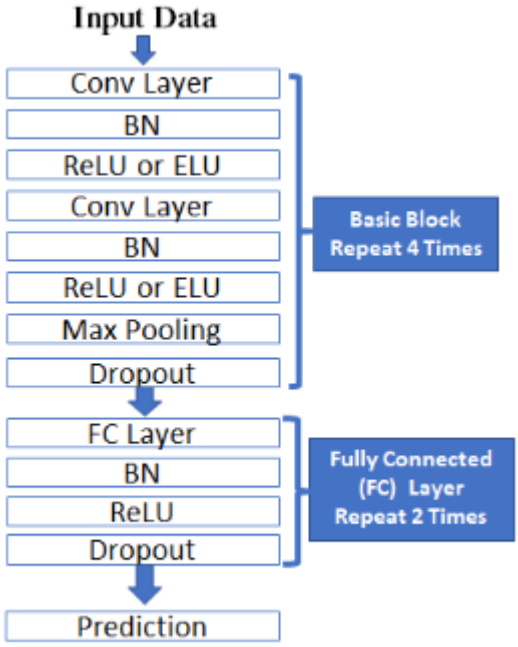

Рис.1.5 Узагальнена архітектура моделі Deep Fingerprint

- Базовий шар виконує вилучення ознак із вхідної послідновності, кількість нейронів на кожній ітерації збільшується (32, 64, 128, 256), що дозволяє поглиблюватись у абстрактність ознак, виявляючи складніші патерни [16].
- Повнозв'язний шар виконує класифікацію ознак, визначаючи фінальний результат [16].

У моделі також є додаткові шари, що дозволяють оптимізувати аналіз стабілізуючи процесс навчання (batch normalization layer), вводячи нелінійність (Rectified Linear Unit або Exponential Linear Unit), виділяючи найбільш значущі ознаки (Max Pooling), знижуючи ризик перенавчання (Dropout) [16].

Одним із найважливіших рішень у методі DF було використання шарів субдискретизації (Max Pooling). Мережа вчиться ігнорувати випадкові фіктивні пакети, згенеровані захистом, оскільки фокусується лише на найбільш виражених структурних максимумах завантаження сторінки. Модель досягла точності 98% у закритому світі.

Хоч модель DF і стала еталонною у сфері WF атак – вона все ж таки має недолік у вигляді обмеженого «поля зору», модель найкраще справляється з короткими прикладами трафіку, але погано корелює пакети, розділені великими часовими інтервалами.

У 2019 році Санджит Бхат та його команда розробили архітектуру Var-CNN, яка розв'язує проблему аналізу довгострокових залежностей завдяки використанню глибших мереж та зміненому формату згортки [18].

Модель Var-CNN має достатньо складну багатошарову архітектуру, тому в межах даної роботи доцільно зосередитися виключно на її основних механізмах роботи без детального розбору топології кожного шару. Головною інновацією алгоритму є паралельна обробка двох незалежних вимірів мережевої сесії. Var-CNN використовує дві відокремлені нейромережі: перша приймає на вхід вектор напрямків пакетів, а друга обробляє масив кумулятивних часових затримок між їхнім прибуттям. Простіше кажучи – матеріалом для навчання моделі є патерни напрямку трафіку та час прстою між відправками пакетів. Обидві гілки побудовані на базі архітектури ResNet, яка застосовує механізм залишкового навчання. Використання прямих зв'язків (skip connections) дозволяє передавати сигнал в обхід проміжних блоків, що вирішує проблему згасаючого градієнта (тенеденція до деградації працездатності шарів) та дає змогу ефективно оптимізувати ваги у глибоких мережах.

Для захоплення довгострокових залежностей між віддаленими у часі пакетами застосовано дилатовані згортки (Dilated Convolutions). Ця операція застосовує згортковий фільтр до елементів масиву із заданим кроком, що експоненційно збільшує рецептивне поле мережі без додавання нових обчислювальних параметрів. На фінальному етапі вектори ознак, видобуті з обох згорткових мереж, конкатенуються з масивом глобальних статистичних метаданих траси. Фінальний результат отримується при комбінуванні виводу класифікаторів двох моделей у блоці Softmax. Метод є високоефективним і показує результати ~98% в умовах закритого світу і ~98% в умовах відкритого світу навіть при високих показниках шумового трафіку [18].

Робота у складних реалістичних умовах та ранішня класифікація (2019-2024)

У 2019 році було розпочато дослідження експлуатації вразливостей ранніх стадій з'єднання та таймінгів пакетів. Рахманн та команда розробили нову

модель Tik-Tok, що здатна максимально достовірно визначати приховані патерни особливо в умовах використання активного захисту або/та в умовах сильних шумів [19]. Tik-Tok базується на моделі DF, але використовує нове представлення даних – спрямований час (directional timing), що поєднує послідовність напрямків пакетів із часовою інформацією. Дослідники довели, що більшість існуючих систем обфускації працюють реактивно і залишають перші мікро-сплески трафіку (початок завантаження) без достатнього шумового покриття. Використовуючи ці ранні таймінги, Tik-Tok змогла успішно нівелювати ефективність систем додавання фіктивних пакетів. Атака досягла 98,4% точності на незахищеному трафіку Tor та 93,5% при умовах використання методів захисту з генерацією шуму (буде розглянуто у наступних розділах) [9; 19].

Цю парадигму довів до абсолюту алгоритм Holmes (2024 рік, запропонований Денгом та іншими), який розв'язав проблему ідентифікації ресурсів у динамічних мережах із нестабільними затримками [20]. Holmes базується на просторово-часовому аналізі та використовує метод керованого контрастного навчання (Supervised Contrastive Learning), коли нейромережа-енкодер навчається групувати точки раннього та повного трафіку одного сайту близько одна до одної в низьковимірному латентному просторі. Архітектура моделі здатна математично зіставити фрагмент невідомого трафіку з еталонними профілями сайтів, навіть якщо перехоплено лише початкову фазу сесії. Експериментальні дані довели, що Holmes здатний з високою точністю ідентифікувати цільовий веб-ресурс у той момент, коли сторінка завантажилася лише на 20-25% [20]. Атака продемонструвала здатність ідентифікувати сайти в даркнеті, коли в середньому завантажено лише 21,71% даних, з точністю близько 85,19% [20].

Нижче – порівняльна таблиця методів деанонімізації (Website Fingerprinting) (табл. 1.2), що базуються на глибокому навчанні (Deep Learning), у хронологічному порядку їх появи та розвитку.

Таблиця 1.2 – Порівняння методів атак WF, що базуються на глибокому навчанні (DL) у хронологічному порядку їх появи та розвитку

Метод (Рік)	**Основні ознаки трафіку**	**Механізм роботи**	**Закритий світ**	**Відкритий світ**	**Основні слабкості**
SDAE (~2016)[9]	Послідовності напрямків пакетів (+1/-1).	Використовує SDA для автоматичного навчання низьковимірних представлень ознак із «шумного» трафіку.	~88%.	~ 86%	Потребує великої кількості даних; не перевершував тогочасні класичні методи (SVM) за точністю.
AWF (~2017)[15]	Послідовності напрямків, порядок пакетів, сплески (bursts).	Перше систематичне дослідження CNN та LSTM для автоматичного вилучення ознак та підбору параметрів.	~94%.	~71%	Вразливий до сучасного захисту (WTF-PAD); фіксовані гіперпараметри обмежують здатність до навчання.
DF (Deep Fingerprinting) (2018)[17]	Сирі послідовності напрямків пакетів (до 5000 елементів).	Глибока згорткова мережа (CNN), натхненна архітектурою VGG. Використовує функцію активації ReLU та субдискретизації MaxPooling	98.3%.	~95.7%	Хутливість до часового дрейфу даних (traffic drift) та змагальних атак.

Продовження таблиці 1.2

Var-CNN (2019)[18]	Ансамбль напрямку, часу пакетів та статистичних метаданих.	Базована на ResNet-18 із використанням диляційних згорткових шарів для вловлювання довгострокових залежностей.	98.8%	98.01%	Висока обчислювальна складність навчання через складність моделі та ансамблювання.
TikTok (~2019/2020)[19]	Часові характеристики сплесків трафіку (напр., медіанна мітка MED).	Модифікована модель DF, що використовує спрямований час (directional timing) та гістограми характеристик сплесків.	98.4%.	~97%	Залежність від точності сегментації сплесків; втрачає актуальність при швидких змінах контенту сайтів.
Holmes (2024)[20]	Агреговані ознаки (TAF) ранніх етапів завантаження (перші 20–40% трафіку).	Використовує SCL для групування фрагментів раннього трафіку з повними відбитками в латентному просторі.	~90.6% (40% завантаження сторінки).	94.58%	Неефективний в умовах змішаного трафіку (багатокладкові сесії) та при зміні структури шаблонів сайтів.

1.4 Висновки до розділу 1

У першому розділі проведено комплексний аналіз теоретичних засад та сучасного стану проблеми забезпечення анонімності в мережі Tor. Розглянуто архітектуру мережі: принцип цибулевої маршрутизації, трирівнева ланцюгова структура (Guard – Middle – Exit), механізм узгодження ключів за протоколом Діффі–Геллмана та роль Directory Authorities. Встановлено, що алгоритм зваженого випадкового вибору вузлів є структурною вразливістю: зловмисник із потужнішим обладнанням здатний маніпулювати вибором реле та підвищити ймовірність деанонімізації, що кількісно підтверджується наведеною математичною моделлю.

Детально вивчена атака класу Website Fingerprinting – найбільш практично значуща загроза, за якої локальний пасивний спостерігач ідентифікує відвідуваний ресурс виключно через аналіз метаданих трафіку: послідовності та напрямку комірок, щільності сплесків і часових патернів. Еволюція класичних методів машинного навчання – від Наївного Баєса і SVM до k-FP та CUMUL – демонструє спільну фундаментальну ваду: жорстку залежність від ручного проектування ознак (feature engineering). Дослідники вручну відбирали метрики, і будь-яка зміна трафіку чи активний захист типу padding руйнував статистичні патерни, змушуючи переписувати модель.

Ця обмеженість стала поштовхом до переходу у сферу глибокого навчання. Нейронні мережі самостійно виявляють нелінійні просторово-часові залежності у сирих послідовностях трафіку без ручного відбору метрик. Deep Fingerprinting досягла точності 98% у закритому світі, Var-CNN – порівнянних результатів у відкритому, а Holmes продемонстрував здатність до ідентифікації вже при завантаженні 20–25% даних сесії. Пук

Попри вражаючі показники, DL-моделі не є досконалим інструментом. Вони схильні до часового дрейфу даних: точність знижується при розходженні реального трафіку з навчальною вибіркою. Результати в умовах відкритого світу суттєво поступаються лабораторним, а стійкість до засобів активного захисту залишається предметом активних досліджень.

Отримані знання про вразливості алгоритму вибору шляху та механізми WF-атак безпосередньо визначають вектор подальшого дослідження – розроблення модифікованого методу, здатного протистояти сучасним DL-класифікаторам на рівні архітектури передачі трафіку.

РОЗДІЛ 2 ІСНУЮЧІ РІШЕННЯ ЗАБЕЗПЕЧЕННЯ АНОНІМНОСТІ В МЕРЕЖІ TOR

2.1 Методи додаткового захисту анонімності користувача мережі Tor

Розвиток методів деанонімізації, розглянутих у попередньому розділі, наочно продемонстрував, що стандартні механізми шифрування та багатошарової маршрутизації в мережі Tor не здатні забезпечити абсолютну конфіденційність в умовах пасивного спостереження. Критична вразливість перед атаками типу Website Fingerprinting виникає через збереження унікальних статистичних характеристик мережевого трафіку, таких як послідовність пакетів, їхній об’єм та часові інтервали, які залишаються доступними для аналізу локальному спостерігачу. Сучасні дослідження підтверджують, що навіть за умови використання багатошарового шифрування, атакуючий може ідентифікувати веб-ресурси з високою точністю, що робить розробку додаткових засобів захисту пріоритетним завданням для наукової спільноти.

У відповідь на ці виклики було розроблено широкий спектр механізмів, спрямованих на обфускацію або зміну форми мережевих трас. Згідно з найактуальнішими оглядами галузі, ці стратегії класифікуються за кількома ключовими доменами: адаптивне наповнення трафіку (adaptive padding), жорстка регуляризація (traffic regularization), морфінг трафіку (traffic morphing) та змагальні збурення (adversarial perturbation). Кожен із цих підходів намагається вирішити проблему балансу між приватністю та зручністю. Додатково повна зміна екосистеми Tor є неможливою через архітектурну осифікацію мережі.

Фундаментальною складністю розробки ефективних засобів протидії є необхідність вирішення критичної дилеми щодо балансу між приватністю, зручністю використання та загальною продуктивністю системи. Більшість теоретично стійких моделей захисту вимагають значних накладних витрат у вигляді надлишкового трафіку (bandwidth overhead) або штучного внесення затримок (latency overhead), що суттєво погіршує користувацький досвід та

створює надмірне навантаження на інфраструктуру ретрансляторів. Саме цей компроміс часто стає причиною того, що перспективні наукові рішення залишаються на етапі прототипів і не інтегруються в основне програмне забезпечення Tor Browser.

Системний аналіз існуючих рішень у межах цього розділу дозволяє простежити еволюцію захисних механізмів від ранніх статичних методів до сучасних адаптивних систем на основі генеративних моделей. Такий підхід необхідний для виявлення слабких сторін існуючих стратегій, зокрема їхньої вразливості до новітніх атак на основі глибокого навчання або нерентабельності з точки зору ресурсних витрат. Ретельне вивчення накопиченого досвіду стає необхідним підґрунтям для подальшого обґрунтування та розробки модифікованого методу підвищення рівня анонімності, який би забезпечував надійний захист при збереженні прийнятної швидкодії мережі.

Ранні методи та стратегії "камуфляжу"

Перші спроби розробки механізмів протидії атакам типу Website Fingerprinting були тісно пов’язані з усвідомленням того факту, що базове шифрування мережі Tor не приховує метадані трафіку, зокрема розмір і час передачі пакетів. Історично одним із перших цілеспрямованих методів захисту стала стратегія маскування, відома як Camouflage. Вона була запропонована дослідницькою групою А. Панченка у 2011 році одночасно з презентацією їхнього класифікатора на основі методу опорних векторів (SVM)[12]. Головна ідея цього підходу полягала у приховуванні справжнього «відбитка» цільової веб-сторінки шляхом штучного змішування її трафіку з фоновим шумом або даними іншої «приманкової» сторінки (decoy page) [9; 12].

У межах методу Camouflage клієнтське програмне забезпечення генерувало фіктивні запити до популярних веб-ресурсів паралельно із завантаженням того сайту, який користувач насправді хотів відвідати. Спостерігач міг бачите лише незв’язний трафік, що статистично дуже відрізнявся від звичного трафіку, і це суттєво ускладнювало роботу тогочасних статистичних класифікаторів. Якщо зловмисник намагався виявити патерни

специфічного ресурсу, додатковий трафік від одночасного завантаження стороннього порталу деформував загальну послідовність, об’єм та часові інтервали передачі даних.

Практичне тестування та подальший аналіз виявили фундаментальні недоліки цієї стратегії, які зробили її масове впровадження на рівні всієї архітектури Tor – неможливим. Основним бар’єром стали колосальні накладні витрати мережевих ресурсів. Оскільки для створення надійного маскування доводилося завантажувати повноцінні сторонні веб-ресурси або генерувати великі обсяги фіктивних пакетів, додаткові витрати пропускної здатності каналу (bandwidth overhead) часто перевищували допустимо-комфортну норму користувача [9]. Для анонімної мережі, яка історично функціонує в умовах обмеженої пропускної здатності та залежить від вузлів-добровольців, подібне навантаження є критично неприйнятним.

Розвиток моделей машинного навчання показав, що навіть такий захист стає все менш ефективним, адже моделі навчилися математично відокремлювати фіктивний шум та ідентифікувати специфічні сплески (bursts) реального трафіку навіть на тлі паралельного завантаження. Хоча ранній метод Camouflage заклав концептуальне підґрунтя для дослідження методів захисту, він яскраво продемонстрував необхідність зберігання балансу між ефективністю методу та продуктивністю мережі.

Регуляризація трафіку (BuFLO та CS-BuFLO)

Наступним етапом розвитку систем захисту став перехід до стратегій жорсткої регуляризації трафіку. Ці методи намагалися повністю нівелювати статистичні відмінності між потоками даних різних веб-сторінок.

Алгоритм BuFLO (Buffered Fixed-Length Obfuscator) [22] був представлений у 2012 році групою дослідників під керівництвом Кевіна Дайера. Згодом, у 2014 році, команда під керівництвом Сяо Кая запропонувала вдосконалену версію – CS-BuFLO, яка враховувала завантаженість каналу зв'язку [22]. Причиною створення цих методів стала доведена вразливість

попередніх стратегій «камуфляжу» перед новими на той час способами аналізу послідовностей пакетів як VNG++ та SVM [9; 22].

Принцип роботи BuFLO полягав у створенні штучного «сталого потоку» даних, який повністю приховував реальну активність користувача. Алгоритм боровся з витоками інформації через три основні параметри: фіксований розмір кожного пакету, відправка даних через суворо визначені часові інтервали та продовження сесії протягом фіксованого періоду. Якщо користувач не передавав реальну інформацію, система автоматично заповнювала канал фіктивними пакетами. У ситуаціях, коли об'єм реальних даних перевищував встановлену швидкість, вони накопичувалися в буфері та відправлялися поступово. Це дозволяло ефективно приховати унікальні сплески трафіку, характерні для конкретних сайтів [22].

Ефективність такого підходу була надзвичайно високою, оскільки точність ідентифікації ресурсів зловмисником знижувалася до мінімальних 5%. Головною перевагою методу стала його здатність протистояти практично всім тогочасним статистичним класифікаторам. Але критичним мінусом виявилися колосальні накладні витрати пропускної здатності (overhead), які часто сягали >100%. Окрім надмірного споживання трафіку, користувачі відчували значні затримки через постійну буферизацію даних. Через таку низьку продуктивність методи жорсткої регуляризації так і не були впроваджені в реальну інфраструктуру Tor [9; 22].

Легковагове адаптивне наповнення (WTF-PAD, Walkie-Talkie та FRONT)

Критичні недоліки та низька продуктивність методів жорсткої регуляризації змусили дослідників шукати компромісні рішення. Головною метою нових розробок стало створення систем захисту, які б надійно приховували статистичні ознаки трафіку, але не створювали критичного навантаження на канали зв'язку. У 2016 році група вчених у складі Марка Хуареса, Мохсена Імані, Майка Перрі, Клаудії Діаз та Меттью Райта запропонувала алгоритм адаптивного наповнення трафіку WTF-PAD [24]. Вже наступного 2017 року дослідники Тао Ванг та Іан Голдберг представили

альтернативний підхід під назвою Walkie-Talkie [23]. Обидва методи створювалися для активної протидії тогочасним класичним атакам машинного навчання, таким як SVM, k-NN та CUMUL, які демонстрували високу точність деанонімізації через аналіз форми сплесків трафіку та інтервалів між пакетами [9; 23; 24].

Алгоритм WTF-PAD функціонує на основі адаптивних автоматів станів, які використовують математичні гістограми для моделювання інтервалів між пакетами. Особливість методу заключається у заповненні фіктивними пакетами лиш природних пауз трафіку, а не генерації безперервного монотонного потоку. Коли реальний обмін даними припиняється, автомат починає відлік часу за заздалегідь визначеним розподілом. Якщо пауза між пакетами стає занадто тривалою, система впроваджує в канал поодинокі фіктивні пакети, які імітують продовження завантаження. Важливою особливістю WTF-PAD є нульова затримка реального трафіку, та наявність двосторонніх автоматів, що працюють як у режимі клієнт-вузол, так і у режимі вузол-клієнт, заповнюючи трасу фіктивним трафіком в обидві сторони. Алгоритм ніколи не зупиняє і не затримує легітимні пакети користувача, а лише заповнює штучним шумом інформативні прогалини. Це дозволяє ефективно руйнувати часові патерни та маскувати початок і завершення великих сплесків даних, зберігаючи мінімальні витрати пропускної здатності [9; 24].

Метод Walkie-Talkie використовує принципово іншу стратегію, яка базується на концепції напівдуплексного зв'язку та формування супер-послідовностей [9; 23]. Замість випадкового додавання шуму цей алгоритм прагне зробити мережеві сліди двох різних веб-сторінок повністю однаковими. Для реалізації цього підходу реальна сторінка користувача об'єднується з випадковою сторінкою-приманкою. Браузер примусово переводиться в режим напівдуплексу, де він може або лише надсилати, або лише отримувати дані в один момент часу. Це значно спрощує структуру трафіку та усуває хаотичні коливання. Система штучно вирівнює кількість та напрямок пакетів для обох сторінок, створюючи єдиний шаблон. Пасивний локальний спостерігач бачить

ідентичний набір сплесків як для захищеного ресурсу, так і для звичайного сайту, що позбавляє його можливості провести точне розпізнавання [23].
Ефективність захисту цих методів значно зросла у порівнянні з розглянутими раніше методами. Алгоритм WTF-PAD на момент своєї публікації успішно знижував точність класичних атак з 90% до безпечних 20% [24] . Головною перевагою методу стали помірні накладні витрати на додатковий трафік, які становили менше шістдесяти відсотків, а також повна відсутність затримок у завантаженні сторінок. Однак його суттєвим мінусом виявилася слабкість до атак глибокого навчання. Сучасні нейромережі типу Deep Fingerprinting навчилися ігнорувати адаптивне наповнення та відновлювати точність класифікації до початкових показників. Метод Walkie-Talkie виявився набагато стійкішим. Його математична модель обмежує теоретичну точність будь-якої атаки рівнем у п'ятдесят відсотків, що дорівнює випадковому вгадуванню [23]. Навіть найскладніші згорткові нейромережі не здатні подолати цей бар'єр, оскільки самі вхідні дані для двох сайтів стають абсолютно ідентичними.

Аналіз застосування цих методів у реальних умовах демонструє протилежні результати. Алгоритм WTF-PAD виявився дуже практичним і життєздатним рішенням. Завдяки низьким накладним витратам його адаптовану версію офіційно інтегрували в ядро мережі Tor [9; 2]. Це підтверджує повну сумісність підходу із сучасними стандартами анонімності. Натомість метод Walkie-Talkie є абсолютно нереалістичним для використання на практиці. Для його роботи клієнтський додаток повинен постійно зберігати та оновлювати величезну базу мережевих відбитків. Крім того, примусовий напівдуплексний режим повністю руйнує переваги сучасних протоколів HTTP/2 та HTTP/3. Це викликає критичні затримки та робить швидкий перегляд сторінок неможливим для користувача.

Нівелювати слабкість WTF-PAD до атак глибокого навчання намагались дослідники з Гонконзького університету науки й технологій Цзяцзюнь Гун та Тао Ванг, що у 2020 році представили метод захисту під назвою FRONT (Fast and Robust Obfuscation of Network Traffic) [9; 25]. Розробники виявили

очевидний прорахунок попередників – початок трафіку найбільш інформативний.

Суть захисту полягає в цілеспрямованому спотворенні найважливішої частини мережевого сліду – початку передачі даних. Клієнт та вхідний вузол Tor абсолютно незалежно один від одного обчислюють випадкову кількість фіктивних пакетів для штучного додавання у канал зв'язку [9]. Для планування часу їхньої відправки система використовує статистичний розподіл Релея. Це означає, що основна маса «шуму» застосовується до самого початку сесії, де інформативність трафіку для зловмисника найвища, а потім інтенсивність наповнення поступово згасає [25]. Реальний потік не зупиняється. Завдяки архітектурі zero-delay легітимні дані користувача проходять крізь захисний шар миттєво без жодного штучного гальмування чи буферизації. Кожна окрема сесія отримує унікальний шаблон. Це забезпечує trace-to-trace рандомізацію, тобто навіть два звернення до одного і того ж сайту виглядають абсолютно по-різному для класифікатора зловмисника, що руйнує процес навчання нейромереж. [25].
Робота дослідників містить в собі результати двох експериментів застосування FRONT з різними конфігураціями та відповідно різним балансом ефективності. У версії з помірними витратами (FT-1) FRONT перевершує WTF-PAD, знижуючи точність атак глибокого навчання (як-от Deep Fingerprinting) суттєво ефективніше. Проте проти спеціалізованих часових атак, як-от Tik-Tok, точність залишається на рівні близько 66% [25]. Також overhead варіюється від 33% (легка конфігурація) до 119% (важка конфігурація FRONT-2500) і затримка мережі майже нульова [9; 25].

Гібридні підходи (2018–2024): TrafficSliver, A2A, WF-UAP, ALERT, RUDOLF

Критичний аналіз попередніх методів виявив, що кожна окрема категорія захисту атакує лише один рівень абстракції WF-ознак: адаптивне padding маскує локальні timing-характеристики, але не впливає на глобальну структуру трафіку; регуляризація нівелює всі ознаки, але створює неприйнятний overhead. Саме усвідомлення цих обмежень призвело до появи гібридних рішень, що

намагалися об'єднати переваги різних підходів, компенсуючи їхні слабкі сторони.

Одним із перших методів цієї категорії став TrafficSliver, представлений у 2020 році командою дослідників на чолі з Сяо Де [26]. Інновація полягала у відмові від традиційної архітектури Tor, де весь трафік користувача проходить через один вхідний вузол (Guard node). Замість цього TrafficSliver реалізував концепцію розподілу потоку даних між кількома паралельними entry-ретрансляторами. Алгоритм створював не один, а декілька незалежних Tor-ланцюжків одночасно, кожен з яких мав власний набір Guard-Middle-Exit вузлів. TCP-потік веб-сеансу розбивався на фрагменти, які динамічно розподілялися між цими паралельними шляхами. Як результат, жоден окремий вхідний вузол не отримував доступу до повного трафіку користувача – кожен спостерігач бачив лише частковий фрагмент, зазвичай близько тридцяти-тридцяти п'яти відсотків від загального обсягу даних [26].

