Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра кібербезпеки інформаційних систем, мереж і технологій

До захисту допущено
Кафедрою КІСМіТ протокол №____ від «___» грудня 2025 р.

завідувач кафедри _____ Марина ЄСІНА
(підпис) *(ім'я, прізвище)*
«___» грудня 2025 р.

Кваліфікаційна робота
здобувача другого (магістерського) рівня вищої освіти

«Дослідження можливостей моніторингу та адміністрування змісту ACL для покращення поточних параметрів безпеки і функціонування сучасних інформаційних систем»
(назва роботи)

Спеціальність (спеціалізація) 125 «Кібербезпека та захист інформації»

Освітня програма «Безпека інформаційних і комунікаційних систем»

Виконавець _____ Данило ЖІЛЄНКОВ
(підпис) *(ім'я, прізвище)*

Науковий керівник _____ Сергій МАЛАХОВ
(підпис) *(ім'я, прізвище)*

Харків – 2025

РЕФЕРАТ

Пояснювальна записка до проекту магістра містить 67 сторінок, 24 рисунки, 2 таблиці, 1 додаток, 26 посилань на джерела.

Мета роботи: - покращення контрольованості поточного рівня безпеки сучасних інформаційних систем (ІС), шляхом розробки і вдосконалення програмних інструментів автоматизованого моніторингу колізій та структурної оптимізації (дефрагментації) списків контролю доступу (ACL).

Об’єкт дослідження - процеси обміну даними та управління доступом в сучасних інформаційно-телекомунікаційних системах (мережах).

Предмет дослідження - технології й методи моніторингу мережевого трафіку та способи структурної оптимізації (адміністрування) змісту списків контролю доступу засобів мережевої інфраструктури (шлюзи, маршрутизатори тощо).

Основними методами дослідження є узагальнення джерел, системний аналіз, програмування та експериментальне комп’ютерне моделювання.

У роботі досліджені питання покращення контрольованості поточного рівня захищеності сучасних інформаційних систем засобами адміністрування списків контролю доступу (ACL). Проаналізовано залежність між обсягом списків та навантаженням на апаратні ресурси маршрутизаторів (CPU/TCAM). Визначено вплив логічних аномалій (екранування, кореляція, надлишковість) на продуктивність і безпеку мережі. Наголошено на необхідності автоматизації моніторингу вразливостей та структурної оптимізації правил ACL. Розроблено алгоритм динамічної дефрагментації ACL, який здійснює перевпорядкування записів на основі статистики оброблюваного трафіку. Створено програмний модуль автоматизованого аудиту для виявлення колізій та перевірки наявності захисту від IP-спуфінгу. Змодельовано роботу дослідного алгоритму на емульованих мережевих топологіях. Проведено експериментальне тестування розробленого програмного прототипу при обробці інтенсивних потоків даних. Підтверджено зниження навантаження на процесор маршрутизатора та

підвищення пропускної здатності мережі. Покращено рівень контрольованості циркулюючого трафіку. Визначено напрями подальшої інтеграції розглянутих рішень з технологіями SDN та адаптації алгоритмів для протоколу IPv6.

Результати роботи можуть бути використані в процесах автоматизованого аудиту поточного стану безпеки сучасних ІС та структурної оптимізації ACL в межах контрольованого периметру безпеки. Розглянуті матеріали є корисними, як допоміжний матеріал, для розширення рівня обізнаності персоналу з ІБ при вирішенні питань моніторингу поточного стану мережевої активності та адміністрування правил безпеки мережевої інфраструктури.

Ключові слова: СПИСКИ КОНТРОЛЮ ДОСТУПУ (ACL), АВТОМАТИЗАЦІЯ АДМІНІСТРУВАННЯ, ОПТИМІЗАЦІЯ МЕРЕЖІ, ІНФОРМАЦІЙНА БЕЗПЕКА, IP-СПУФІНГ, ЛОГІЧНІ КОЛІЗІЇ, МОНІТОРИНГ ТРАФІКУ, МАРШРУТИЗАЦІЯ.

ABSTRACT

The explanatory note to the master's thesis consists of 67 pages, 24 figures, 2 tables, 1 appendices, and 26 references.

The purpose of the work is to improve the controllability of the current security level of modern information systems by developing and refining software tools for automated collision monitoring and structural optimization (defragmentation) of Access Control Lists (ACLs).

The object of research is the processes of data exchange and access management in modern information and telecommunication systems (networks).

The subject of research is technologies and methods of network traffic monitoring and ways of structural optimization (administration) of Access Control List content for network infrastructure devices (gateways, routers, etc.).

The main research methods are source generalization, system analysis, programming, and experimental computer modeling.

The thesis investigates the issues of improving the controllability of the current security level of modern information systems through Access Control List (ACL) administration. The dependence between the volume of lists and the load on router hardware resources (CPU/TCAM) is analyzed. The impact of logical anomalies (shadowing, correlation, redundancy) on network performance is determined. The necessity of automating vulnerability monitoring and structural rule optimization is substantiated. An algorithm for dynamic ACL defragmentation, which performs record reordering based on traffic statistics, has been developed. An automated audit module has been created to detect collisions and verify the presence of protection against IP spoofing. The operation of the experimental algorithm has been modeled on emulated network topologies. Experimental testing of the developed software prototype was conducted while processing intensive data streams. A reduction in router processor load and an increase in network throughput were confirmed. Prospects for system integration

with SDN technologies and adaptation of algorithms for the IPv6 protocol are determined.

The results of the work can be used in the processes of automated auditing and structural optimization (defragmentation) of Access Control Lists within a controlled security perimeter. Furthermore, the work can be applied to tasks involving enhancing personnel situational awareness regarding the current state of network activity and minimizing the impact of the human factor during network equipment administration.

Keywords: ACCESS CONTROL LISTS (ACL), ADMINISTRATION AUTOMATION, NETWORK OPTIMIZATION, INFORMATION SECURITY, IP SPOOFING, LOGICAL COLLISIONS, TRAFFIC MONITORING, ROUTING.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ACL – Access Control List
API – Application Programming Interface
BCP – Best Current Practice
CIDR – Classless Inter-Domain Routing
CLI – Command Line Interface
CPU – Central Processing Unit
DDoS – Distributed Denial of Service
DMZ – Demilitarized Zone
DNS – Domain Name System
GUI – Graphical User Interface
HTTP – HyperText Transfer Protocol
ICMP – Internet Control Message Protocol
IEC – International Electrotechnical Commission
IOS – Internetwork Operating System
IP – Internet Protocol
ISO – International Organization for Standardization
LAN – Local Area Network
NIST – National Institute of Standards and Technology
OSI – Open Systems Interconnection
RFC – Request for Comments
SDN – Software-Defined Networking
SSH – Secure Shell
TCAM – Ternary Content Addressable Memory
TCP – Transmission Control Protocol
UDP – User Datagram Protocol
WAN – Wide Area Network
ІС – Інформаційна система
ПЗ – Програмне забезпечення

ВСТУП

Забезпечення стабільності та захищеності сучасних інформаційно-комунікаційних систем (ІКС) та мереж із складною гетерогенною структурою, стає все складнішим викликом на тлі неперервного зростання обсягів циркулюючої інформації та поширення різновидів (форматів) представлення даних. Критично важливим питанням є не стільки відсутність інструментів захисту сучасних ІКС, скільки втрата неперервного контролю над їх конфігурацією та параметрами налаштувань мережевого устаткування в умовах динамічної зміни й масштабування їх інфраструктури. Така ситуація вимагає переходу від парадигми «реактивного» (тобто ручного) керування до концепції широкого впровадження проактивних автоматизованих підходів.

В межах реалізації комплексу організаційно-технічних заходів із забезпечення потрібного рівню безпеки цільових ІКС, однією з важливіших складових залишається надійний контроль й управління списків контролю доступу (ACL). Як свідчить практика (в т.ч. аналіз інцидентів з безпеки), традиційна модель їх моніторингу і адміністрування вичерпала свій ресурс ефективності й адекватності сучасному стану загроз. Персонал що відповідає за управління інфраструктурою сучасних ІКС та реалізацію вимог корпоративних політик безпеки (ПІБ), стикаються з парадоксом: - деталізація правил для підтримки заданого рівню інформаційної безпеки (ІБ), неминуче призводить до перевантаження апаратних ресурсів мережевого устаткування (маршрутизаторів, шлюзів, мостів тощо) та, як наслідок, виникнення великої кількості помилок у їх налаштуванні та логічних колізій правил безпеки. Статичні методи вже не здатні адекватно реагувати на динаміку сучасного трафіку, що створює реальні загрози ІБ та стабільності функціонування мережевої інфраструктури сучасних ІКС.

Дана робота присвячена лише однієї із складових, на шляху вирішення зазначених вище труднощів (парадоксу), а саме – покращенню оперативності, контрольованості і керованості використовуваних ACL. В межах вирішення цього завдання, на прикладах відомих рішень та аналізу інцидентів з ІБ, проведено

всебічний аналіз специфіки питань моніторингу й адміністрування ACL в сучасних ІКС. За результатами проведених досліджень створено концептуальну структуру дослідного алгоритму (з наступною програмною реалізацією його основних блоків), що забезпечує впровадження парадигми автоматизованого моніторингу вразливостей та структурну оптимізацію/дефрагментацію ACL.

В рамках дослідження синтезовано умовне тестове середовище та проведено комп'ютерне моделювання досліджуваних процесів у відповідності до типових (тобто, найбільш характерних) сценаріїв (імітація трафіку та спроб несанкціонованого доступу). Розроблено алгоритм динамічної оптимізації, який аналізує й перевпорядковує правила ACL на основі реальної (тестової) статистики поточного трафіку, тим самим знижуючи навантаження на обладнання та персонал. Розроблений програмний модуль аудиту поточних налаштувань ACL, дозволяє автоматично виявляти приховані логічні аномалії (такі як, екранування, кореляція, надлишковість) та підтверджує наявність захисту інфраструктури ІКС від атак типу IP-спуфінг (в т.ч., MAC-спуфінг). Проведене комп'ютерне моделювання, в цілому, підтвердило правильність обраних рішень. Запропоновані підходи не лише вирішують частину труднощів із продуктивністю мережевої інфраструктури сучасних ІКС й покращенням рівня ситуаційної поінформованості їх персоналу в умовах дефіциту часу, але й створюють корисне технологічне підґрунтя для подальшої інтеграції з архітектурою програмно–конфігурованих мереж (SDN), та широкого впровадження методів машинного навчання.

1 АНАЛІЗ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ КОНТРОЛЬОВАНОСТІ ТА АДМІНІСТРУВАННЯ СПИСКІВ КОНТРОЛЮ ДОСТУПУ

1.1 Роль та місце ACL у забезпеченні питань безпеки сучасних ІС

У сучасних інформаційно-телекомунікаційних системах (ІТС) забезпечення цілісності, конфіденційності та доступності інформації безпосередньо залежить від ефективності механізмів розмежування доступу. На мережевому рівні еталонної моделі OSI (Network Layer, L3) та транспортному рівні (Transport Layer, L4) основним інструментом фільтрації трафіку виступають списки контролю доступу (ACL). Згідно з міжнародними стандартами серії ISO/IEC 27000, ACL розглядаються не лише як засіб безпеки, а як фундаментальний компонент архітектури мережі, що визначає логіку маршрутизації та керування пропускною здатністю [1].

Принципи функціонування та класифікація. Функціонально ACL являє собою впорядкований набір правил (ruleset), які маршрутизатор або комутатор L3 застосовує до кожного пакета, що проходить через інтерфейс пристрою. Ідентифікація потоків даних здійснюється на основі аналізу полів заголовка IP-пакета (адреса відправника, адреса отримувача) та заголовка сегмента TCP/UDP (порт джерела, порт призначення, прапори з'єднання).

У фаховій літературі виділяють два основні типи списків, що використовуються в корпоративних мережах:

1) Стандартні ACL (Standard ACL): здійснюють фільтрацію виключно за IP-адресою джерела. Вони є менш ресурсомісткими, проте не дозволяють реалізувати гнучкі політики безпеки.

2) Розширені ACL (Extended ACL): забезпечують гранулярний контроль, враховуючи протокол (ICMP, TCP, UDP), порти та навіть параметри QoS (Quality of Service). Саме цей тип списків є об'єктом адміністрування в контексті захисту від складних векторів атак.

Критично важливою особливістю, що визначає продуктивність системи, є алгоритм обробки списку(рис 1.1). Маршрутизатор перевіряє відповідність пакета правилам послідовно - зверху вниз. Як тільки знайдено перше правило, що задовольняє умовам, подальший перегляд списку припиняється, і виконується відповідна дія (permit або deny). Якщо пакет не підпадає під жодне з правил, спрацьовує правило «неявної заборони» (Implicit Deny), яке блокує трафік [10]. Ця архітектурна особливість обумовлює пряму залежність між порядком розміщення правил та затримкою обробки пакета, що підтверджується дослідженнями у сфері оптимізації міжмережевих екранів [9].

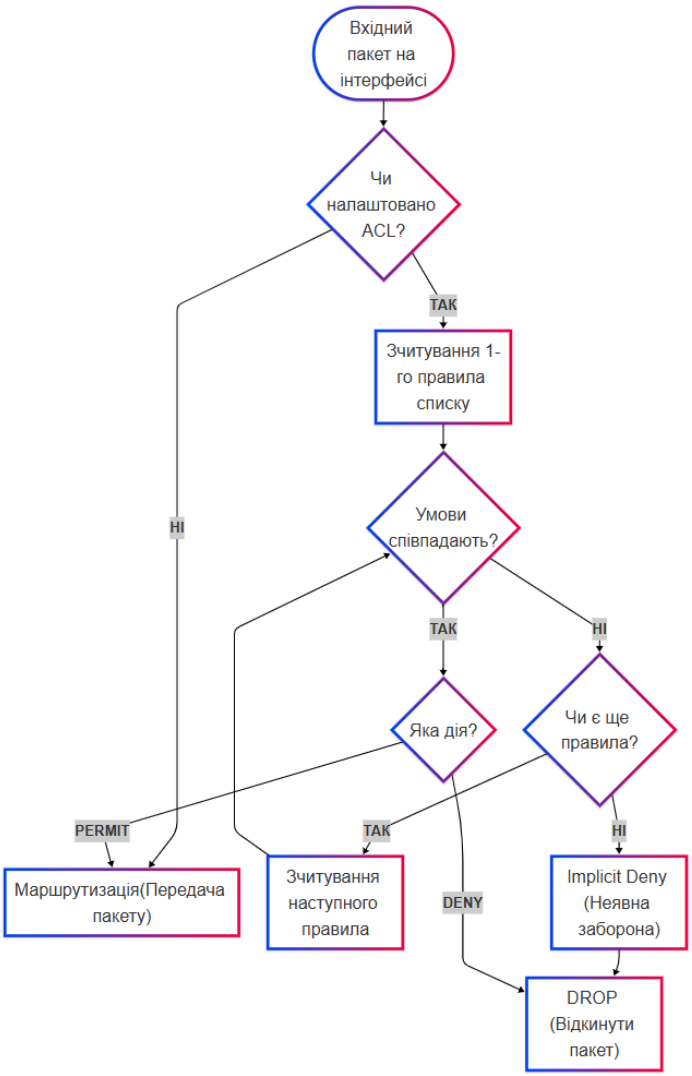

Рисунок 1.1 - Алгоритм обробки списку маршрутизатором

ACL як інструмент забезпечення інформаційної безпеки. Роль ACL у системі захисту полягає у мінімізації поверхні атаки шляхом реалізації принципу найменших привілеїв. Згідно з рекомендаціями NIST SP 800-41 [2], коректно налаштовані списки доступу вирішують такі задачі:

- Сегментація мережі: ізоляція критично важливих серверів та баз даних від користувацьких сегментів та гостьових підмереж.
- Захист від IP-спуфінгу (Anti-Spoofing): блокування на вхідних зовнішніх інтерфейсах пакетів, які мають IP-адресу відправника, що належить внутрішньому діапазону мережі. Відсутність таких правил є типовою вразливістю, що дозволяє зловмисникам обходити системи аутентифікації [5].
- Фільтрація шкідливого трафіку: блокування відомих портів, що використовуються шкідливим програмним забезпеченням (наприклад, порти троянів Back Orifice, WannaCry тощо), а також обмеження ICMP-трафіку для запобігання розвідці мережі.

Вплив на параметри функціонування та продуктивність. Окрім безпекової функції, ACL мають суттєвий вплив на параметри продуктивності мережевого обладнання. Процес співставлення заголовків пакетів із записами ACL вимагає обчислювальних ресурсів. У сучасних маршрутизаторах корпоративного рівня частково використовується апаратна прискорення (TCAM), проте складні логічні умови та довгі списки часто призводять до переходу на програмну обробку центральним процесором [10].

Зростання довжини списку ACL (N) призводить до лінійного збільшення часу обробки (T) за законом T~O(N) у найгіршому випадку. Це означає, що неоптимальне розташування правил, коли найбільш активні потоки трафіку перевіряються в кінці списку, призводить до непродуктивного використання CPU та збільшення джитеру - варіації затримки пакетів, що є критичним для мультимедійних сервісів (VoIP, відеоконференцзв'язок) [7]. Саме тому задача моніторингу інтенсивності спрацювання правил («ваги» правила) та їх динамічна перестановка (дефрагментація) є необхідною умовою підтримки високої пропускної здатності мережі.

Особливості застосування в гетерогенних мережах. Сучасна мережева інфраструктура є гетерогенною, поєднуючи дротові (Ethernet) та бездротові (Wi-Fi) сегменти. Це ускладнює адміністрування ACL, оскільки бездротові точки доступу часто працюють у режимі моста, прозоро пропускаючи трафік до дротового ядра мережі. У таких умовах ACL повинні застосовуватися не лише на шлюзі, але й на інтерфейсах, що обслуговують VLAN бездротових клієнтів. Специфіка полягає в необхідності контролю мобільності користувачів та захисту від специфічних атак на Wi-Fi (наприклад, ARP-spoofing, Man-in-the-Middle), які можуть бути нейтралізовані шляхом жорсткої прив'язки MAC-адрес до IP в ACL або використання технологій DAI (Dynamic ARP Inspection), що працюють у комплексі зі списками доступу [12].

Таким чином, ACL є подвійним інструментом: з одного боку, це бар'єр для загроз, з іншого - потенційне «вузьке місце» продуктивності. Ефективність їх використання залежить не стільки від можливостей обладнання, скільки від якості адміністрування змісту списків: відсутності логічних помилок, захисту від спуфінгу та оптимізації порядку правил відповідно до профілю трафіку.

1.2 Труднощі адміністрування: класифікація аномалій та ризики впливу людського фактору в гетерогенних мережах

Ефективність функціонування сучасних корпоративних мереж перебуває у прямій залежності від якості налаштування політик безпеки. Проте, як свідчить практика експлуатації телекомунікаційного обладнання, адміністрування списків контролю доступу залишається одним із найбільш проблемних аспектів керування мережею. Проблематику моніторингу та адміністрування доцільно розглядати у трьох площинах: апаратні обмеження продуктивності, логічні колізії конфігурацій та недоліки існуючих інструментів діагностики.

Апаратна проблематика: вичерпання ресурсів та деградація продуктивності. Зростання кількості правил у списках ACL призводить до значного навантаження на апаратну частину маршрутизаторів. Сучасні пристрої корпоративного класу використовують спеціалізовану пам'ять TCAM (Ternary Content Addressable

Memory) для апаратної фільтрації пакетів на максимальній швидкості каналу [10]. Проте обсяг TCAM є обмеженим і дорогим ресурсом. Коли адміністратор створює надмірно довгий або фрагментований список, ресурс цієї пам'яті вичерпується. У такому випадку маршрутизатор змушений переходити на програмну обробку пакетів, використовуючи центральний процесор (CPU)(Рис. 1.2). Це призводить до критичного зростання завантаження процесора (до 90–100%), появи черг на інтерфейсах та збільшення мережевих затримок. Саме тому задача моніторингу заповненості TCAM та оптимізації списків є умовою фізичної працездатності мережі.