Механізм TrafficSliver існував у двох варіантах реалізації: TrafficSliver-Net та TrafficSliver-App. Перша версія функціонувала на рівні мережевого стеку, модифікуючи сам протокол передачі даних Tor. Друга працювала як проміжний шар між браузером та мережею Tor, що спрощувало впровадження без зміни базової інфраструктури. В цілому ідея обох варіантів залишалася однаковою: розподіл руйнує глобальну статистичну картину трафіку, яка є ключовою для WF-атак. Класифікатори зловмисника спираються на аналіз повної послідовності пакетів, включаючи характерний початок сеансу, burst-структури середини та специфічний кінець завантаження. Коли ця інформація розподілена між кількома вузлами без можливості кореляції, точність ідентифікації різко падає [26].

Однак TrafficSliver продемонстрував критичне обмеження: локальні статистичні ознаки у кожному фрагменті трафіку зберігалися без змін. Burst-піки, розподіли міжпакетних інтервалів (IAT) та послідовності розмірів пакетів залишалися доступними для DL-аналізу. Сучасні нейромережі виявилися здатними частково відновлювати інформацію про відвідуваний ресурс навіть з

неповних фрагментів. Саме ця проблема мотивувала розробку наступного покоління методів, що комбінували traffic splitting з додатковими механізмами обфускації локальних ознак.

Наступним кроком еволюції стала категорія adversarial perturbation – методів, що використовують принципи змагального машинного навчання для генерації спеціально розрахованих збурень. У 2021 році Сінлін Хоу та співавтори представили метод A2A (Attack-to-Attack), який реалізував концепцію ітеративної атаки на класифікатори зловмисника [27]. Система функціонувала як proxy між клієнтом Tor та вхідним вузлом мережі, перехоплюючи вихідний трафік у режимі реального часу. На відміну від простого додавання випадкового шуму, A2A використовував substitute model – власну копію передбачуваного класифікатора атакуючого, навчену на публічних датасетах WF-трафіку (простіше – імітація атакуючого).

Механізм роботи базувався на ітеративному процесі оптимізації. На кожному кроці алгоритм обчислював градієнт функції втрат класифікатора (що імітує атакуючого) відносно вхідної послідовності пакетів. Цей градієнт вказував, які саме зміни у трафіку найбільше вплинуть на рішення моделі. Система вставляла мінімальну кількість dummy-пакетів у найбільш чутливі позиції послідовності, змушуючи класифікатор помилятися у передбаченні. Процес повторювався ітеративно, поступово накопичуючи збурення, доки точність класифікації не знижувалася до заданого порогу або не вичерпувався бюджет додаткового трафіку [27].

Критичною перевагою A2A став надзвичайно низький overhead – лише 2,2% додаткового bandwidth, найкращий показник серед усіх відомих методів захисту на той момент. При цьому точність атак знижувалася до 57,5%, що робило деанонімізацію практично неефективною [9; 27]. Однак метод мав фундаментальну вразливість: якщо зловмисник знав про застосування A2A і перенавчав свою модель з урахуванням можливих збурень (adversarial training), ефективність захисту різко падала. Substitute model більше не відповідала реальній моделі атакуючого, що робило генеровані збурення неоптимальними.

Вирішенням проблеми adversarial training став метод ALERT (trace-Agnostic LightwEight Transferable Robust defense), представлений у 2024 році командою Цяо та співавторів [9; 28]. Ключова інновація полягала у відмові від аналізу реального трафіку користувача під час генерації збурень. Замість цього ALERT використовував GAN-архітектуру (Generative Adversarial Network), навчену офлайн на великих публічних датасетах Tor-трафіку. Генератор трафіку навчався створювати універсальні adversarial perturbations – шаблони dummy-пакетів, які ефективно обманювали широкий клас DL-класифікаторів незалежно від конкретного трафіку, до якого вони застосовувалися [28].

Принципова відмінність від A2A полягала у стратегії застосування: збурення генерувалися не у реальному часі (на основі поточного трафіку), а заздалегідь, ще до початку веб-сеансу. Система підготовлювала універсальний патерн dummy-пакетів, який потім накладався на будь-який вихідний потік користувача. Це усувало потребу у повному сліді трафіку для розрахунку об’єму, позицій та інших характеристик dummy-пакетів, що було найбільшим обмеженням попередніх змагальних методів. Додатковою інновацією стала стратегія random website imitation: для кожного користувача ALERT випадково обирав цільовий веб-сайт з великого пулу (зазвичай кілька тисяч ресурсів) і генерував збурення, що маскували справжній трафік саме під обраний ресурс [28].

Такий алгоритм роботи суттєво ускладнював тренування моделі зловмисника. Навіть якщо атакуючий намагався перенавчити свою модель з урахуванням захисту, він не міг передбачити, під який саме сайт буде замасковано трафік конкретного користувача. Кожна сесія отримувала унікальний випадковий шаблон обфускації. Експериментальні дані продемонстрували вражаючу ефективність: точність атаки Deep Fingerprinting знижувалася з 97,5% до лише 12,5% при bandwidth overhead близько 20% [28]. Це був значний прорив – метод забезпечував захист рівня найкращих регуляризаційних підходів, але з overhead у десяток разів меншим, ніж у BuFLO.

Найновішим представником категорії став RUDOLF (Reinforcement learning-based Universal Defense), представлений у 2024 році командою Цзян та співавторів [29]. Цей метод застосував принципово інший підхід до проблеми генерації збурень – замість заздалегідь навчених GAN-моделей або substitute classifiers використовувалася технологія reinforcement learning, по якій модель навчалась людиноподібно, по принципу «спроб та помилок». Система реалізовувала SAC-агента (Soft Actor-Critic), що навчався приймати рішення про вставку dummy-пакетів у режимі реального часу без потреби у буферизації повного трафіку.

Архітектура RUDOLF базувалася на концепції burst-level decision making. Замість аналізу кожного окремого пакета, агент працював з послідовностями burst – групами пакетів одного напрямку. Під час навчання RL-агент отримував зворотний зв'язок через dual-component reward function, що балансувала два протилежні критерії: максимізацію ймовірності помилки класифікатора (evasion success) та мінімізацію bandwidth overhead. Така архітектура дозволяла системі динамічно адаптуватися до конкретних характеристик трафіку, вибираючи оптимальну стратегію обфускації для кожного burst окремо [29].

Критичною перевагою reinforcement learning стала здатність до узагальнення: навчений на обмеженому наборі веб-сайтів, SAC-агент демонстрував ефективність і проти нових, раніше не бачених ресурсів. Це робило RUDOLF стійким до змін веб-контенту та появи нових сайтів без потреби у повторному навчанні моделі. Експериментальні результати показали точність атак на рівні 15%-20% при overhead <30% [9; 29]. Хоча це дещо поступалося показникам ALERT за ефективністю, RUDOLF не потребував офлайн-навчання GAN на великих датасетах і міг адаптуватися до нових типів атак через механізм online learning. Але застосовність методу у умовах реального використання в мережі Tor фактично є неможливою – це не дозволяє велика ресурсозатратність.

2.2 Висновок до розділу 2

Порівняльний аналіз методів захисту анонімності в мережі Tor демонструє критичну закономірність: жоден з окремих підходів не забезпечує достатнього захисту проти сучасних атак на основі глибокого навчання при збереженні прийнятної продуктивності мережі. Таблиця 2.1 містить детальне порівняння ключових методів за основними критеріями ефективності та практичної застосовності. У таблиці ефективність показує, на скільки відсотків середньо знижується точність Deep Fingerprinting атаки. «Часткова» ефективність означає, що метод руйнує глобальну структуру, але локальні ознаки залишаються.

Таблиця 2.1: Порівняння методів захисту від атак WF

Метод захисту	Ефект. проти DF (%)	Принцип роботи	Переваги	Недоліки
Camouflage (2011) [12]	51%	Паралельне завантаження «приманкових» сторінок для маскування справжнього трафіку користувача.	Простий механізм; не вимагає змін у Tor	Неприйнятно высокий overhead (>150%); вразливий до DL-аналізу
BuFLO (2012) [22]	50%	Жорстка регуляризація з фіксованими розмірами пакетів, інтервалами передачі та буферизацією всіх даних.	Теоретично ідеальний захист; ефективний проти всіх атак	Неможливий overhead >200%; критичні затримки; непрактичний для реального використання
WTF-PAD (2016) [24]	25.5%	Адаптивне заповнення dummy-клітинками на основі аналізу burst-послідовностей у режимі реального часу без буферизації.	Офіційно розгорнутий у Tor; нульова затримка; низький overhead (<60%)	Недостатня ефективність проти DL-атак; DF все ще досягає ~73% точності
Walkie-Talkie (2017) [23]	75%	Half-duplex комунікація з формуванням супер-послідовностей, де трафік цільового сайту маскується під decoy-сайт.	Висока ефективність; стійкий до DF; zero-latency	Вимагає попереднього знання про сайти; потребує модифікації браузера; неприйнятні затримки на практиці
FRONT (2020) [25]	54%	Рандомізована вставка dummy-пакетів на початку сеансу (найінформативнішої частини) з використанням розподілу Релея.	Обфускує найважливішу частину трафіку; zero-latency; помірний overhead (16-25%)	Залежить від точності timestamp; Слабкий проти таймінг-атак

Продовження таблиці 2.1

TrafficSliver (2020) [26]	Часткова	Розподіл TCP-потоку між кількома паралельними Guard-вузлами так, щоб кожен спостерігач отримував лише ~30-35% трафіку.	Руйнує глобальну структуру трафіку; низький overhead; без буферизації	Локальні ознаки (bursts, IAT) зберігаються у фрагментах; ДЛ-моделі можуть частково відновити інформацію
A2A (2021) [27]	~42.4%	Ітеративна вставка dummy-пакетів на основі gradient-атаки на substitute classifier, що імітує модель атакуючого.	Мінімальний overhead (2.2%); высока ефективність проти DL-атак	Вразливий до adversarial training; потребує proxy-архітектури; залежить від якості substitute model
ALERT (2024) [28]	84.37%	GAN-генератор створює універсальні adversarial perturbations офлайн, які застосовуються до будь-якого трафіку. Стратегія random website imitation маскує справжній сайт під випадково обраний.	Найвища ефективність (DF 97%→12.7%); стійкий до adversarial training; без потреби у повному сліді	Вимагає офлайн-навчання GAN на великих датасетах; залежність від substitute classifier; overhead ~20%
RUDOLF (2024) [29]	~76%	SAC-агент приймає рішення про вставку dummy-пакетів на burst-рівні у реальному часі. Dual-component reward балансує evasion success та bandwidth overhead.	Адаптивний до конкретних характеристик трафіку; узагальнюється на нові веб-сайти; без потреби у буферизації	Вимагає RL-навчання; більш складна архітектура; overhead~30%; менш ефективний за ALERT

Ранні підходи (Camouflage, BuFLO) ілюструють вибір між теоретичною стійкістю та практичністю: BuFLO забезпечує 50% зниження точності атак, але його 200%+ overhead робить його абсолютно непрактичним. Методи адаптивного наповнення (WTF-PAD, FRONT) досягли золотої середини з прийнятним overhead (<60%) та нульовою затримкою, що забезпечило впровадження WTF-PAD в офіційну архітектуру Tor [2], однак їхня ефективність проти DL-атак залишається недостатньою (25-54%). Traffic splitting (TrafficSliver) пропонує низький overhead, але локальні ознаки у фрагментах залишаються вразливими.

Принципово новий напрямок представляють методи змагального машинного навчання. A2A демонструє мінімальний overhead (2.2%), однак вразливий до adversarial training. ALERT досягає найбільшої ефективності (84.37%),

знижуючи точність DF до 12.68%, при прийнятному overhead ~20%, проте вимагає складного GAN-навчання на великих датасетах. RUDOLF пропонує адаптивність та узагальнення на нові сайти, але його ефективність дещо поступається ALERT.

Висновок порівняльного аналізу підтверджує, що жоден метод окремо не вирішує проблему: ранні методи вибирають між ефективністю та практичністю; адаптивне padding зберігає практичність але втрачає ефективність (через надвисокий overhead); traffic splitting руйнує глобальну, але не локальну структуру (не ефективно, якщо DL модель здатна класифікувати сайт на ранніх етапах); adversarial методи досягають найбільшої ефективності, але залишаються складними у реалізації. Саме це обґрунтовує необхідність розробки гібридних підходів, що комбінують переваги розподілу трафіку (низький overhead, реальний час) з adversarial perturbation (висока ефективність), як це демонструється у пропонованому методі дипломної роботи.

РОЗДІЛ 3 ПЛАНУВАННЯ АРХІТЕКТУРИ МОДИФІКОВАНОГО МЕТОДУ ПІДВИЩЕННЯ АНОНІМНОСТІ В МЕРЕЖІ TOR

3.1 Постановка задачі та концепція взаємодоповнюваності методів

Проведений у розділах 1 та 2 аналіз існуючих методів WF-атак та методів захисту дав чітке розуміння існуючої проблеми – жоден метод сам по собі не може одночасно ефективно протистояти атакам на основі глибокого навчання і при цьому залишатись в прийнятних рамках продуктивності для зручного користування, наприклад: методи адаптивного наповнення (WTF-PAD, FRONT) зберігають низький overhead та нульову затримку, але залишаються вразливими до DL-атак, знижуючи точність Deep Fingerprinting лише до 72%; методи регуляризації (BuFLO, CS-BuFLO) досягають теоретично сильного захисту, але їхній bandwidth overhead понад 100% робить їх непридатними у практичному використанні; методи розподілу трафіку (TrafficSliver) ефективно руйнують глобальну структуру послідовності пакетів, однак дрібні, локальні статистичні ознаки у фрагментах залишаються доступними для класифікаторів. Змагальні методи (A2A, ALERT, RUDOLF) демонструють найвищу ефективність, знижуючи точність атак до 12-20%, проте більшість реалізацій вимагають або офлайн-навчання складних моделей, або доступу до повного сліду трафіку для генерації оптимальних збурень, що робить їх ефективними але складними у реалізації на практиці.

Метою дипломної роботи є розробка модифікованого методу підвищення рівня анонімності у мережі Tor, що нівелює основні прогалини у роботі вже існуючих методів шляхом взаємодоповнюваності. Задача полягає у пошуку ефективного синергічного поєднання методів, при якому основні слабкості кожного окремого методу будуть компенсуватись сильнішими сторонами іншого. Така стратегія принципово відрізняється від модифікації конкретного методу поодиноко, оскільки дозволяє досягти якісно іншого рівня захисту через описану взаємодію компонентів.

При розробці важливою частиною є постановка критеріїв, по яким буде виповнюватись оцінка ефективності методу. Процес оцінки ефективності буде за такими кількісними та якісними критеріями:

Кількісні критерії:

1. Метод повинен знижувати точність сучасних DL-атак (Deep Fingerprinting, Var-CNN, Holmes) до ≤ 30%.
2. Накладні витрати пропускної здатності мають бути ≤ 30%, оскільки більший overhead суттєво погіршує користувацький досвід у мережі Tor, де швидкість з'єднання вже обмежена багаторівневим шифруванням.
3. Додаткова затримка від захисних механізмів має залишатись у межах 500-1000 мілісекунд, щоб бути непомітною на фоні природної варіативності Tor-з'єднань, зберігаючи ефективність.

Якісні критерії:

1. Метод повинен функціонувати у режимі реального часу без потреби у буферизації повного сеансу (або буферизацією без видимої для користувача затримки), що забезпечує застосовність до потокового трафіку.
2. Реалізація має бути можливою на споживацькому обладнанні без вимог до спеціалізованих апаратних прискорювачів, що критично для широкого впровадження.

Як вже було оголошено раніше, основою запропонованого підходу буде принцип взаємодоповнюваності методів захисту. Розглянуті методи захисту діють на різних рівнях абстракції представлення трафіку, фокусуючись на порушеннях різних ознак трафіку: глобальний рівень (макроструктура повної послідовності, загальний обсяг, тривалість сеансу) та локальний рівень (мікроструктура burst-послідовностей, розподіли міжпакетних інтервалів, специфічні патерни розмірів пакетів). Але жоден метод не розрахований на всі ознаки трафіку. Таким чином для кожного методу з певним класом слабкостей існує інший метод, чий механізм роботи природньо усуває ці слабкості, і навпаки. В свою чергу атаки WF використовують ознаки обох рівнів одночасно,

тому захист лише одного рівня залишає інший доступним для класифікатора. Розглянути як кожен з методів здатен компенсувати один одного можна у таблиці 3.1.

Таблиця 3.1 Демонстрація виявлених пар комплементарних методів та механізми їхньої взаємної компенсації.

Метод	Фундаментальна слабкість	Компенсуючий метод	Механізм компенсації
Traffic Splitting	Локальні статистичні ознаки (burst-структури, Inter-Arrival Time розподіли) зберігаються у кожному фрагменті	Adversarial Perturbation	Спотворює локальні ознаки через вставку dummy-пакетів, роблячи фрагменти складнішими до розпізнавання
Adversarial Perturbation	Потреба у повному сліді трафіку для обчислення оптимальних збурень; буферизація унеможливлює real-time роботу	Traffic Splitting	Створює малі фрагменти (~33% від повного сеансу), достатньо короткі для обробки у режимі реального часу
Adaptive Padding	Детерміністичні схеми заповнення розпізнаються DL-моделями; залишається вразливим до сучасних атак	Adversarial Perturbation	Генерує недетерміністичні збурення, спеціально розраховані для введення класифікатора в оману
Regularization	Екстремально високий overhead (>200%) через повну уніфікацію всіх характеристик трафіку	Traffic Splitting + Adversarial	Селективна модифікація лише критичних ознак замість тотальної регуляризації знижує overhead до прийнятного рівня

Аналіз матриці порівняння (табл. 3.1) дає можливість сформувати головну гіпотезу дослідження: поєднання розподілу трафіку та змагального шуму створює синергію. Разом ці методи захищають користувача набагато ефективніше, ніж просто за сумою їхніх окремих можливостей. Якщо метод A знижує точність атаки до величини p_A, а метод B – до величини p_B, то за умови незалежності рівнів ознак, що атакуються цими методами, комбінований метод C досягає точності $p_C \leq p_A \times p_B$. Подібна ймовірнісна модель можлива тому, що класифікатор зловмисника втрачає доступ до ознак відразу двох рівнів абстракції, наприклад: traffic splitting руйнує глобальну структуру через фрагментацію, а adversarial perturbation спотворює локальні статистичні патерни у кожному фрагменті сформованого трафіку. DL-модель не може компенсувати втрату інформації на одному рівні через аналіз іншого, оскільки обидва рівні пошкоджені одночасно і незалежно один від одного.

Практична реалізація методу вимагає вирішення певних задач. По-перше, важливо визначити оптимальну роботу частини розподілу трафіку, адже від цього буде залежати загальне навантаження на кожен з вузлів: велика кількість сформованих шляхів N зменшить overhead та точність класифікатора атакуючого (при розділенні трафіку з більшою частотою будуть руйнуватись і локальні ознаки), але ускладнить передачу інформації та збільшить ризик втрати інформації, а замала кількість N зменшить ефективність, збільшивши overhead та матиме менші негативні наслідки на результат WF-атаки. По-друге, треба розробити або модифікувати механізм генерації adversarial perturbations, який може функціонувати на фрагментах обмеженого розміру без доступу до повного сеансу, зберігаючи при цьому достатню ефективність проти DL-класифікаторів. По-третє, потрібна координація між модулями розділення трафіку та змагального шуму, що забезпечує коректну роботу системи у режимі реального часу з прийнятними затримками для кінцевого користувача. Рішення до представлених задач та прийняття конкретних тенічних рішень буде розглянуто у наступному розділі роботи.

3.2 Планування архітектури та обґрунтування варіантів гібридного методу

Загальний образ методу було окреслено у підрозділі 3.1. За кістяк системи буде взято метод TrafficSliver, який дозволить позбутись глобальних ознак, але відкритим залишається питання локальних ознак – яким чином реалізовувати змагальний шум. Було розглянуто три альтернативних варіанти, що відрізняються за своїми підходами та складністю реалізації.

Варіант А – ітеративне змагальне збурення

В основі першого варіанту лежить A2A-подібний метод. Для роботи задіюється substitute model – нейромережа, навчена на публічному датасеті до початку сесії, що буде виступати в ролі тренера. Після поділу у модулі TrafficSliver, фрагментований трафік проходить через модель змагальних збурень типу A2A, яка генерує необхідний шум. Схематичне зображення

алгоритму на рис.3.1. Вставлені пакети маркуються спеціальним внутрішнім прапорцем. Ця мітка дозволяє кінцевому вузлу мережі безболісно відкинути їх перед виходом корисних даних у відкритий інтернет.

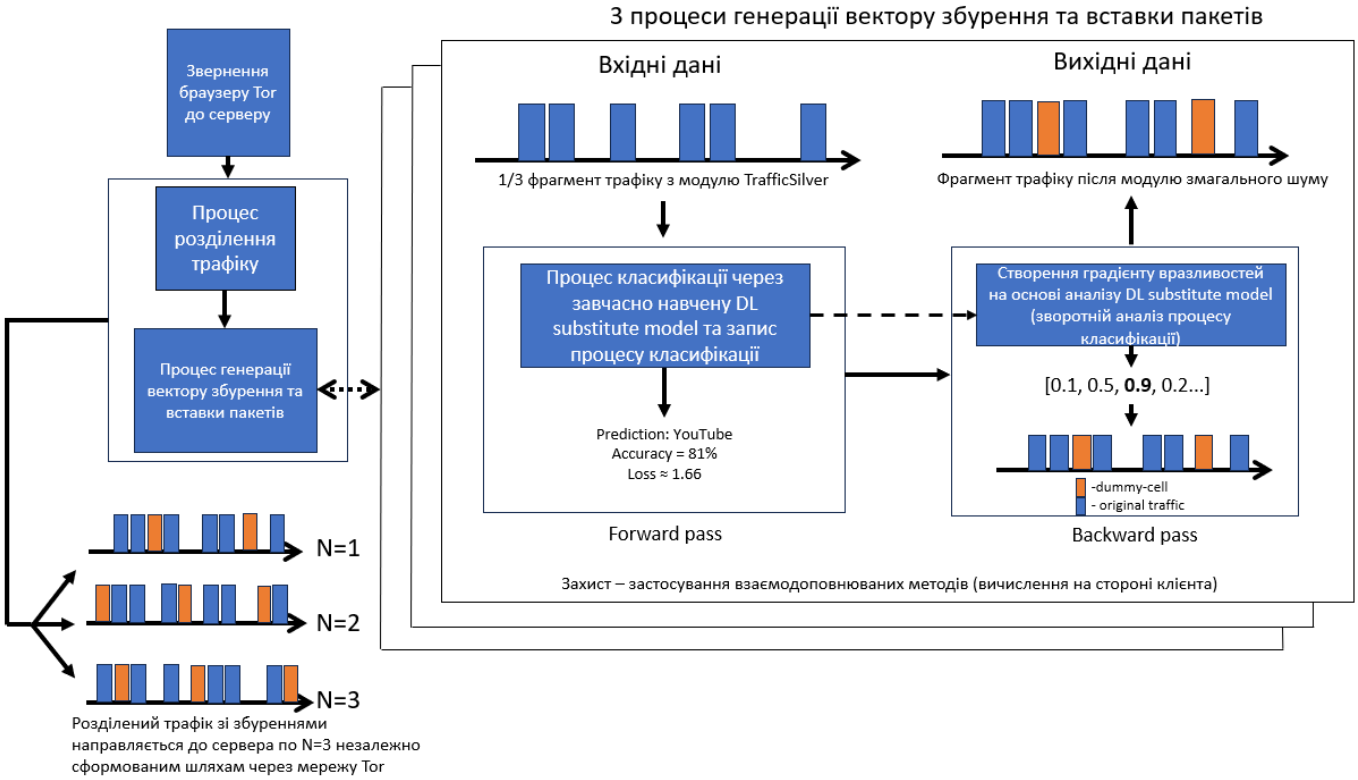

Рис. 3.1 Схематичне зображення роботи модуля змагального збурення

Важливим є те, що вектор для кожної частини траси є унікальним, і генерується в реальному часі, таким чином надійність методу при змагальному перетренуванні залишається незмінною. Підхід добре описаний у літературі, а substitute model тренується на локальному обладнанні за кілька годин. Але подібний підхід вимагає значних обчислювальних ресурсів під час активної сесії.

Варіант B – фрагментно-оптимізовані універсальні збурення

Другий варіант переносить усі важкі обчислення в офлайн і залишає клієнту лише тривіальну операцію накладення готового шаблону. Механізм полягає у створенні набору з п'яти–десяти універсальних векторів збурень δ, кожен з яких обманює класифікатор на широкому класі вхідних фрагментів, а не лише на одній конкретній трасі.

Метод концептуально базується на підході універсальних змагальних збурень (Universal Adversarial Perturbations, UAP), запропонованому у роботі Moosavi-Dezfooli та ін. [30]. Цей метод має змінений механізм роботи. Стандартні методи шукають збурення, ефективні проти класифікатора, що бачить повний трафік. Пропонований метод змінює область пошуку ознак, тут substitute model навчається не на повному датасеті, а його частині, симулюючи саме спостерігача, що бачить лише один фрагмент трафіку.

Навчання та створення шаблонів відбувається у хмарі задля збереження ресурсів користувача, адже процес потребує великої кількості відеопам'яті. Публічний датасет ділиться на N кількість частин, що повторює роботу модуля TrafficSliver. Фрагменти потрапляють у forward та backward pass, де тренується substitute model і генеруються збурення. Після цього з клієнтської сторони відбувається розділення трафіку на N=3 частини, до яких потім застосовується вектор збурення (у паралельному або послідовному режимі обчислення). Схематично цей варіант реалізації зображено на рисунку 3.2.