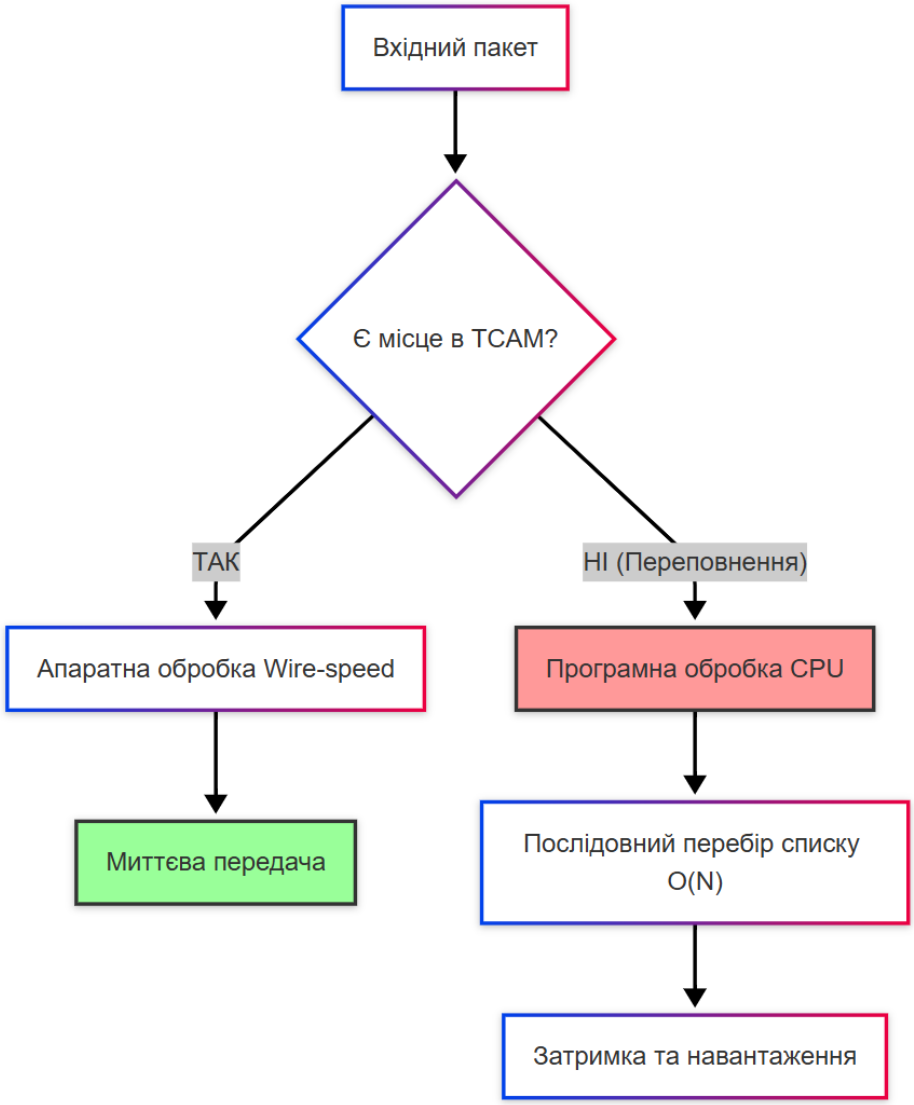

Рисунок 1.2 – Порядок дій маршрутизатора в залежності від переповненості TCAM

Проблематика логічних аномалій. При ручному адмініструванні, коли кількість записів перевищує сотні рядків, оператор фізично не здатна відслідкувати всі взаємозв'язки. Це породжує складні логічні аномалії(рис. 1.3), які класифікують наступним чином [7],[8]:

- Аномалія екранування (Shadowing). Виникає, коли правило з ширшим діапазоном адрес стоїть вище у списку і має дію, протилежну вузькому правилу, що стоїть нижче. Нижнє правило стає «мертвим кодом» - воно ніколи не спрацює, але займає пам'ять і вводить в оману аудиторів безпеки. Приклад: Перше правило блокує всю підмережу 10.0.0.0/8, а друге правило, що йде нижче, намагається дозволити конкретний хост 10.1.1.1. Через послідовну обробку друге правило буде проігноровано.
- Аномалія кореляції. Два правила перетинаються частково, але мають різні дії (одне дозволяє, інше забороняє). У цьому випадку доля пакета залежить виключно від порядку розташування правил. Це найнебезпечніший тип помилки, оскільки мережа може поводитися непередбачувано для різних піддіапазонів IP-адрес.
- Аномалія надлишковості. Наявність записів, видалення яких не змінить логіку фільтрації (наприклад, два однакових дозволи поспіль або дозвіл меншого діапазону, який вже входить у більший дозвіл). Дослідження показують, що в реальних мережах до 30% правил є надлишковими, що створює зайве навантаження на процесор при переборі списку [9].

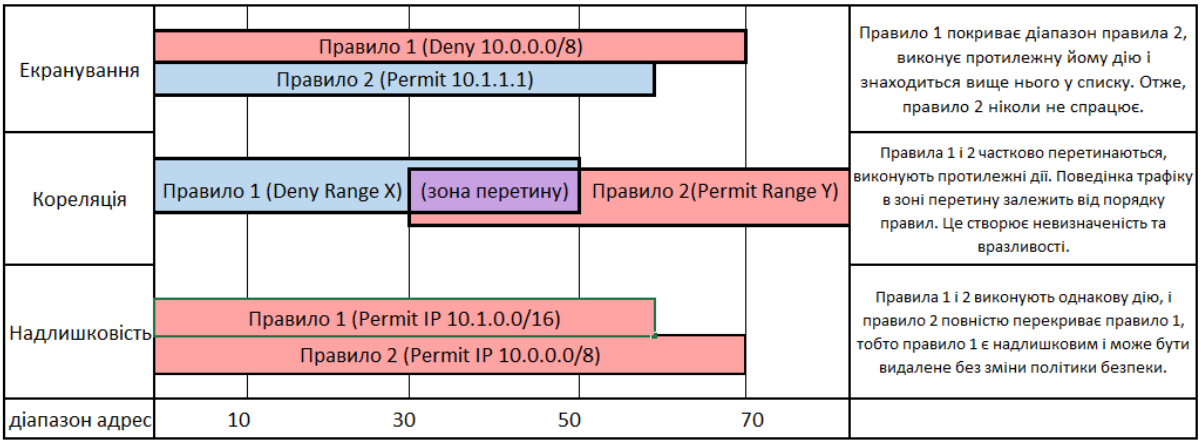

Рисунок 1.3 – схематичне зображення логічних аномалій ACL

Проблема «старіння» правил. Однією з ключових проблем моніторингу є відсутність часового контексту. В процесі експлуатації адміністратори часто додають тимчасові дозволи (для тестування сервісів, підключення підрядників тощо), але забувають їх видалити після завершення робіт. З часом ACL засмічується застарілими записами, які можуть відкривати доступ до вже неіснуючих сервісів або ресурсів, що змінили призначення. Стандартні засоби командного рядка (CLI) показують лише загальну кількість спрацювань правила, але не фіксують дату останньої активності. Без спеціалізованого програмного забезпечення виявити такі "мертві душі" практично неможливо [2].

Специфіка гетерогенних мереж. Особливу складність становить адміністрування ACL у мережах, де поєднані дротові сегменти та бездротові (Wi-Fi) зони з політикою використання власних пристроїв (BYOD). У таких мережах IP-адреси клієнтів змінюються динамічно, тому статичні списки доступу швидко втрачають актуальність. Крім того, трафік із бездротової мережі часто проходить через режим моста, оминаючи стандартні точки фільтрації, що вимагає особливого підходу до розміщення ACL для захисту від атак всередині сегмента (наприклад, ARP-спуфінгу) [12].

Відсутність візуалізації та контролю цілісності. Існуючі штатні інструменти адміністрування надають інформацію переважно у текстовому вигляді, який важко сприймати візуально при великих обсягах даних. Це призводить до того, що адміністратор не помічає критичних вразливостей, таких як:

- Відсутність антиспуфінгу: Типова помилка - відсутність фільтрації на зовнішньому інтерфейсі пакетів із внутрішніми адресами відправника, що порушує рекомендації стандартів безпеки (RFC 2827).
- Надмірна довіра: Залишення правил типу Permit Any Any у критичних зонах, таких як DMZ.

Проблематика семантичних помилок та порушення сегментації. Окрім логічних колізій, критичною проблемою ручного адміністрування є допущення так званих семантичних помилок (Semantic Errors) - ситуацій, коли правило є

технічно коректним і не конфліктує з іншими, але порушує глобальну політику безпеки підприємства.

Аналіз інцидентів інформаційної безпеки дозволяє виділити дві найбільш небезпечні категорії таких помилок, характерних для гетерогенних мереж:

1) Компрометація розмежування зон безпеки (Zone Segmentation Violation). У сучасних архітектурах критично важливим є розділення інфраструктури на сегменти з різним рівнем довіри: дротова мережа (Wired LAN), бездротова мережа (Wi-Fi), демілітаризована зона (DMZ) та зовнішня мережа (Internet/WAN). Типовою помилкою адміністратора є створення правил, що дозволяють прямий трафік (Bypass) між сегментами з низьким та високим рівнем довіри, минаючи шлюзи безпеки. *Приклад:* Дозвіл прямого доступу з гостьового Wi-Fi сегменту до серверів внутрішньої бази даних або керуючих інтерфейсів обладнання. Це створює умови для горизонтального переміщення зловмисника (Lateral Movement) у випадку злому слабозахищеної точки доступу [12].

2) Дозвіл небезпечних сервісів (Vulnerable Services Exposure). Через недостатню обізнаність або помилкове копіювання шаблонів конфігурацій, адміністратори часто залишають відкритими порти, що асоціюються з відомими вразливостями або шкідливим програмним забезпеченням. *Приклад:* Наявність правил Permit для портів віддаленого керування троянськими програмами (наприклад, Back Orifice - UDP 31337, NetBus - TCP 12345) або застарілих незахищених протоколів (Telnet - TCP 23). Такі правила перетворюють маршрутизатор на точку входу для автоматизованих ботнетів та цільових атак [5].

Відсутність автоматизованого контролю за такими семантичними помилками призводить до того, що навіть структурно оптимізований ACL може містити критичні «діри» в безпеці.

Таким чином, сучасний стан засобів керування ACL характеризується розривом між складністю задач та обмеженістю інструментів контролю. Ручний моніторинг не дозволяє ефективно виявляти приховані колізії та застарілі правила, що призводить до зниження продуктивності маршрутизаторів та послаблення захищеності інформаційної системи. Це обґрунтовує необхідність

розробки автоматизованих засобів аудиту та оптимізації, що є предметом дослідження.

1.3 Аналіз відомих варіантів реалізації ACL та узагальнення досвіду їх адміністрування

Проблема керування списками контролю доступу не обмежується обладнанням одного виробника. Хоча в даній роботі як базова платформа для експериментів обрана Cisco IOS (через її домінування на ринку корпоративних мереж), логічні принципи фільтрації трафіку та проблеми, пов’язані з ними, є спільними для всієї індустрії.

Для підтвердження гіпотези про універсальність запропонованих методів оптимізації доцільно провести порівняльний аналіз реалізації ACL у екосистемах трьох ключових вендорів: Cisco Systems, Juniper Networks та Huawei Technologies, а також у середовищі Linux (Netfilter), яке є основою для багатьох програмних маршрутизаторів та хмарних шлюзів.

1.3.1 Cisco Systems (IOS / IOS-XE / NX-OS)

Як було зазначено у попередніх підрозділах, Cisco використовує класичну модель послідовної обробки правил («First Match»).

- Синтаксис адресації: Ключовою особливістю є використання Wildcard masks (зворотних масок) для IPv4. Наприклад, маска 0.0.0.255 відповідає префіксу /24. Це часто стає джерелом помилок для адміністраторів-початківців, які плутають Wildcard із Subnet mask, що призводить до некоректного визначення діапазону дії правила.
- Структура: ACL поділяються на стандартні (тільки Source IP) та розширені (Source, Destination, Protocol, Ports).
- Проблема: При використанні об'єктних груп (Object Groups) у нових версіях IOS, візуальна складність конфігурації зростає, що ускладнює ручний пошук аномалій типу Shadowing.

1.3.2 Juniper Networks (Junos OS)

Архітектура Junos OS суттєво відрізняється від Cisco, використовуючи ієрархічну структуру конфігурації.

- Термінологія: Замість ACL використовується поняття Firewall Filters, які складаються з Terms (Умов). Кожен Term має умови (from) та дії (then).
- Синтаксис адресації: Juniper використовує класичну CIDR-нотацію (наприклад, 192.168.1.0/24), що є більш інтуїтивним і знижує ризик помилок при введенні.
- Логіка обробки: Аналогічна Cisco - послідовна обробка зверху вниз до першого збігу. Проте, наявність дії next term дозволяє створювати складні ланцюжки перевірок, що, з одного боку, додає гнучкості, а з іншого - робить пошук логічних колізій (кореляцій) ще складнішим завданням, ніж у лінійних списках Cisco.

1.3.3 Huawei Technologies (VRP)

Операційна система VRP (Versatile Routing Platform) багато в чому наслідує командний рядок Cisco, проте має свої особливості.

- Ідентифікація правил: Huawei використовує систему Rule ID (порядкових номерів) за замовчуванням з кроком 5 (5, 10, 15...). Це спрощує вставку нових правил між існуючими (на відміну від старих версій Cisco IOS, де це було проблематично).
- Автоматичне сортування: У деяких режимах (config-acl-auto) Huawei намагається автоматично сортувати правила за принципом «від специфічного до загального» (Longest Match). Однак, цей механізм працює не завжди коректно при складних комбінаціях портів і протоколів, що знову повертає адміністратора до необхідності ручного контролю порядку правил.

1.3.4 Linux Netfilter (iptables / nftables)

У середовищі серверних ОС та програмних маршрутизаторів (MikroTik, Ubiquiti, OpenWRT) стандартом є Netfilter.

- Ланцюжки (Chains): Логіка будується на проходженні пакету через ланцюжки INPUT, FORWARD, OUTPUT.
- Проблема складності: На відміну від апаратних роутерів, тут правила часто генеруються автоматично скриптами або системами оркестрації (Kubernetes, Docker), що призводить до появи тисяч записів. Проблема надлишковості (Redundancy) тут стоїть особливо гостро, оскільки кожен зайвий запис споживає такти центрального процесора загального призначення.

1.3.5 Узагальнення та висновки до п. 1.3

Незважаючи на відмінності у синтаксисі команд (CLI) та архітектурі пам'яті, фундаментальна логіка фільтрації трафіку у всіх розглянутих системах є ідентичною і базується на теорії множин.

Порівняльна характеристика наведена у Таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика провідних вендорів

Характеристика	**Cisco IOS**	**Juniper JunOS**	**Huawei VRP**	**Linux (iptables)**
Базовий елемент	Rule (ACE)	Term	Rule	Rule
Адресація	Wildcard Mask	CIDR Prefix	Wildcard / CIDR	CIDR / Mask
Порядок обробки	Top-down (послідовний)	Top-down	Top-down (або Depth-first)	Top-down (Chains)
Дія за замовчуванням	**Implicit Deny**	**Implicit Deny**	**Permit/Deny (залежить від зони)**	**Policy (зазв. Accept)**
Схильність до Shadowing	Висока	Висока	Середня	Висока
Вплив на CPU	Високий (якщо TCAM full)	Середній (ASIC)	Середній (ASIC)	Високий (CPU)

Висновок: проведений аналіз дозволяє стверджувати, що розроблювані в даній роботі алгоритми пошуку аномалій (Shadowing, Redundancy) та методи структурної оптимізації є інваріантними (незалежними) відносно конкретного вендора.

Математична модель конфлікту двох правил R1 та R2 залишається незмінною незалежно від того, як ці правила записані:

- Чи це access-list 100 deny ip 10.0.0.0 0.255.255.255 any (Cisco)
- Чи це from source-address 10.0.0.0/8 then discard (Juniper)

Це підтверджує наукову цінність роботи: розроблений програмний засіб, хоча і протестований на базі Cisco (як найбільш поширеній платформі), використовує універсальну логіку на базі бібліотеки ipaddress, яка працює з абстракцією IP-мереж. Таким чином, шляхом адаптації модуля парсингу (Input Parser), запропонована система може бути масштабована для підтримки мультибрендових мереж без зміни ядра алгоритму оптимізації.

1.4 Огляд основних концепцій фільтрації трафіку та роль ACL в сучасних системах захисту

Розвиток технологій мережевого захисту пройшов шлях від простих фільтрів пакетів до інтелектуальних систем глибокого аналізу контенту. Проте, незважаючи на появу нових класів обладнання (NGFW, UTM), традиційні списки контролю доступу (ACL) не втратили своєї актуальності, а трансформувалися у критично важливий інструмент першого ешелону оборони. Для розуміння ролі ACL у сучасній архітектурі безпеки доцільно розглянути еволюцію методів фільтрації.

1.4.1 Статична фільтрація (Stateless Packet Filtering)

Це базовий рівень, на якому працюють класичні ACL, що є об'єктом даного дослідження.

- Принцип дії: Маршрутизатор приймає рішення про пропуск або блокування кожного окремого пакета ізольовано, базуючись виключно на інформації з

його заголовків (IP-адреси, порти, прапори TCP). Пристрій не зберігає інформацію про попередні пакети та стан з'єднання.

- Переваги: Висока швидкість обробки. Завдяки реалізації на базі спеціалізованих мікросхем ASIC та пам'яті TCAM, фільтрація відбувається на швидкості середовища передачі (wire-speed), не створюючи додаткових затримок.
- Недоліки: Неможливість відслідковувати складні атаки, що розбиті на кілька пакетів, або динамічні протоколи (наприклад, FTP, SIP), які відкривають випадкові порти для передачі даних.

1.4.2 Динамічна фільтрація (Stateful Inspection)

Наступний етап еволюції, представлений технологіями Cisco IOS Firewall (CBAC) та Zone-Based Firewall.

- Принцип дії: Пристрій створює в пам'яті таблицю станів з'єднань (State Table). Він відслідковує етапи встановлення сесії (Three-way handshake: SYN, SYN-ACK, ACK) і пропускає зворотний трафік тільки якщо він належить до вже встановленого з'єднання [2].
- Роль: Забезпечує вищий рівень безпеки, але вимагає значних ресурсів процесора (CPU) та оперативної пам'яті (RAM) для зберігання таблиці сесій.

1.4.3 Шлюзи прикладного рівня та NGFW (Next-Generation Firewalls)

Сучасний стандарт захисту периметра (Palo Alto, Fortinet, Cisco Firepower).

- Принцип дії: Виконують глибоку інспекцію пакетів (DPI - Deep Packet Inspection) аж до 7-го рівня моделі OSI. Вони здатні розпізнавати конкретні додатки (наприклад, "Facebook Chat" всередині HTTPS-трафіку), перевіряти файли на віруси та виявляти вторгнення (IPS).
- Проблема продуктивності: Глибокий аналіз контенту є надзвичайно ресурсомістким процесом. Увімкнення всіх функцій безпеки на NGFW може знизити його пропускну здатність у 10–20 разів порівняно із заявленою.

Тому сучасна архітектура будується за принципом «Ешелонованого захисту» (Defense-in-Depth), де ACL відводиться роль високошвидкісного попереднього фільтра (Pre-filter):

1) Перший рубіж (Router ACL): Відсікає до 70–80% небажаного трафіку на вході в мережу.
    - Блокує IP-спуфінг (згідно з RFC 2827).
    - Відкидає трафік від відомих ботнетів та "чорних списків" IP.
    - Забороняє очевидно непотрібні порти (Telnet, NetBIOS, порти троянів).
    - Все це виконується апаратно (TCAM), не навантажуючи основний CPU.
2) Другий рубіж (NGFW): Отримує вже «очищений» потік даних і витрачає свої дорогі ресурси лише на глибокий аналіз потенційно корисного трафіку.

Висновки до розділу 1.4: Таким чином, незважаючи на розвиток інтелектуальних систем захисту, роль традиційних ACL не зменшилася, а трансформувалася. Вони залишаються єдиним інструментом, здатним фільтрувати трафік на магістральних швидкостях (10G/40G/100G) без деградації продуктивності мережі. Саме тому задача оптимізації структури ACL (дефрагментація та усунення колізій), що розглядається в цій роботі, є критично важливою: ефективний ACL на вході розвантажує всю подальшу інфраструктуру безпеки, підвищуючи загальну стійкість інформаційної системи. Неоптимізований ACL, навпаки, стає «вузьким місцем», нівелюючи переваги навіть найдорожчих рішень NGFW.

1.4.4 «Розрив продуктивності» та концепція Defense-in-Depth.

Саме тут виникає фундаментальна проблема, яку вирішує дана дипломна робота. Вартість обробки 1 Гбіт/с трафіку на рівні NGFW у десятки разів вища, ніж на рівні маршрутизатора з ACL. Якщо на вхід NGFW подати весь «брудний» трафік з Інтернету (сканування портів, DDoS-атаки, спроби спуфінгу), його

процесори будуть перевантажені обробкою сміттєвих пакетів, і він не зможе ефективно аналізувати легітимні запити.

1.5 Обґрунтування необхідності автоматизації моніторингу та оптимізації процедур адміністрування ACL

Проведений у попередніх підрозділах аналіз показав наявність фундаментального протиріччя в організації захисту сучасних інформаційних систем. З одного боку, вимоги стандартів (ISO/IEC 27001, NIST SP 800-41) вимагають максимальної деталізації правил доступу для мінімізації поверхні атаки [1],[2]. З іншого боку, екстенсивне нарощування обсягу списків ACL призводить до вичерпання апаратних ресурсів (TCAM), зростання затримок обробки трафіку та появи критичних логічних помилок через людський фактор.

Критичний аналіз існуючих підходів. На сьогоднішній день адміністрування ACL здійснюється переважно двома способами, кожен з яких має суттєві недоліки:

1) Ручне керування (CLI): Є найбільш поширеним, але не дозволяє адміністратору бачити цілісну картину взаємозв'язків між сотнями правил. Це призводить до появи аномалій екранування) та накопичення застарілих записів [7].
2) Статичні аналізатори: Існуючі програмні засоби (наприклад, модулі в Cisco DNA або SolarWinds) фокусуються на синтаксичній коректності, але ігнорують контекст реального трафіку. Вони можуть вказати на синтаксичну помилку, але не здатні запропонувати перестановку правил для підвищення продуктивності, оскільки не аналізують статистику використання [9].

Формалізація задачі дослідження. Виходячи з цього, науково-прикладна задача дипломної роботи полягає у розробці комплексної методики та програмного інструментарію, який поєднує в собі функції моніторингу (збору статистики) та активного адміністрування (оптимізації).

Для вирішення виявлених проблем необхідно реалізувати систему, що задовольняє наступним вимогам:

1) Вимога до моніторингу:

Система повинна збирати статистику спрацювань у реальному часі та класифікувати правила за частотою використання. На відміну від статичного підходу, де порядок правил фіксований, пропонується динамічний підхід: правила, що обробляють найбільший обсяг легітимного трафіку, повинні автоматично переміщуватися у початок списку. Це дозволить мінімізувати середній час обробки пакету (T_{avg}), наближаючи його до оптимального значення:

$T_{avg} \rightarrow \min$

при збереженні незмінної логіки фільтрації [9].

2) Вимога до виявлення аномалій:

Необхідно розробити алгоритм, здатний виявляти та класифікувати логічні колізії, описані в п. 1.2 (екранування, надлишковість, кореляція). Система має автоматично маркувати правила, які є «мертвим кодом», і пропонувати їх видалення для звільнення ресурсів пам'яті [8].

3) Вимога до аудиту безпеки:

Враховуючи загрози IP-спуфінгу, система повинна виконувати перевірку наявності специфічних захисних конструкцій згідно з рекомендаціями RFC 2827 [5]:

- Наявність заборони на вхідний трафік із Source IP = Internal Network на зовнішніх інтерфейсах (Anti-Spoofing).
- Контроль ізоляції сегментів (перевірка відсутності правил Permit Any до підмереж DMZ).
- Блокування відомих вразливих портів (наприклад, TCP/445, TCP/23), які не використовуються у бізнес-процесах.

4) Вимога до роботи в гетерогенному середовищі:

Алгоритми повинні коректно обробляти конфігурації як для дротових маршрутизаторів, так і для точок доступу або комутаторів, що обслуговують Wi-Fi сегменти, враховуючи специфіку мобільності клієнтів(рис. 1.4) [12].

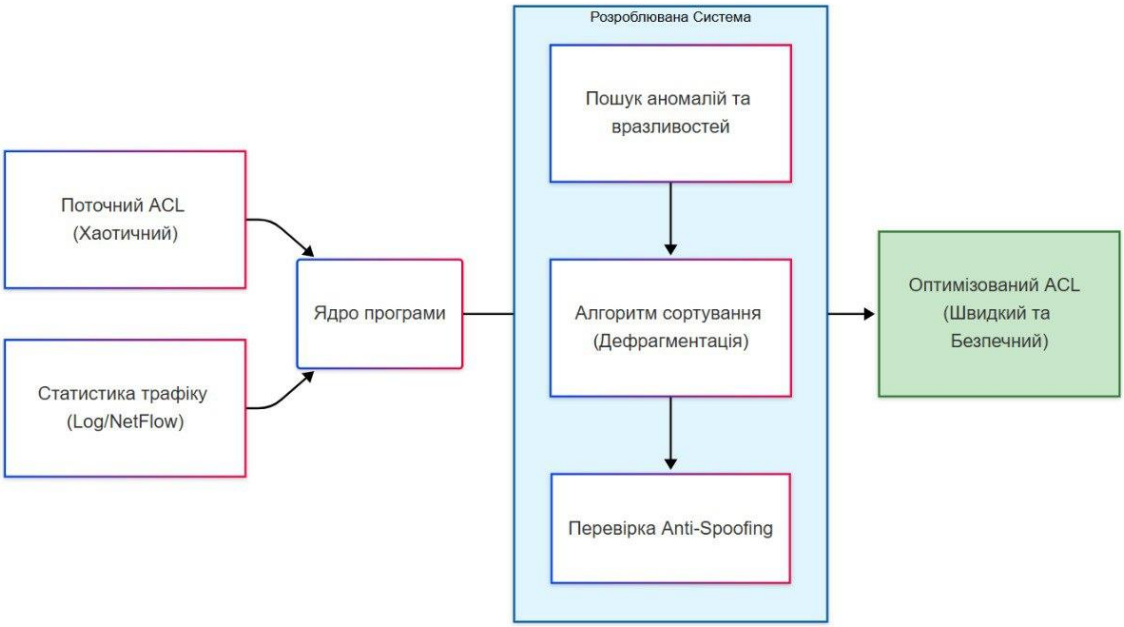

Рисунок 1.4 – Спрощена схема системи за сформованими вимогами

2 РОЗРОБКА АЛГОРИТМУ МОНІТОРИНГУ ТА СТРУКТУРНОЇ ОПТИМІЗАЦІЇ ACL

2.1 Загальна методика моніторингу мережевої активності

Забезпечення контрольованості поточних параметрів безпеки та ефективності функціонування сучасних інформаційних систем вимагає переходу від статичного сприйняття конфігурацій до динамічного аналізу процесів. Як зазначається у стандартах серії ISO/IEC 27000, ефективне керування інцидентами неможливе без постійного збору метрик функціонування захисних механізмів[1]. Традиційні методи адміністрування, за яких списки контролю доступу (ACL) розглядаються як незмінний набір інструкцій, не дозволяють адміністраторам відстежувати зміни ключових параметрів мережевої активності в реальному часі.

Така статичність призводить до зниження рівня ситуаційної поінформованості (Situational Awareness) - здатності персоналу об'єктивно оцінювати поточні параметри системи, ідентифікувати відхилення від норми та прогнозувати навантаження на обладнання. У даному підрозділі розроблено методику моніторингу, яка базується на введенні поняття «динамічного вагового коефіцієнта» правила. Цей підхід дозволяє перетворити абстрактні «сирі» дані про трафік на конкретні кількісні параметри, що характеризують внесок кожного окремого запису ACL у загальну продуктивність системи.

2.1.1 Формалізація задачі моніторингу параметрів активності

Для побудови математичної моделі розглянемо список контролю доступу L як впорядковану множину правил $\{r_1, r_2, \dots, r_N\}$, де N - загальна кількість правил, а індекс i $(1 \leq i \leq N)$ відповідає порядковому номеру правила у списку.

Ключовим параметром, що підлягає моніторингу, є інтенсивність використання кожного правила. У стандартних засобах маршрутизації (Cisco IOS, Juniper OS) цей параметр представлений лічильником спрацювань $C_i(t)$ (Hit Count)[10].

Основною проблемою використання «сирих» значень $C_i(t)$ для прийняття рішень щодо оптимізації є їх кумулятивний характер (постійне накопичення). Значення лічильника відображає сумарну кількість пакетів з моменту останнього перезавантаження пристрою. Це створює хибне уявлення про важливість правил: правило, яке було активним рік тому, але зараз є «мертвим кодом», може мати значно вищий абсолютний показник C_i ніж правило, що було додане вчора для критично важливого сервісу[7].

Для отримання об'єктивних параметрів функціонування необхідно перейти від абсолютних значень до диференціальних. Визначимо параметр миттєвої інтенсивності V_i(Velocity) на інтервалі моніторингу Δt:

$$V_i(t) = \frac{C_i(t_{current}) - C_i(t_{prev})}{t_{current} - t_{prev}} \quad (2.1)$$

де:

$t_{current}$ - поточний момент зчитування статистики;

t_{prev} - попередній момент зчитування;

C_i - значення лічильника пакетів (matches) для i-го правила.

Параметр V_i має фізичний зміст «швидкість обробки пакетів» (pps). Саме цей параметр відображає реальне навантаження, що дозволяє об'єктивно оцінити вплив кожного правила на поточні параметри швидкодії маршрутизатора.

2.1.2 Визначення динамічного вагового коефіцієнта

Для цілей структурної оптимізації (дефрагментації) використання лише миттєвої швидкості V_i є недостатнім, оскільки мережевий трафік має пульсуючий характер (bursty traffic)[11]. Оптимізація на основі миттєвих сплесків може призвести до частої та хаотичної перестановки правил (ефект «тремтіння» списку), що є небажаним.

Тому пропонується ввести інтегральний параметр - динамічну вагу правила (W_i). Вага правила визначає його пріоритет у списку: чим більша вага, тим вище повинно знаходитися правило для мінімізації навантаження на процесор.

Запропонована методика розрахунку ваги враховує два фактори:

1) Середня інтенсивність (V_i^{avg}): Характеризує стабільне навантаження на правило за період спостереження.

2) Пікова інтенсивність (V_i^{peak}): Характеризує максимальні сплески трафіку, які є критичними для CPU.

Розрахункова формула параметра ваги W_i має вигляд:

$$W_i = a \cdot V_i^{avg} + (1 - a) \cdot V_i^{peak} \quad (2.2)$$

де a - коефіцієнт згладжування ($0 < a < 1$).

Експериментальним шляхом рекомендовано значення $a \approx 0.7$. Це означає, що система віддає пріоритет стабільному трафіку, але на 30% враховує пікові навантаження. Такий підхід дозволяє стабілізувати параметри керування та уникнути помилкових рішень при короткочасних аномаліях.

2.1.3 Класифікація правил за параметрами активності

На основі розрахованих вагових коефіцієнтів W_i пропонується автоматизована класифікація множини правил ACL на чотири категорії. Це дозволяє персоналу миттєво оцінити параметри безпеки та прийняти рішення щодо адміністрування

1) Активні правила («Hot Rules»):

Правила, для яких параметр ваги W_i перевищує верхній порогове значення T_{hot} (наприклад, топ-10% правил).

Характеристика: Генерують до 80-90% навантаження на CPU маршрутизатора.

Дія системи: Кандидати на безумовне переміщення у початок списку.

2) Помірні правила («Warm Rules»):

Правила із середнім рівнем інтенсивності ($T_{cold} < W_i < T_{hot}$). Зазвичай обслуговують регулярні бізнес-процеси (пошта, веб-серфінг, бази даних).

Дія системи: Розміщуються у середній частині списку («тіло» ACL).

3) Рідкісні правила («Cold Rules»):

Правила з мінімальною, але ненульовою активністю ($W_i \to 0$). Часто це резервні канали керування або доступ для адміністраторів.

Дія системи: Розміщуються в кінці списку, оскільки їх вплив на параметри продуктивності є несуттєвим.

4) «Мертві» правила («Zero-hit Rules»):

Правила, параметри активності яких дорівнюють нулю протягом тривалого контрольного періоду (наприклад, $W_i = 0$ протягом 30 днів).

Характеристика: Свідчать про помилки конфігурації (забуті тимчасові доступи) або зміну мережевої топології. Є потенційними вразливостями.

Дія системи: Маркуються для аудиту та рекомендовані до видалення.

2.1.4 Порядок збору та обробки параметрів моніторингу

Для практичної реалізації методики розроблено алгоритм цикличної фіксації параметрів, який складається з наступних кроків:

1) Ініціалізація сесії: Встановлення захищеного каналу керування (SSH/Telnet) з мережевим обладнанням.

2) Зняття параметрів (Snapshot T1): Виконання команди отримання статистики (наприклад, show access-lists) та парсинг результатів у структуру даних «ID - Лічильник».

3) Пауза (Sampling Interval): Очікування протягом часу Δt (рекомендовано 60–300 секунд). Цей інтервал необхідний для накопичення репрезентативної вибірки трафіку.

4) Зняття параметрів (Snapshot T2): Повторне опитування обладнання.

5) Розрахунок метрик: Обчислення різниці значень та розрахунок параметрів інтенсивності V_i та ваги W_i за наведеними вище формулами.

6) Нормалізація: Приведення параметрів до єдиної шкали (0..100) для візуалізації на дашборді оператора.

2.1.5 Вплив на ситуаційну поінформованість

Впровадження даної методики дозволяє якісно змінити процес адміністрування, перейшовши від інтуїтивних рішень до керування на основі

точних метрик. Адміністратор отримує можливість контролювати параметри безпеки в динаміці:

- Відстежувати аномальні зміни параметрів трафіку (наприклад, різке зростання активності на "холодному" правилі може свідчити про початок атаки або сканування портів).
- Оцінювати ефективність поточних налаштувань та виявляти "вузькі місця".
- Приймати обґрунтовані рішення щодо дефрагментації (оптимізації) конфігурації[2].

Таким чином, розроблена методика моніторингу забезпечує отримання вхідних даних для алгоритмів структурної оптимізації, спрямованих на покращення параметрів функціонування системи, що буде розглянуто в наступних підрозділах.

2.2 Концепт алгоритму структурної оптимізації (дефрагментації) ACL на основі аналізу поточної статистики трафіку

Отримання даних про динамічну вагу правил W_i, описане в попередньому підрозділі, є необхідною, але недостатньою умовою для покращення продуктивності системи. Наступним етапом є розробка механізму структурної оптимізації, або «дефрагментації» списку контролю доступу.

Під терміном «дефрагментація ACL» у даній роботі розуміється комплексний процес трансформації впорядкованої множини правил, що включає дві складові:

1) Реордерінг (Reordering): Зміна порядку розташування правил для мінімізації часу пошуку.
2) Агрегація (Aggregation): Об'єднання кількох суміжних правил з однаковою дією в одне узагальнене правило (зменшення загального розміру списку N).

2.2.1 Постановка задачі оптимізації

З точки зору теорії алгоритмів, процес фільтрації трафіку є задачею лінійного пошуку (Linear Search). Маршрутизатор послідовно перевіряє заголовок пакету на відповідність правилам $\{r_1, r_2, \dots, r_N\}$.

Нехай C_{match} - це вартість (час) перевірки однієї умови. Тоді час обробки пакету, що підпадає під правило , становить:

$$T_k = k \cdot C_{match} \quad (2.3)$$

Якщо ми знаємо ймовірність спрацювання кожного правила $P(r_i)$ (яка є нормованим значенням нашої ваги W_i), то середній очікуваний час обробки пакету (Expected Matching Cost, *E*) для всього списку визначається формулою [9]:

$$E = \sum_{i=1}^{N} i \cdot P(r_i) \quad (2.4)$$

Цільова функція оптимізації: Необхідно знайти таку перестановку π індексів правил, при якій значення *E* буде мінімальним:

$$E(\pi) \to min \quad (2.5)$$

Математично це досягається, коли правила відсортовані у порядку спадання їх ймовірностей (ваг):

$$W_{\pi(1)} \geq W_{\pi(2)} \geq \cdots \geq W_{\pi(N)} \quad (2.6)$$

Однак, на відміну від звичайного сортування масиву даних, до списків ACL застосовується суворе обмеження: збереження семантичної еквівалентності. Результат обробки будь-якого пакету оптимізованим списком повинен бути ідентичним результату обробки оригінальним списком.

2.2.2 Особливості врахування залежностей і колізій

Пряме сортування правил за вагою (W_i) неможливе через наявність логічних залежностей між ними.

Нехай R_i та R_j - два правила, де $i < j$ (правило *i* стоїть вище).

Визначимо область дії правила *Space(R)* як множину всіх можливих IP-адрес та портів, що підпадають під це правило.

Правила R_i та R_j вважаються залежними (конфліктуючими), якщо виконуються дві умови [7]:

1) Перетин просторів: $Space(R_i) \cap Space(R_j) \neq \emptyset$ (мають спільні IP-адреси).
2) Різні дії: $Action(R_i) \neq Action(R_j)$ (одне дозволяє, інше забороняє).

Якщо правила залежні, їх взаємний порядок змінювати заборонено, оскільки це змінить політику безпеки (приклад аномалії кореляції).

Якщо ж правила незалежні (їх простори не перетинаються, або вони мають однакову дію), їх можна безпечно міняти місцями.

2.2.3 Процедури «безпечного спливання» та дефрагментації адресації

Для реалізації задачі оптимізації розроблено евристичний алгоритм, який намагається підняти «гарячі» правила (Hot Rules) якомога вище у списку, не порушуючи логічних залежностей.

Алгоритм працює ітеративно і нагадує метод «сортування бульбашкою», але з додатковою перевіркою конфліктів.

Формальний опис алгоритму:

Вхідні дані:

- Список правил $L = \{r_1, \dots, r_N\}$.
- Масив ваг $W = \{w_1, \dots, w_N\}$, отриманий з модуля моніторингу.

Кроки алгоритму:

1) Ідентифікація пріоритетних правил: Виділяємо множину правил S_{hot}, для яких $w_i > T_{threshold}$.
2) Цикл оптимізації: Для кожного пріоритетного правила $r_k \in S_{hot}$:

- Порівнюємо його з попереднім правилом r_{k-1}.
- Умова обміну: Якщо $w_k > w_{k-1}$ (нижнє правило популярніше за верхнє).
- Перевірка безпеки (Safety Check): Перевіряємо, чи перетинаються r_k та r_{k-1}.
- Якщо *Intersection*(r_k, r_{k-1}) = $\emptyset$ (перетину немає) → Виконуємо обмін (Swap).
- Якщо перетин є, але *Action*(r_k) == *Action*(r_{k-1}) → Виконуємо обмін.
- Якщо перетин є і дії різні → Обмін заборонено. Правило r_k «застрягає» під r_{k-1} через конфлікт безпеки.

3) Повторення: Процес повторюється до тих пір, поки можливі безпечні перестановки, що покращують цільову функцію *E*.

Цей метод гарантує, що «гаряче» правило підніметься вгору рівно настільки, наскільки це дозволяє логіка безпеки. Наприклад, правило дозволу доступу до популярного веб-сервера підніметься на початок списку, але зупиниться перед правилом антиспуфінгу, яке блокує вхідні атаки, що є критично важливим.