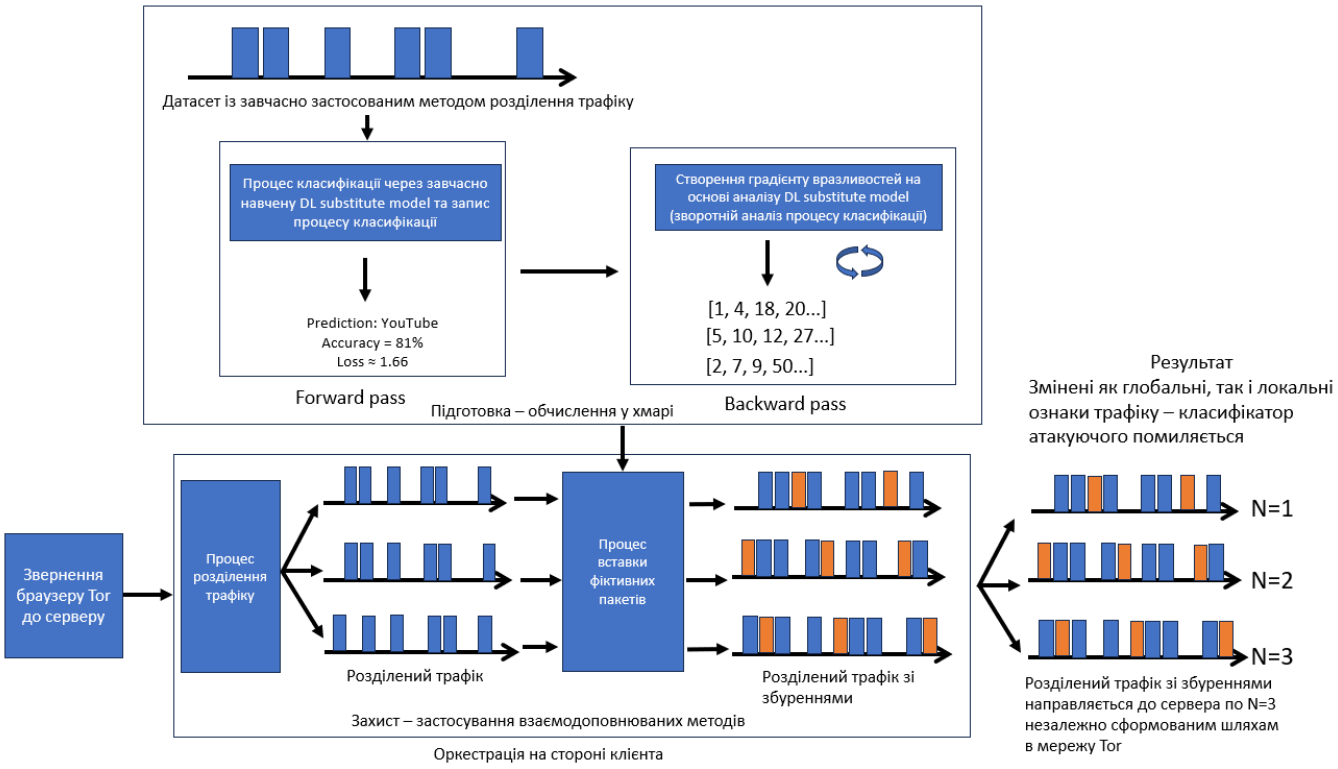

Рис. 3.2 Схематичне зображення реалізації варіанту B

Або порядок дій буде змінено: тренування substitute model на цілих, не поділених публічних датасетах у хмарі з генерацією векторів збурення, і вже

робота з готовими даними на стороні клієнта – процес збурення з подальшим розділенням трафіку, як зображено на рисунку 3.3.

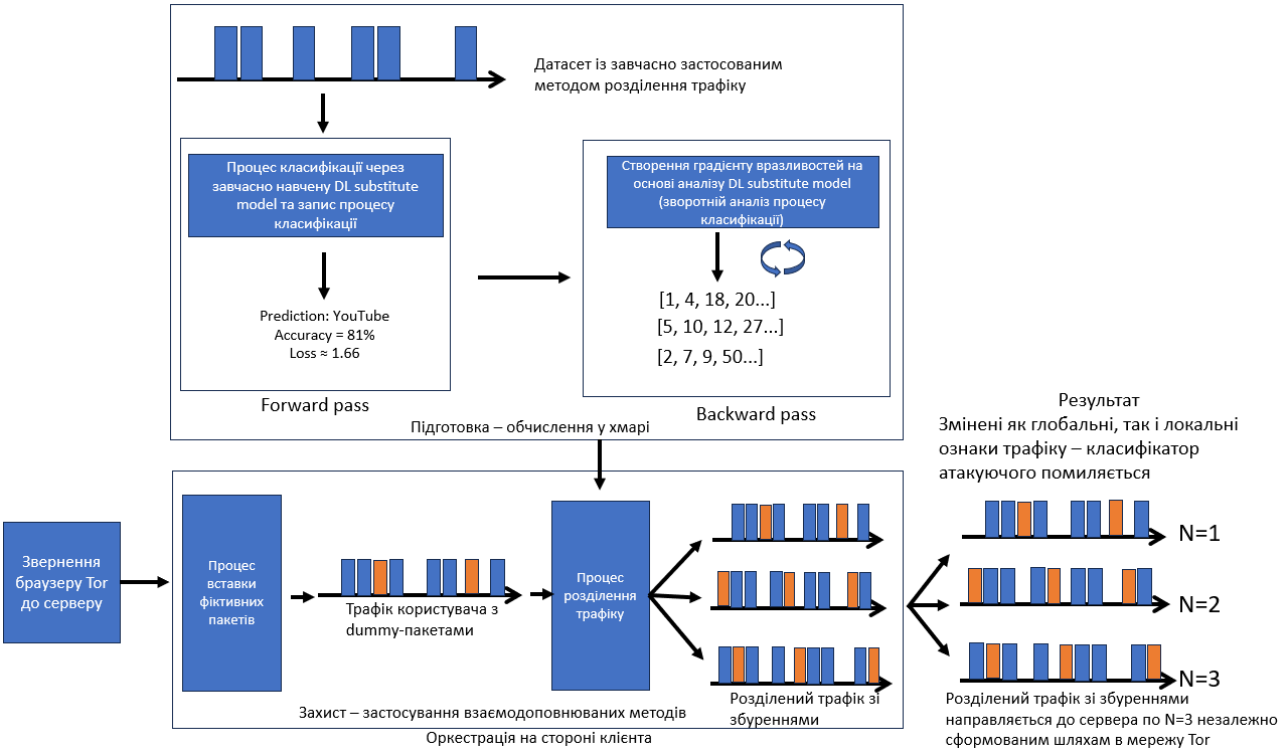

Рис. 3.3. Схематичне зображення реалізації варіанту В

Один з варіантів буде обрано на основі оцінки ефективності у наступних підрозділах.

Таким чином система складається з трьох основних модулів. UAP Application Module зберігає набір з 5-10 pre-computed універсальних шаблонів GGI і на початку кожної сесії випадково обирає один з них. Всі клітинки, що проходять через модуль, доповнюються фіктивними (dummy) пакетами згідно з обраним вектором збурень. Модуль розділювач приймає вихідний потік модифікованого трафіку і розподіляє Tor-клітинки між трьома паралельними ланцюжками (N=3) за алгоритмом зваженого round-robin (лабораторні умови) або механізмом TrafficSliver (реалістичні умови). Оркестратор забезпечує синхронізацію між модулями та підтримує мінімальний burst-буфер. Жодного з вузлів Tor модифікувати не потрібно – вся логіка зосереджена на клієнтській стороні.

Додатково буде розглянуто можливість модифікації самого методу універсальних збурень, окремо від Traffic splitting методу.

Бюджет збурень та його зв'язок з мережевим overhead

Будь-який метод вставки dummy-пакетів визначається єдиним ключовим параметром – бюджетом збурень B, що задає максимальну кількість фіктивних клітинок, які дозволено додати до оригінальної траси довжиною T. Зв'язок між бюджетом і мережевим overhead є лінійним:

$$overhead = \frac{B}{T}$$

де T = 5000 клітинок. Для цільового B = 500 overhead = 10%, що втричі нижче допустимого ліміту <= 30% (критерій 3.1). Бюджет B є єдиним параметром, що контролює цей баланс: збільшення B посилює захист, але збільшує overhead. Таблиця 3.2 демонструє цей компроміс для типових значень.

Таблиця 3.2 – Залежність бюджету збурень та bandwidth overhead (T = 5000)

Бюджет B	**Overhead**	**Оцінка практичності**
100	2%	Незначний вплив на трафік; ймовірно недостатньо для значного захисту
500	10%	Цільовий бюджет роботи: баланс захисту та продуктивності
1000	20%	Підвищена ефективність; помітне сповільнення для користувача Tor
1500	30%	Максимально допустимий за критерієм; граничне навантаження на Tor

Для обох варіантів реалізації (A та B) бюджет B = 500 є фіксованим, і метод вставки не впливає на overhead – виключно на розміщення та ефективність збурень. Обидва варіанти таким чином задовольняють критерій overhead <= 30%.

Порівняльна оцінка варіантів та обґрунтування вибору напряму реалізації

Варіанти A та B порівнюються за п'ятьма критеріями, що безпосередньо відповідають вимогам підрозділу 3.1 (табл. 3.3). Overhead у обох варіантів однаковий (10%), тому порівняння зосереджується на продуктивності та стійкості.

Таблиця 3.3 – Порівняльна оцінка варіантів реалізації модуля збурень

Критерій	**Варіант A (ітеративне AT)**	**Варіант B (офлайн UAP)**
Overhead (B=500)	10%	10%
Латентність на клієнті	Висока: backward pass per burst (потенційно >1000 мс)	<5 мс: накладання шаблону
GPU на стороні клієнта	Необхідний (>>512 МБ VRAM)	Не потрібен
Пам'ять клієнта	~500 МБ (substitute model)	~кілька КБ (шаблони)

Продовження таблиці 3.3

Стійкість до AT	Висока: унікальне delta per trace	Середня: кінцевий набір шаблонів
Придатність для Tor	Ні: неприйнятні затримки	Так: < 1000 мс

Варіант A відкидається через критичну несумісність з вимогами Tor: backward pass через substitute model для кожного burst-у і паралельно для трьох ланцюгів утворює затримки у тисячі мілісекунд – неприйнятно при вже існуючих, природних затримках Tor 300–1000 мс. Варіант A залишається теоретичним еталоном порівняння (верхня межа ефективності при необмежених ресурсах). Варіант B задовольняє всі вимоги і приймається як основний напрям реалізації.

Окремо слід обґрунтувати порядок застосування компонентів. Було прийнято рішення, що в архітектурі методу збурення застосовується до повної траси і лише потім вона розподіляється між ланцюгами, а не навпаки. Зворотний порядок (розділення -> збурення) вимагав би витрачати бюджет B окремо на кожен із N фрагментів, що при N = 3 та B = 500 потроює сумарний overhead до 30% – одразу на межі допустимого ліміту, при іншому розподілі пакетів, з дотриманням умов поставлених раніше, частота dummy вставок буде набагато меншою, що відобразиться на ефективності методу. Порядок збурення-розділення є більш оптимальним, для зберігання overhead = 10% та коректної калібрацію пулу.

Варіант B у базовій формі спирається на статичний набір шаблонів збурень. Теоретично це означає, що адаптивний атакуючий, накопичивши достатній обсяг захищеного трафіку, може ідентифікувати шаблони та адаптуватись. Перспективним напрямком усунення цього обмеження є рандомізація вибору позицій у межах ширшого попередньо побудованого пулу. Конкретна реалізація, параметри та оцінка ефективності цієї ідеї підлягають систематичному дослідженню у четвертому розділі, де метод коригується емпірично-ітеративно за результатами експериментів.

3.3 Математичне обґрунтування та попередня оцінка ефективності методу

Цей підрозділ присвячений математичному опису того, як саме працює запропонований метод і чому він теоретично має бути ефективним. Усі кількісні оцінки нижче є теоретичними: вони ґрунтуються на математичному аналізі і є попередніми гіпотезами, які перевіряються у четвертому розділі за результатами реальних експериментів. Параметри методу (бюджет збурень, розмір пулу, кількість ланцюгів) можуть уточнюватись у ході досліджень.

Формальна постановка задачі захисту з бюджетним обмеженням

Перш ніж описувати математику методу, необхідно формально визначити, що саме є вхідними даними, з чим працює система і чого вона намагається досягти. Це дозволяє точно і однозначно говорити про ефективність та обмеження методу.

Трафікова траса. Tor-з'єднання у нашому датасеті представлене у вигляді послідовності клітинок фіксованого розміру. Кожна клітинка є або вихідним пакетом клієнта, або вхідним пакетом від сервера. Математично трасу записують так:

$$x = (x_1, x_2, ..., x_T) \quad (3.1)$$

де x – траса трафіку (послідовність клітинок одного з'єднання);

x_i – значення і-ї клітинки: +1 (вихідний пакет від клієнта), -1 (вхідний пакет від сервера), 0 (порожня позиція);

T – загальна довжина траси (у датасеті CW прийнято $T = 5000$)

Простими словами: трасу можна уявити як рядок із п'яти тисяч символів, де "+" – "трафік клієнт-сервер", "-" – "трафік сервер-клієнт", "0" – тиша, значення елементу – часовий проміжок між останньою не пустою позицією. Саме цей рядок аналізує класифікатор атакуючого.

Класифікатор атакуючого – це глибока нейронна мережа (наприклад, DF, VarCNN або Holmes), яка навчена на трасах відомих веб-сайтів і намагається за трасою нового з'єднання визначити, який сайт відвідав користувач. Позначатимемо його:

$$f_\theta : R^T -> R^C \quad (3.2)$$

де f_θ – класифікатор атакуючого з параметрами θ;

R^T – простір вхідних трас (послідовності довжиною T);

R^C – простір вихідних оцінок (C = 95 класів-сайтів у датасеті CW);

θ – внутрішні параметри (ваги) нейронної мережі.

Функція втрат $L(f_\theta (x), y)$ показує наскільки класифікатор "помиляється" на трасі x, якщо правильною відповіддю є клас y. Чим більше L, тим більше класифікатор не впевнений у правильному класі. Мета захисту – змусити L зростати.

Задача захисту з бюджетним обмеженням. Необхідно знайти вектор збурення Δ, такий що:

$$(1)\ f_\theta(x + \Delta) \neq y$$

$$(2)\ |\Delta|_0 \leq B$$

$$(3)\ \Delta_i \geq 0 \text{ для всіх i = 1,...,T}$$

$$(4)\ overhead = \frac{B}{T} \leq overhead\_\max = 0.3 = 30\% \quad (3.3)$$

де Δ – вектор збурення (показує, куди і скільки додати dummy);

$|\Delta|_0$ – кількість ненульових елементів у delta, тобто кількість доданих dummy-пакетів;

B – бюджет збурень: максимально дозволена кількість dummy;

overhead – відносне збільшення обсягу трафіку;

$overhead_{max} = 0.30$ – цільове обмеження (30%) згідно критеріїв підрозділу 3.1.

Головною умовою (1) є помилка класифікатора. Умова (2) – обмеження ресурсів: не можна додати більше ніж B пакетів. Умова (3) – технічне обмеження клієнта: він може лише додавати власні вихідні клітинки, але не видаляти або підробляти пакети сервера. Умова (4) – зв'язок із мережевим навантаженням: кожен доданий dummy збільшує трафік. При обраному бюджеті B = 500 та T = 5000:

$$overhead = \frac{500}{5000} = 0.10 = 10\% \quad (3.4)$$

Це втричі менше допустимого ліміту 30%, що залишає запас для можливого коригування бюджету у ході експериментів четвертого розділу.

Градієнтно-керована ін'єкція: математичний механізм

Центральне питання: куди саме у трасу x вставляти dummy-клітинки, щоб класифікатор помилявся якнайчастіше? Інтуїтивно – туди, де класифікатор є найбільш "чутливим": де навіть невелика зміна трафіку найсильніше впливає на його рішення. Градієнт функції втрат по входу – це саме такий індикатор чутливості.

Карта чутливості градієнтно-керованої ін'єкції

Для кожної позиції i в трасі обчислюється, наскільки сильно зміна значення x_i впливає на функцію втрат у середньому по всіх трасах навчальної вибірки:

$$S[i] = \left(\frac{1}{|D_{train}|}\right) * \sum_{\{(x,y)in\ D_{train}\}} \left|\frac{dL(f_\theta(x),y)}{dx_i}\right| \quad (3.5)$$

де S[i] – чутливість позиції i (невід'ємне число: чим більше, тим важливіша ця позиція для класифікатора);

$|D_{train}|$ – кількість трас у навчальній вибірці;

$\frac{dL}{dx_i}$ – часткова похідна (градієнт) функції втрат по значенню x_i: показує, як зміна x_i впливає на L;

|...| – модуль (беремо абсолютне значення, бо нас цікавить сила впливу);

SUM() – сума по всіх трасах навчального датасету D_{train}.

Так ми перевірямо впевненість моделі щодо остаточного рішення в залежності від позицій комірок у трафіку. Чим більше S[i] – тим менше впевненості у класифікації дає модель, і тим вигідніше вставити dummy саме туди у позицію на якій модель почала змінювати результат класифікації.

Пул градієнтно-чутливих позицій

З усіх T = 5000 позицій відбираємо p найчутливіших, формуючи пул P – множину позицій для можливої вставки dummy:

$$P = \{\, i : S[i] \text{входить до } top - p \text{ значень } S \,\} \quad (3.6)$$

де P – пул позицій: множина з p найчутливіших позицій траси;

p – розмір пулу (параметр, що визначається експериментально; орієнтовне значення: p = 1500, тобто 30% від T);

"top-p" – p позицій з найбільшими значеннями S[i].

Важливо: пул будується один раз офлайн і зберігається як файл. Це одноразова обчислювально важка операція (потребує GPU і кількох годин). Після цього клієнт використовує готовий пул без жодних важких обчислень у реальному часі – проста математична операція вставки до масиву.

Оцінка ефективності вставки (лінійне наближення). Після вставки dummy у множину позицій $Z \subseteq P$ змінюється функція втрат класифікатора. Через розклад Тейлора першого порядку (лінійне наближення поблизу точки x):

$$DL(Z) = L(f_\theta(x + \Delta_Z), y) - L(f_\theta(x), y)$$
$$\approx \sum_{\{i\ in\ Z\}} \left(\frac{dL}{dx_i}\right) * \Delta_i$$
$$= \sum_{\{i\ in\ Z\}} S[i] \quad (3.7)$$

де DL(Z) – зміна функції втрат після вставки dummy у позиції Z;

Δ_Z – вектор збурення: $\Delta_i = +1$ якщо $i \in Z$, 0 інакше (додаємо вихідний dummy-пакет);

"≈" – наближена рівність (лінійне наближення Тейлора);

$\sum_{\{i\ in\ Z\}} S[i]$ – сума чутливостей по обраних позиціях.

Оскільки S[i] > 0 для всіх $i \in P$ (ми відібрали лише позиції з ненульовою чутливістю), то DL(Z) > 0 – функція втрат зростає. Це означає: класифікатор стає менш впевненим у правильному класі.

Можна встановити нижню і верхню межі для DL(Z):

$$B * min_{\{i\ in\ Z\}} S[i] \leq DL(Z) \leq B * max_{\{i\ in\ Z\}} S[i] \quad (3.8)$$

де B = |Z| – кількість вставлених dummy (бюджет збурень);

min S[i] – найменша чутливість серед обраних позицій;

max S[i] – найбільша чутливість серед обраних позицій.

Нижня межа гарантує: навіть якщо ми обрали не найбільш оптимальні позиції з пулу P, кожен dummy все одно робить ненульовий внесок у збурення

класифікатора. При B = 500 dummy це накопичується до значень, достатніх для переведення класифікатора у режим помилки. Точний відсоток правильно збурених трас визначається архітектурою конкретного класифікатора і встановлюється у четвертому розділі.

Ймовірнісна модель рандомізованого вибору позицій

У базовому варіанті метод використовує фіксований набір позицій для вставки dummy – однакових для всіх трас. Це ефективно проти класифікаторів, які не знають про захист, але потенційно вразливе до атакуючих, що можуть перенавчитись. Далі наведено теоретичне обґрунтування того, чому рандомізація вибору позицій у межах пулу P могла б суттєво посилити стійкість методу. Це є теоретичним напрямом, що підлягає перевірці у четвертому розділі.

Припустимо, що замість фіксованого набору для кожної траси і незалежно і рівномірно випадково обирається підмножина Z_i розміром k з пулу P:

$$Z_i \sim U(\{ A \subseteq P : |A| = k \}) \quad (3.9)$$

де Z_i – випадково обрана підмножина позицій для і-ї траси;

"∼" – позначає "має розподіл";

U – Unofrom, рівномірний розподіл (кожна підмножина рівноймовірна);

{ A ⊆ P : |A| = k } – множина всіх підмножин пулу P розміром k;

k – розмір вибірки (кількість вставлених dummy; k = B = 500).

Варіантів вставки з пулу існує багато – це кількість способів обрати k позицій з пулу розміром p:

$$|\omega| = C(p,k) = \frac{p!}{k! * (p-k)!} \quad (3.10)$$

де |ω| – кількість можливих різних реалізацій збурення;

C(p, k) – біноміальний коефіцієнт ("p вибрати k");

p – розмір пулу;

k – розмір вибірки (бюджет dummy);

"!" – факторіал числа (наприклад, 5! = 5*4*3*2*1 = 120).

При орієнтовних значеннях p = 1500 та k = 500 (пул втричі ширший за бюджет):

$$C(1500, 500) \approx 10^{370} \quad (3.11)$$

Для порівняння: кількість атомів у всьому спостережуваному Всесвіті становить приблизно 10^80. Тобто простір можливих реалізацій збурення більший за цю кількість у сотні порядків. Навіть якби атакуючий зібрав мільярд (10^9) захищених трас, він охопив би лише частку 10^9 / 10^370 = 10^(-361) від загального простору – фактично нуль. Але, як було сказано вище, це гіпотеза – обґрунтоване припущення, і точні дані будуть визначені у наступному розділі експериментальним шляхом

Ймовірність того, що дві незалежно захищені траси i та j мають однаковий набір позицій dummy:

$$P(Z_i = Z_j) = \frac{1}{C(p,k)} \approx \frac{1}{10^{370}} \approx 0 \quad (3.12)$$

де $P(Z_i = Z_j)$ – ймовірність збігу двох незалежних реалізацій;

$1/C(p, k)$ – обернена кількість можливих підмножин.

Практичне значення цього результату: послідовний класифікатор (такий як DF), що шукає стабільний просторовий патерн у трасі, не знайде жодного повторюваного патерну – бо кожна траса містить унікальний набір позицій dummy. Це підтверджується формально: оскільки Z_i обирається незалежно від класу сайту y, взаємна інформація між патерном вставки і класом дорівнює нулю:

$$I(Z_i; Y) = 0 \quad (3.13)$$

де $I(Z_i; Y)$ – взаємна інформація між набором позицій dummy Z_i та класом сайту Y;

0 – точна рівність (не наближена), оскільки Z_i обирається незалежно від x та y за означенням.

Отже знаючи лише те, де стоять dummy (Z_i), неможливо дізнатись, який сайт відвідав користувач. Тому послідовний атакуючий, що спирається на позиційний патерн, не отримує жодної корисної інформації для перенавчання своєї моделі.

Фундаментальна межа для агрегатних атакуючих. Ситуація кардинально інша для класифікаторів, що аналізують не позиції пакетів, а їхню кількість у

часових вікнах (наприклад, Holmes через TAF-ознаки). Визначимо TAF-ознаку для часового вікна w:

$$n_w(x) \;=\; |\{\, i : \; t_i \text{ в } w \text{ та } \; x_i \;=\; +1 \,\}| \quad (3.14)$$

де $n_w(x)$ – кількість вихідних пакетів у вікні w для траси x;

t_i – час відправки i-ї клітинки;

"в w" – належить часовому вікну w;

$x_i = +1$ – умова: рахуємо лише вихідні пакети клієнта;

|...| – кількість елементів у множині.

Після вставки k dummy-пакетів ця ознака змінюється на:

$$\Delta_{n_w} \;=\; |Z\_i \;\cap\; \{\, j : \; t_j \text{ в } w \,\}| \quad (3.15)$$

де Δ_{n_w} – зміна кількості вихідних пакетів у вікні w;

$Z_i \cap \{...\}$ – перетин: позиції dummy, що потрапили у вікно w;

"∩" – знак перетину множин.

Очікуване значення цієї зміни (при рівномірному розподілі Z_i по пулу P):

$$E\left[\Delta_{n_w}\right] = \; k \; * \frac{|P \cap w_{positions}|}{|P|} \quad (3.16)$$

де E[...] – математичне сподівання (середнє значення);

|P ∩ w_positions| – кількість позицій пулу P, що знаходяться у часовому вікні w;

|P| – загальний розмір пулу.

Якщо пул P рівномірно покривав би всю трасу, $E[\Delta_{n_w}] = k/W$ було б однаковим для всіх вікон w – і не залежало б від класу сайту. Тоді захист був би ефективним. Але на практиці пул P концентрований у першій частині трафіку (де S[i] найбільші). Тому Δ_{n_w} для різних вікон різне, і через особливості трафіку конкретних сайтів ця різниця корелює з класом. Взаємна інформація не зникає:

$$I(\, n_w(x');\; Y\,) \;>\; 0 \quad \text{для частини вікон w} \quad (3.17)$$

де $n_w(x')$ – TAF-ознака захищеної траси x';

Y – клас сайту;

I(...; ...) > 0 – ознака зберігає інформацію про клас.