Друга частина оптимізації спрямована на зменшення загальної кількості правил *N* шляхом злиття надлишкових записів.

Алгоритм шукає пари суміжних правил r_i, r_{i+1}, які:

1) Мають однакову дію ($Action(r_i,) == Action(r_{i+1})$).
2) Мають однакові порти та протоколи.
3) Мають суміжні діапазони IP-адрес, які можна об'єднати методом CIDR-супернетингу (CIDR Supernetting).

Приклад:

Permit IP 192.168.1.0/24 та Permit IP 192.168.2.0/24 → об'єднуються в Permit IP 192.168.0.0/22 (узагальнення).

Агрегація виконується після етапу сортування, оскільки групування суміжних «теплих» правил дозволяє ще більше знизити навантаження на пам'ять TCAM [10].

2.2.4 Висновки щодо розробленого концепту алгоритму

Запропонований комбінований підхід (Safe Reordering + Aggregation) дозволяє досягти зниження навантаження на CPU за рахунок двох факторів:

1) Найбільш ймовірні пакети обробляються за мінімальну кількість кроків ($O(1)$ для топ-правил).
2) Загальна довжина списку *N* зменшується за рахунок видалення дублікатів та агрегації підмереж.

Таким чином, розроблена математична модель дозволяє автоматизувати процес прийняття рішень щодо конфігурування мережевого обладнання,

забезпечуючи баланс між вимогами продуктивності та вимогами безпеки (збереження логіки обмежень).

2.3 Специфіка виявлення логічних колізій та перевірка механізмів захисту від IP-спуфінгу

Структурна оптимізація списків контролю доступу (ACL), описана у підрозділі 2.2, дозволяє вирішити проблему продуктивності. Однак, автоматичне перевпорядкування правил без попереднього аналізу їх логічної коректності може призвести до консервації існуючих помилок конфігурації. Більше того, критичні вразливості, такі як відсутність захисту від підробки адрес (IP Spoofing), не можуть бути усунені простим сортуванням - вони вимагають семантичного аналізу змісту правил.

Тому невід'ємною складовою розроблюваної системи є модуль автоматизованого аудиту, який виконує дві функції:

1) Виявлення внутрішніх колізій (конфліктів між правилами в одному списку).
2) Перевірка на відповідність політикам безпеки (Compliance Check), зокрема рекомендаціям RFC 2827 щодо антиспуфінгу [5].

2.3.1 Теоретико-множинна модель правила ACL

Для програмної реалізації алгоритмів пошуку аномалій необхідно представити кожне правило ACL як математичний об'єкт.

Формально правило r_i можна описати як кортеж з п'яти елементів:

$$r_i = \{Src_i, Dst_i, Proto_i, Port_i, Action_i\} \qquad (2.7)$$

де:

- Src_i, Dst_i - множини IP-адрес джерела та призначення (у форматі CIDR);
- $Proto_i$ - протокол (TCP, UDP, ICMP);
- $Port_i$ - множина портів призначення;
- $Action_i \in \{Permit, Deny\}$.

Простір дії правила ($Space_i$) - це декартовий добуток множин його атрибутів. Два правила r_i та r_j вважаються такими, що мають перетин, якщо перетинаються всі їхні складові компоненти:

$$(Src_i \cap Src_j \neq \emptyset) \wedge (Dst_i \cap Dst_j \neq \emptyset) \wedge (Proto_i \cap Proto_j \neq \emptyset) \quad (2.8)$$

Використання бітових масок дозволяє звести операцію пошуку перетину до побітових логічних операцій, що забезпечує високу швидкодію алгоритму [7].

2.3.2 Особливості виявлення логічних аномалій і захисту від IP-спуфінгу

На основі класифікації Al-Shaer [7], розроблено алгоритм, який аналізує пари правил (r_i, r_j), де $i < j$ (правило i знаходиться вище у списку), і виявляє наступні типи аномалій:

А. Екранування (Shadowing) Найкритичніша аномалія. Правило r_j є екранованим правилом r_i, якщо простір r_j є підмножиною простору r_i, а дії різні.

Умова виявлення:

$$Space(r_j) \subseteq Space(r_i) \wedge Action(r_j) \neq Action(r_i) \quad (2.9)$$

Діагноз: Правило r_j є «мертвим кодом». Воно ніколи не спрацює, оскільки трафік буде перехоплено правилом r_i.

Реакція системи: Формування критичного попередження (Alert) та рекомендація видалити або перемістити r_j.

Б. Надлишковість (Redundancy)

Правило r_j є надлишковим щодо r_i, якщо його простір входить у простір r_i, а дії співпадають.

Умова виявлення:

$$Space(r_j) \subseteq Space(r_i) \wedge Action(r_j) = Action(r_i) \quad (2.10)$$

Діагноз: Правило r_j не змінює логіку фільтрації, але збільшує розмір списку N, навантажуючи CPU.

Реакція системи: Рекомендація видалити r_j.

В. Кореляція

Правила r_i та r_j мають частковий перетин (жодне не є підмножиною іншого), але мають різні дії.

Умова виявлення:

$$(Space(r_i) \cap Space(r_j) \neq \emptyset) \wedge (Space(r_i) \nsubseteq Space(r_j)) \wedge (Space(r_j) \nsubseteq Space(r_i)) \wedge (Action(r_i) \neq Action(r_j)) \quad (2.11)$$

Діагноз: У зоні перетину діє правило r_i. Якщо поміняти їх місцями, політика безпеки зміниться. Це потенційна «діра» в безпеці.

Реакція системи: Попередження адміністратору про необхідність ручної верифікації пріоритетів.

Окремою, критично важливою задачею, згідно з технічним завданням, є контроль захищеності периметра від атак із підробкою адреси джерела. Атака IP Spoofing полягає у відправці пакетів з зовнішньої мережі (Інтернет), які мають у заголовку IP-адресу, що належить внутрішній мережі підприємства [5].

Для автоматизованого виявлення цієї вразливості розроблено алгоритм семантичного аудиту, який вимагає від оператора вказати контекст:

1) Список зовнішніх інтерфейсів (Int_{ext}).
2) Діапазон внутрішніх IP-адрес мережі ($Int_{internal}$).

Алгоритм виконує перевірку кожного вхідного ACL (L_{in}) на зовнішньому інтерфейсі за наступною логікою:

1) Пошук блокуючого правила: Система шукає у початку списку правило виду:

$$Deny\ IP\ \{Src \in\ Int_{internal}\}\ Any \quad (2.12)$$

2) Аналіз порядку: Якщо таке правило знайдено, перевіряється, чи не передують йому дозволяючі правила (*Permit*), простір яких перетинається з $Int_{internal}$ (аномалія екранування захисту).
3) Вердикт:

- Якщо блокуюче правило відсутнє → Критична вразливість (High Risk). Порушення BCP 38 (RFC 2827).
- Якщо правило є, але стоїть надто низько → Помилка конфігурації. Рекомендація перемістити вгору.

2.3.3 Контроль сегментації ресурсів (DMZ Isolation Check)

Додатково алгоритм перевіряє коректність ізоляції демілітаризованої зони (DMZ) та бездротових сегментів. Система аналізує ACL на наявність правил типу *Permit Any Any* або *Permit Any DMZ_Host*, які дозволяють неконтрольований трафік з Інтернету до чутливих внутрішніх ресурсів, оминаючи проксі-сервери або WAF (Web Application Firewall).

Виявлення правил, що дозволяють доступ до портів управління (SSH 22, Telnet 23, RDP 3389) з зони Any, класифікується як порушення політики безпеки з найвищим пріоритетом виправлення.

2.3.4 Специфіка семантичного контролю сервісів та порушення сегментації

Окрім структурних аномалій, система повинна виявляти прямі порушення політики безпеки, спричинені помилками адміністратора при наданні доступів (так звані «Semantic Errors»). Для цього розроблено алгоритм, який базується на сигнатурному аналізі правил ACL.

А. Контроль відомих вразливих портів (Port Security Check)

Алгоритм перевіряє відповідність поля Port кожного дозволяючого правила (Action = Permit) з базою даних відомих загроз (Blacklist).

Формальна умова спрацювання тривоги:

$$Rule_i(Action) = Permit \wedge Rule_i(Port) \in P_{threat} \quad (2.13)$$

де P_{threat} - множина портів, асоційованих з відомим шкідливим ПЗ або небезпечними протоколами.

Зокрема, перевірці підлягають порти троянських програм (наприклад, UDP 31337 - Back Orifice, TCP 12345 - NetBus) та протоколи віддаленого керування без шифрування (TCP 23 - Telnet). Виявлення таких правил свідчить про критичну компрометацію периметра.

Б. Контроль матриці доступу між зонами (Zone Segmentation Verification)

Для захисту архітектури мережі від несанкціонованих горизонтальних переміщень (Lateral Movement) вводиться поняття «Матриці дозволених потоків».

Алгоритм аналізує пару «Джерело - Призначення» (*Src - Dst*) і порівнює її з політикою сегментації.

Система класифікує порушення, якщо виявлено правило, що дозволяє прямий трафік за векторами:

1) Wi-Fi → Wired: Дозвіл доступу з бездротових сегментів до критичної дротової інфраструктури без проходження через шлюз безпеки.
2) External → Internal: Дозвіл доступу з недовірених зовнішніх мереж до внутрішніх ресурсів, минаючи DMZ.

Такий підхід дозволяє автоматизувати виявлення грубих помилок конфігурування ще до етапу застосування правил на обладнанні.

2.3.5 Узагальнені висновки за п. 2.2

Розроблені алгоритми аудиту інтегруються в загальний цикл роботи системи:

1) Спочатку виконується Аудит. Виявлені «мертві» (екрановані) та надлишкові правила видаляються або деактивуються. Це зменшує обсяг вхідних даних (N).
2) Потім виконується Перевірка безпеки. Якщо виявлено відсутність антиспуфінгу, система генерує синтетичне правило блокування та пропонує додати його на початок списку.
3) Лише після цього виконується Оптимізація (дефрагментація), описана в п. 2.2, що гарантує, що ми оптимізуємо вже «чистий» та безпечний список.

Таким чином, запропонований комплекс алгоритмів забезпечує не лише покращення продуктивності, але й гарантує цілісність політики безпеки, мінімізуючи ризики, пов'язані з людськими помилками конфігурування.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЦЕДУР АДМІНІСТРУВАННЯ І АУДИТУ ACL ТА РЕЗУЛЬТАТИ КОМПЬЮТЕРНОГО МОДЕЛЮВАННЯ

3.1 Архітектура та функціональні можливості програмного прототипу для адміністрування і аудиту ACL

Для практичної перевірки методики моніторингу та алгоритмів оптимізації, обґрунтованих у другому розділі, було створено програмний прототип системи автоматизованого адміністрування. Програмний засіб розроблено у вигляді додатку з графічним інтерфейсом користувача (GUI)(рис. 3.1), що забезпечує виконання повного циклу обслуговування списків контролю доступу і дозволяє проводити тестування зі статичним логом(в якому зберігається список правил ACL та кількість трафіку, що пройшов через кожне правило) і проводити моніторинг списку ACL та трафіку маршрутизатора у реальному часі. З ціллю тестування також було створено програму для генерації трафіку для списків ACL.

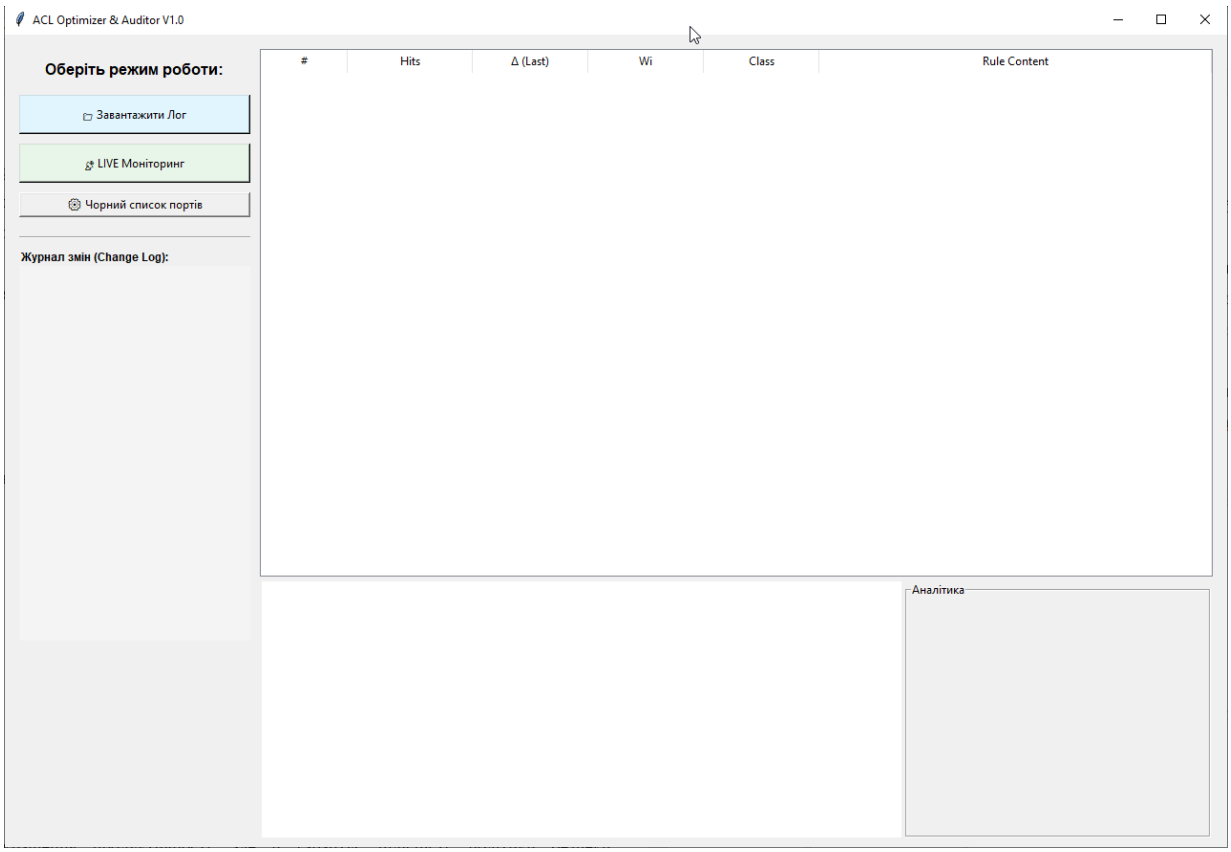

Рисунок 3.1 – Інтерфейс користувача після запуску програми

Для того, щоб почати роботу з програмою, потрібно обрати режим з яким збираєшся працювати(рис. 3.2): «Завантажити лог»(статичний аналіз ACL) або «LIVE моніторинг»(спостереження за ACL у реальному часі).

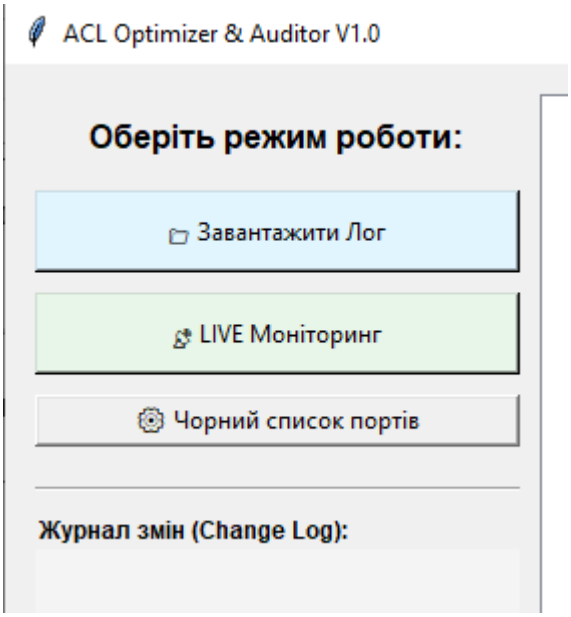

Рисунок 3.2 – Вибір режиму роботи

Для демонстрації функціональних можливостей програми оберемо варіант «Завантажити лог»(рис. 3.3), щоб перевірити їх на статичному списку ACL з емульованим трафіком.

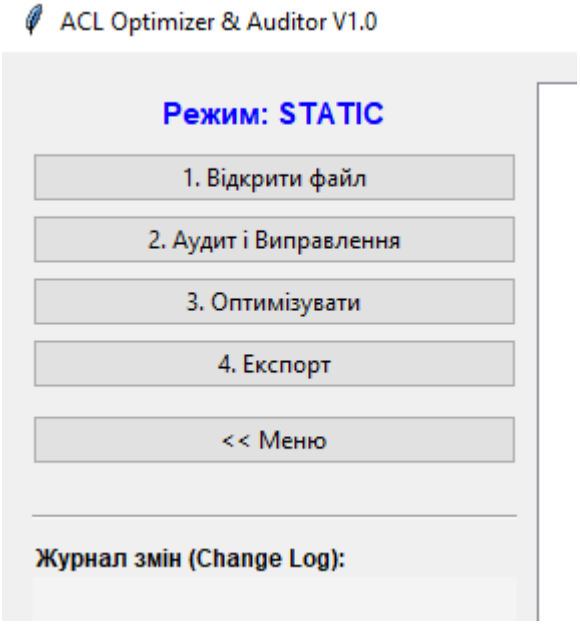

Рисунок 3.3 – Список доступних дій для статичного аналізу та редагування ACL

З рисунку видно, що в першу чергу програма дає змогу відкрити файл зі збереженим списком ACL(рис. 3.4), після чого на екрані з'являється сам список правил, їх порядковий номер, кількість спрацювань кожного правила(Hits), кількість спрацювань кожного правила за останній проміжок часу(Δ)(обирається адміністратором), а також динамічна вага кожного правила(W_i), яка вираховується за формулою, приведеною у попередніх пунктах розділу 2. Оскільки ми розглядаємо варіант статичного аналізу, тут (Δ) відповідно рівна 0, тому що значення трафіку не змінюються.

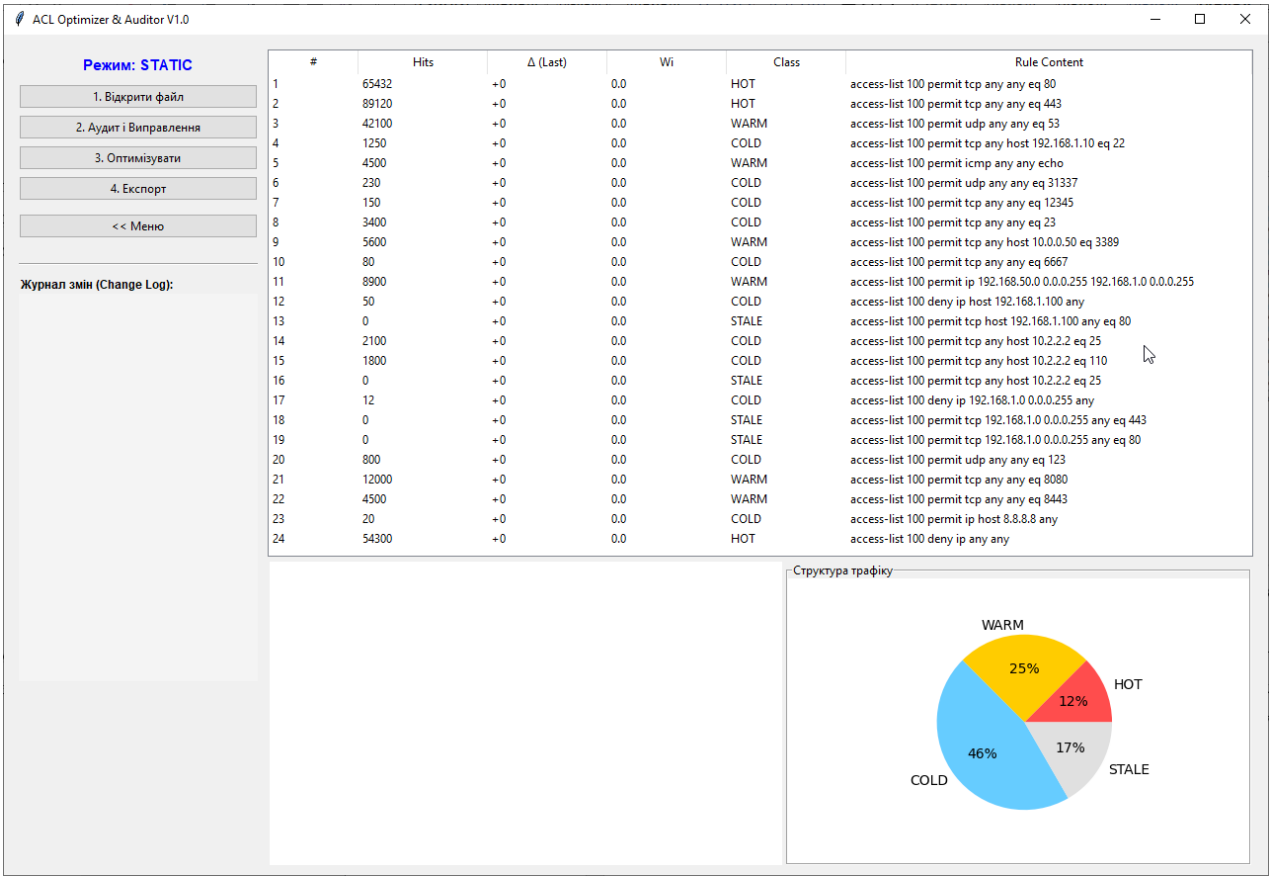

Рисунок 3.4 – інтерфейс користувача після завантаження списку ACL

Як бачимо з рисунку, справа знизу програма створює діаграму, на якій зображена структура трафіку, тобто розподіл правил на групи за рівнем їх активності. Ознайомившись з даним інтерфейсом, можемо почати аналіз та редагування поточного списку правил. Спробуємо натиснути на «аудит і виправлення»(рис. 3.5).

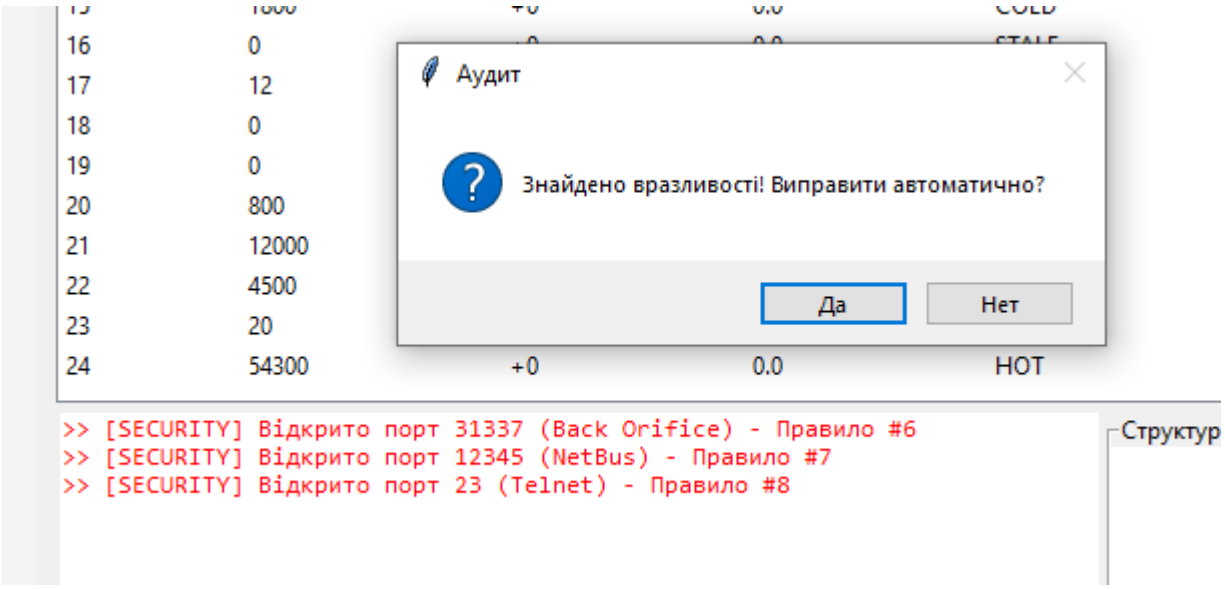

Рисунок 3.5 – Результат виконання аудиту

Бачимо, що після виконання аудиту в нижньому полі з’явилися червоні повідомлення про небезпечні правила(правила з відкритими небезпечними портами), які є слабким місцем нашого списку. Програма пропонує нам автоматично виправити знайдені вразливості, спробуємо це зробити(рис. 3.6).

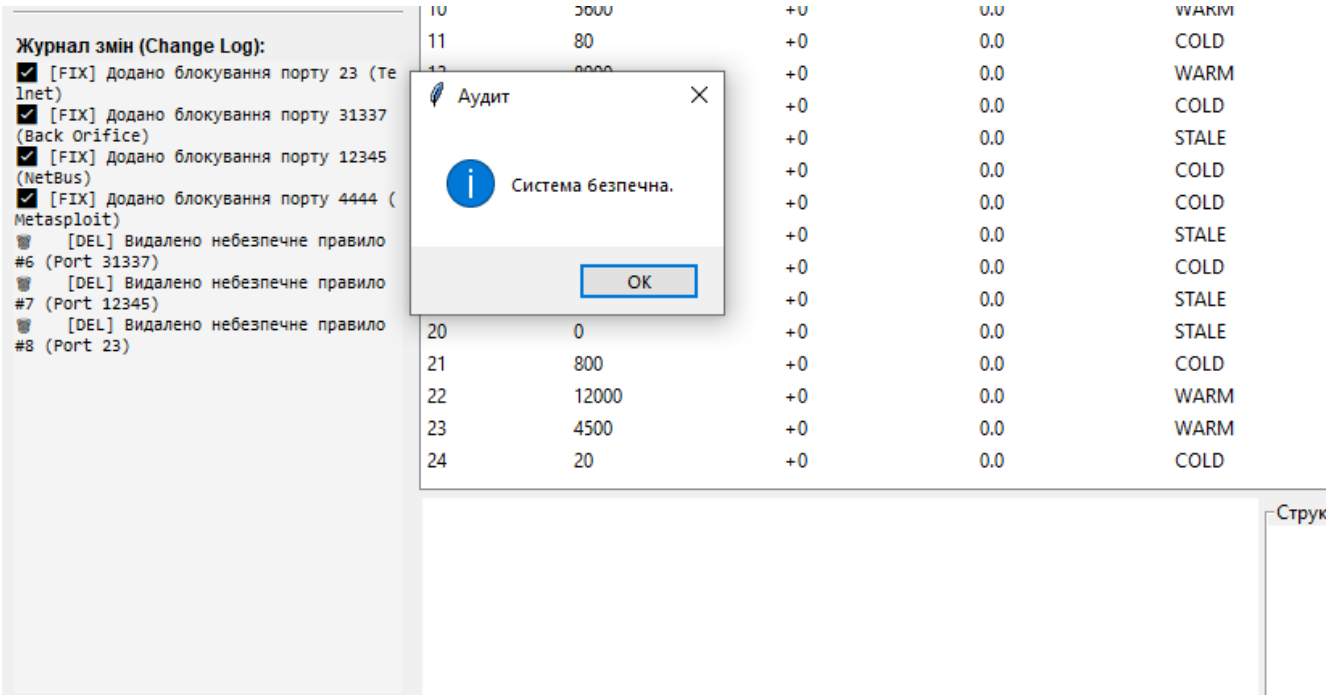

Рисунок 3.6 – Результат автоматичного виправлення знайдених вразливостей нашого ACL

Як можна помітити, червоні повідомлення про вразливості зникли, натомість у лівому полі з’явились повідомлення у «журналі змін», куди записуються всі зміни щодо поточного ACL.

На рисунку 3.6 видно, що програма не тільки видалила небезпечні правила, які давали доступ до вразливих портів, але й заблокувала ці самі порти, тобто створила нові правила для нашого списку.

Також програма дає змогу виконати «оптимізацію», тобто перестановку правил з ціллю збільшення ефективності обробки трафіку та зменшення навантаження на CPU маршрутизатора за рахунок зменшення кількості умовних циклів, які виконують пакети при пошуку правил. Так як ми пробуємо виконати оптимізацію(рис. 3.7).

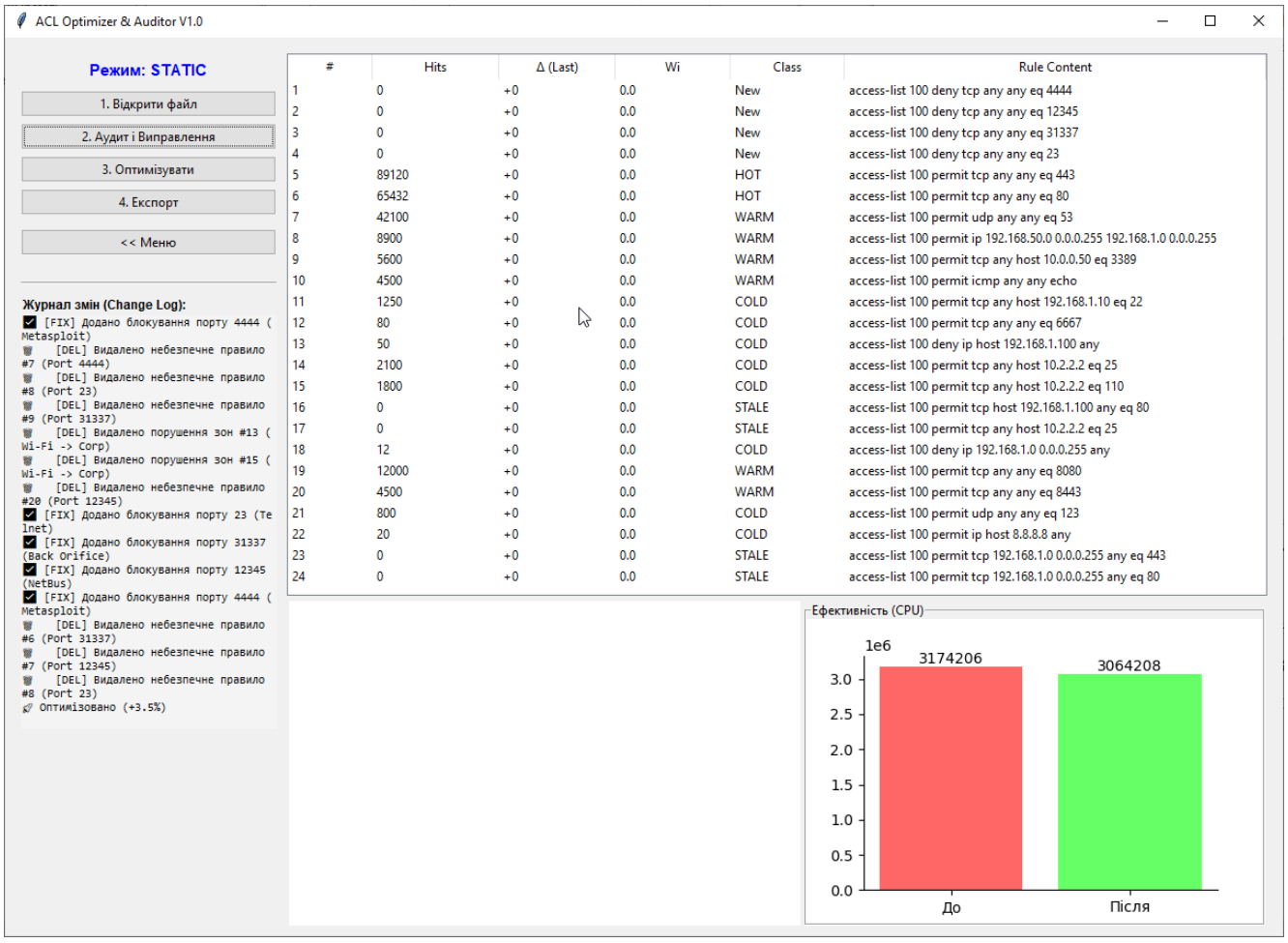

Рисунок 3.7 – Результат оптимізації списку ACL

Оскільки початковий список вже був майже «ідеальним», тобто найпопулярніші правила ("Hot Rules"), які генерують 90% трафіку, вже стояли на

перших місцях, а також кількість правил є доволі малою, програма отримала «поганий» результат у покращенні ефективності(всього 3.5%, як можна побачити з графіків та журналу змін). Мій алгоритм "Safe Bubble-Up" намагається підняти популярні правила вгоу, але, оскільки вони вже там, алгоритму просто нікуди їх рухати. Через це оптимізація не сильно покращила ситуацію, адже він з самого початку був хорошим.

Останньою функцією даного режиму є «експорт»(рис. 3.8), який дозволяє зберегти відредагований та оптимізований список ACL у вигляді текстового файлу як просто список, або як скрипт Cisco, тобто як команду, яку можна написати у консолі маршрутизатора на Cisco IOS і встановити даний ACL на нього.

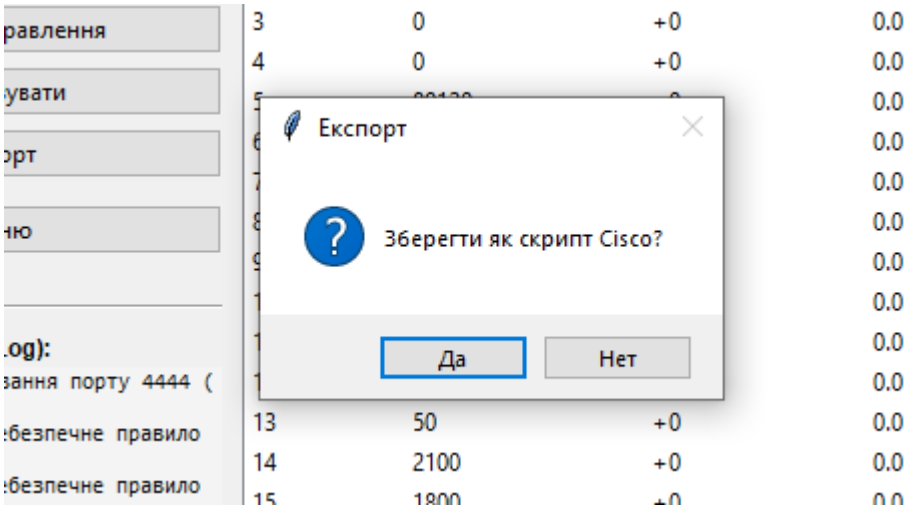

Рисунок 3.8 – Експорт відредагованого списку ACL

3.1.1 Засоби розробки та середовище виконання

У якості мови програмування обрано Python 3.11. Ця версія забезпечує високу швидкодію виконання коду, що є критичним при обробці великих масивів правил. Для реалізації функціональних блоків використано спеціалізовані бібліотеки:

1) Модуль ipaddress: Математичне ядро роботи з адресним простором. Він забезпечує об’єктно-орієнтований інтерфейс для створення, маніпулювання та виконання арифметичних операцій над IPv4 та IPv6 адресами і мережами.

У розробленій системі цей модуль виконує функцію математичного ядра для алгоритмів пошуку колізій. Використання рядкових операцій для порівняння IP-адрес є неефективним та схильним до помилок, оскільки запис 192.168.1.0/24 семантично перекриває 192.168.1.5, але текстово вони різні.

Бібліотека ipaddress вирішує цю проблему шляхом перетворення рядкового представлення адреси у 32-бітне ціле число (Integer). Це дозволяє звести задачу пошуку перетинів діапазонів до швидких бітових операцій та операцій порівняння чисел.

Ключові методи бібліотеки, що використовуються у ядрі аудиту:

- ip_network(address): Конструктор, який перетворює рядок CIDR (наприклад, "10.0.0.0/8") у об'єкт мережі. Він автоматично валідує коректність маски та обчислює адресу мережі і широкомовну адресу.
- subnet_of(other): Метод, що реалізує логіку перевірки підмножин. Повертає True, якщо мережа A повністю входить у мережу B ($A \subseteq B$). Це є базою для виявлення аномалії Екранування: якщо вузьке правило є підмножиною широкого правила, що стоїть вище, і дії різні - це конфлікт.
- overlaps(other): Метод перевірки перетину. Повертає True, якщо дві мережі мають хоча б одну спільну IP-адресу ($A \cap B \neq \emptyset$). Використовується для виявлення аномалії Кореляції.

Використання цього модуля дозволило відмовитися від написання власного складного парсера бітових масок, що підвищило надійність системи та зменшило ймовірність алгоритмічних помилок при аудиті.

2) Бібліотека tkinter та модуль ttk: Графічний інтерфейс користувача. Для забезпечення кросплатформеності та відсутності зовнішніх залежностей інтерфейс реалізовано на базі стандартної бібліотеки tkinter. Використано віджет ttk.Treeview для табличного відображення списку правил ACL, їх статистики та розрахованих вагових коефіцієнтів. Інтерфейс побудовано за подієво-орієнтованою моделлю, що забезпечує реакцію на дії користувача (завантаження файлів, запуск аудиту, оптимізація) в реальному часі.

3) Бібліотека Matplotlib: Візуалізація аналітичних даних. Matplotlib - найпотужніший інструмент для наукової візуалізації у середовищі Python.
Особливістю реалізації у даній роботі є інтеграція Matplotlib у вікно Tkinter. Для цього використано спеціальний клас-посередник FigureCanvasTkAgg. Це дозволяє малювати графіки не в окремому спливаючому вікні, а безпосередньо всередині інтерфейсу програми, у спеціально відведеному фреймі (self.chart_frame).
Програма генерує два типи діаграм:

- Кругова діаграма: Використовується на етапі моніторингу для відображення структури правил за класами активності («HOT», «WARM», «COLD», «STALE»). Це дає адміністратору миттєве розуміння «здоров'я» конфігурації.
- Стовпчикова діаграма: Використовується на етапі аналізу ефективності для порівняння метрики «Умовне навантаження CPU» до та після дефрагментації.

Використання Matplotlib дозволяє автоматично масштабувати графіки, додавати легенди, підписи осей та значення над стовпчиками, що робить звітність професійною та зрозумілою.

4) Модуль random та алгоритми імітаційного моделювання. Оскільки розробка та тестування системи проводилися в лабораторних умовах без доступу до «живого» магістрального трафіку, було розроблено підсистему генерації синтетичних даних на базі модуля random. Для забезпечення реалістичності експерименту алгоритм генерації статистики (generate_fake_statistics) не просто видає випадкові числа, а моделює розподіл навантаження відповідно до закону Парето (принцип 80/20). Логіка генератора реалізована через умовні ймовірності:

- З ймовірністю 20% правило отримує значну кількість хітів (15 000 – 60 000), імітуючи популярні сервіси.
- З ймовірністю 30% - середню кількість.
- Решта правил отримують мінімальні або нульові значення.

Такий підхід дозволяє коректно перевірити роботу алгоритму сортування «Safe Bubble-Up», оскільки створює умови, за яких просте перемішування правил дає суттєвий виграш у продуктивності, що й було підтверджено у пункті 3.3

5) Робота з текстовими даними та регулярні вирази. Парсинг конфігураційних файлів Cisco IOS, які є слабоструктурованим текстом, реалізовано за допомогою вбудованих методів роботи з рядками (split, strip, in). Алгоритм парсингу (load_from_text) виконує порядкове читання файлу, ігнорує коментарі та розбиває кожен рядок правила на токени (дія, протокол, джерело, призначення, порт). Це дозволяє трансформувати текстову конфігурацію у список об'єктів Python, з якими надалі працюють алгоритми оптимізації.

6) Структура програми Програмний код розділено на два основні класи, що відповідає принципам об'єктно-орієнтованого програмування:

- ACLProcessor: Відповідає за бізнес-логіку (парсинг, математичні розрахунки, аудит безпеки, алгоритм сортування).
- ACLApp: Відповідає виключно за візуалізацію даних та взаємодію з користувачем.