Це і є фундаментальним обмеженням методу: рандомізація позицій не усуває класовий сигнал у TAF-просторі. Агрегатний атакуючий (Holmes) дивиться не на те, "де" стоять пакети, а на те, "скільки" їх у кожному вікні – і ця кількість залишається передбачуваною навіть після вставки dummy. Подолання цього обмеження є окремим предметом дослідження у четвертому розділі.

Математична формалізація Traffic Split

Компонент розподілу трафіку реалізовано на основі методу TrafficSliver. Його механізм полягає у відкритті трьох паралельних незалежних з'єднань з Guard-вузлами, і розподіленням між ними захищеної траси x' випадковими пачками (Bandwidth-Weighted Random, BWR). На відміну від строго почергового розподілу, BWR передає кожному ланцюгу цілісні групи пакетів довільного розміру, що є реалістичнішою моделлю поведінки мережевого трафіку. У результаті кожен Guard-вузол бачить приблизно T/N ≈ 5000/3 ≈ 1667 клітинок – третину трафіку, без доступу до повної структури траси.

Для математичного опису та розрахунків у даній роботі використовується спрощена модель – round-robin розподіл, за якого клітинки призначаються ланцюгам суворо по черзі. Це спрощення обрано через значно меншу обчислювальну складність і прозорість формалізації, при цьому ключові метрики ефективності залишаються однаковими для обох варіантів розподілу. У round-robin фрагмент, що спостерігає Guard-вузол ланцюга c:

$$x'_c = \{ x'_i : (i-1)\ mod\ N = c-1 \},\quad c = 1,\ldots,N \qquad (3.18)$$

де x'_c – фрагмент захищеної траси, що бачить Guard c;

x'_i – i-та клітинка захищеної траси x';

mod – залишок від ділення;

N = 3 – кількість ланцюгів.

Для оцінки ефективності захисту використовуються дві метрики. Перша – середня точність одного Guard-вузла circ_acc, що є часткою фрагментів, на яких класифікатор дав правильну відповідь. Рахується як звичайна точність (accuracy), але одиницею виміру є фрагмент одного ланцюга, а не повна траса.

Друга – samp_fool, що є часткою трас, де всі N Guard-вузлів одночасно помилилися. За умови незалежності ланцюгів:

$$samp\_fool = (1 - circ\_acc)^N \quad (3.20)$$

де samp_fool – ймовірність одночасної помилки всіх N Guards;

Формула однакова як для round-robin, так і для BWR (TrafficSliver), оскільки залежить лише від N та circ_acc, а не від правила призначення пакетів. Основною метрикою у четвертому розділі є circ_acc: вона відображає реальну загрозу від одного скомпрометованого Guard-вузла, тоді як samp_fool є оптимістичною оцінкою, що передбачає одночасну компрометацію всіх трьох.

3.4 Висновок до розділу 3

Запропонований метод є модифікацією двох існуючих підходів, що раніше застосовувались незалежно. TrafficSliver розподіляє трафік між кількома Tor-ланцюгами, позбавляючи атакуючого доступу до повного відбитку сайту, але не захищає від аналізу локальних ознак у фрагментах. Методи змагального збурення (A2A, ALERT) вставляють dummy-пакети в градієнтно-чутливі позиції траси та ефективно спотворюють локальні статистичні ознаки, проте вимагають значних обчислювальних ресурсів клієнта в реальному часі. Жоден з методів окремо не атакує обидва рівні ознак одночасно, схематично це зображено на рисунку 3.4.

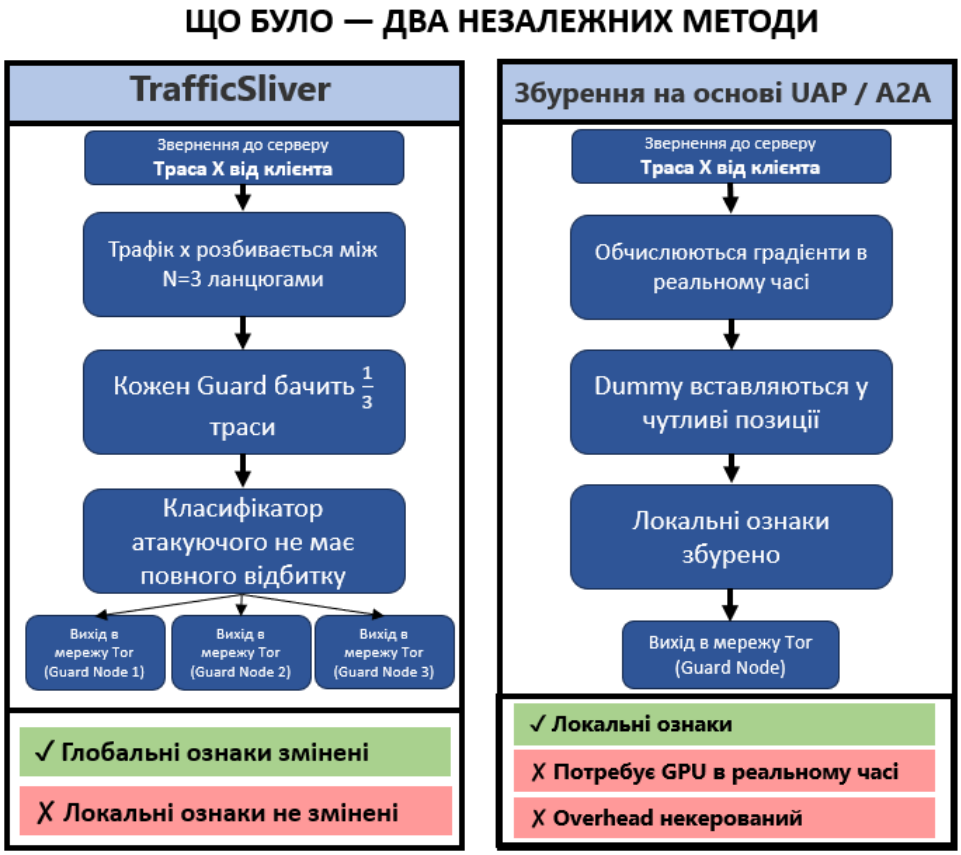

Рис. 3.4. Схематичне зображення батьківських методів

Модифікація полягає в інтеграції цих підходів у єдиний пайплайн із трьома ключовими архітектурними рішеннями. По-перше, обчислення збурення винесено в офлайн: пул градієнтно-чутливих позицій будується один раз на основі карти чутливості S[i] і зберігається у вигляді файлу розміром близько 6 КБ – це усунуло вимогу до GPU на стороні клієнта. По-друге, встановлено порядок застосування компонентів: збурення застосовується до повної траси до її розподілу між ланцюгами, що зберігає overhead на рівні 10% проти 30% при зворотному порядку. По-третє, введено єдиний бюджет B, що одночасно визначає силу збурення та overhead трафіку (overhead = B/T).

У результаті метод працює у двох фазах, які можна побачити на рисунку 3.5. Підготовча фаза виконується один раз: обчислюється карта чутливості S[i] через градієнти функції втрат, формується пул позицій P і зберігається на клієнті. Захисна фаза активується при кожному з'єднанні: у вхідну трасу x вставляється B dummy-пакетів у позиції з пулу P, після чого захищена траса x` = x + B розподіляється між N=3 Tor-ланцюгами через TrafficSliver. Кожен Guard-вузол отримує приблизно третину вже збуреної траси – без доступу до повної послідовності та її незбуреної форми.

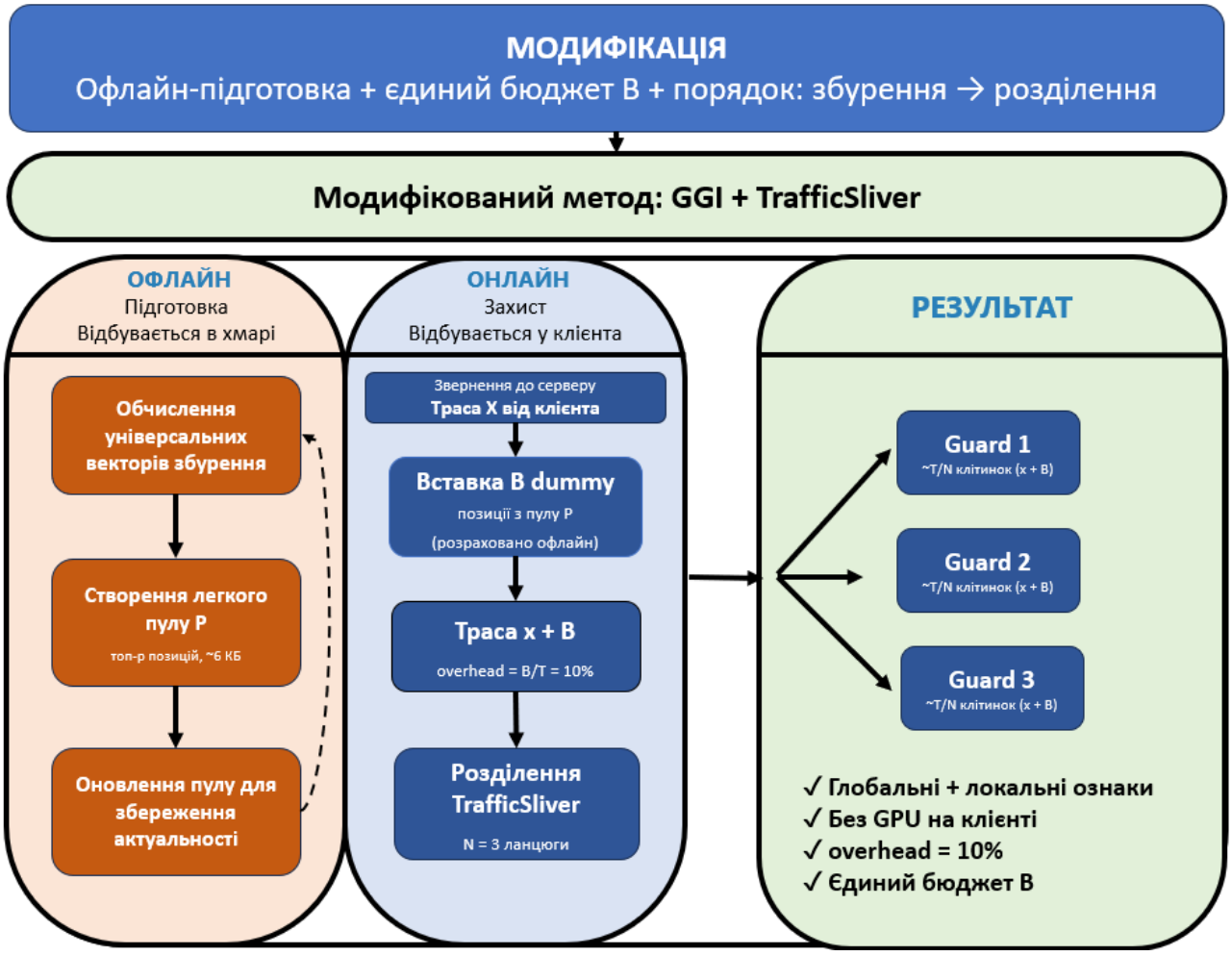

Рис. 3.5 Схематичне зображення модифікованого методу

Математичний аналіз підтвердив теоретичну обґрунтованість підходу і водночас чесно зафіксував його межі. З одного боку, вставка dummy у позиції з найбільшою чутливістю S[i] гарантує зростання функції втрат класифікатора, а рандомізація вибору цих позицій теоретично позбавляє атакуючого стабільних даних для перенавчання – $I(Z_i ; Y) = 0$. З іншого боку, агрегатні ознаки трафіку

(кількість пакетів у часових вікнах) зберігають класовий сигнал незалежно від вибору позицій: $I(n_w(x') ; Y) > 0$. Усунення цієї межі вимагає або рівномірного розподілу dummy по всіх часових вікнах, або значного збільшення бюджету – обидва шляхи суперечать цільовому overhead <= 30%. Це протиріччя є центральним об'єктом дослідження у четвертому розділі.

Метод на даному етапі є концептом. Конкретні параметри – розмір пулу p, правило вибору позицій, кількість ланцюгів N – визначаються не теоретично, а ітеративно, за результатами систематичних експериментів. Усі кількісні оцінки ефективності, наведені у цьому розділі, є теоретичними гіпотезами, що підлягають верифікації або спростуванню у четвертому розділі. Підсумовані дані розділу зібрані у таблицю 3.4.

Таблиця 3.4 – Підсумок теоретичних результатів розділу 3

Результат	**Зміст**	**Статус**
Концепт методу	Офлайн-UAP (B=500, overhead=10%) + TrafficSliver N=3	Обґрунтовано
Ефективність GGI	DL(Z) >= B * min S[i] > 0 – кожен dummy збурює класифікатор	Теоретично доведено
Стійкість до AT (послідовні)	$I(Z_i ; Y) = 0$ – рандомізація усуває навчальний сигнал для DF	Теоретично обґрунтовано
Межа методу (агрегатні)	$I(n_w ; Y) > 0$ – TAF-ознаки зберігають класовий сигнал	Теоретично встановлено
Метрики оцінки	circ_acc (основна) та samp_fool = $(1\text{-circ\_acc})^N$	Визначено
Числові оцінки	Теоретичні гіпотези по 4 сценаріях атакуючих	Верифікуються у розділі 4

РОЗДІЛ 4 ПРАКТИЧНА ОЦІНКА ЕФЕКТИВНОСТІ МОДИФІКОВАНОГО МЕТОДУ ЗАХИСТУ У МЕРЕЖІ TOR

4.1 Середовище тестування, методологія дослідження та система метрик оцінки ефективності

Цей підрозділ описує умови та інструменти, за допомогою яких проводилась оцінка методу: апаратне та програмне середовище, набір даних, моделі атакуючих і те, чим принципово відрізняються між собою їхні підходи, підхід до організації оцінки, а також систему метрик. Конкретні результати кожного етапу наведено у підрозділах 4.3–4.6.

Апаратне та програмне середовище

Усі експерименти проводились на одній робочій станції під керуванням операційної системи Linux Mint на машині з характеристиками, що описані у таблиці 4.1.

Таблиця 4.1 – Апаратна конфігурація дослідницького стенду

Компонент	**Характеристика**	**Роль в експериментах**
Процесор	Intel Core i5-12450H (12th Gen, 8 ядер)	CPU-операції, генерація захищеного трафіку
GPU	NVIDIA GeForce RTX 2050, 4 GB VRAM	Навчання та інференс нейронних мереж
RAM	16 GB	Завантаження датасету, батч-обробка
ОС	Linux Mint	Середовище розробки та запуску пайплайну

Обмеження у 4 ГБ VRAM безпосередньо вплинуло на організацію дослідження: навчання ресурсоємних моделей (зокрема Holmes) вимагало зменшеного розміру батчу і значних витрат часу. Це обмеження відображено у відповідних методологічних рішеннях підрозділів 4.4 та 4.6.

Програмний стек, що використовувався для отримання результатів описано у таблиці 4.1.2:

Таблиця 4.2 – Програмний стек

Компонент	**Версія**	**Призначення**
Python	3.8	Основна мова реалізації
PyTorch	>=1.13	Навчання та інференс нейронних мереж
WFlib	ACM CCS 2024	Бібліотека атакуючих (DF, VarCNN, Holmes)
scikit-learn	>=1.1	Допоміжні метрики та статистика
NumPy	>=1.23	Обробка трас та генерація збурень
matplotlib	>=3.5	Візуалізація результатів

Ключовою зовнішньою залежністю є бібліотека WFlib [31] – стандартизоване середовище для WF-досліджень, представлене на ACM CCS 2024. Вона надає реалізації атакуючих моделей, завантажувач датасету CW та засоби генерації ознак (DIR, DT2, TAF). Код захисного методу, пайплайн та система оцінки є внеском даної роботи, розробленим поверх WFlib.

Датасет

Для оцінки використано датасет CW (Closed-World) – стандартний еталонний набір у галузі Website Fingerprinting, наданий WFlib. Характеристики датасету надані у таблиці 4.3:

Таблиця 4.3 – Характеристики датасету CW

Параметр	**Значення**
Кількість класів	95 веб-сайтів
Навчальна вибірка	≈ 85 641 трас
Валідаційна вибірка	≈ 9 516 трас
Тестова вибірка	10 573 траси
Формат траси	Послідовність 5000 клітинок: значення {-1, 0, +1} × timestamp
Ознака DIR	Напрямок клітинок {-1, +1} – використовується DF
Ознака DT2	Напрямок × часовий інтервал – використовується VarCNN
Ознака TAF	Лічильник вихідних пакетів per ~40 мс вікно – використовується Holmes

Моделі атакуючих та їх характеристики у контексті захисту

Розуміння того, які ознаки трафіку аналізує кожен атакуючий, є необхідним для правильної інтерпретації результатів: один і той самий захист по-різному взаємодіє з різними типами класифікаторів, як описано у таблиці 4.1.4. Точність

кожного методу на не зміненому трафіку виведене експериментально, для підтвердження дієздатності методу атаки.

Таблиця 4.4 – Моделі атакуючих: архітектура та базова точність

Атакуючий	**Архітектура**	**Ознака**	**Точність (clean)**
DF	1D-CNN (8 блоків + 3 FC)	DIR – послідовність напрямків	97.76%
VarCNN	Variable-depth CNN	DT2 – напрямок × часовий інтервал	98.02%
Holmes	SupConLoss encoder + kNN (k=10)	TAF – лічильник пакетів per вікно	98.35%

DF аналізує послідовність напрямків пакетів безпосередньо через 1D-CNN. Класифікація спирається на конкретні позиції burst-ів у трасі – де саме в послідовності з'являються характерні групи вхідних або вихідних пакетів. Тому DF є чутливим до просторового розташування пакетів.

VarCNN, крім напрямку, враховує часові інтервали між пакетами (IPI). Ознака DT2 кодує одночасно напрямок і часовий проміжок до наступного пакету. Це робить VarCNN чутливим не лише до позицій, але й до часових характеристик трафіку – зокрема до аномально малих або нульових IPI.

Holmes принципово відрізняється від обох. Він обчислює Time-Aggregated Features (TAF): кількість вихідних пакетів у кожному фіксованому часовому вікні (~40 мс), після чого навчає embedding-простір через contrastive learning (SupConLoss) і класифікує через kNN. Ключова особливість: Holmes дивиться на те, скільки пакетів у вікні, а не на те, де вони розташовані. Це принципово змінює взаємодію з позиційно-рандомізованим захистом – детальний аналіз наведено у підрозділах 4.4 та 4.5. Узагальнені ознаки атакуючих моделей охараткеризовано у таблиці 4.5:

Таблиця 4.5 – Тип ознак атакуючих як ключ до розуміння результатів

Атакуючий	**Аналізує**	**Чутливий до**
DF	Послідовність позицій пакетів	Змін у просторовому розташуванні burst-ів
VarCNN	Позиції + IPI	Змін позицій і аномальних IPI (IPI=0)
Holmes	Кількість пакетів за часове вікно	Змін у кількості пакетів per вікно

Підхід до організації оцінки

Дослідження організовано як ітеративний процес з дев'яти фаз, де кожна наступна фаза виходила з результатів і питань попередньої. Такий підхід дозволив послідовно досліджувати ефективність методу, виявляти його межі та перевіряти гіпотези щодо їх подолання. Детальний опис кожної фази наведено у підрозділах 4.3–4.6.

Важливою характеристикою дослідження є еволюція самого протоколу оцінки: у ранніх фазах адаптивний атакуючий навчався на одній фіксованій реалізації захисту, що давало завищену оцінку ефективності захисту. У фінальній фазі (Tier-1) цей протокол замінено на online-augmentation: нова рандомізована реалізація захисту генерується на кожній епосі навчання, що відповідає реальному розподілу (кожне з'єднання – свій унікальний Z_i). Методологічна верифікація та зіставлення результатів двох протоколів докладно розглянуті у підрозділі 4.6.

Щодо кількості повторень: адаптивні експерименти (Tier-1) проводились з трьома незалежними сідами (seed = 0, 1, 2), результати звітуються як mean ± std. Неадаптивні – з одним сідом: навчання детермінованих класифікаторів на фіксованому датасеті демонструє мінімальну варіацію між запусками.

Система метрик оцінки ефективності

Для оцінки захисту в умовах Traffic Split використовується система з п'яти метрик (табл. 4.6), вибір яких визначається моделлю загрози: атакуючий контролює один Guard-вузол і бачить один фрагмент трафіку за одне з'єднання.

Таблиця 4.6 – Система метрик оцінки ефективності захисту

Метрика	**Визначення**	**Роль**
fool	1 − acc(f, D_test) Fooling rate на повній трасі без split	Для конфігурацій без розподілу трафіку
circ_acc	P(f(x'_c) = y) Точність на одному Guard-фрагменті	Основна операційна метрика

Продовження таблиці 4.6

circ_fool	1 − circ_acc	Fooling rate одного Guard-вузла
samp_fool	P(∀c: f(x'_c) ≠ y) ≈ (1−circ_acc)^N Всі N Guards помилилися одночасно	Оптимістична метрика; для порівнянності з літературою
linking_acc	acc(f, D_test) на повній x' без split Верхня межа для linking-атакуючого	Ефективність GGI незалежно від split

Основною метрикою у підрозділах 4.3–4.6 є circ_acc – вона відповідає реалістичному сценарію, де атакуючий спостерігає один Guard-вузол. Метрика samp_fool також наводиться для порівнянності з літературою, але є оптимістичною: вона вимагає одночасної помилки всіх трьох Guards, що менш вимогливо для атакуючого. Для адаптивних результатів Tier-1 наводиться circ_acc ± std по трьох сідах.

4.2 Програмна реалізація методу захисту – code-review

Повний вихідний код ключових модулів наведено у додатку А. Усі трафікові траси зберігаються у форматі .npz – двовимірний масив X форми (n_traces, seq_len = 5000), де кожен елемент кодує одночасно напрямок і час пакету: `X[i, j] = sign(direction) × timestamp_in_seconds`

Значення +0.135 означає вихідний пакет, що прибув через 135 мс після початку з'єднання; -0.072 – вхідний пакет через 72 мс. Нульові значення – порожні позиції (zero-padding) для коротких трас, що не досягли 5000 клітинок.

Атакуючі моделі не читають цей формат безпосередньо – кожна модель очікує свій тип ознак (табл. 4.7). Модуль data_io.py відповідає за перетворення.

Таблиця 4.7 – Ознаки трафікових трас для різних атакуючих моделей

Ознака	Модель	Форма тензора	Як обчислюється
DIR	DF	(N, 1, 5000)	sign(X) – лише знак: +1 / -1 / 0
DT2	VarCNN	(N, 2, 5000)	Канал 0: sign(X); Канал 1: IPI = diff(|X|), clip ≥ 0
TAF	Holmes	(N, 3×2×W)	n_packets, n_bursts, mean_burst per 40 мс вікно (WFlib)

Перетворення до DT2 є особливо важливим для розуміння взаємодії з захистом. Міжпакетний інтервал (IPI) обчислюється як:

```
IPI[j] = clip( |X[j]| - |X[j-1]|, 0, None )
```

Від'ємні значення (коли наступний пакет прибуває "раніше" попереднього через особливості запису) обрізаються до нуля. Dummy-пакети у методі захисту отримують timestamp попереднього пакету, що означає IPI=0 – і ця аномалія є видимою у каналі DT2. Саме через це VarCNN після перенавчання частково ідентифікує захищений трафік через часовий канал.

TAF-ознаки для Holmes генеруються окремим інструментом WFlib (gen_taf_adap.py) поза основним пайплайном: для кожного варіанту захисту окремо готується набір TAF-векторів, що використовуються для навчання та оцінки Holmes.

Побудова карти чутливості та пулу позицій (gradient_pool.py)

Функція build_sensitivity_map (див. додаток А.1) реалізує ключову офлайн-операцію: визначення, де у трасі класифікатор найбільш чутливий до змін. Алгоритм ітерується по навчальній вибірці протягом кількох епох і накопичує абсолютні значення градієнтів функції втрат по вхідному тензору:

```
x_batch.requires_grad_(True)
loss = cross_entropy(model(x_batch), y_batch)
loss.backward()
grad_magnitude += x_batch.grad.squeeze(1).abs().mean(dim=0)
```

Ключовий момент: параметри моделі заморожені (requires_grad_(False) для всіх p в model.parameters()), тому градієнт тече виключно до вхідного тензору. Функція squeeze(1) прибирає розмірність каналу (1D-ознака DIR), mean(dim=0) усереднює по батчу – в результаті отримується вектор форми (seq_len,), де кожен елемент – середнє $\left|\frac{\partial L}{\partial x_i}\right|$ по всьому батчу. Після 7 епох усереднення по n_batches дає стабільну карту S[i].

Функція select_top_positions повертає топ-k позицій у відсортованому порядку (зростання) – це важливо для коректної правосторонньої вставки dummy:

```
top_idx = np.argsort(grad_magnitude)[::-1][:k]
return np.sort(top_idx).astype(np.int64)
```

Результат зберігається у vectors/rand_pool.pt – PyTorch-словник із масивом позицій (1500 int64), повною картою grad_magnitude, параметрами overhead та seq_len. Зберігання повної карти дозволяє при необхідності побудувати пул іншого розміру без повторного запуску навчання.

Реалізація захистів (defenses/ggi.py, defenses/traffic_split.py)

Вставка dummy у трасу (defenses/ggi.py)(див. додаток А.2). Базова операція _insert_outgoing_dummies реалізує вставку dummy-клітинок із критично важливою деталлю – обробкою позицій у спадному порядку (справа наліво):

```
for k in np.sort(positions)[::-1]:
    prev_abs = abs(float(trace[k-1])) if k > 0 else 0.0
    dummy = prev_abs if prev_abs > 0 else EPS_TIMESTAMP
    trace = np.concatenate([trace[:k], [dummy], trace[:length-1][k:]])
```

Обробка справа наліво вирішує проблему зміщення індексів: вставка dummy на позицію k зсуває всі наступні елементи вправо. Якщо обробляти зліва направо, наступна цільова позиція k' > k вже не вказуватиме на потрібний елемент. При обробці справа наліво вже оброблені (більші) позиції не впливають на ще не оброблені (менші). Після всіх вставок trace = trace[:length] обрізає хвіст до T=5000.

Timestamp dummy дорівнює абсолютному значенню timestamp попереднього пакету (prev_abs). Це означає нульовий IPI між попереднім пакетом і dummy: обидва відправляються в один момент часу. Якщо dummy є першим у трасі (k=0), використовується мале значення EPS_TIMESTAMP = 1e-9 (майже нуль, але не рівно нуль), щоб dummy не відфільтрувався як порожня позиція (zero-padding).

Static vs Rand-GGI відрізняються лише вибором позицій:

```
# Static GGI – однакові позиції для всіх трас:
apply_static_ggi(X_raw, positions)  # positions = top-500

# Rand-GGI – нова вибірка для кожної траси:
```

```
rng = np.random.default_rng(seed)
sampled = rng.choice(pool_positions, size=n_insertions, replace=False)
apply_rand_ggi(X_raw, pool_positions, n_insertions, seed)
```

rng.choice з replace=False гарантує, що кожна з 500 обраних позицій є унікальною – dummy не вставляються двічі на одну позицію. Параметр seed дозволяє відтворити точно ту саму рандомізацію при повторному запуску: якщо seed=None, NumPy генерує новий при кожному виклику.

Реалізація Traffic Split (defenses/traffic_split.py) (див. додаток А.3). Round-robin використовує зрізи NumPy з кроком n_circuits:

```
for c in range(n_circuits):
    sub = active[c::n_circuits]   # кожен N-й пакет, починаючи з c
    X_split[i*3 + c, :len(sub)] = sub[:seq_len]
```

Функція active = trace[:active_len] – лише реальні (ненульові) клітинки траси, без zero-padding. Фрагмент sub для ланцюга c складається з пакетів c, c+N, c+2N, ... і зберігається у масив форми (seq_len,) з нуль-паддингом до кінця. Мітка y дублюється для кожного з N фрагментів – це дозволяє стандартним класифікаторам оцінювати кожен фрагмент незалежно за звичайним supervised learning.

TrafficSliver (BWR) реалізує реалістичнішу стратегію – випадкові пачки пакетів:

```
j = 0
while j < active_len:
    circuit = int(rng.integers(n_circuits))
    run = int(rng.integers(1, bwr_max + 1))  # Uniform[1, 4]
    assignment[j:j+run] = circuit
    j += run
```

Кожен "run" – блок від 1 до bwr_max=4 consecutive пакетів – надсилається до одного випадкового ланцюга. На відміну від round-robin, де кожен Guard бачить рівно T/N розріджених пакетів, TrafficSliver дає кожному Guard цілісні шматки трафіку змінного розміру – це ближче до реального мережевого розгортання, де пакети однієї передачі йдуть разом.

Реалізація метрик (metrics.py)

Функція predict виконує батчевий інференс у режимі model.eval() із вимкненим обчисленням градієнтів (@torch.no_grad()), що скорочує час і пам'ять GPU:

```
preds[i:i+batch_size] = model(batch).argmax(1).cpu().numpy()
```

argmax(1) повертає індекс класу з найбільшим logit-значенням – передбачений клас. .cpu().numpy() переносить результат з GPU до NumPy для подальшого порівняння з мітками (див. додаток А.6)

Для конфігурацій зі split функція circuit_metrics обчислює всі три метрики з одного масиву передбачень:

```
correct = (preds == y_split)           # булевий масив
circ_acc = correct.mean() * 100        # середня точність sub-трас

correct_per_visit = correct.reshape(n_orig, n_circuits)
sample_acc = correct_per_visit.any(axis=1).mean() * 100
samp_fool  = 100 - sample_acc
```

reshape(n_orig, n_circuits) перетворює плоский масив у матрицю n_visits × n_circuits: кожен рядок відповідає одному відвіданню, кожен стовпець – одному Guard-ланцюгу. any(axis=1) повертає True для рядка, де хоча б один ланцюг класифікований правильно (тобто атакуючий успішний через хоча б один Guard). samp_fool = 1 - sample_acc: частка відвідувань, де жоден Guard не вгадав.

icirc_acc – це "наскільки часто один Guard правий". samp_fool – це "наскільки часто всі три Guard одночасно помилилися". При circ_acc = 10% кожен Guard правий лише у 10% випадків, але samp_fool ≈ (0.9)^3 = 72.9% – значно нижча цифра захисту. Ця різниця у ~23 відсоткових пункти визначає, чому circ_acc є чеснішою метрикою для заявленої моделі загрози (один Guard).

Пайплайн оцінки: скрипти 3 та 4

Скрипт 3_apply_defense.py (див. додаток А.7) є диспетчером: він приймає параметр --defense і викликає відповідну функцію з модуля defenses. Принциповим архітектурним рішенням є роздільне застосування збурення та split: скрипт 3 виробляє захищену трасу x' у форматі .npz, а скрипт 4 при

необхідності ділить її на фрагменти. Це дозволяє комбінувати будь-який захист з будь-якою стратегією split, не дублюючи код.

Скрипт 4_evaluate.py (див. додаток А.8) реалізує два режими через --split. При split=none обчислюється звичайна accuracy і fooling rate на повній трасі. При split=roundrobin або tsilver спочатку обчислюється linking_acc – точність на несплітованій захищеній трасі x': це верхня межа для атакуючого, що може реасемблювати всі фрагменти назад.

```
# linking_acc: accuracy без split (upper bound)
result.update(accuracy_and_fooling(model, X_raw, y, ...))
result["linking_acc"] = result.pop("acc")

# circuit metrics: з split
X_split, y_split = split_round_robin(X_raw, y, n_circuits)
result.update(circuit_metrics(model, X_split, y_split, n_circuits, ...))
```

Результат виводиться як єдиний рядок JSON у stdout:

```
RESULT {"tag": "...", "circ_acc": 1.12, "samp_fool": 98.61,
"linking_acc": 4.09}
```

Такий формат зручний для збору результатів кількох запусків: grep "RESULT" з лог-файлу дає чистий JSON-потік, який легко агрегувати скриптом для обчислення mean ± std по сідах.

Навчання адаптивного атакуючого (train_attacker.py) та online-аугментація

Скрипт train_attacker.py (див. додаток А.10) підтримує два режими: clean (навчання на чистому трафіку) та adaptive (online-аугментація). Режим adaptive є ключовим для коректної оцінки рандомізованого захисту і реалізує принципово важливу ідею: свіжа рандомізована реалізація захисту на кожній епосі.

```
# Режим clean:
X_ep, y_ep = Xtr, ytr   # ті самі дані кожну епоху

# Режим adaptive (online-aug):
X_ep, y_ep = apply_defense(Xtr, ytr, args, pools,
                           seed=args.seed * 1000 + epoch)
```

Формула seed для кожної епохи: args.seed * 1000 + epoch. Це означає, що при seed=0, epoch=0: seed=0; при seed=0, epoch=1: seed=1; при seed=1, epoch=0:

seed=1000. Таким чином, різні сіди (runs) і різні епохи дають гарантовано різні рандомізації, але результати повністю детерміновані і відтворювані за заданим seed. Це принципово важливо для рандомізованого захисту, тому, що за умови навчання атакуючого на одній фіксованій реалізації захисту, він фактично запам'ятовує одну конкретну підмножину Z_i позицій dummy для кожної траси. У реальному розгортанні кожне з'єднання матиме свою унікальну реалізацію. Online-аугментація відтворює цей реальний розподіл: атакуючий вимушений навчитися класифікувати трафік незалежно від конкретного вибору Z_i – тобто знайти ознаки, що залишаються стабільними попри рандомізацію.

Для навчання використовується class-balanced підмножина: n_per_class=200 трас на клас з навчальної вибірки (200 × 95 = 19 000 трас). Це робить одну епоху affordable – замість повних 85 641 трас. Вибір найкращої епохи здійснюється за macro F1 (а не accuracy) на фіксованому захищеному validation set:

```
f1 = f1_score(yv_true, preds, average="macro")
if f1 > best_f1:
    best_f1, best_epoch = f1, epoch
    torch.save(model.state_dict(), save_file)
```

Macro F1 є кращою метрикою для вибору моделі, ніж accuracy, оскільки рівномірно враховує ефективність на всіх 95 класах – клас з малою кількістю прикладів не "тоне" у загальному показнику. Validation set для вибору найкращої епохи захищений фіксованим seed (10000 + args.seed), щоб різні epoch-и порівнювались на одному розподілі, тоді як train-set кожну епоху отримує нову рандомізацію. Для кращого розуміння основні параметри навчального процесу атакуючого зібрані у таблиці 4.8.

Таблиця 4.8 – Ключові технічні параметри навчання атакуючого

Параметр	Значення	Обґрунтування
n_per_class	200	Баланс між репрезентативністю і часом однієї епохи
train_epochs	12	Достатньо для збіжності; best epoch зберігається

Продовження таблиці 4.8

batch_size	64	OOM-safe для 4 GB VRAM при DT2 ознаках VarCNN

optimizer	Adamax	Адаптивний; стабільний на sparse gradient даних
learning_rate	2e-3	Стандарт для DF/VarCNN з WFlib
Метрика відбору	Macro F1	Рівномірна вага всіх 95 класів
Seed формула	seed*1000 + epoch	Детермінована унікальна рандомізація per epoch

Часові збурення: оптимізація вектора delta (timing_uap.py, defenses/timing.py)

DF читає лише напрямок пакетів (ознака DIR), тому Rand-GGI, що додає лише вихідні dummy, вже достатньо для нього. VarCNN використовує DT2 – двоканальну ознаку, де другий канал є міжпакетним інтервалом (IPI). Dummy з нульовим IPI (timestamp = попередній пакет) є очевидним артефактом у каналі IPI. Щоб атакувати VarCNN по обох каналах, застосовується додаткове часове збурення: затримка вихідних пакетів на невеликий позиційно-залежний відсоток.

Функція generate_timing_delta, у timing_uap.py (див. додаток А.4), знаходить вектор delta форми (seq_len,) методом проєктованого градієнтного піднесення (projected gradient ascent). Delta має дві фізичні обмеження: delta[i] >= 0 (клієнт може лише затримати власні пакети, але не прискорити) і delta застосовується лише до вихідних пакетів (direction > 0). Після кожного кроку delta обрізається у бюджет [0, overhead]:

```
scale = 1.0 + delta.view(1,1,-1) * outgoing.unsqueeze(1)
x_adv = torch.cat([direction, ipi * scale], dim=1)
...
delta = torch.clamp((delta + alpha * grad_delta.sign()).detach(), 0.0,
overhead)
```

Два архітектурних рішення заслуговують окремої уваги. По-перше, оптимізація ведеться лише на трасах, які ще не обдурено (keep = ~fooled). Це концентрує градієнтний сигнал на важких прикладах і прискорює збіжність: коли модель вже помилково класифікує трасу, витрачати обчислення на її додаткове збурення безглуздо. По-друге, використовується знаковий крок (grad_delta.sign()), а не масштабований градієнт. Знаковий крок рівномірно

оновлює всі позиції незалежно від магнітуди градієнта, що забезпечує більш рівномірне використання бюджету overhead по всій трасі.

Функція apply_timing (defenses/timing.py, (див. додаток А.5)) застосовує збережений вектор delta до тестових трас. Реалізація використовує форвардні різниці (IPI[i] = |t[i+1]| − |t[i]|), оскільки саме так DT2 читає часові проміжки. Затримка нараховується на пакет, що приходить пізніше:

```
ipi = np.diff(abs_t, axis=1)                          # форвардні IPI
outgoing_next = (X_raw[:, 1:] > 0).astype(np.float32)
scale = 1.0 + delta[:length-1] * outgoing_next
new_ipi = ipi * scale
rebuilt = first + np.cumsum(new_ipi, axis=1)          # відновлення абс.
часу
```

Абсолютні timestamps відновлюються кумулятивною сумою збурених IPI. Чисто математично це гарантує монотонність результуючих timestamps (оскільки delta >= 0 і IPI >= 0, new_ipi >= ipi >= 0) і коректно зберігає напрямки пакетів (sign(X_raw) не змінюється). Важливо, що лише вихідні пакети отримують затримку (outgoing_next) – вхідні пакети клієнт не контролює і їх timestamps залишаються незмінними.

Альтернативні та експериментальні захисти

У процесі дослідження були реалізовані та протестовані чотири додаткові захисти, жоден з яких не переміг адаптивного атакуючого Holmes. Вони задокументовані у кодовій базі (defenses/poisson_ggi.py, defenses/temporal_jitter.py, defenses/burst_ggi.py, defenses/rand_bbd.py)(див. додаток А.11 – А.14) разом із чесним поясненням причин невдачі. Цей огляд описує технічну реалізацію кожного і пояснює, чому саме вони не усунули слабкість Rand-GGI проти Holmes.

Poisson-GGI (defenses/poisson_ggi.py). Rand-GGI рандомізує позиції dummy у межах пулу, але пул рівномірно розподілений у часі, тому кожне TAF-вікно все одно отримує приблизно однакову кількість dummy. Holmes читає саме кількість пакетів/пачок на вікно (TAF-ознаку), тому стабільний per-window count є стабільним відбитком. Poisson-GGI намагається рандомізувати count: для

кожного TAF-вікна кількість dummy n_w ~ Poisson(lambda), де lambda = n_insertions / n_windows:

```
pool_window = (abs_ms[pool_clipped] / window_ms).astype(np.int32)
windows, inverse = np.unique(pool_window, return_inverse=True)
lam = n_insertions / len(windows)
for w in range(len(windows)):
    count = int(rng.poisson(lam))
    sampled.extend(rng.choice(in_window, size=count,
replace=False).tolist())
```

Проблема в тому, що lambda обчислюється з timestamps самої траси, а timestamps є класозалежними – різні сайти мають різні часові профілі завантаження. Адаптивний Holmes просто навчився цій залежності: навіть при Poisson-рандомізації per-window count залишається класово-характеристичним, бо середнє lambda, а не лише сама реалізація Poisson, несе інформацію про сайт. Ключовий збій: рандомізація реалізацій при незмінному класозалежному середньому не прибирає взаємну інформацію I(count; site).

Temporal Jitter (defenses/temporal_jitter.py). Ідея проста: зсунути всі timestamps траси на одну випадкову константу shift ~ Uniform(-jitter_ms, +jitter_ms), щоб пакети потрапляли в різні TAF-вікна від запуску до запуску:

```
shift = rng.uniform(-jitter_s, jitter_s)
new_abs = np.where(real, np.maximum(abs_t + shift, 1e-9), 0.0)
X_out[i] = (np.sign(trace) * new_abs).astype(np.float32)
```

Захист коштує нуль накладних витрат і є найпростішим у реалізації. Але він зберігає відносну структуру траси: різниця timestamps будь-яких двох пакетів залишається незмінною (t'[j] − t'[i] = t[j] − t[i]). Holmes переважно покладається на відносні часові інтервали, а не на абсолютні позиції у часі. Після перенавчання на jittered-трасах модель просто навчилась ігнорувати абсолютне зміщення і відновила точність на відносних ознаках.

Burst-GGI (defenses/burst_ggi.py). Rand-GGI вставляє dummy з IPI = 0 (timestamp = попередній пакет). Адаптивний VarCNN навчається ідентифікувати ці нульові IPI як артефакти захисту. Burst-GGI намагається вставляти dummy всередину реальних вихідних пачок із реалістичним IPI. Функція

build_position_stats сканує навчальну вибірку і збирає статистику по кожній позиції: як часто там є придатний gap у burst (freq) та середнє і стандартне відхилення half-gap (mean_hg, std_hg):

```
for pos, half_gap in get_outgoing_burst_gaps(trace, min_gap):
    freq[pos] += 1
    sum_hg[pos] += half_gap
    sum_hg2[pos] += half_gap * half_gap
```

При застосуванні IPI dummy вибирається як Normal(mu_hg, sigma_hg * 0.25), обрізане у [MIN_IPI, 2*mu_hg]. Коефіцієнт 0.25 зменшує дисперсію: ціль – отримати правдоподібне значення, а не відтворити повний розподіл gap. Burst-GGI прибрав ознаку нульового IPI, але адаптивний атакуючий все одно переміг: навіть реалістично виглядаючі dummy змінюють загальний burst-профіль траси способом, що залишається класово-характеристичним.

Rand-BBD (defenses/rand_bbd.py). Це єдиний захист у проєкті, що теоретично правильно атакує TAF-ознаку Holmes – а саме кількість пачок на вікно, а не кількість пакетів. Rand-BBD має два компоненти. Компонент A розбиває реальні пачки: вставляє один вихідний dummy безпосередньо перед кожною зміною напрямку, розділяючи одну пачку на дві. Компонент B створює фіктивні пачки: у рандомно вибраних вікнах, що не мають реальних меж пачок, вставляється пара (+1, -1) dummy, що імітує burst:

```
# Компонент A: розбити реальні пачки
burst_idx = np.where(signs[1:] != signs[:-1])[0] + 1
# Компонент B: фіктивні пачки у порожніх вікнах
empty_windows = list(set(range(last_window)) - burst_windows)
fake_windows = rng.choice(empty_windows, size=take, replace=False)
```

Ключова відмінність від попередніх підходів: фіктивні вікна вибираються незалежно від сайту (Bernoulli-вибір), що теоретично дає клас-незалежний шум у per-window burst count. Саме цього бракувало Poisson-GGI (там lambda залежала від траси). Rand-BBD залишений у кодовій базі як гіпотеза майбутньої роботи: масштабне тестування проти адаптивного Holmes не проводилось у межах цього дипломного проєкту.

Таким чином, програмна реалізація методу є цілісною системою з чіткими межами відповідальності: data_io.py обробляє вхідні дані, gradient_pool.py будує офлайн-артефакт, defenses/ реалізує захисти (Rand-GGI, часові збурення та Traffic Split), metrics.py обчислює результат, а скрипти 3–4 і train_attacker.py координують їх у відтворюваний пайплайн. Чотири експериментальні варіанти (Poisson-GGI, Temporal Jitter, Burst-GGI, Rand-BBD) задокументовані з поясненням причин невдачі, що чесно окреслює межі підходу і вказує на Rand-BBD як перспективу майбутньої роботи. Ключовим методологічним принципом залишається онлайн-аугментація у train_attacker.py – вона є обов'язковою умовою для коректної оцінки рандомізованого захисту.

4.3 Побудова пулу позицій та оцінка базових конфігурацій захисту

Цей підрозділ охоплює перші три фази експерименту: побудову карти чутливості і пулу позицій, верифікацію принципу GGI на Static GGI та виявлення його вразливості до adversarial training, рандомізацію в Rand-GGI і інтеграцію з Traffic Split.

Перший крок дослідження – побудова карти чутливості S[i] – результату офлайн розрахунків, що визначає в яких позиціях вставляти dummy. Функція build_sensitivity_map ітерується по навчальній вибірці (D_train, 85 641 траса) протягом 7-20 епох (коригується вручну емпіричним методом) і накопичує усереднені абсолютні значення градієнтів функції втрат: S[i] = $E_x \left|\frac{dL}{dx[i]}\right|$.

Важливо: Параметри моделі заморожені, градієнт визначається виключно за вхідним набором даних.

Результат виявив виразну просторову концентрацію чутливості у першій частині траси. Тобто модель атакуючого найбільш чутлива на перших етапах з'єднання. Для Static GGI (топ-500) усі 500 найчутливіших позицій зосереджені в діапазоні 0–553 (перші 11% траси довжиною T = 5000). Для Rand-GGI пул з 1500 позицій охоплює діапазон 0–2021 (перші 40%). Ця концентрація відображає фазу встановлення зв'язку (DNS-запит, TLS-handshake, перші обміни), де патерни найкраще розрізняють сайти.

Static GGI – базова ефективність та вразливість до AT

Метою першої фази була перевірка базового принципу: чи здатне цілеспрямоване вставляння 500 dummy у топ-500 позицій з S[i] суттєво знизити точність WF-класифікатора проти неадаптивного атакуючого. Результати фази експерименту описані у таблиці 4.9. У Static GGI множина позицій Z є фіксованою однаковою для кожної траси.

Таблиця 4.9 – Static GGI проти неадаптивних атакуючих (overhead 10%)

Атакуючий	Accuracy атакуючого (без захисту)	Accuracy атакуючого (Static GGI)	Fooling_rate
DF	97.76%	10.87%	89.13%
VarCNN	98.02%	7.66%	92.73%

Результати підтвердили ефективність принципу GGI. Кросс-модельне перенесення (92.73% проти VarCNN при пулі, обчисленому на градієнтах DF) свідчить про спільну зону чутливості: позиції 0–553 є інформативними для обох моделей. Проте перевірка стійкості до adversarial training (AT) виявила критичну вразливість: після перенавчання DF на захищеному трафіку (DF_adap) точність відновилась до 98.45% – fooling пав з 89.13% до 1.55%. Причиною є фіксований патерн – завжди ті самі 500 позицій – атакуючий легко виявляє закономірність враховуючи частоту dummy по позиціях.

Rand-GGI – рандомізація як відповідь на AT

Ключова ідея Rand-GGI – збільшення пулу до |P| = 1500 позицій і рандомізований вибір підмножини Z_i розміром k = 500 незалежно для кожної траси. Бюджет overhead залишається незмінним (B/T = 500/5000 = 10%), але вибрана підмножина є унікальною для кожної сесії. Результати експерименту Rand-GGI описані у таблиці 4.10:

Таблиця 4.10 – Static GGI vs Rand-GGI проти неадаптивних атакуючих

Атакуючий	Static GGI (fool%)	Rand-GGI (fool%)	Виграш
DF	89.13%	95.04%	+5.91 p.p.
VarCNN	92.73%	96.32%	+3.59 p.p.

Rand-GGI перевершує Static GGI на ~4–6 p.p. (p.p. – percentage points) проти обох неадаптивних атакуючих: ширший пул охоплює більше градієнтно-чутливих позицій. Була проведена також додаткова перевірка (не включена в таблицю) Rand-GGI проти DF_adap (навченого на Static GGI трасах) – 98.87% fool. Хоча це і є тестом стійкості з певними нюансами (DF_adap не бачив Rand-GGI), він підтверджує, що adversarial training на фіксованому патерні не переноситься на рандомізоване – ціль досягнута. Повноцінна адаптивна оцінка розглядається в підрозділі 4.4.

Traffic Split та комбінований метод Rand-GGI + Split-3

Третя фаза досліджувала розділення трафіку як другий компонент захисту. Захищена траса x’ розподілялась між N = 3 Tor-ланцюгами за принципом round-robin, і кожен Guard-вузоль оцінювався незалежно. Результати експерименту описані у таблиці 4.11. Експеримент може вважатись коротким підтвердженням ефективності методу окремо від змагального збурення.

Таблиця 4.11 – Traffic Split N=3 самостійно проти неадаптивних атакуючих

Атакуючий	samp_fool%	Примітка
DF (non-adap)	96.36%	Захист ефективний
VarCNN (non-adap)	92.01%	Захист ефективний
Holmes (non-adap)	83.22%	найнижчий серед трьох, модель атакуючого орієнтується на 3 ознаки трафіку
DF3_adap (split-aware)	2.92%	повний провал

Самостійний Traffic Split провалився проти DF3_adap – еталонної у дослідженнях моделі DF, перенавченого спеціально на фрагментах чистого трафіку (sub-трасах): samp_fool пав до 2.92%. Кожен фрагмент чистого трафіку зберігає характерні патерни сайту – зокрема, кожен 3-й пакет round-robin зберігає свою локальну послідовність burst’ів.

Комбінований метод Rand-GGI + Split-3 вирішує цю проблему: GGI спотворює локальні ознаки до розподілу по ланцюгах, позбавляючи кожний Guard-вузоль сталого орієнтиру (табл. 4.12).

Таблиця 4.12 – Rand-GGI + Split-3: результати базової конфігурації

Атакуючий	Без захисту (acc%)	samp_fool%
DF (non-adap)	97.76%	99.01%
VarCNN (non-adap)	98.02%	96.47%
Holmes (non-adap)	98.35%	96.91%
DF3_adap	= DF (non-adap)	90.07%

Ключовий результат фази – 90.07% samp_fool проти DF3_adap при повному провалі Traffic Split самостійно (2.92%). Це підтверджує взаємодоповнюваність компонентів – рандомізовані вставки dummy позбавляють атакуючого сталого патерну навіть на рівні окремих фрагментів.

За результатами трьох базових фаз сформовано основну конфігурацію – Rand-GGI + Traffic Split N=3 (round-robin) – яка показує 96–99% samp_fool проти неадаптивних атакуючих та 90.07% проти реалістичного split-aware атакуючого. Наступне питання, що постало перед дослідженням: як метод поводиться проти повноцінного адаптивного атакуючого, що бачив Rand-GGI під час навчання, – це охоплює підрозділ 4.4.

4.4 Оцінка стійкості методу проти адаптивних атакуючих

Після того, як базова конфігурація Rand-GGI + Split-3 показала високу ефективність проти неадаптивних атакуючих, наступне і найважливіше питання: що станеться, якщо атакуючий знає про захист і перенавчається? Цей підрозділ послідовно розглядає три сценарії адаптації зростаючої сили: атакуючий, що знає лише про split (але не про GGI-збурення), атакуючий з часовою ознакою (DT2), та повністю адаптований атакуючий, який бачив саме захищений трафік під час навчання.