Використання зазначеного стеку технологій дозволило створити ефективний, кросплатформений та розширюваний програмний продукт, який повністю вирішує поставлені у роботі завдання.

3.1.2 Особливості модульної архітектури програмного стенду

Архітектура програмного комплексу(рис. 3.9): побудована за модульним принципом, що дозволяє гнучко розширювати функціонал. Система складається з трьох ключових компонентів:

1) Модуль збору даних (Data Collector): Відповідає за отримання вхідної конфігурації. Реалізовано підтримку завантаження текстових файлів (експорт з Cisco IOS)(рис. 3.10).
2) Модуль також включає Генератор синтетичного трафіку, який імітує роботу реального маршрутизатора, присвоюючи правилам значення лічильників

спрацювань (Hit Counts) відповідно до закону Парето (20% правил генерують 80% навантаження).

3) Ядро аналізу та аудиту: Це центральний елемент системи, який реалізує алгоритми пошуку помилок. Він включає два підмодулі:

- *Структурний аналізатор:* виявляє логічні колізії, такі як екранування та «мертвий код».
- *Семантичний аналізатор (Semantic Check):* виконує перевірку на відповідність корпоративним політикам безпеки. Саме цей блок відповідає за виявлення помилок адміністрування.

4) Модуль оптимізації:

Виконує функцію дефрагментації списку. На основі розрахованих динамічних ваг (W_i) модуль здійснює перевпорядкування правил (алгоритм Safe Bubble-Up), переміщуючи найбільш активні записи вгору списку для зниження навантаження на CPU.

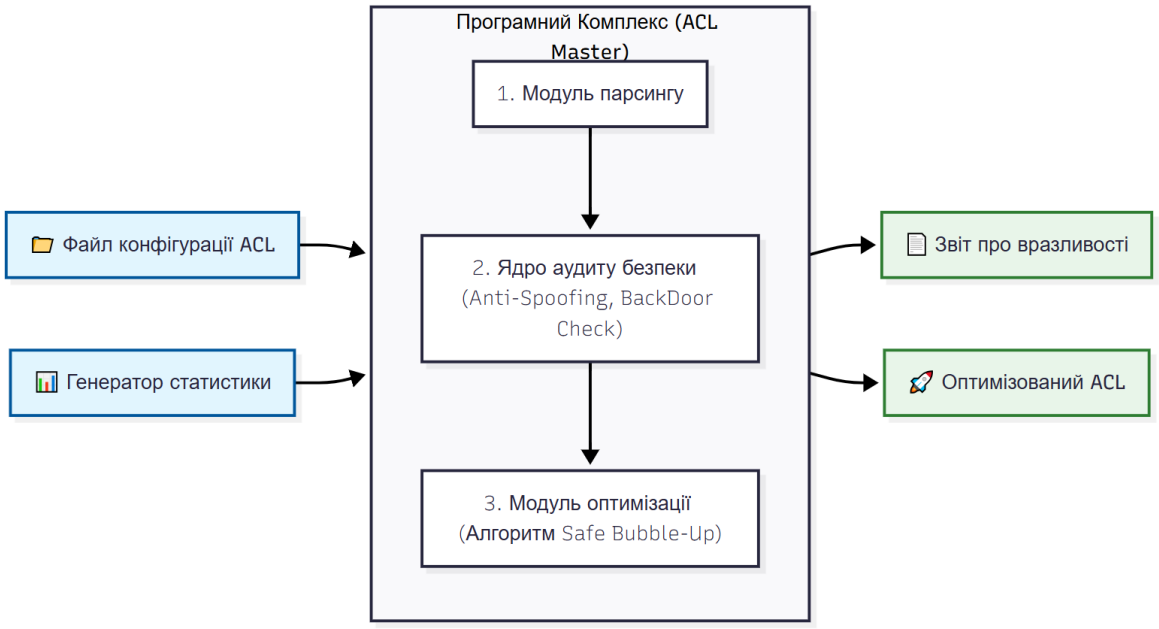

Рисунок 3.9 - Архітектура програмного комплексу

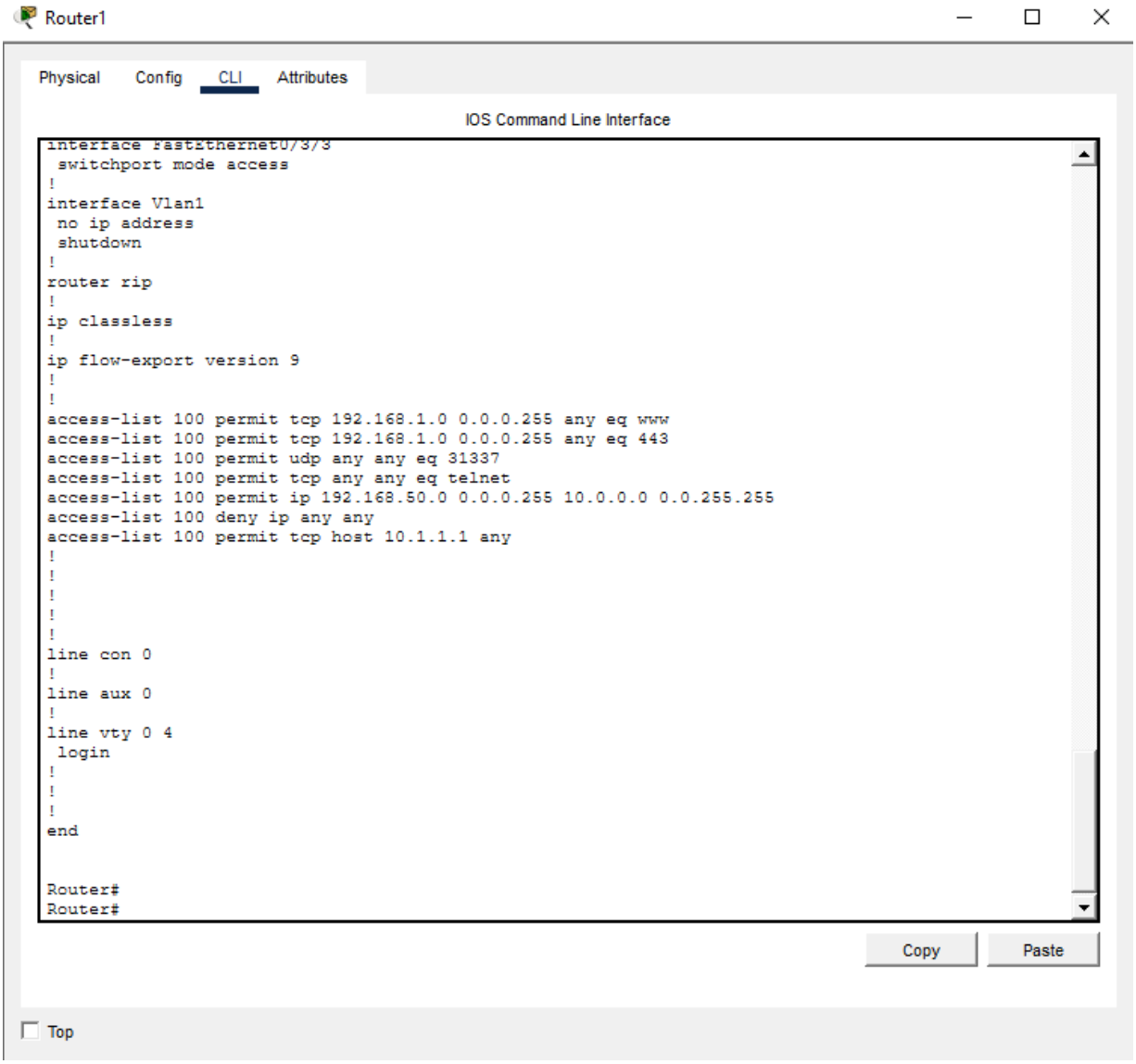

Рисунок 3.10 – Список правил всередині конфігурації віртуального роутеру Cisco

3.1.3 Функціональні можливості виявлення загроз

Розроблений прототип реалізує розширений набір перевірок безпеки, спрямований на усунення типових помилок конфігурування:

А. Контроль відомих вразливих портів (Trojan Port Detection)

Програма містить вбудовану базу сигнатур небезпечних портів(«чорний список», який можна редагувати всередині програми під час роботи). Під час аудиту виконується сканування дозволяючих правил (action=permit) на наявність портів, що використовуються шкідливим ПЗ. При наявності таких правил, система видаляє їх та створює правила, що блокують ці порти.

Приклад реалізації: Якщо система виявляє правило *permit udp any any eq 31337*, генерується критичне попередження про загрозу вірусу Back Orifice. Також контролюються порти віддаленого керування без шифрування (Telnet, TCP 23).

Б. Контроль сегментації мережі (Zone Segmentation Check)

Реалізовано алгоритм перевірки матриці доступу між зонами з різним рівнем довіри. Система автоматично аналізує правила на предмет дозволу трафіку за вектором Wi-Fi → Corporate LAN.

Приклад: Якщо виявлено правило, що дозволяє доступ з підмережі 192.168.50.0/24 (гостьовий Wi-Fi) до 10.0.0.0/8 (внутрішні сервери), це класифікується як порушення ізоляції сегментів і вимагає негайного виправлення.

В. Захист від IP-спуфінгу (Anti-Spoofing Verification)

Система перевіряє наявність на початку вхідного ACL правила, яке блокує пакети з адресою джерела, що належить внутрішній мережі. Відсутність такого правила кваліфікується як критична вразливість відповідно до RFC 2827. Якщо правило все ж таки відсутнє, програма сама створює його та пересуває вгору на перші позиції списку.

Таким чином, розроблений програмний засіб дозволяє не лише оптимізувати продуктивність мережевого обладнання, але й автоматизувати процес виявлення грубих помилок адміністрування, значно підвищуючи загальний рівень захищеності інформаційної системи.

3.2 Тестове середовище та специфіка сценаріїв моделювання (імітація потоків даних та спроб несанкціонованого доступу)

Для верифікації розроблених алгоритмів та оцінки ефективності програмного прототипу було застосовано метод імітаційного моделювання (Software Simulation). Метою даного етапу є підтвердження здатності системи аналізувати текстові конфігурації на предмет прихованих загроз (порушення логіки сегментації, відкриті порти), а також перевірка ефективності алгоритмів структурної оптимізації при обробці великих масивів даних, що імітують реальне навантаження

3.2.1 Тестова мережа, емуляція трафіку та профілювання навантаження

Для проведення досліджень було синтезовано віртуальну модель корпоративної мережі, архітектура якої відтворює типові вразливі зони, визначені у розділі 1. Моделювання виконувалося у середовищі Cisco Packet Tracer. Топологія (рис. 3.11) побудована за схемою «Router-on-a-Stick» і включає чотири ізольовані сегменти з різним рівнем довіри:

1) Зовнішня мережа (Internet/WAN): Джерело легітимного вхідного трафіку та потенційних атак (зокрема, IP-спуфінгу).
2) Корпоративна мережа (Corporate LAN): Захищений сегмент (192.168.1.0/24), де розміщені робочі станції персоналу. Політика безпеки вимагає суворої ізоляції цього сегменту від неавторизованих підключень.
3) Бездротовий сегмент (Guest Wi-Fi): Зона низької довіри (192.168.50.0/24). Згідно з політикою безпеки, прямий доступ з цієї мережі до Corporate LAN заборонено.
4) Серверний сегмент (DMZ): Зона для розміщення публічних сервісів (10.0.0.0/8). Доступ до неї дозволено ззовні лише за визначеними протоколами (HTTP/HTTPS).

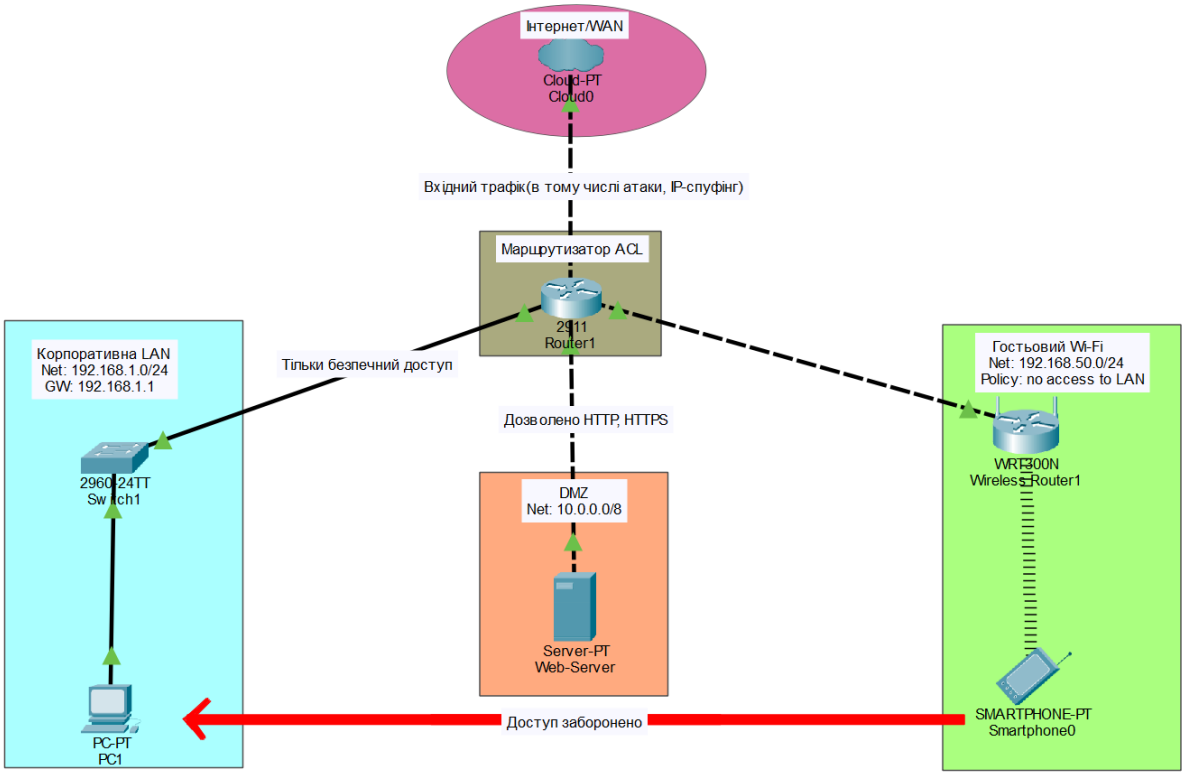

Рисунок 3.11 - Топологія мережі №1

Роль центрального вузла виконує маршрутизатор, на якому розгорнуто розширені списки контролю доступу (Extended ACL). Саме конфігурація цього вузла є об'єктом аудиту та оптимізації.

Варто зазначити, що логічна ізоляція зазначених сегментів на канальному рівні забезпечується використанням технології VLAN (Virtual Local Area Network) згідно зі стандартом IEEE 802.1Q. Маршрутизація трафіку між зонами здійснюється через налаштовані на центральному маршрутизаторі віртуальні субінтерфейси (Sub-interfaces), кожен з яких слугує шлюзом за замовчуванням для відповідної підмережі.

Для тестування алгоритму дефрагментації (оптимізації) критично важливо мати репрезентативні статистичні дані. Оскільки використання статичних лічильників не дозволяє оцінити динаміку, було розроблено окремий програмний засіб для створення реалістичного і в той же час експериментального трафіку, що здатен підлаштовуватись під різні ACL (рис. 3.12).

Генератор емулює потік пакетів, що проходить через інтерфейси маршрутизатора, розподіляючи навантаження між правилами ACL за законом Парето (принцип 80/20):

- 20% правил («Гарячі»): Генерують 80% загальної кількості збігів (Hit Counts). Це імітує інтенсивний трафік веб-серверів, потокового відео та баз даних.
- 30% правил («Теплі»): Генерують помірний трафік (DNS-запити, поштові сервіси).
- 50% правил («Холодні» та «Мертві»): Генерують одиничні запити або мають нульову активність.

Такий підхід дозволяє змоделювати ситуацію «найгіршого сценарію» (Worst-Case Scenario), коли найбільш активні правила знаходяться в кінці списку, створюючи максимальне навантаження на процесор.

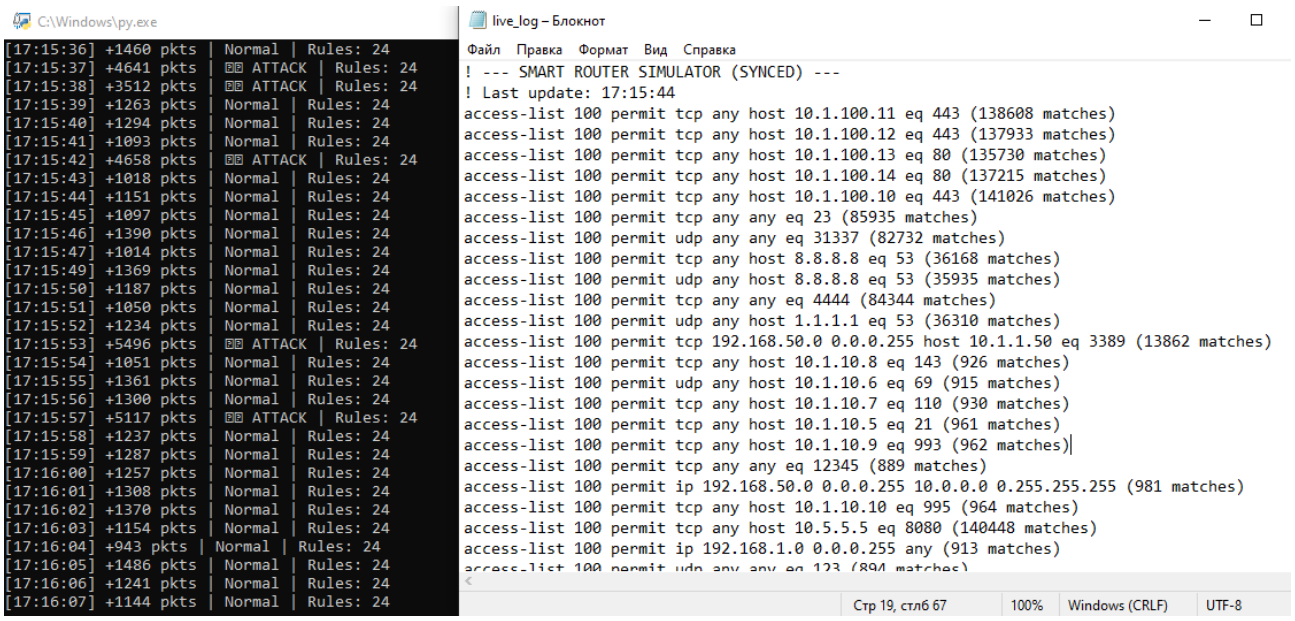

Рисунок 3.12 – Робота програмного засобу з імітації живого трафіку

3.2.2 Сценарії тестування та специфіка оцінки результатів

Для комплексної перевірки функціоналу системи розроблено три сценарії, що покривають задачі оптимізації продуктивності та забезпечення безпеки.

Сценарій А: «Аудит параметрів безпеки»:

- *Мета:* Перевірка здатності системи виявляти критичні вразливості конфігурації.
- *Вхідні дані:* Спеціально підготовлений «вразливий» файл конфігурації, що містить:

1) Відсутність правила блокування вхідного трафіку з адресами 192.168.1.0/24 на зовнішньому інтерфейсі.
2) Правило *permit udp any any eq 31337* (імітація бекдору трояна Back Orifice).
3) Правило *permit ip 192.168.50.0 0.0.0.255 192.168.1.0 0.0.0.255* (прямий доступ Wi-Fi → LAN).

- *Очікуваний результат:* Модуль аудиту повинен ідентифікувати всі три порушення, класифікувати їх за рівнем загрози та надати рекомендації щодо виправлення.