Два протоколи адаптації та їх різниця

У дослідженні використовувались два різні протоколи навчання адаптивного атакуючого. Перший – split-aware протокол: атакуючий знає, що трафік розділено на ланцюги, і навчається класифікувати окремі фрагменти (sub-траси) чистого трафіку, але не посвячений у GGI-збурення. Це реалістичний сценарій: Guard-вузоль бачить фрагменти багатьох користувачів, що дає йому достатньо даних для навчання, але не знає точного алгоритму захисту. Другий протокол – повністю адаптивний (worst-case): атакуючий знає про GGI і split, отримує захищені фрагменти для навчання. Він відповідає пасивному атакуючому, який спостерігає уже застосований захист і виносить захищений трафік для перенавчання.

Для рандомізованого захисту другий протокол вимагає особливого підходу до навчання – online-аугментації. Якщо навчати атакуючого на одній фіксованій реалізації GGI-маски, він запам'ятовує одну конкретну підмножину позицій dummy для кожного сайту, що переоцінює стабільність сигналу. Коректним підходом буде використання свіжої реалізації захисту з новим seed (args.seed * 1000 + epoch) при кожній епосі навчання. Це змушує модель навчатись ознакам, які інваріантні до рандомізації позицій, а не запам'ятовувати одну маску – це і є сутністю worst-case сценарію, тестування при цьому відбувається з найкращої для атакуючого позиції. Навчання проводилось на 19 000 sub-трас (200/клас × 95 класів), результати TRUE ADAPTIVE (Tier 1, DF і VarCNN) отримано з трьома незалежними сідами.

Реалістичний сценарій: атакуючий знає про split

Перший адаптивний атакуючий – DF3_adap – це DF, ексклюзивно навчений на чистих sub-трасах після round-robin split-3, без будь-якого знання про GGI. Він вчиться розпізнавати сайт з одного чистого фрагменту трафіку. Як показано в підрозділі 4.3, незахищений Traffic Split проти нього повністю провалився (2.92% samp_fool). Rand-GGI + Split-3 видає 90.07% samp_fool. Це означає, що рандомізація позицій dummy ефективно позбавляє DF3_adap сталого патерну навіть на рівні окремих фрагментів: DF читає DIR-

послідовність, тобто модель вразлива до позиційної рандомізації. DF3_adap на повних чистих трасах має лише 1.42% точності (98.58% fool) – він оптимізований під sub-траси, а не повні записи. На захищених фрагментах circ_acc падає до 6.72% – рандомізація працює з позиційним атакуючим на відмінному рівні.

Другий атакуючий у цьому сценарії – VarCNN_adap, навчений на чистих DT2 sub-трасах (best val F1 = 0.9216). Проти Rand-GGI + Split-3 він дає circ_acc = 24.02%, або 67.47% samp_fool – гірше ніж DF3_adap (90.07%). Різниця пояснюється ознакою DT2: крім напрямку пакетів, VarCNN читає міжпакетні інтервали (Inter-Packet-Interval, IPI). Rand-GGI вставляє dummy з IPI ≈ 0 (timestamp = timestamp попереднього пакету), що створює характерний вплив в IPI-каналі: там, де поставлено dummy, IPI = 0. Такі нульові спайки не притаманні чистим sub-трасам і VarCNN_adap зберігає часткову класифікаційну здатність попри ці аномалії – IPI=0 спайки заважають класифікації, але не повністю стирають сайт-специфічний сигнал у DT2. Саме це, витік через IPI-канал, було виявлено як окрему вразливість і стало мотивацією для Burst-GGI та timing-вектора, що розглянутий у наступному абзаці.

Повністю адаптивний атакуючий: тренування на захищеному трафіку

Найбільш об'єктивне тестування – коли атакуючий навчається на самому захищеному трафіку з online-аугментацією. Такий сценарій дає критичну для захисту оцінку, що є одним з найгірших сценаріїв – worst case (табл. 4.13).

Таблиця 4.13 – Rand-GGI + Split-3 проти адаптивних атакуючих (усі протоколи)

Атакуючий	Протокол	circ_acc	samp_fool	Статус
DF3_adap	split-aware (чистий трафік)	6.72%	90.07%	Захист ефективний
VarCNN_adap	split-aware (чистий трафік)	24.02%	67.47%	Достатньо за кількісним критерієм з розділу 3
Holmes_adap	fully adaptive (захищений трафік, TAF)	72.79%	17.30%	Захист не ефективний

Продовження таблиці 4.13

DF_adap (online-aug)	захищений трафік (DIR)	~71.65%	11.28% *	Захист не ефективний
VarCNN_adap (online-aug)	захищений трафік (DT2)	~94.58%	4.77%	Захист не ефективний

Коли атакуючий бачить захищений трафік з online-аугментацією, результати погіршуються драматично. DF_adap circ_acc ~71.65% – випадковість позицій погіршує класифікацію DF, але не повністю. 71% circ_acc означає, що більше половини фрагментів атакуючий класифікує правильно (при умові компрометації одного з 3-х вузлів).

VarCNN_adap circ_acc ~94.58% – навчаючись на захищеному трафіку, VarCNN стає інваріантним до рандомізації позицій і опирається на інші ознаки – зокрема, IPI ≈ 0 в позиціях dummy.

Holmes_adap (навчений на TAF-ознаках захищеного трафіку): samp_fool = 17.30%. Це найвище значення серед повністю адаптивних, але не через ефективність захисту, а через catastrophic forgetting, про що нижче.

Чому Holmes в worst-case ефективніше – catastrophic forgetting

Цікавий і важливий ефект, спостережений у всіх адаптивних атакуючих – “катастрофічне забування”: перенавчаючись на захищеному трафіку, атакуючий втрачає здатність класифікувати чистий, не змінений трафік. Holmes, який генерує 98.35% на чистому трафіку, після адаптації зберігає лише 22.17% на чистому. Це не є перевагою захисту в реальному сценарії, адже реальний Guard-вузоль бачить трафік багатьох користувачів, більшість з яких не використовують захист – атакуючий на практиці не може собі дозволити таке вибіркове застосування моделі. Holmes_adap дає 17.30% samp_fool на Rand-GGI+Split-3 – це не означає, що захист “утримує” 17%: circ_acc = 72.79% означає, що більше 70% окремих фрагментів класифікується правильно – захист проти Holmes_adap фактично не працює, але в минулому підрозділі 4.3 було визначено помірну ефективність проти не адаптованого Holmes, що задовольняє кількісний критерій описаний у розділі 3.1.

Чому рандомізація позицій не захищає від Holmes

Holmes використовує Time-Aggregated Features (TAF): кількість вихідних пакетів у кожному часовому вікні (~40 мс). Rand-GGI вставляє рівно N = 500 dummy в кожну трасу – розподілених по перших ~40% траси, але в цілому – однаково для всіх трас цього сайту. Скільки dummy потрапляє у кожне вікно визначається вибіркою позицій, але в середньому цей розподіл стабільний (~500/W_active dummy на активне вікно Holmes). Holmes_adap бачить цей стабільний "зсув" лічильників та навчається його компенсувати, діставшись до внутрішнього патерну сайту. Рандомізація позицій не знижує взаємну інформацію I(n_w; site) між кількістю пакетів в вікні і міткою сайту. отже, TAF-агрегат залишається класово-характеристичним. Це відрізняє Holmes від DF, що читає послідовність (вразливу до позиційної рандомізації), Holmes – агрегат (інваріантний до позицій).

Ensemble пул: чи допомагає пул на основі VarCNN

Окремо було перевірено гіпотезу, не менш важливу та цінну для дослідження: чи змінить ситуацію пул, побудований на градієнтах VarCNN або їх поєднанні з DF.

Вимірювання перекриття пулів виявило, що 1328 з 1500 позицій спільні для DF і VarCNN – 88.5% перекриття. Обидва атакуючі чутливі до одних і тих самих позицій – початку траси, де патерни найхарактерніші. У коректному протоколі (TRUE ADAPTIVE, Exp 7 PROPER – атакуючий тренується на захищеному трафіку з online-аугментацією) VarCNN-пул (α=0.0) дає 3.78% samp_fool проти VarCNN_adap, тоді як стандартний Rand-GGI – 4.77%. Тобто ensemble-пул виявився гіршим на 0.99 в.п., а не кращим. Аналіз показав, що проблема VarCNN_adap не в позиціях пулу, а в самому характері dummy – IPI ≈ 0 є ознакою, яку VarCNN_adap навчається виявляти незалежно від того, в яких позиціях стоять dummy.

Таким чином, експериментальним шляхом було виявлено чітку межу методу. Проти split-aware атакуючого (DF3_adap) – 90.07% samp_fool, це робоча точка. Проти VarCNN_adap того самого протоколу – 67.47%, що вказує на витік

IPI-каналу. Проти повністю адаптивного worst-case (online-aug): DF_adap circ_acc ~71.65%, VarCNN_adap ~94.58%, Holmes_adap 72.79% – захист провалюється за всіма трьома. Наступний абзац ставить за ціль подолання цих обмежень.

Спроби подолати межу проти Holmes: Poisson-GGI та Temporal Jitter

Встановивши, що рандомізація позицій не впливає на TAF-агрегат, дослідження перейшло до двох нових гіпотез, спрямованих безпосередньо на руйнування стабільності лічильників в часових вікнах. Обидві виявились невдалими (табл. 4.14), але через різні причини, що само по собі є інформативним результатом.

Poisson-GGI переходить від фіксованої кількості dummy до випадкової: для кожного TAF-вікна кількість $n_w \sim \text{Poisson}(\lambda_w)$, де λ_w = n_insertions/W_pool(trace) – кількості вікон 40 мс, у які потрапляють позиції пулу. Це повинно було створити дисперсію в per-window count та запобігти тому, щоб Holmes_adap навчався стабільному зсуву лічильників. Проте параметр λ_w = n_insertions/W_pool(trace) залежить від часових характеристик самої траси – кількість активних вікон W_pool різна для різних сайтів, бо часовой профіль завантаження є класо-специфічним. Отже, адаптивний Holmes навчається різним середнім λ_w для різних класів – і це дає йому навіть більше інформації, ніж базовий Rand-GGI.

Таблиця 4.14 – Порівняння спроб подолати Holmes_adap

Захист	Адаптивний атакуючий	samp_fool	circ_acc	Результат
Rand-GGI+Split-3	Holmes_adap	17.30%	72.79%	Базовий рівень
Poisson-GGI+Split-3	Holmes_poisson_adap	10.51%	81.88%	Гірше за базовий
Jitter+Rand-GGI+Split-3	Holmes_jitter_adap	19.38%	70.16%	Базовий рівень

Poisson-GGI показав 10.51% samp_fool – гірше за базовий Rand-GGI. Пояснюється подібний результат тим, що Poisson-варіація count-per-window центрується навколо класо-специфічного середнього. Holmes_adap навчається

цьому середньому – тобто цей підхід не вилучає інформацію про сайт з TAF, а лише змінює форму її представлення.

Значущим побічним висновком став cross-defense ефект: Rand-GGI+Split-3 проти Holmes_poisson_adap (навченого на іншому захисті) дає 30.63% – на 13.33 p.p. вище базових 17.30%. Це підтверджує, що атакуючий, навчений на одному типі захисту, гірше розпізнає інший.

Temporal Jitter (~+20 мс) додає випадкову константу зсуву до всіх timestamps траси, щоб пакети переходили між TAF-вікнами. Однак результат (19.38% samp_fool) практично ідентичний базовому (17.30%): +2.08 p.p. в межах шуму. Побічне вимірювання підтверджує причину: Rand-GGI+Split-3 (без jitter) проти Holmes_jitter_adap дає 19.36% – тобто Jitter+Rand-GGI+Split-3 = 19.38% від атакуючого, який бачив цей захист. Таким чином, сам jitter додає рівно 0.02 p.p.. Причиною подібному результату є одна константа Δt для всієї траси, що зберігає відносну часову структуру незмінною: $|t_j - t_i| = |t'_j - t'_i|$ для будь-яких i, j. TAF-патерни всередині вікон – переходи burst, розподіл пакетів всередині – залишаються незмінними, що бачить Holmes_jitter_adap.

Burst-GGI: маскування IPI та різноманіття

Паралельно з подоланням Holmes досліджувалась вразливість Rand-GGI, що полягала IPI ≈ 0 у dummy-пакетах. Перевірка показала, що частка IPI ≈ 0 у чистому трафіку становить 0.1423, а після Rand-GGI зростає до 0.1982. Burst-GGI намагається вставляти dummy в реальні паузи всередині вихідних burst'ів з середнім IPI ≈ 41.92 мс – натуральне значення замість аномального нуля. IPI sanity check підтвердив, що Burst-GGI успішно знижує частку нульових IPI до 0.1457. Мета була досягнута. Водночас, через менший пул (185 замість 500 dummy), Burst-GGI дещо поступається Rand-GGI навіть проти неадаптивних атакуючих: −0.69 p.p. для DF (97.29% → 96.60%) та −2.24 p.p. для VarCNN (96.47% → 94.23%), але це залежить від бюджету збурень – збільшивши його, покращимо samp_fool.

Cross-defense результати – коли атакуючий навчався на Rand-GGI, а тестувався на Burst-GGI – були вражаючими: samp_fool зростав з 4.77% до 35.72% для

VarCNN_adap і з 12.01% до 54.59% для DF_adap. це підтвердило гіпотезу: атакуючий, вивчений на IPI≈0, бачить менше статистичних закономірностей чим при IPI≈42мс.

Справжній адаптивний тест (атакуючий навчався саме на Burst-GGI-захищеному трафіку) виявив ще гірший результат: геометричне середнє samp_fool = 4.35%, тоді як Rand-GGI дає 7.33% (джерело: Exp7 TRUE ADAPTIVE – 11.28% для DF, 4.77% для VarCNN; в Exp8 cross-defense таблиці базовий Rand-GGI vs DF_adap виміряно як 12.01%, різниця між запусками ~0.73 п.п.). Burst-GGI поступився Rand-GGI. Варто зазначити, що навчання VarCNN_burst_adap було зупинено на epoch 5 з 20 запланованих (best checkpoint – epoch 3, F1=0.9406) через обчислювальні обмеження; повністю натренована модель, ймовірно, дала б ще нижчий samp_fool для Burst-GGI.

Таблиця 4.15 – Порівняння Burst-GGI і Rand-GGI проти власних адаптивних атакуючих

Захист	Атакуючий	circ_acc	samp_fool	Результат
Burst-GGI+Split-3	DF_burst_adap	84.98%	4.79%	Погано
Burst-GGI+Split-3	VarCNN_burst_adap	93.65%	3.96%	Погано
Rand-GGI+Split-3	DF_rand_adap	≈71.65%	11.28%	Краще за рахунок Cross-defense
Rand-GGI+Split-3	VarCNN_rand_adap	≈94.58%	4.77%	Краще за рахунок Cross-defense

Причиною подібного результату є пул, адже у Burst-GGI він містить лише 185 позицій (проти 1500 у Rand-GGI), локалізованих у діапазоні 2–316. Оскільки 185 ≤ n_insertions = 500, кожна траса отримує однаковий набір dummy – детермінований патерн «185 зайвих outgoing пакетів у позиціях 2–316». DF_burst_adap знаходить цей патерн через DIR-канал і досягає val F1 = 0.89 – навіть вище, ніж DF_rand_adap ≈ 0.74, бо детермінований патерн легше вивчити. Burst-GGI замінив IPI=0 на іншу – кластер IPI ≈ 42 мс у позиціях 2–316.

У ході експерименту було отримано важливі дані, що показали на скільки різноманіття збурення важливіше за натуральність IPI. Rand-GGI: IPI=0 (аномальний) + C(1500, 500) різноманіття → захищає DF. Burst-GGI: IPI=42мс (натуральний) + 1 комбінація → не захищає нікого.

Фундаментальна межа клієнтського захисту

Пять незалежних спроб – Poisson-GGI, Temporal Jitter, Burst-GGI, Ensemble pool, та зміна бюджету на найдискримінативніші позиції не подолали адаптивного worst-case атакуючого.

У всіх перелічених спроб є спільна структурна причина – взаємна інформація $I(n_w; site)$ між кількістю пакетів у вікні та міткою сайту залишається високою. Для інформаційно стійкого захисту потрібно $I(n_w; site) \to 0$, що вимагає або вирівнювання лічильників до спільної стелі (constant-rate), що в свою чергу порушує вимогу низького overhead, або руйнування самих burst-меж через клас-незалежне впровадження фіктивних burst – це залишається невипробуваним напрямком (Rand-BBD).

Це є головним результатом розділу 4.4. Rand-GGI + Traffic Split N=3 є найкращим практичним низько-overhead захистом і його межа проти worst-case адаптивного атакуючого є не дефектом реалізації, а фундаментальним наслідком обмежень клієнтського insertion-only захисту з низьким overhead.

Проти реалістичного split-aware атакуючого (DF3_adap) метод забезпечує 90.07% samp_fool – це фінальна точка для завершення дослідження. Проти повністю адаптованого (online-aug) – це невирішувана проблема в заявленому класі захистів, що потребує глибшого вивчення.

4.5 Висновок до розділу 4

У цьому підрозділі узагальнено результати експериментів, описаних у підрозділах 4.3 та 4.4, та обґрунтовано фінальне рішення щодо запропонованого методу захисту. Порівняння виконано за двома осями: ефективність базової конфігурації проти атакуючих різного типу та стійкість методу у трьох

протоколах адаптації, які моделюють реалістичний і граничний (worst-case) рівні тиску атакуючого.

Базова конфігурація Rand-GGI + Traffic Split N = 3 при overhead 10 % дає найкращі показники серед усіх протестованих захистів. Результати зведено у таблиці 4.16.

Таблиця 4.16 – Ефективність захистів проти неадаптивних атакуючих та реалістичного split-aware DF3_adap (samp_fool, %, вище – краще для захисту)

Конфігурація захисту	**DF**	**Var-CNN**	**Holmes**	**DF3_adap (split-aware)**
Без захисту	2.24 %	1.98 %	1.65 %	1.42 %
Static GGI	89.13 %	92.73 %	—	1.55 %
Rand-GGI	95.04 %	96.32 %	93.87 %	1.13 %
Traffic Split N=3	96.36 %	92.01 %	83.22 %	2.92 %
Rand-GGI + Split-3	**99.01** %	**96.47** %	**96.91** %	**90.07** %

Перевага комбінованого методу над окремими компонентами підтверджена кількісно: Rand-GGI + Split-3 перевершує самостійний Rand-GGI на 3.97 п.п. проти DF та на 3.04 п.п. проти Holmes, а самостійний Traffic Split N = 3 – на 2.65 п.п. проти DF та на 13.69 п.п. проти Holmes. Принципова перевага комбінованого методу видно проти DF3_adap: 90.07 % samp_fool проти 2.92 % у самостійного Split-3 – це у тридцять разів вищий показник. Відсутність результату для Static GGI проти Holmes пояснюється недоцільністю проведення експерименту через апріорі низькі очікувані результати при значних обчислювальних витратах.

Оцінку ефективності в умовах, наближених до реальних, виконано у трьох протоколах адаптації атакуючого, які зведено у таблиці 4.17. Реалістичний split-aware протокол є основною робочою точкою методу. Worst-case online-aug протокол, коли атакуючий має повний доступ до самого захищеного трафіку, задає верхню межу можливого тиску атакуючого.

Таблиця 4.17 – Стійкість Rand-GGI + Split-3 у трьох протоколах адаптації атакуючого

Атакуючий	**Протокол навчання**	**Ознака**	**circ_acc / samp_fool, %**	**Реалістичність**
DF3_adap	split-aware (без GGI)	DIR	samp_fool 90.07	реалістичний
Var-CNN_adap	split-aware (без GGI)	DT2	samp_fool 67.47	проміжний
DF_adap	online-aug (GGI+Split)	DIR	circ_acc 71.65 ± 0.17	worst-case
Var-CNN_adap	online-aug (GGI+Split)	DT2	circ_acc 94.58 ± 0.67	worst-case
Holmes_adap	online-aug (GGI+Split)	TAF (агрегат)	samp_fool 17.30	worst-case

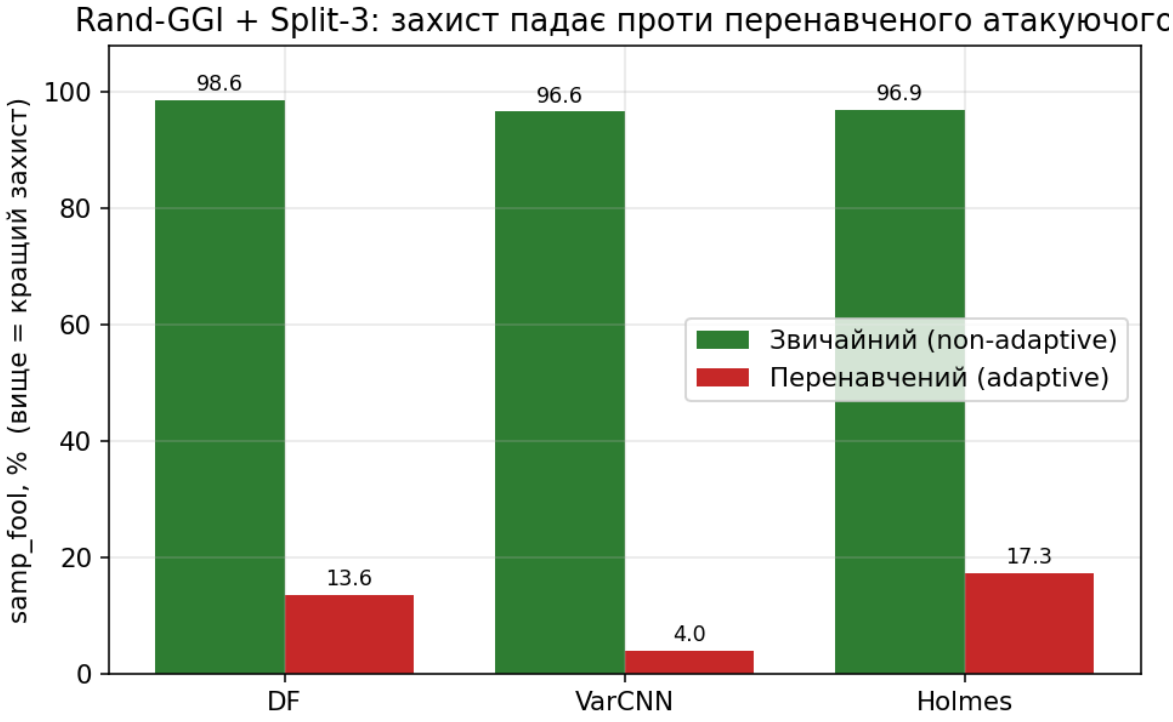

Рисунок 4.1 – Контраст non-adaptive vs adaptive

Інтерпретація результатів однозначна. Проти реалістичного split-aware атакуючого метод дає 90.07 % samp_fool – це робоча точка для практичного розгортання. Проти Var-CNN_adap у проміжному протоколі ефективність падає до 67.47 % через витік ознаки DT2 (timing-канал). Проти повністю адаптивного Var-CNN_adap у worst-case протоколі circ_acc досягає 94.58 ± 0.67 %. Метод

впирається у фундаментальну межу: випадковість позицій ускладнює класифікацію DF (circ_acc 71.65 %), але не впливає на агрегатні TAF-ознаки, які залишаються незмінними при перестановках dummy-пакетів.

Для подолання цієї межі експериментально перевірено п'ять модифікацій методу. Усі вони виявились невдалими з однієї спільної причини – позиційна рандомізація не зменшує взаємну інформацію I(n_w; site) між агрегатними ознаками та міткою сайту. Підсумок перевірок наведено у таблиці 4.18.

Таблиця 4.18 – Перевірені модифікації для подолання межі та причини їх відмови

Модифікація	**Ідея**	**Причина відмови**	**Результат**
Burst-GGI	Вставка dummy у природні паузи між burst (IPI ≈ 42 мс)	Пул має лише 185 позицій у діапазоні 2–316 проти 1500 у Rand-GGI; такого різноманіття замало, тому атакуючий вивчає патерн	geo_mean 4.35 % (гірше за Rand-GGI 7.33 %)
Poisson-GGI	Рандомізація кількості dummy на TAF-вікно за Пуассоном	Параметр λ_w виходить різний для різних сайтів, тому Holmes_adap вивчає це середнє і впізнає сайт	samp_fool 10.51 %
Temporal Jitter	Константний зсув усіх таймстемпів на Δt	Відносна часова структура траси не змінюється, а TAF дивиться саме на неї	samp_fool 19.38 %
Ensemble Pool	Об'єднання градієнтів DF та Var-CNN у спільний пул	Позиції DF і Var-CNN перекриваються на 88.5 %, тому пул майже не змінюється	Δ –0.70 p.p. до Rand-GGI
opt-budget	Перенесення бюджету у найбільш дискримінативні позиції	Атакуючий дивиться на агрегати, а не на конкретні позиції, тому такий перерозподіл нічого не дає	vs Var-CNN_adap 94.93 %, vs DF_adap 87.57 %

Спільний для всіх п'яти модифікацій висновок такий: у класі клієнтських insertion-only захистів без затримок проти worst-case адаптивного атакуючого, що читає агрегатні ознаки, межа є невирішуваною (або вкрай складною для реалізації) у межах заявлених обмежень. Окреслення цієї межі через критерій I(n_w; site) → 0 є самостійним науковим результатом роботи.

Обмеження запропонованого методу зафіксовано експериментально. Метод не дає стійкості проти worst-case адаптивного атакуючого, що читає TAF-агрегати (Holmes_adap samp_fool 17.30 %, Var-CNN_adap circ_acc 94.58 %). Чотири з п'яти модифікацій, які мали подолати цю межу, дали ефективність гіршу за базову конфігурацію. Метод протестовано лише в режимі closed-world на одному датасеті CW і при фіксованому overhead 10 % – перевірка для open-world сценарію та інших бюджетів overhead винесена у майбутню роботу.

Напрямки подальших досліджень визначено на основі виявлених обмежень і зведено у таблиці 4.19.