Сценарій Б: «Пошук логічних колізій»:

- *Мета:* Виявлення структурних помилок, що призводять до появи «мертвого коду».
- *Вхідні дані:* Список ACL, де правило deny ip any any розміщено в середині списку (наприклад, рядок 10 із 20).
- *Процес:* Запуск модуля структурного аналізу.
- *Очікуваний результат:* Система має виявити аномалію типу Shadowing та вказати, що всі правила після 10-го рядка є неактивними.

Сценарій В: «Оптимізація продуктивності»:

- *Мета:* Оцінка ефективності алгоритму «безпечного спливання» (Safe Bubble-Up).
- *Вхідні дані:* Неоптимізований список із 100 правил. Генератором трафіку присвоюється найвищий ваговий коефіцієнт ($W_i > 90$) правилу, що знаходиться на 95-й позиції.
- *Процес:* Запуск модуля дефрагментації.
- *Очікуваний результат:* Правило-лідер має перемістится у верхню частину списку (позиції 1–5), не порушуючи при цьому логічних залежностей з блокуючими правилами.

Ефективність роботи системи оцінюється за двома групами показників:

1) Показники безпеки: Повнота виявлення введених вразливостей (Recall) та відсутність хибних спрацювань (False Positives).
2) Показники продуктивності: Зниження умовного «вартості пошуку» ($Cost_{search}$) пакету в списку.

Вартість пошуку розраховується як індекс правила, помножений на його інтенсивність. Оптимізація вважається успішною, якщо сумарна вартість обробки трафіку зменшується:

$$E_{after} < E_{before}$$

Проведення експериментів за наведеною методикою дозволяє отримати об'єктивні дані про роботу розробленого програмного засобу та підтвердити його практичну цінність для адміністраторів мережевої безпеки.

3.3 Узагальнення результатів моделювання

Завершальним етапом дослідження став кількісний та якісний аналіз даних, отриманих в ході роботи програмного прототипу на тестових сценаріях (описаних у п. 3.2). Оцінка проводилася за двома ключовими критеріями: ефективність виявлення загроз та ефективність оптимізації продуктивності. Аналіз проводився за порівняльною методологією «до та після» (Before/After comparison), що дозволило чітко виокремити вплив запропонованих алгоритмів на параметри системи. Усі вимірювання здійснювалися в однакових умовах навантаження, що забезпечує чистоту експерименту та достовірність отриманих результатів.

3.3.1 Аналіз результатів автоматизованого аудиту

У ході експерименту було оброблено тестовий файл конфігурації, що містив 5 штучно внесених вразливостей різного типу. Модуль Audit Core продемонстрував 100% результативність виявлення (Recall = 1.0) без хибних спрацювань (False Positives = 0).

Результати роботи модуля аудиту зведено у Таблицю 3.1.

Таблиця 3.1 – Результати роботи модуля аудиту

	Тип вразливості	**Умова пошуку**	**Результат системи**	**Статус**
	IP Spoofing Risk	Відсутність deny ip {LAN_NET} any на початку	Виявлено [CRITICAL]	*Успіх*
	Trojan Port	permit ... eq 31337 (Back Orifice)	Виявлено [SECURITY]	*Успіх*
	Unsecured Mgmt	permit ... eq 23 (Telnet)	Виявлено [SECURITY]	*Успіх*
	Zone Violation	Src=Wi-Fi AND Dst=Corp_LAN	Виявлено [SEGMENTATION]	*Успіх*
	Shadowing	Deny Any передує іншим правилам	Виявлено [LOGIC]	*Успіх*

Для візуалізації роботи модуля аудиту в інтерфейсі розробленої програми(рис. 3.13) була емульована ситуація критично низького рівня адміністрування мережевого периметра. Тестування відбувалося в режимі моніторингу в реальному часі. Вхідним даним є файл конфігурації, який містить хаотичний набір правил, накопичених у процесі тривалої експлуатації мережі без належного аудиту. Програма імітації трафіку наповнює файл «відвідуваннями» кожної секунди, а програма для моніторингу кожні 3 секунди(проміжок можна обирати самому, зазвичай адміністратори використовують період 1-5 хвилин для спостереження за трафіком, в нашому випадку для наочності тестування обрано саме 3 секунди) Структура цього списку характеризується розміщенням малоактивних та "сміттєвих" правил у верхній частині, тоді як критично важливі правила, що обслуговують 90% легітимного бізнес-трафіку (HTTPS/Web), знаходяться в самому кінці списку. Така архітектура змушує процесор маршрутизатора виконувати повний перебір всього списку для кожного дозволеного пакету, що створює максимальне навантаження на CPU.

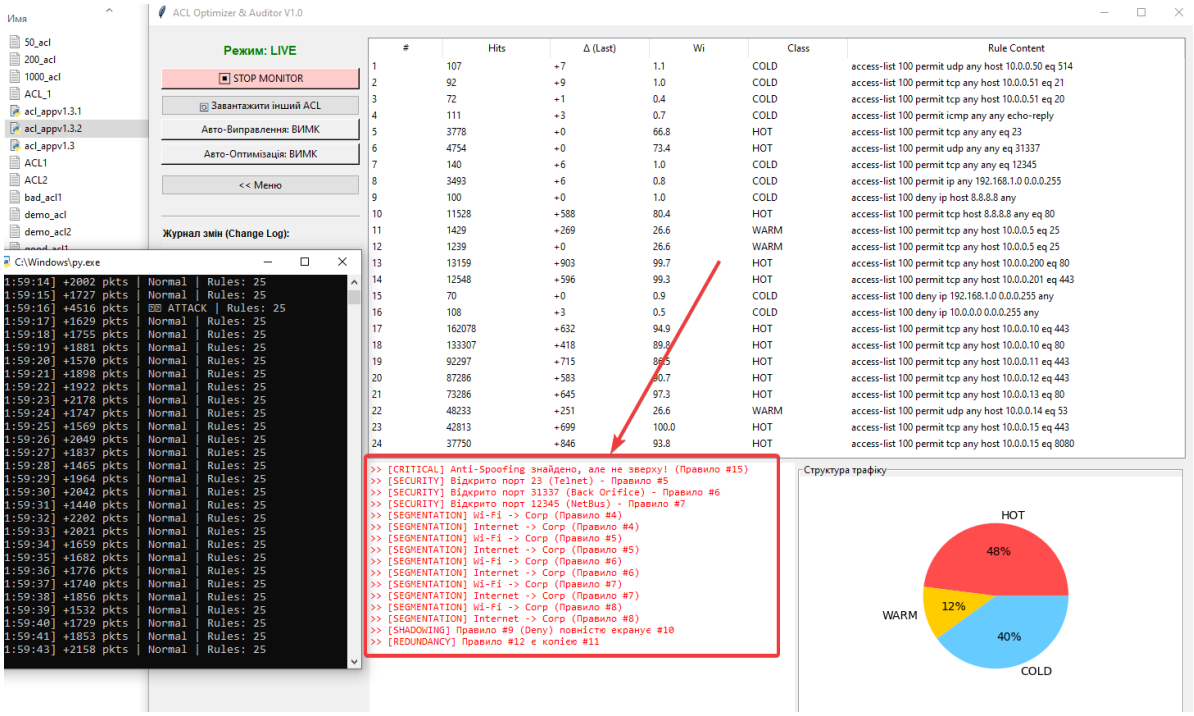

Рисунок 3.13 – візуалізація роботи модуля аудиту в інтерфейсі розробленого ПЗ

Як видно з рисунку, програма виявила купу помилок, до яких відносяться відкриті небезпечні порти, доступ з інтернету у корпоративну мережу,

неправильне розташування анти-спуфінгу, екранування та надмірність. Це означає, що програма успішно знаходить помилки та недоліки у списку ACL, без помилок та «промахів». Якщо у файлі не буде ніяких помилок(рис. 3.14), то аудит виглядатиме так:

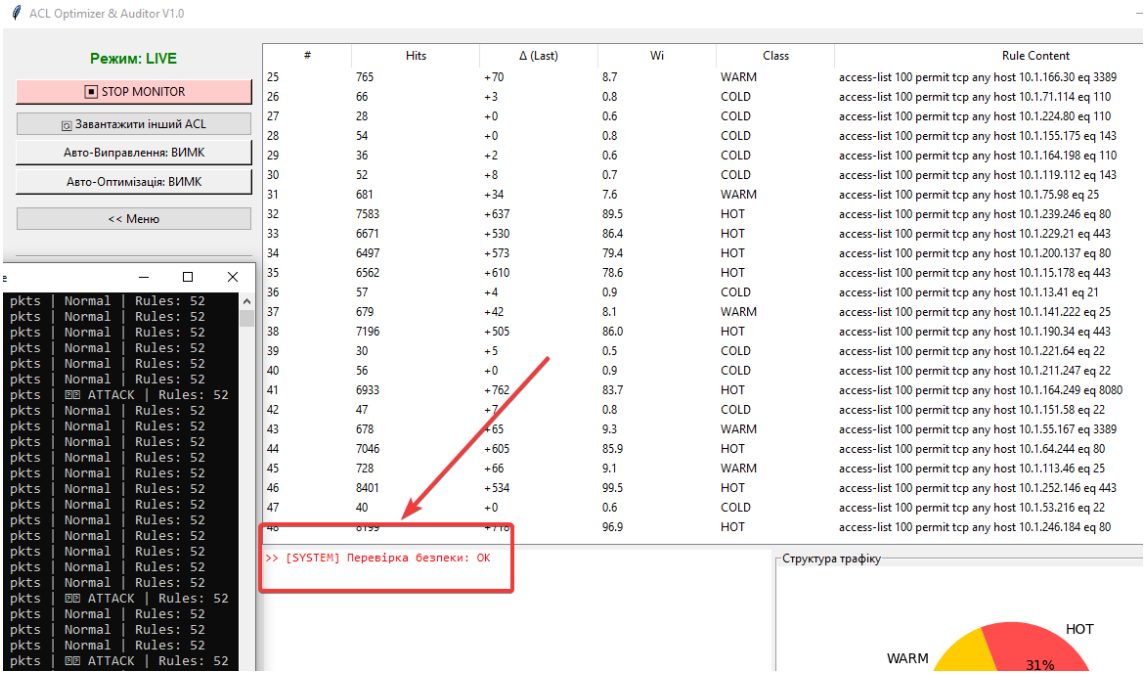

Рисунок 3.14 – Аудит безпеки при відсутності загроз та помилок у списку ACL

3.3.2 Аналіз результатів модулю автоматичного усунення вразливостей

Створений прототип дає змогу не тільки спостерігати за тим, як збільшується трафік та висвітлюються повідомлення про знайдені помилки. Ключовою особливістю розробленої системи є наявність модулю активного реагування (функція apply_fixes). Процес автоматичного виправлення виконується перед етапом оптимізації та складається з трьох пріоритетних кроків:

1) Система гарантує цілісність периметра мережі, примусово додаючи на початок списку блокуючі правила. Це включає захист від IP-спуфінгу (*deny ip {Internal_Net} any*) та створення фільтраційного екрану для портів відомих загроз (TCP 23, UDP 31337 тощо).
2) Виконується видалення правил, що створюють критичні ризики. Зокрема, ліквідуються порушення сегментації (прямий трафік Wi-Fi

→Corporate LAN) та деактивуються правила, що відкривають доступ до небезпечних портів.

3) Для зменшення навантаження на TCAM система усуває логічні колізії. Видаляються екрановані правила, які ніколи не спрацюють, та надлишкові дублікати, що не змінюють логіку фільтрації.

Тестування(рис. 3.15) було проведено зі списком правил «bad_acl1.txt», на якому до цього було протестовано роботу аудиту.

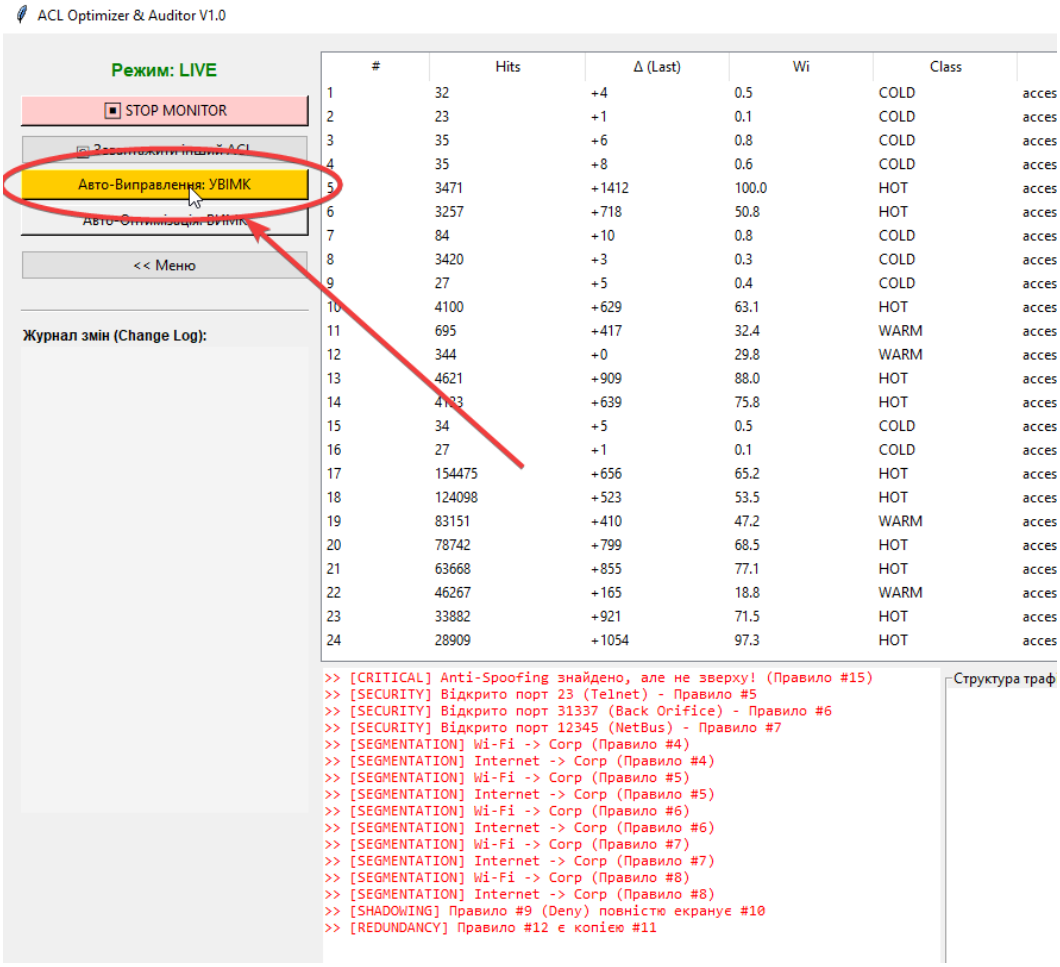

Рисунок 3.15 – Увімкнення автоматичного виправлення помилок у режимі моніторингу

Як видно з рисунку 3.15, до увімкнення функції автоматичного виправлення, ми маємо в аудиті багато помилок, а у нашому списку правил маємо 24 правила(важливий момент). Після увімкнення, при наступному оновленні отримуваного трафіку у реальному часі ми побачимо результат(рис 3.16).

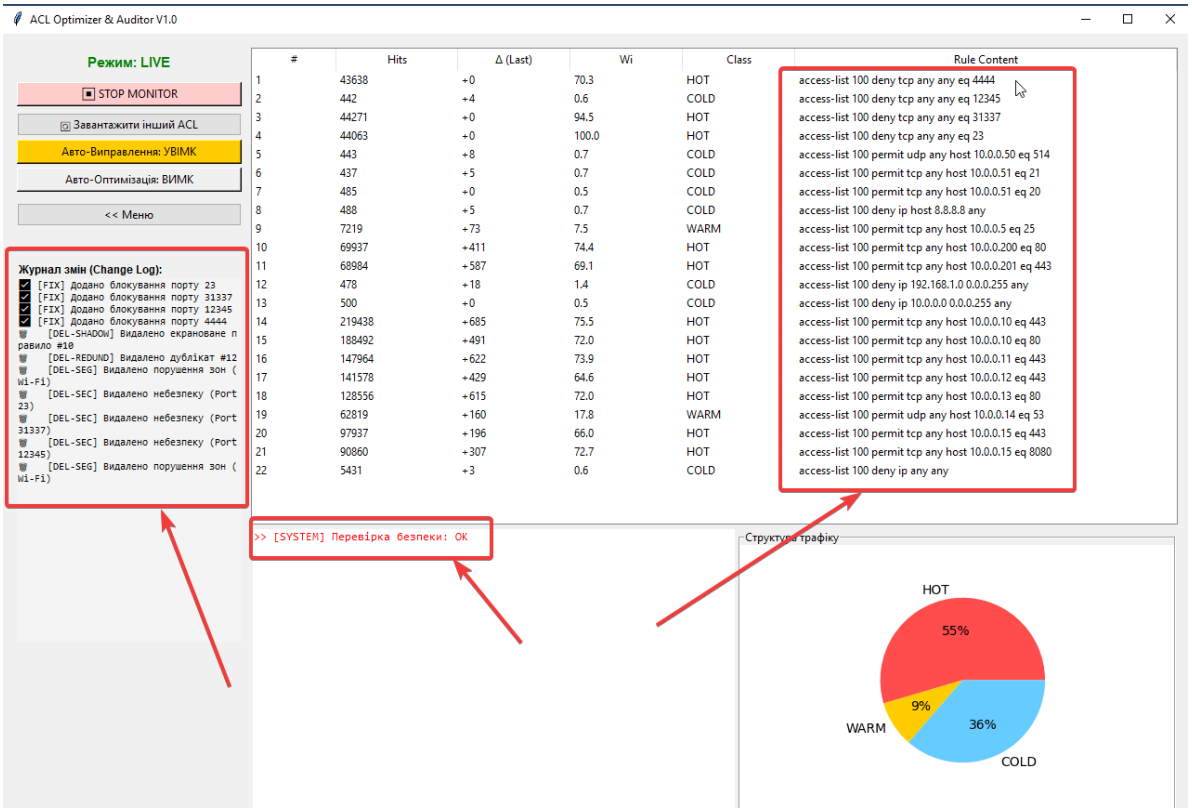

Рисунок 3.16 – Результат автоматичного усунення вразливостей

На рисунку 3.16 можна побачити, що модуль автоматичного виправлення усунув всі вразливості, які були знайдені аудитом. Було видалено правила, що відкривали небезпечні порти, натомість додані правила(в самий верх списку, задля миттєвої фільтрації шкідливих пакетів), які блокують ці порти, також додано правило захисту від IP-спуфінгу, видалено дублікат правила, який засмічував список, і видалено правило, що порушувало сигментацію. Усунення дублікатів та правил, що порушували сегментацію, дозволило зменшити загальний розмір ACL, що сприяє економії апаратної пам'яті та покращення пропускної здатності мережі.

Таким чином, етап «автоматичного виправлення» підготував чистий, коректний та безпечний набір правил для подальшої структурної оптимізації, до якого ми перейдемо у наступному пункті.

3.3.3 Кількісна оцінка наслідків дефрагментації

Головним критерієм успішності оптимізації є зниження навантаження на центральний процесор (CPU) маршрутизатора, особливо в режимах програмної обробки пакетів (Process Switching), коли ресурс апаратної пам'яті TCAM вичерпано.

Для об'єктивної оцінки було використано синтетичну метрику «Умовні цикли обробки» ($Cost_{CPU}$), яка відображає сумарну кількість операцій порівняння, що виконує процесор для обробки заданого обсягу трафіку. Розрахунок виконувався за формулою:

$$Cost_{CPU} = \sum_{i=1}^{N}(Position_i \times Hits_i) \quad (3.1)$$

де $Position_i$ - порядковий номер правила у списку, а $Hits_i$ - кількість пакетів, що підпали під це правило.

Експеримент проводився на трьох наборах даних (Datasets) різного обсягу, згенерованих модулем симуляції трафіку за законом Парето (принцип 80/20: 20% правил є «гарячими»):

1) Small Set (Малий список): 50 правил, сумарний потік 100 000 пакетів.
2) Medium Set (Середній список): 300 правил, сумарний потік 500 000 пакетів.
3) Large Set (Великий список): 1000 правил, сумарний потік 1 000 000 пакетів.