Таблиця 4.19 – Напрямки майбутніх досліджень з пріоритезацією

Напрямок	**Зміст та обґрунтування**	**Очікуваний ефект**
Rand-BBD (Randomized Burst Boundary Disruption)	Замість рандомізації окремих позицій – рандомізація на рівні burst-переходів: вставка пар (+1, –1) у випадкові вікна, де реальних burst немає. Метод атакує саме статистику n_bursts[w], яка у всіх п'яти попередніх модифікаціях залишалась клас-специфічною. Реалізація на рівні коду готова (defenses/rand_bbd.py), масштабне тестування відкладено через нестачу обчислювальних ресурсів (4 ГБ GPU).	Прямо атакує TAF-агрегат; за теоретичною оцінкою – samp_fool вище 17.30 % проти Holmes_adap
Count-equalization (constant-rate)	Підняття лічильника n_packets[w] до однієї спільної стелі для всіх класів. Це єдиний теоретично надійний спосіб опустити взаємну інформацію I(n_w; site) до нуля.	Гарантована стійкість проти worst-case, але overhead ≥ 30 % – виходить за межі дешевого захисту для Tor
Чутливість до бюджету overhead	Перевірка методу на overhead {5 %, 10 %, 15 %, 20 %} замість фіксованих 10 % – побудова кривої overhead ↔ fooling.	Уточнення оптимальної робочої точки
Open-world оцінка	Тестування на датасеті OW (відкритий світ із фоновим трафіком) – доповнення метрик closed-world показниками Precision та Recall.	Оцінка частоти хибно-позитивних спрацювань атакуючого

Продовження таблиці 4.19

Емпіричний вимір у реальному Tor	Перенесення методу з симуляції у testbed мережі Tor, щоб напряму виміряти затримки та overhead при реальній маршрутизації.	Підтвердження придатності до розгортання

Першочерговий напрямок – експериментальне тестування Rand-BBD, єдиної теоретично обґрунтованої гіпотези, що атакує не статистику n_packets, а статистику n_bursts: у випадкові часові вікна додаються фіктивні burst-переходи незалежно від класу. Реалізація методу на рівні коду готова, повноцінне тестування потребує більших обчислювальних ресурсів. Альтернативний шлях – count-equalization – теоретично гарантує стійкість, але виходить за межі дешевого захисту через overhead понад 30 %, а це суперечить вимозі придатності до перевантаженої мережі Tor.

Фінальне рішення щодо методу таке: для практичного розгортання обрано базову конфігурацію Rand-GGI + Traffic Split N = 3, що при overhead 10 % забезпечує 96–99 % samp_fool проти неадаптивних атакуючих DF, Var-CNN, Holmes та 90.07 % samp_fool проти реалістичного split-aware DF3_adap – це найкращий практичний показник серед розглянутих захистів даного класу. Метод придатний до розгортання як додаткове клієнтське програмне забезпечення для мережі Tor без модифікації існуючих Tor-вузлів. Виявлена фундаментальна межа проти worst-case адаптивного атакуючого, експериментально підтверджена п'ятьма незалежними модифікаціями, окреслює простір клієнтських insertion-only захистів і задає вектор подальших досліджень.

ЗАГАЛЬНІ ВИСНОВКИ

Досліджено сучасні види атак на основі аналізу трафіку у мережі Tor. Встановлено, що найбільшу загрозу складають атаки класу Website Fingerprinting у реалізаціях Deep Fingerprinting, Var-CNN та Holmes, які на незахищеному трафіку демонструють точність 97–98 %. Проаналізовано еволюцію цих атак від класичних статистичних методів до моделей глибокого навчання, що автоматично виявляють складні нелінійні просторово-часові закономірності у мережевих потоках. Підтверджено, що велика кількість вузлів Tor наразі контролюється, і контроль дає технічну можливість застосовувати ці атаки до перехопленого трафіку реальних користувачів.

Проаналізовано існуючі рішення забезпечення анонімності в мережі Tor: Camouflage, BuFLO, CS-BuFLO, WTF-PAD, Walkie-Talkie, FRONT, TrafficSliver, A2A, ALERT та RUDOLF. Виявлено системну закономірність – ускладнення моделей атак провокує симетричну відповідь у розвитку захистів, проте жодне з розглянутих рішень одночасно не задовольняє вимоги високої ефективності проти DL-атак, низького overhead до 30 % та відсутності значних додаткових затримок. Показано, що однокомпонентні захисти втрачають ефективність проти сучасних нейромережевих класифікаторів. Цей висновок став теоретичним обґрунтуванням принципу взаємодоповнюваності – основи запропонованого модифікованого методу.

Модифіковано метод підвищення рівня анонімності користувача в мережі Tor, що поєднує багатошляхову передачу та генерацію змагального шуму. Розроблено гібридний пайплайн з трьома архітектурними рішеннями: винесення обчислення збурення в офлайн (універсальний вектор ≈ 6 КБ замість GPU на стороні клієнта); встановлення порядку «збурення -> розподіл», що утримує overhead на рівні 10 % замість 30 % при зворотному порядку; введення єдиного бюджету B, що визначає силу збурення та накладні витрати. Метод працює у дві фази: підготовча обчислює пул чутливих позицій P; захисна вставляє 500 dummy-пакетів у випадково обрану підмножину пулу $|P| = 1500$ та розподіляє трасу між трьома Tor-ланцюгами.

Оцінено ефективність запропонованого рішення на стандартному датасеті CW (95 класів, 10 573 тестових траси) проти трьох сучасних атакуючих моделей у трьох протоколах адаптації з трьома незалежними сідами. Базова конфігурація Rand-GGI + Traffic Split N = 3 при overhead 10 % забезпечує 96–99 % samp_fool проти неадаптивних атакуючих та 90.07% проти реалістичного split-aware DF3_adap – найкращий практичний показник серед захистів цього класу. Експериментально виявлено фундаментальну межу клієнтських insertion-only захистів проти worst-case адаптивних атакуючих, що читають агрегатні TAF-ознаки, та окреслено напрямки подальших досліджень – Rand-BBD і техніки вирівнювання лічильників часових вікон. Мету дипломної роботи досягнуто.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 The Tor Project [Електронний ресурс]. URL: https://www.torproject.org

2 The Tor Project Specifications [Електронний ресурс]. URL: https://spec.torproject.org/

3 Meng, X.; Liang, M. A Traffic Splitting Algorithm for Load Balancing in Tor. Entropy 2022, 24, 807., URL: https://doi.org/10.3390/e24060807

4 Rahalkar C. Analyzing Trends in Tor / C. Rahalkar, A. Virgaonkar, K. Varadan // arXiv:2208.11149 [cs.CR]. 2024. URL: https://arxiv.org/abs/2208.11149

5 Tor Overview [Електронний ресурс]. URL: https://www.privacyguides.org/en/advanced/tor-overview/

6 Tor Security Advisory: "Relay Early" Traffic Confirmation Attack [Електронний ресурс] / The Tor Project. URL: https://blog.torproject.org/tor-security-advisory-relay-early-traffic-confirmation-attack/

7 Was threat actor KAX17 de-anonymizing the Tor network? [Електронний ресурс] / Malwarebytes. 2021. URL: https://www.malwarebytes.com/blog/news/2021/12/was-threat-actor-kax17-de-anonymizing-the-tor-network

8 Is KAX17 Performing De-Anonymization Attacks against Tor Users? [Електронний ресурс] / nusenu. URL: https://nusenu.medium.com/is-kax17-performing-de-anonymization-attacks-against-tor-users-42e566defce8

9 Теоретичні засади та класичні методи деанонімізації на основі Website Fingerprinting

10 Cui Y. A Comprehensive Survey of Website Fingerprinting Attacks and Defenses in Tor: Advances and Open Challenges / Y. Cui [та ін.] // arXiv:2510.11804 [cs.CR]. 2025. URL: https://arxiv.org/abs/2510.11804

11 Liberatore M. Inferring the Source of Encrypted HTTP Connections / M. Liberatore, B. N. Levine // ACM Digital Library. 2006. URL: https://dl.acm.org/doi/pdf/10.1145/1180405.1180437

12 Herrmann D. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier / D. Herrmann, R.

Wendolsky // Proceedings of the 2009 ACM workshop on Cloud computing security (CCSW '09). New York : ACM, 2009. P. 139–150. URL: https://dl.acm.org/doi/pdf/10.1145/1655008.1655013

13 Panchenko, L. Niessen, A. Zinnen, and T. Engel, “Website fingerprinting in onion routing based anonymization networks,” in Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, 2011, pp. 103–114.

14 Hayes J. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique / J. Hayes, G. Danezis // Proceedings of the 25th USENIX Security Symposium. Austin, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes

15 Panchenko A. Website Fingerprinting at Internet Scale / A. Panchenko [та ін.] // Proceedings of the 23rd ISOC Network and Distributed System Security Symposium (NDSS '16). San Diego, 2016. URL: https://www.ndss-symposium.org/ndss2016/programme/website-fingerprinting-internet-scale/

16 Сучасні методи деанонімізації з використанням глибокого навчання (Deep Learning)

17 Rimmer V. Deep Learning Website Fingerprinting Features : Master’s thesis / Vera Rimmer ; KU Leuven. Leuven, 2017. 87 p. URL: https://cosicdatabase.esat.kuleuven.be/backend/publications/files/these/280

18 What is a convolutional neural network? [Електронний ресурс]. URL: https://cloud.google.com/discover/what-are-convolutional-neural-networks

19 Sirinam P. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning / P. Sirinam, M. Juarez, M. Imani, M. Wright // arXiv:1801.02265 [cs.CR]. 2018. URL: https://arxiv.org/abs/1801.02265

20 Bhat S. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning / S. Bhat, D. Lu, A. Kwon, S. Devadas // Proceedings on Privacy Enhancing Technologies. 2019. Vol. 2019, No. 4. URL: https://www.researchgate.net/publication/336185142_Var-CNN_A_Data-Efficient_Website_Fingerprinting_Attack_Based_on_Deep_Learning

21 Tik-Tok: The Utility of Packet Timing in Website Fingerprinting Attacks / M. S. Rahman, P. Sirinam, N. Mathews [та ін.] // Proceedings on Privacy Enhancing Technologies. 2020. Vol. 2020, No. 3. P. 1–20. URL: https://arxiv.org/pdf/1902.06421

22 Deng X. Robust and Reliable Early-Stage Website Fingerprinting Attacks via Spatial-Temporal Distribution Analysis / X. Deng, Q. Li // arXiv preprint. 2024. arXiv:2407.00918. URL: https://arxiv.org/pdf/2407.00918

23 Захист

24 Dyer, Kevin P., Scott E. Coull, Thomas Ristenpart and Thomas Shrimpton. / "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail." *2012 IEEE Symposium on Security and Privacy* (2012): 332-346. URL: https://www.semanticscholar.org/paper/Peek-a-Boo%2C-I-Still-See-You%3A-Why-Efficient-Traffic-Dyer-Coull/af7887649aaec82c123ea6d140a04e2209fdcf73

25 X. Cai, R. Nithyanand, and R. Johnson, "Cs-buflo: A congestion sensitive website fingerprinting defense," in Proceedings of the 13th Workshop on Privacy in the Electronic Society, 2014, pp. 121–130. URL: https://doi.org/10.1145/2665943.2665949

26 T. Wang and I. Goldberg, "{Walkie-Talkie}: An efficient defense against passive website fingerprinting attacks," in 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 1375–1390. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tao

27 M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright, "Toward an efficient website fingerprinting defense," in Computer Security– ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21. Springer, 2016, pp. 27–46. URL: https://dl.acm.org/doi/abs/10.1016/j.jnca.2025.104125

28 J. Gong and T. Wang, "Zero-delay lightweight defenses against website fingerprinting," in 29th USENIX security symposium (USENIX security 20),

2020, pp. 717–734. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/gong

29 TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting / W. De la Cadena, A. Mitseva, J. Pennekamp [та ін.] // Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20). New York, 2020. P. 1971–1985. URL: https://dl.acm.org/doi/pdf/10.1145/3372297.3423351

30 C. Hou, J. Shi, M. Cui and Q. Yang, "Attack versus Attack: Toward Adversarial Example Defend Website Fingerprinting Attack," *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Shenyang, China, 2021, pp. 766-773, URL: https://ieeexplore-custom.ieee.org/document/9724464

31 Qiao, Litao ; Wu, Bang ; Li, Heng et al. / Trace-agnostic and Adversarial Training-resilient Website Fingerprinting Defense. IEEE International Conference on Computer Communications. 2024. pp. 1-18

32 M. Jiang, B. Cui, J. Fu, T. Wang, L. Yao, and B. K. Bhargava, “Rudolf: An efficient and adaptive defense approach against website fingerprinting attacks based on soft actor-critic algorithm,” IEEE Transactions on Information Forensics and Security, 2024

33 S. -M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi and P. Frossard, "Universal Adversarial Perturbations," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA, 2017, pp. 86-94, doi: 10.1109/CVPR.2017.17, URL: https://ieeexplore.ieee.org/document/8099500 [взято принцип роботи з опису]

34 Xinhao Deng, Qi Li, Ke Xu. *Robust and Reliable Early-Stage Website Fingerprinting Attacks via Spatial-Temporal Distribution Analysis*. ACM CCS 2024, URL: https://github.com/FIND-Lab/Website-Fingerprinting-Library

35 Юшко Д.С., Курдеча В.В. Модифікований метод підвищення рівня анонімності користувача в мережі Tor// Перспективи розвитку інформаційно-телекомунікаційних технологій та систем (ПРІТС-2026) :

тези XVIII наук.-техн. конф. студентів та аспірантів, 13–17 квіт. 2026 р., Київ. – Київ : КПІ ім. Ігоря Сікорського, 2026. – С. 443.

ДОДАТОК А Програмна реалізація модифікованого методу захисту

У додатку А наведено вихідний код ключових модулів програмної реалізації модифікованого методу підвищення анонімності користувача в мережі Tor. Зміст підрозділів відповідає описам у підрозділі 4.2. Реалізація написана мовою Python з використанням бібліотек NumPy та PyTorch і інтегрується з середовищем WFlib (ACM CCS 2024).

А.1 Побудова карти чутливості та пулу позицій

Модуль gradient_pool.py реалізує офлайн-операцію побудови карти чутливості S[i] через градієнти функції втрат по вхідному тензору. Функція build_sensitivity_map усереднює абсолютні значення градієнтів за кілька епох; функція select_top_positions повертає k найчутливіших позицій у відсортованому порядку. Результат зберігається як артефакт обсягом близько 6 КБ і використовується захистом Rand-GGI під час роботи.

```
import numpy as np
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset


def build_sensitivity_map(model, X_feature, y, device, epochs=7, batch_size=256):
    model.eval()
    for p in model.parameters():
        p.requires_grad_(False)

    seq_len = X_feature.shape[-1]
    grad_magnitude = torch.zeros(seq_len, device=device)
    n_batches = 0

    loader = DataLoader(TensorDataset(X_feature, y),
                        batch_size=batch_size, shuffle=True)

    for epoch in range(epochs):
        for x_batch, y_batch in loader:
            x_batch = x_batch.to(device).requires_grad_(True)
            y_batch = y_batch.to(device)

            loss = F.cross_entropy(model(x_batch), y_batch)
            loss.backward()

            # x_batch.grad has shape (B, 1, seq_len); drop the channel axis.
            g = x_batch.grad.squeeze(1)
```

```
            grad_magnitude += g.abs().mean(dim=0)
            n_batches += 1
            x_batch.grad = None

        print(f"  epoch {epoch + 1}/{epochs}  batches so far: {n_batches}",
              flush=True)

    grad_magnitude /= n_batches
    return grad_magnitude.cpu().numpy()


def select_top_positions(grad_magnitude, k):
    """Return the k most sensitive positions, sorted in ascending order."""
    top_idx = np.argsort(grad_magnitude)[::-1][:k]
    return np.sort(top_idx).astype(np.int64)
```

A.2 Реалізація захистів Static GGI та Rand-GGI

Модуль defenses/ggi.py містить реалізацію Gradient-Guided Injection. Функція _insert_outgoing_dummies вставляє dummy-пакети у вказані позиції з обробкою у спадному порядку (справа наліво), що уникає зсуву індексів. apply_static_ggi використовує однаковий набір позицій для всіх трас; apply_rand_ggi для кожної траси формує нову випадкову підмножину з пулу, що нейтралізує adversarial training атакуючого.

```
import numpy as np

EPS_TIMESTAMP = 1e-9   # tiny stand-in so a leading dummy is not trimmed away

def _insert_outgoing_dummies(trace, positions):
    length = len(trace)
    for k in np.sort(positions)[::-1]:
        k = int(k)
        prev_abs = abs(float(trace[k - 1])) if k > 0 else 0.0
        # Same timestamp as the previous packet => IPI 0 (no extra delay).
        dummy = prev_abs if prev_abs > 0 else EPS_TIMESTAMP
        trace = np.concatenate([trace[:k], [dummy], trace[:length - 1][k:]])
    return trace


def apply_static_ggi(X_raw, positions):
    X_out = np.empty_like(X_raw)
    for i in range(len(X_raw)):
        X_out[i] = _insert_outgoing_dummies(X_raw[i].copy(), positions)
        if (i + 1) % 2000 == 0:
            print(f"  static_ggi apply: {i + 1}/{len(X_raw)}", flush=True)
    return X_out


def apply_rand_ggi(X_raw, pool_positions, n_insertions, seed=None):
    rng = np.random.default_rng(seed)
    X_out = np.empty_like(X_raw)
```

```
    for i in range(len(X_raw)):
        sampled = rng.choice(pool_positions, size=n_insertions, replace=False)
        X_out[i] = _insert_outgoing_dummies(X_raw[i].copy(), np.sort(sampled))
        if (i + 1) % 2000 == 0:
            print(f"  rand_ggi apply: {i + 1}/{len(X_raw)}", flush=True)
    return X_out
```

А.3 Реалізація багатошляхового розподілу трафіку

Модуль defenses/traffic_split.py реалізує дві стратегії розподілу трафіку між кількома Tor-ланцюгами. Функція split_round_robin розподіляє пакети послідовно за принципом «кожен N-й пакет — у ланцюг c». Функція split_trafficsliver реалізує BWR-стратегію TrafficSliver: пачки з 1–4 послідовних пакетів передаються випадковому ланцюгу, що ближче до реального розгортання у мережі.

```
import numpy as np


def split_round_robin(X_raw, y, n_circuits=3, seq_len=5000):
    n = len(y)
    X_split = np.zeros((n * n_circuits, seq_len), dtype=np.float32)
    y_split = np.empty(n * n_circuits, dtype=np.int64)

    for i in range(n):
        trace = X_raw[i]
        nonzero = np.flatnonzero(trace)
        active_len = int(nonzero[-1]) + 1 if len(nonzero) else seq_len
        active = trace[:active_len]

        for c in range(n_circuits):
            sub = active[c::n_circuits]
            keep = min(len(sub), seq_len)
            X_split[i * n_circuits + c, :keep] = sub[:keep]
            y_split[i * n_circuits + c] = y[i]

    return X_split, y_split


def split_trafficsliver(X_raw, y, n_circuits=3, seq_len=5000, bwr_max=4,
                        seed=None):
    rng = np.random.default_rng(seed)
    n = len(y)
    X_split = np.zeros((n * n_circuits, seq_len), dtype=np.float32)
    y_split = np.empty(n * n_circuits, dtype=np.int64)

    for i in range(n):
        trace = X_raw[i]
        nonzero = np.flatnonzero(trace)
        active_len = int(nonzero[-1]) + 1 if len(nonzero) else 0

        # Decide a circuit for each packet, one run (batch) at a time.
        assignment = np.empty(active_len, dtype=np.int64)
```

```
        j = 0
        while j < active_len:
            circuit = int(rng.integers(n_circuits))
            run = int(rng.integers(1, bwr_max + 1))
            assignment[j:j + run] = circuit
            j += run

        active = trace[:active_len]
        for c in range(n_circuits):
            sub = active[assignment == c]
            keep = min(len(sub), seq_len)
            X_split[i * n_circuits + c, :keep] = sub[:keep]
            y_split[i * n_circuits + c] = y[i]

    return X_split, y_split
```

A.4 Оптимізація вектора часових збурень

Модуль timing_uap.py обчислює універсальний вектор часових збурень delta методом проєктованого градієнтного піднесення. Delta застосовується лише до вихідних пакетів (клієнт не може прискорювати чужі пакети) і обмежується бюджетом overhead. Оптимізація ведеться лише на трасах, які ще не обдурено, що концентрує градієнтний сигнал на важких прикладах.

```
import numpy as np
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset


def generate_timing_delta(model, X_dt2, y, device,
                          overhead=0.10, epochs=10, alpha=0.002, batch_size=64):
    model.eval()
    for p in model.parameters():
        p.requires_grad_(False)

    seq_len = X_dt2.shape[-1]
    delta = torch.zeros(seq_len, device=device)
    loader = DataLoader(TensorDataset(X_dt2, y),
                        batch_size=batch_size, shuffle=True)

    for epoch in range(1, epochs + 1):
        fooled_n, used_n = 0, 0

        for x_batch, y_batch in loader:
            x_batch = x_batch.to(device)
            y_batch = y_batch.to(device)

            direction = x_batch[:, 0:1, :]
            ipi = x_batch[:, 1:2, :]
            outgoing = (x_batch[:, 0, :] > 0).float()

            # Apply the current delta only to outgoing packets.
```

```
            scale = 1.0 + delta.view(1, 1, -1) * outgoing.unsqueeze(1)
            x_adv = torch.cat([direction, ipi * scale], dim=1)

            with torch.no_grad():
                preds = model(x_adv).argmax(1)
                fooled = preds != y_batch
            fooled_n += int(fooled.sum())
            used_n += len(y_batch)

            # Only keep optimizing on the traces we have NOT fooled yet.
            keep = ~fooled
            if int(keep.sum()) == 0:
                continue

            x_keep = x_batch[keep]
            y_keep = y_batch[keep]
            out_keep = (x_keep[:, 0, :] > 0).float()
            ipi_keep = x_keep[:, 1:2, :]

            ipi_adv = (ipi_keep * (1.0 + delta.view(1, 1, -1) *
                                   out_keep.unsqueeze(1)))
            ipi_adv = ipi_adv.detach().requires_grad_(True)
            x_adv_keep = torch.cat([x_keep[:, 0:1, :], ipi_adv], dim=1)

            # Ascend the loss (note the minus sign before backward()).
            (-F.cross_entropy(model(x_adv_keep), y_keep)).backward()

            grad_delta = (ipi_adv.grad[:, 0, :] *
                          ipi_keep[:, 0, :] * out_keep).mean(dim=0)
            # One signed step, then project back into the [0, overhead] budget.
            delta = torch.clamp(
                (delta + alpha * grad_delta.sign()).detach(), 0.0, overhead)

        fooling = fooled_n / used_n * 100.0
        d = delta.cpu().numpy()
        print(f"  epoch {epoch:2d}/{epochs}  fooling={fooling:.1f}%"
              f"  nonzero={(d > 1e-6).sum()}/{seq_len}"
              f"  max={d.max() * 100:.1f}%", flush=True)

    return delta.cpu().numpy()
```

A.5 Застосування часових збурень

Модуль defenses/timing.py застосовує збережений вектор delta до тестових трас. Реалізація використовує форвардні різниці IPI, оскільки саме так DT2-ознака читає часові проміжки. Абсолютні timestamps відновлюються кумулятивною сумою збурених інтервалів, що гарантує монотонність результуючих міток часу.

```
import numpy as np


def apply_timing(X_raw, delta):
```

```
    n_rows, length = X_raw.shape
    abs_t = np.abs(X_raw)

    # Forward inter-packet intervals, shape (N, L-1).
    ipi = np.diff(abs_t, axis=1)
    ipi = np.clip(ipi, 0.0, None).astype(np.float32)

    # A delay applies to interval i only if the later packet (i+1) is outgoing.
    outgoing_next = (X_raw[:, 1:] > 0).astype(np.float32)
    scale = 1.0 + delta[:length - 1] * outgoing_next
    new_ipi = ipi * scale

    # Rebuild absolute timestamps: keep t[0], then cumulative-sum the intervals.
    first = abs_t[:, :1]
    rebuilt = first + np.cumsum(new_ipi, axis=1)
    new_abs = np.concatenate([first, rebuilt], axis=1)

    return (np.sign(X_raw) * new_abs).astype(np.float32)
```

A.6 Система метрик оцінки ефективності

Модуль metrics.py обчислює систему з п'яти метрик, описаних у таблиці 4.6. Функція predict виконує батчевий інференс у режимі model.eval() без обчислення градієнтів. Функція accuracy_and_fooling — для несплітованого трафіку, circuit_metrics — для конфігурацій з розділенням на N ланцюгів. Основна метрика circ_acc відповідає реалістичному сценарію спостереження за одним Guard-вузлом.

```
import numpy as np
import torch

from .data_io import to_feature_tensor

@torch.no_grad()
def predict(model, X_tensor, device, batch_size=256):
    """Run the model over a feature tensor and return predicted labels."""
    model.eval()
    preds = np.empty(len(X_tensor), dtype=np.int64)
    for i in range(0, len(X_tensor), batch_size):
        batch = X_tensor[i:i + batch_size].to(device)
        preds[i:i + batch_size] = model(batch).argmax(1).cpu().numpy()
    return preds


def accuracy_and_fooling(model, X_raw, y, feature, device, batch_size=256):
    X_tensor = to_feature_tensor(X_raw, feature)
    preds = predict(model, X_tensor, device, batch_size)
    acc = float((preds == y).mean()) * 100.0
    return {"acc": round(acc, 2), "fool": round(100.0 - acc, 2)}


def circuit_metrics(model, X_split, y_split, n_circuits, feature, device,
```

```
                batch_size=256):
    X_tensor = to_feature_tensor(X_split, feature)
    preds = predict(model, X_tensor, device, batch_size)
    correct = (preds == y_split)

    circ_acc = float(correct.mean()) * 100.0

    n_orig = len(y_split) // n_circuits
    correct_per_visit = correct.reshape(n_orig, n_circuits)
    sample_acc = float(correct_per_visit.any(axis=1).mean()) * 100.0

    return {
        "circ_acc": round(circ_acc, 2),
        "circ_fool": round(100.0 - circ_acc, 2),
        "sample_acc": round(sample_acc, 2),
        "samp_fool": round(100.0 - sample_acc, 2),
    }
```

А.7 Скрипт застосування захисту до трафіку

Скрипт 3_apply_defense.py є диспетчером: він приймає параметр --defense і викликає відповідну функцію з модуля defenses. Принциповим архітектурним рішенням є роздільне застосування збурення та split: скрипт виробляє захищену трасу у форматі .npz, а розділення між ланцюгами виконується на етапі оцінки.

```
import argparse
import os

import _bootstrap  # noqa: F401
import numpy as np
import torch

from wf_defense import data_io, defenses


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--defense", required=True,
                        choices=["rand_ggi", "static_ggi", "timing", "combined",
                                 "poisson_ggi", "jitter", "rand_bbd", "burst_ggi"])
    parser.add_argument("--in_file", required=True, help="raw .npz input traces")
    parser.add_argument("--out_file", required=True, help="raw .npz output path")
    parser.add_argument("--pool", default="vectors/rand_pool.pt",
                        help="rand_pool.pt (for rand_ggi / poisson_ggi)")
    parser.add_argument("--dir_plan", default="vectors/dir_plan.pt",
                        help="dir_plan.pt (for static_ggi)")
    parser.add_argument("--timing_vec", default="vectors/timing_delta.npz",
                        help="timing_delta.npz (for timing / combined)")
    parser.add_argument("--burst_pool", default="vectors/burst_pool.pt",
                        help="burst_pool.pt (for burst_ggi)")
    parser.add_argument("--n_insertions", type=int, default=500)
    parser.add_argument("--seq_len", type=int, default=5000)
    parser.add_argument("--seed", type=int, default=42)
    args = parser.parse_args()
```

```
    print(f"Loading {args.in_file} ...")
    X_raw, y = data_io.load_raw(args.in_file, args.seq_len)
    print(f"  {len(y)} traces, seq_len={args.seq_len}")

    print(f"Applying defense: {args.defense}")
    if args.defense == "rand_ggi":
        pool = torch.load(args.pool, map_location="cpu", weights_only=False)
        X_def = defenses.apply_rand_ggi(X_raw, pool["positions"],
                                        args.n_insertions, seed=args.seed)
    elif args.defense == "static_ggi":
        plan = torch.load(args.dir_plan, map_location="cpu", weights_only=False)
        X_def = defenses.apply_static_ggi(X_raw, plan["positions"])
    elif args.defense == "timing":
        delta = np.load(args.timing_vec)["delta"].astype(np.float32)
        X_def = defenses.apply_timing(X_raw, delta[:args.seq_len])
    elif args.defense == "combined":
        delta = np.load(args.timing_vec)["delta"].astype(np.float32)
        pool = torch.load(args.pool, map_location="cpu", weights_only=False)
        X_def = defenses.apply_timing(X_raw, delta[:args.seq_len])
        X_def = defenses.apply_rand_ggi(X_def, pool["positions"],
                                        args.n_insertions, seed=args.seed)
    elif args.defense == "poisson_ggi":
        pool = torch.load(args.pool, map_location="cpu", weights_only=False)
        X_def = defenses.apply_poisson_ggi(X_raw, pool["positions"],
                                           args.n_insertions, seed=args.seed)
    elif args.defense == "jitter":
        X_def = defenses.apply_temporal_jitter(X_raw, seed=args.seed)
    elif args.defense == "rand_bbd":
        X_def = defenses.apply_rand_bbd(X_raw, seed=args.seed)
    elif args.defense == "burst_ggi":
        bp = torch.load(args.burst_pool, map_location="cpu", weights_only=False)
        X_def = defenses.apply_burst_ggi(X_raw, bp["positions"], bp["mu_hg"],
                                         bp["sigma_hg"], args.n_insertions,
                                         seed=args.seed)
    else:
        raise ValueError(args.defense)

    os.makedirs(os.path.dirname(os.path.abspath(args.out_file)), exist_ok=True)
    data_io.save_raw(args.out_file, X_def, y)
    print(f"Saved defended traces -> {args.out_file}")


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

A.8 Скрипт оцінки ефективності захисту

Скрипт 4_evaluate.py реалізує два режими через параметр --split. При split=none обчислюється accuracy і fooling rate на повній трасі. При split=roundrobin або tsilver спочатку обчислюється linking_acc — верхня межа для атакуючого, що може реасемблювати всі фрагменти, після чого обчислюються метрики circ_acc та samp_fool на розділеному трафіку. Результат виводиться у форматі JSON для подальшої агрегації.

```
import argparse
import json

import _bootstrap  # noqa: F401
import torch

from wf_defense import data_io, models_io, metrics, defenses


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--model", required=True, choices=["DF", "VarCNN"])
    parser.add_argument("--feature", required=True, choices=["DIR", "DT2"])
    parser.add_argument("--ckpt", required=True)
    parser.add_argument("--in_file", required=True,
                        help="raw .npz traces to classify (clean or defended)")
    parser.add_argument("--split", default="none",
                        choices=["none", "roundrobin", "tsilver"])
    parser.add_argument("--n_circuits", type=int, default=3)
    parser.add_argument("--bwr_max", type=int, default=4)
    parser.add_argument("--seq_len", type=int, default=5000)
    parser.add_argument("--seed", type=int, default=42)
    parser.add_argument("--tag", default="", help="free-text label for the row")
    args = parser.parse_args()

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    X_raw, y = data_io.load_raw(args.in_file, args.seq_len)
    num_classes = int(y.max()) + 1
    model = models_io.load_model(args.model, args.ckpt, num_classes, device)

    result = {"tag": args.tag, "model": args.model, "split": args.split}

    if args.split == "none":
        result.update(metrics.accuracy_and_fooling(
            model, X_raw, y, args.feature, device))
    else:
        # Upper bound for an attacker who relinks the circuits.
        result.update(metrics.accuracy_and_fooling(
            model, X_raw, y, args.feature, device))
        result["linking_acc"] = result.pop("acc")
        result.pop("fool")

        if args.split == "tsilver":
            X_split, y_split = defenses.split_trafficsliver(
                X_raw, y, args.n_circuits, args.seq_len, args.bwr_max,
                seed=args.seed)
        else:
            X_split, y_split = defenses.split_round_robin(
                X_raw, y, args.n_circuits, args.seq_len)

        result.update(metrics.circuit_metrics(
            model, X_split, y_split, args.n_circuits, args.feature, device))

    print("RESULT " + json.dumps(result), flush=True)


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

А.9 Модуль перетворення ознак трафіку

Модуль data_io.py відповідає за перетворення raw DT-формату трас у ознаки, які очікують атакуючі моделі: DIR (напрямок пакетів) для DF, DT2 (напрямок + міжпакетний інтервал) для VarCNN. Перетворення до DT2 є особливо важливим: dummy-пакети у методі захисту мають IPI=0, що є помітною аномалією у часовому каналі.

```
import numpy as np
import torch


def load_raw(path, seq_len=5000):
    """Load a `.npz` file and return raw traces (X) and labels (y).

    The file must contain arrays "X" (signed timestamps) and "y" (labels).
    Traces are truncated or zero-padded to `seq_len` columns.
    """
    data = np.load(path)
    X = data["X"].astype(np.float32)
    y = data["y"].astype(np.int64)

    n_cols = X.shape[1]
    if n_cols > seq_len:
        X = X[:, :seq_len]
    elif n_cols < seq_len:
        X = np.pad(X, ((0, 0), (0, seq_len - n_cols)))
    return X, y


def save_raw(path, X, y):
    """Save raw traces and labels back to a `.npz` file."""
    np.savez(path, X=X.astype(np.float32), y=y.astype(np.int64))


def raw_to_ipi(X_raw):
    """Inter-packet intervals: IPI[i] = |t[i]| - |t[i-1]|, clipped to >= 0."""
    abs_t = np.abs(X_raw)
    ipi = np.diff(abs_t, axis=1, prepend=0.0)
    return np.clip(ipi, 0.0, None).astype(np.float32)


def to_dir_tensor(X_raw):
    """Raw traces -> DIR feature tensor of shape (N, 1, seq_len) for DF."""
    direction = np.sign(X_raw).astype(np.float32)
    return torch.from_numpy(direction[:, None, :])


def to_dt2_tensor(X_raw):
    direction = np.sign(X_raw).astype(np.float32)
    ipi = np.diff(np.abs(X_raw), axis=1, prepend=0.0)
    ipi = np.clip(ipi, 0.0, None).astype(np.float32)
    stacked = np.stack([direction, ipi], axis=1)
```

```
    return torch.from_numpy(stacked)


def to_feature_tensor(X_raw, feature):

    if feature == "DIR":
        return to_dir_tensor(X_raw)
    if feature == "DT2":
        return to_dt2_tensor(X_raw)
    raise ValueError(f"Unknown feature '{feature}' (use 'DIR' or 'DT2')")


def balanced_subset(X, y, per_class, seed=42):

    rng = np.random.default_rng(seed)
    chosen = []
    for label in np.unique(y):
        idx = np.where(y == label)[0]
        take = min(len(idx), per_class)
        chosen.append(rng.choice(idx, take, replace=False))
    chosen = np.concatenate(chosen)
    rng.shuffle(chosen)
    return X[chosen], y[chosen]
```

A.10 Скрипт навчання адаптивного атакуючого

Скрипт train_attacker.py реалізує навчання атакуючих моделей у двох режимах: clean (на чистому трафіку) та adaptive (з online-аугментацією). Режим adaptive є ключовим для коректної оцінки рандомізованого захисту: на кожній епосі генерується свіжа реалізація захисту з новим seed = run·1000 + epoch, що

```
import argparse
import os
import random

import _bootstrap  # noqa: F401
import numpy as np
import torch
from sklearn.metrics import f1_score
from torch.utils.data import DataLoader, TensorDataset

from wf_defense import data_io, models_io, defenses


def apply_defense(X_raw, y, args, pools, seed):
    if args.defense == "rand_ggi":
        X = defenses.apply_rand_ggi(X_raw, pools["rand"]["positions"],
                                    args.n_insertions, seed=seed)
    elif args.defense == "burst_ggi":
        bp = pools["burst"]
        X = defenses.apply_burst_ggi(X_raw, bp["positions"], bp["mu_hg"],
                                     bp["sigma_hg"], args.n_insertions, seed=seed)
    else:  # none
        X = X_raw
```

```
    if args.split == "roundrobin":
        return defenses.split_round_robin(X, y, args.n_circuits, args.seq_len)
    if args.split == "tsilver":
        split_seed = None if seed is None else seed + 500000
        return defenses.split_trafficsliver(
            X, y, args.n_circuits, args.seq_len, args.bwr_max, seed=split_seed)
    return X, y


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--model", required=True, choices=["DF", "VarCNN"])
    parser.add_argument("--feature", required=True, choices=["DIR", "DT2"])
    parser.add_argument("--mode", default="clean", choices=["clean", "adaptive"])
    parser.add_argument("--train_file", required=True)
    parser.add_argument("--valid_file", required=True)
    parser.add_argument("--save_file", required=True)
    # defense settings (used in adaptive mode)
    parser.add_argument("--defense", default="rand_ggi",
                        choices=["none", "rand_ggi", "burst_ggi"])
    parser.add_argument("--split", default="roundrobin",
                        choices=["none", "roundrobin", "tsilver"])
    parser.add_argument("--pool", default="vectors/rand_pool.pt")
    parser.add_argument("--burst_pool", default="vectors/burst_pool.pt")
    parser.add_argument("--n_circuits", type=int, default=3)
    parser.add_argument("--bwr_max", type=int, default=4)
    parser.add_argument("--n_insertions", type=int, default=500)
    # data / optimization
    parser.add_argument("--n_per_class", type=int, default=200)
    parser.add_argument("--n_valid_class", type=int, default=50)
    parser.add_argument("--train_epochs", type=int, default=12)
    parser.add_argument("--batch_size", type=int, default=64)
    parser.add_argument("--learning_rate", type=float, default=2e-3)
    parser.add_argument("--optimizer", default="Adamax")
    parser.add_argument("--seed", type=int, default=0)
    parser.add_argument("--seq_len", type=int, default=5000)
    args = parser.parse_args()

    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.cuda.manual_seed_all(args.seed)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    if os.path.exists(args.save_file):
        print(f"{args.save_file} already exists - nothing to do.")
        return

    # Load pools only if the chosen defense needs them.
    pools = {}
    if args.defense == "rand_ggi":
        pools["rand"] = torch.load(args.pool, map_location="cpu",
                                   weights_only=False)
    if args.defense == "burst_ggi":
        pools["burst"] = torch.load(args.burst_pool, map_location="cpu",
                                    weights_only=False)

    # Class-balanced train / valid subsets (keeps one epoch affordable).
    Xtr, ytr = data_io.load_raw(args.train_file, args.seq_len)
    Xva, yva = data_io.load_raw(args.valid_file, args.seq_len)
    Xtr, ytr = data_io.balanced_subset(Xtr, ytr, args.n_per_class, args.seed)
```

```
Xva, yva = data_io.balanced_subset(Xva, yva, args.n_valid_class, args.seed + 7)
num_classes = int(max(ytr.max(), yva.max())) + 1
print(f"train subset={Xtr.shape}  valid subset={Xva.shape}  "
      f"classes={num_classes}")

# A FIXED defended validation set, used to pick the best epoch.
if args.mode == "adaptive":
    Xv_def, yv_def = apply_defense(Xva, yva, args, pools, seed=10000 + args.seed)
else:
    Xv_def, yv_def = Xva, yva
Xv_feat = data_io.to_feature_tensor(Xv_def, args.feature).to(device)
yv_true = yv_def

model = models_io.build_model(args.model, num_classes).to(device)
optimizer_cls = getattr(torch.optim, args.optimizer)
optimizer = optimizer_cls(model.parameters(), lr=args.learning_rate)
criterion = torch.nn.CrossEntropyLoss()

os.makedirs(os.path.dirname(os.path.abspath(args.save_file)), exist_ok=True)

best_f1, best_epoch = 0.0, -1
for epoch in range(args.train_epochs):
    # Build this epoch's training set.
    if args.mode == "adaptive":
        # Fresh realization every epoch (this is the key to a fair attacker).
        X_ep, y_ep = apply_defense(Xtr, ytr, args, pools,
                                   seed=args.seed * 1000 + epoch)
    else:
        X_ep, y_ep = Xtr, ytr
    X_feat = data_io.to_feature_tensor(X_ep, args.feature)
    y_feat = torch.from_numpy(y_ep.astype(np.int64))

    loader = DataLoader(TensorDataset(X_feat, y_feat),
                        batch_size=args.batch_size, shuffle=True)
    model.train()
    total_loss, n_seen = 0.0, 0
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad()
        loss = criterion(model(xb), yb)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * xb.size(0)
        n_seen += xb.size(0)

    # Validate on the fixed defended set; keep the best epoch.
    model.eval()
    preds = []
    with torch.no_grad():
        for j in range(0, len(yv_true), 256):
            preds.append(model(Xv_feat[j:j + 256]).argmax(1).cpu().numpy())
    preds = np.concatenate(preds)
    f1 = f1_score(yv_true, preds, average="macro")
    if f1 > best_f1:
        best_f1, best_epoch = f1, epoch
        torch.save(model.state_dict(), args.save_file)
    print(f"  epoch {epoch:2d}: loss={total_loss / n_seen:.3f}  "
          f"valid_F1={f1:.4f}  (best ep{best_epoch}={best_f1:.4f})",
          flush=True)
```

```
    print(f"Done -> {args.save_file}  best_epoch={best_epoch}  F1={best_f1:.4f}")


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

A.11 Експериментальний захист Poisson-GGI

Модуль defenses/poisson_ggi.py реалізує експериментальну модифікацію Rand-GGI з пуассонівським розподілом кількості dummy-пакетів по часових вікнах. Спроба подолати адаптивного Holmes виявилась невдалою: параметр lambda обчислюється з timestamps самої траси і залишається класозалежним.

```
import numpy as np

from .ggi import _insert_outgoing_dummies


def apply_poisson_ggi(X_raw, pool_positions, n_insertions,
                      window_ms=40.0, seed=None):
    rng = np.random.default_rng(seed)
    n_rows, length = X_raw.shape
    X_out = np.empty_like(X_raw)
    pool_clipped = np.clip(pool_positions, 0, length - 1)

    for i in range(n_rows):
        trace = X_raw[i]
        abs_ms = np.abs(trace) * 1000.0   # seconds -> milliseconds

        # Which TAF window each pool position falls into for THIS trace.
        pool_window = (abs_ms[pool_clipped] / window_ms).astype(np.int32)
        windows, inverse = np.unique(pool_window, return_inverse=True)
        n_windows = len(windows)
        lam = n_insertions / n_windows      # Poisson mean per window

        sampled = []
        for w in range(n_windows):
            in_window = pool_positions[inverse == w]
            count = int(rng.poisson(lam))
            if count == 0 or len(in_window) == 0:
                continue
            count = min(count, len(in_window))
            sampled.extend(
                rng.choice(in_window, size=count, replace=False).tolist())

        if not sampled:
            # Extremely rare: every Poisson draw was 0 -> insert a small batch.
            fallback = max(1, n_insertions // 10)
            sampled = rng.choice(pool_positions,
                                 size=min(fallback, len(pool_positions)),
                                 replace=False).tolist()

        positions = np.sort(np.array(sampled, dtype=np.int64))
        X_out[i] = _insert_outgoing_dummies(trace.copy(), positions)
        if (i + 1) % 2000 == 0:
```

```
            print(f"  poisson_ggi apply: {i + 1}/{n_rows}", flush=True)

    return X_out
```

A.12 Експериментальний захист Temporal Jitter

Модуль defenses/temporal_jitter.py додає випадкову константу зсуву до всіх timestamps траси для переміщення пакетів між TAF-вікнами. Захист зберігає відносну структуру траси, тому після перенавчання Holmes ігнорує абсолютне зміщення і відновлює точність на відносних ознаках.

```
import numpy as np


def apply_temporal_jitter(X_raw, jitter_ms=20.0, seed=None):
    rng = np.random.default_rng(seed)
    n_rows = len(X_raw)
    X_out = np.empty_like(X_raw)
    jitter_s = jitter_ms / 1000.0   # ms -> seconds (raw traces use seconds)

    for i in range(n_rows):
        trace = X_raw[i]
        real = trace != 0.0
        if not real.any():
            X_out[i] = trace.copy()
            continue

        shift = rng.uniform(-jitter_s, jitter_s)
        abs_t = np.abs(trace)
        # Keep padding at 0; clamp real timestamps to stay strictly positive.
        new_abs = np.where(real, np.maximum(abs_t + shift, 1e-9), 0.0)
        X_out[i] = (np.sign(trace) * new_abs).astype(np.float32)
        if (i + 1) % 2000 == 0:
            print(f"  temporal_jitter apply: {i + 1}/{n_rows}", flush=True)

    return X_out
```

A.13 Експериментальний захист Burst-GGI

Модуль defenses/burst_ggi.py вставляє dummy-пакети всередину реальних вихідних burst-ів з природним значенням IPI замість аномального нуля. Через малий пул (185 позицій) патерн виявився детермінованим і не пройшов адаптивне тестування.

```
import numpy as np

MIN_GAP_S = 0.003   # smallest gap (seconds) worth inserting into
MIN_IPI = 1e-6      # smallest dummy inter-packet time
```

```
def get_outgoing_burst_gaps(trace, min_gap=MIN_GAP_S):
    result = []
    length = len(trace)
    nz_idx = np.flatnonzero(trace)
    if len(nz_idx) < 3:
        return result

    for k in range(1, len(nz_idx) - 1):
        i_prev, i_cur, i_next = nz_idx[k - 1], nz_idx[k], nz_idx[k + 1]
        if i_cur >= length - 1:
            break
        # Need three consecutive outgoing packets (we are inside a burst).
        if trace[i_prev] <= 0 or trace[i_cur] <= 0 or trace[i_next] <= 0:
            continue
        gap = abs(trace[i_next]) - abs(trace[i_cur])
        if gap < min_gap:
            continue
        insert_pos = i_cur + 1
        if insert_pos < length:
            result.append((insert_pos, gap / 2.0))
    return result


def build_position_stats(X_raw, min_gap=MIN_GAP_S, seq_len=5000):
    n = len(X_raw)
    freq = np.zeros(seq_len, dtype=np.float32)
    sum_hg = np.zeros(seq_len, dtype=np.float64)
    sum_hg2 = np.zeros(seq_len, dtype=np.float64)

    for trace in X_raw:
        for pos, half_gap in get_outgoing_burst_gaps(trace, min_gap):
            if pos < seq_len:
                freq[pos] += 1
                sum_hg[pos] += half_gap
                sum_hg2[pos] += half_gap * half_gap

    valid = freq > 0
    mean_hg = np.where(valid, sum_hg / np.maximum(freq, 1), 0.0).astype(np.float32)
    var_hg = np.where(valid, sum_hg2 / np.maximum(freq, 1) - mean_hg ** 2, 1e-12)
    std_hg = np.sqrt(np.maximum(var_hg, 1e-12)).astype(np.float32)
    freq /= n
    return freq, mean_hg, std_hg


def _insert_burst_dummy(trace, position, mu_hg, sigma_hg, rng):
    length = len(trace)
    p = int(position)
    if p <= 0 or p >= length:
        return trace
    prev_abs = abs(float(trace[p - 1])) if trace[p - 1] != 0 else 0.0
    ipi = float(rng.normal(mu_hg, max(sigma_hg * 0.25, MIN_IPI)))
    ipi = float(np.clip(ipi, MIN_IPI, 2.0 * mu_hg))
    dummy_ts = prev_abs + ipi
    return np.concatenate([trace[:p], [dummy_ts], trace[:length - 1][p:]])


def apply_burst_ggi(X_raw, pool_positions, pool_mu_hg, pool_sigma_hg,
                    n_insertions, seed=None):
```

```
    rng = np.random.default_rng(seed)
    n_rows = len(X_raw)
    X_out = np.empty_like(X_raw)

    for i in range(n_rows):
        k = min(n_insertions, len(pool_positions))
        idx = rng.choice(len(pool_positions), size=k, replace=False)
        positions = pool_positions[idx]
        mus, sigmas = pool_mu_hg[idx], pool_sigma_hg[idx]

        trace = X_raw[i].copy()
        for j in np.argsort(positions)[::-1]:   # right to left
            trace = _insert_burst_dummy(trace, int(positions[j]),
                                        float(mus[j]), float(sigmas[j]), rng)
        X_out[i] = trace
        if (i + 1) % 2000 == 0:
            print(f"  burst_ggi apply: {i + 1}/{n_rows}", flush=True)

    return X_out
```

A.14 Експериментальний захист Rand-BBD

Модуль defenses/rand_bbd.py реалізує єдиний у проєкті захист, що теоретично коректно атакує TAF-ознаку Holmes на рівні кількості пачок у вікні. Компонент A розбиває реальні пачки вставкою вихідного dummy перед зміною напрямку; компонент B створює фіктивні пачки у випадкових порожніх вікнах. Залишений як гіпотеза для майбутньої роботи.

```
import numpy as np


def apply_rand_bbd(X_raw, n_fake=400, window_ms=40.0, seed=None):
    """Disrupt real bursts and inject fake ones into empty windows."""
    rng = np.random.default_rng(seed)
    n_rows, length = X_raw.shape
    X_out = np.empty_like(X_raw)

    for i in range(n_rows):
        trace = X_raw[i]
        real_trace = trace[trace != 0.0]
        if len(real_trace) < 2:
            X_out[i] = trace.copy()
            continue

        # Indices where the direction flips => real burst boundaries.
        signs = np.sign(real_trace)
        burst_idx = np.where(signs[1:] != signs[:-1])[0] + 1

        abs_ms = np.abs(real_trace) * 1000.0
        burst_windows = set((abs_ms[burst_idx] / window_ms).astype(int))

        # Windows that are currently active but have no burst boundary.
        last_window = int(abs_ms[-1] / window_ms) + 1
```

```
        empty_windows = list(set(range(last_window)) - burst_windows)
        take = min(n_fake, len(empty_windows))
        fake_windows = rng.choice(empty_windows, size=take, replace=False) \
            if take > 0 else []

        insert_ts, insert_sign = [], []

        # Component A: split each real burst with one outgoing dummy.
        for idx in burst_idx:
            ts = abs_ms[idx] / 1000.0 - 0.0001
            prev_ts = abs_ms[idx - 1] / 1000.0
            if ts <= prev_ts:
                ts = prev_ts + 1e-6
            insert_ts.append(ts)
            insert_sign.append(1.0)

        # Component B: a (+1, -1) pair inside each chosen empty window.
        for w in fake_windows:
            base = w * window_ms / 1000.0
            t_out = base + rng.uniform(0.001, window_ms / 1000.0 - 0.002)
            t_in = t_out + 0.001
            insert_ts.extend([t_out, t_in])
            insert_sign.extend([1.0, -1.0])

        all_ts = np.concatenate([np.abs(real_trace), insert_ts])
        all_sign = np.concatenate([np.sign(real_trace), insert_sign])

        order = np.argsort(all_ts)
        new_trace = (all_ts[order] * all_sign[order])

        # Fix length back to `length`.
        if len(new_trace) > length:
            new_trace = new_trace[:length]
        elif len(new_trace) < length:
            new_trace = np.pad(new_trace, (0, length - len(new_trace)))
        X_out[i] = new_trace.astype(np.float32)

        if (i + 1) % 2000 == 0:
            print(f"  rand_bbd apply: {i + 1}/{n_rows}", flush=True)

    return X_out
```