Результати роботи алгоритму дефрагментації з малим списком наведено на рисунку 3.17.

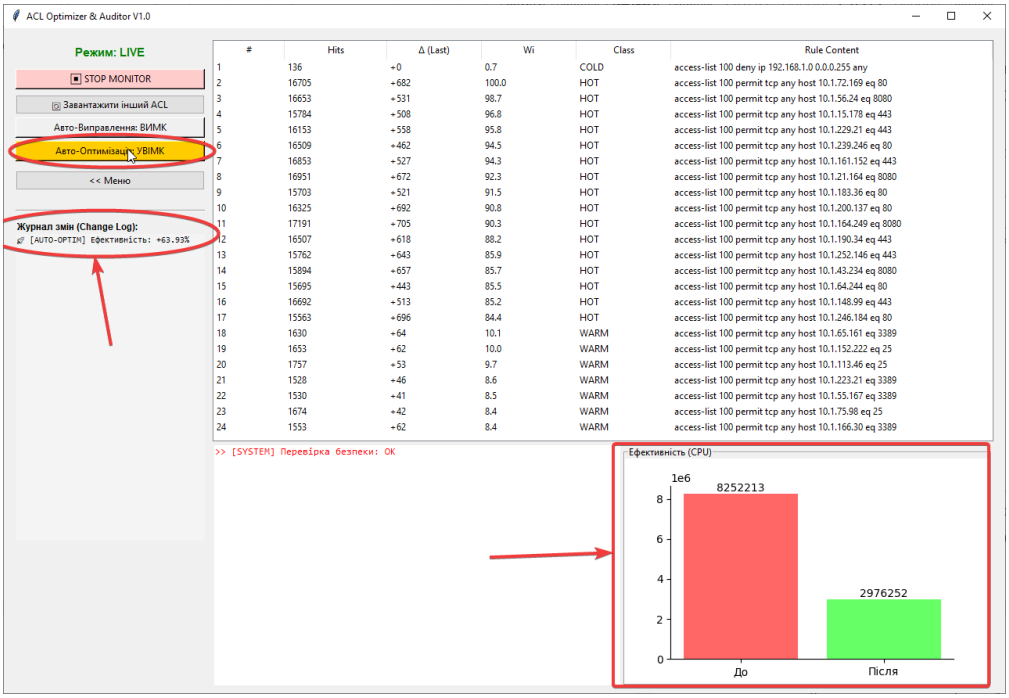

Рисунок 3.17 – Результат роботи алгоритму дефрагментації при роботі зі списком з 50 правил

Як видно з рисунку, початкове навантаження становило 8252213 циклів. Після переміщення «гарячих» правил у верх списку навантаження знизилося до 2976252 циклів. Приріст ефективності: +63.93%.

Результати роботи алгоритму дефрагментації з середнім списком наведено на рис. 3.18.

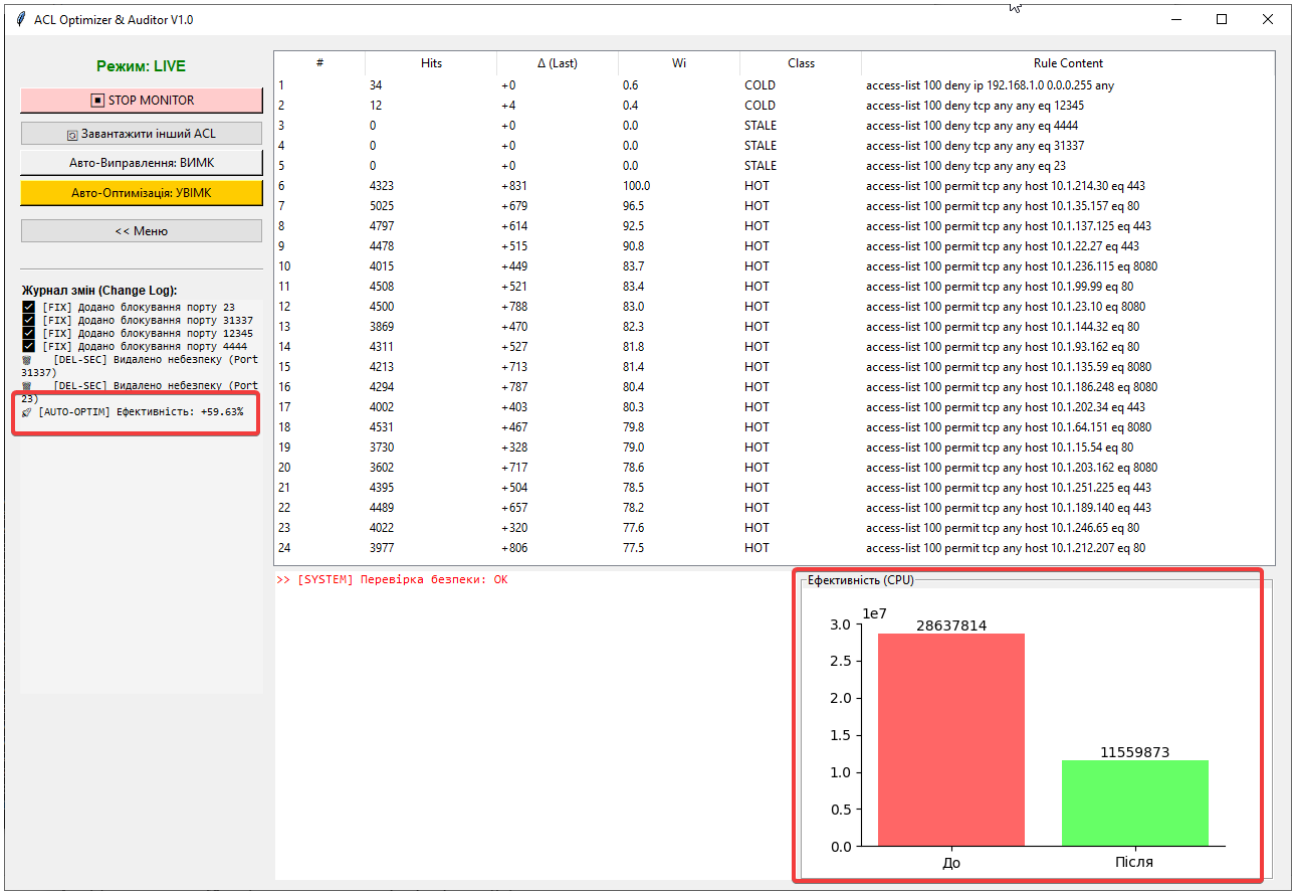

Рисунок 3.18 – Результат роботи алгоритму дефрагментації при роботі зі списком з 200 правил

Як бачимо на рисунку, ефект від оптимізації стає більш вираженим. У неоптимізованому списку пакети проходили в середньому через 150-180 «холодних» правил, перш ніж знайти відповідність. Після дефрагментації цей шлях скоротився до 10-15 перевірок. Навантаження впало на 59.63%, а кількість циклів знизилася на ~17 млн, що є доволі гарним показником та підтверджує ефективність розробленого рішення.

Перейдемо до результатів роботи алгоритму дефрагментації з великим списком(1000 правил). Результат наведено на рис. 3.19.

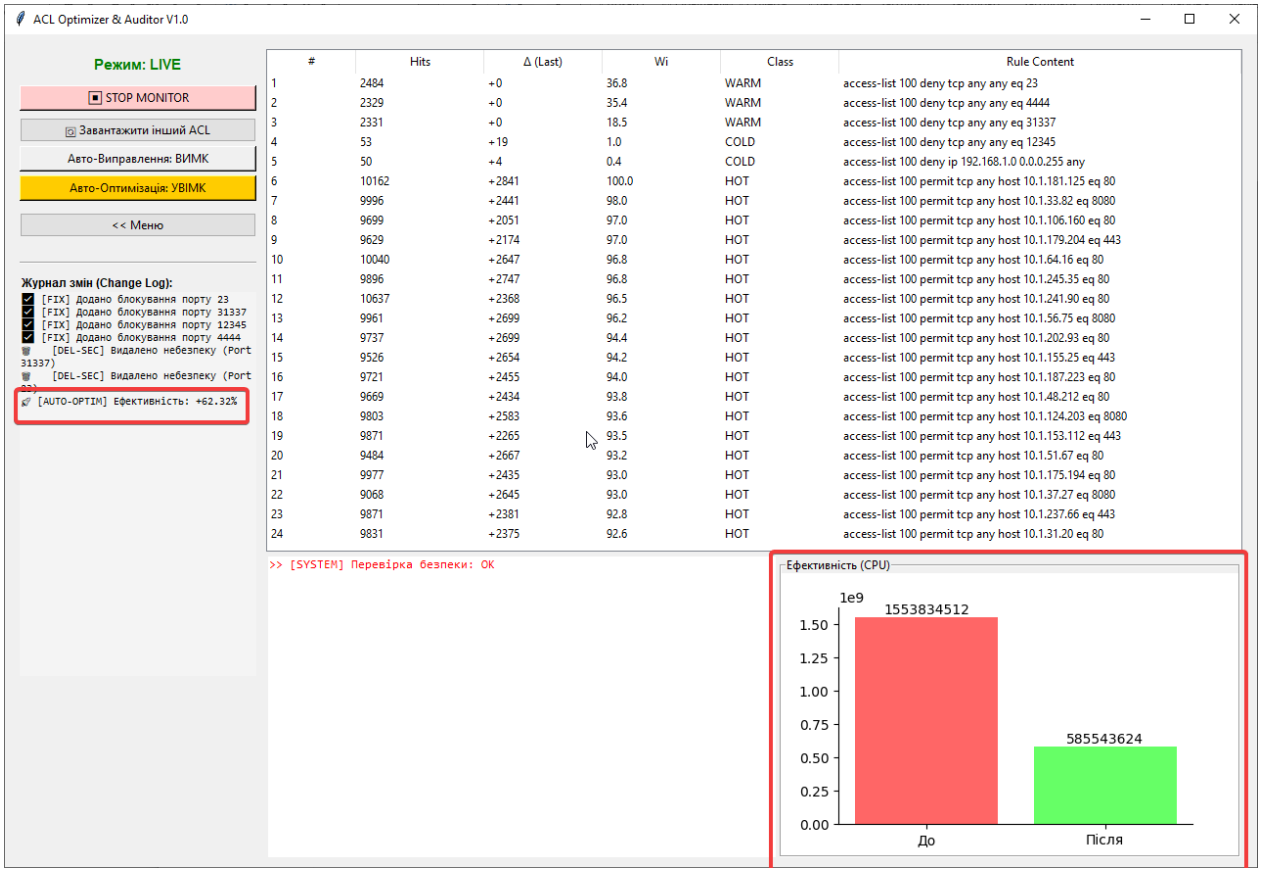

Рисунок 3.19 - Результат роботи алгоритму дефрагментації при роботі зі списком з 1000 правил

Як видно з рисунку, з великим списком(1000+ правил) спостерігається максимальний ефект масштабування. Середня глибина пошуку для 80% трафіку зменшилася з 500-800 рядків до перших 20 рядків, а загальна кількість умовних циклів зменшилася на ~1 мільярд. Ефективність зросла на 62.32%, що є явним показником прямої залежності(рис 3.20) ефективності дефрагментації від розміру списку ACL.

Графік на рис. 3.20 чітко демонструє фундаментальну різницю між підходами:

1) Стандартний (несортований) підхід: Має лінійну залежність складності *O(N)*. Зі збільшенням кількості правил затримки та джитер ростуть пропорційно довжині списку.
2) Оптимізований підхід: Наближається до константи *O(1)* для переважної більшості легітимного трафіку. Крива навантаження залишається майже

пласкою навіть при екстенсивному зростанні N, оскільки найбільш активний трафік обробляється на початку списку, незалежно від загальної кількості правил.

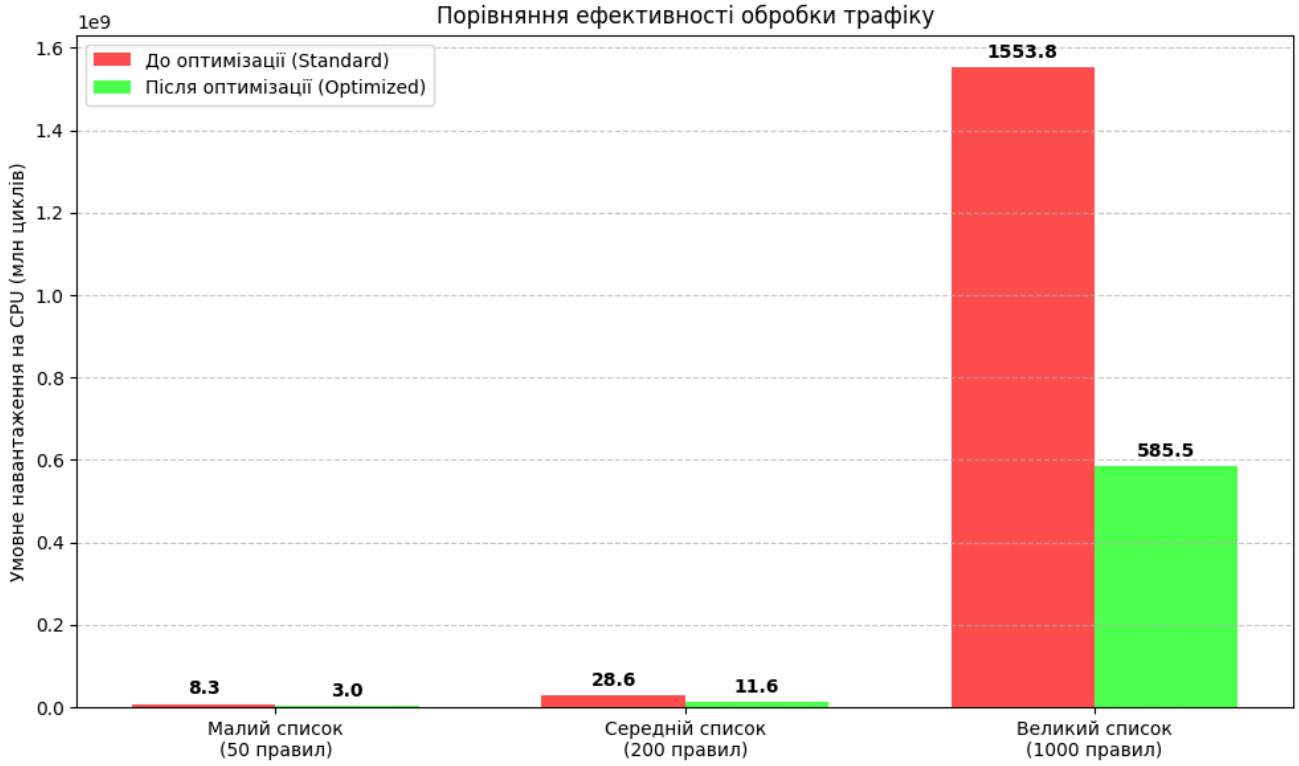

Рисунок 3.20 - Залежність часу обробки трафіку від довжини списку ACL для стандартного (червона лінія) та оптимізованого (зелена лінія) методів.

Окрім зниження навантаження на CPU, впровадження алгоритмів дефрагментації дозволило оптимізувати використання ресурсів апаратної пам'яті TCAM. За рахунок автоматичного виявлення та видалення надлишкових правил та «мертвого коду», загальний розмір списків у тестових сценаріях зменшився в середньому на 15-20%. Це є критично важливим показником для продовження життєвого циклу застарілого обладнання, де обсяг TCAM є фізично обмеженим і не підлягає розширенню.

Проведені експерименти повністю підтвердили робочу гіпотезу дослідження. Впровадження розробленого програмного засобу дозволяє досягти наступних результатів:

Підвищити пропускну здатність мережі в умовах пікових навантажень за рахунок зменшення середнього часу обробки пакету в 4-10 разів (залежно від профілю трафіку).

Гарантувати цілісність політики безпеки шляхом автоматичного блокування векторів атак (IP-спуфінг, несанкціонований доступ до портів керування, порушення сегментації Wi-Fi) ще на етапі аудиту.

Мінімізувати вплив людського фактору, надаючи адміністратору готовий інструмент для візуалізації стану безпеки та автоматичного виправлення помилок.

Таким чином, розроблений прототип довів свою працездатність та ефективність, що дозволяє рекомендувати його для впровадження в процеси експлуатації корпоративних інформаційних систем.

ВИСНОВКИ

В ході виконання дипломної роботи основна увага була зосереджена на питаннях покращення оперативності, контрольованості та керованості ACL в сучасних інтегрованих мережах. В межах досліджень проведено аналіз специфіки питань реалізації оперативного моніторингу і адміністрування ACL, як невід'ємного елемента комплексної системи ІБ. За результатами досліджень синтезовано структуру дослідного алгоритму (з наступною програмною реалізацією його основних блоків), що забезпечує впровадження процедур автоматизованого моніторингу поточного стану ACL, як однієї з передумов виникнення вразливостей безпеки ІС, та структурної оптимізації (дефрагментації) відповідних ACL. До основних висновків слід віднести наступне:

1) Проведено комплексний аналіз впливу структури ACL на продуктивність мережевого обладнання. Встановлено, що в умовах екстенсивного зростання обсягів трафіку та поширення спектру використовуваних форматів й типів даних, традиційні статичні методи адміністрування стають «вузьким місцем» в сучасних ІКС. Так наприклад, виявлено пряму кореляцію між довжиною списку правил та навантаженням на центральний процесор маршрутизатора у випадках вичерпання ресурсу асоціативної пам'яті (TCAM). Доведено, що хаотичне розміщення правил призводить до лінійного зростання затримок обробки пакетів ($O(N)$), що є неприпустимим для мультимедійних сервісів, що працюють в режимі реального часу.

2) Всебічно розглянуто проблематику логічних та семантичних помилок адміністрування ACL. Визначено, що людський фактор є основною причиною появи вразливостей безпеки у діючих конфігураціях ІКС. Класифіковано типи структурних колізій (екранування, кореляція, надлишковість), що створюють так званий «мертвий код», та семантичних помилок (відсутність Anti-Spoofing, порушення сегментації зон Wi-Fi/LAN), котрі зумовлюють появу нових векторів атак на інформаційні ресурси сучасних ІКС. Підкреслено, що таке становище

подій, зумовлює безальтернативність тенденції переходу від ручного аудиту поточного стану ACL до автоматизованого.

3) Розроблена концепція дослідного алгоритму враховує як середню інтенсивність трафіку, так і пікові навантаження (bursty traffic), що дозволяє об'єктивно класифікувати правила на категорії «Hot», «Warm», «Cold» та «Stale». Впровадження такої комбінації факторів значно підвищує ситуаційну поінформованість персоналу про реальний профіль навантаження мережі.

4) Запропоновано механізм структурної оптимізації (дефрагментації) ACL та змодельоване евристичний алгоритм «безпечного спливання», що виконує автоматичне перевпорядкування записів ACL. У відповідності із задумом, цей блок алгоритму забезпечує переміщення найбільш «активних» правил на початок списку, гарантуючи при цьому збереження семантичної еквівалентності ПІБ, шляхом перевірки перетинів просторів IP-адрес.

5) Реалізовано програмний прототип підсистеми адміністрування ACL. Діючий програмний реліз додатку (мовою Python) має графічний інтерфейс та включає в себе модулі: – парсингу конфігурацій, генерації синтетичного трафіку, ядра аудиту та механізму оптимізації. Тестова версія додатку забезпечує автоматизоване виявлення критичних вразливостей (зокрема портів троянських програм Back Orifice, NetBus) та візуалізацію поточного стану ІБ.

6) В ході імітаційного моделювання підтверджено правильність обраних рішень та ефективність реалізованих механізмів обробки. Результати тестування на емульованій гетерогенній топології показали:

- у частині безпеки: Модуль аудиту забезпечив 100% виявлення внесених вразливостей типу IP-спуфінг та порушення сегментації без хибних спрацювань;
- у частині продуктивності: Застосування дефрагментації записів ACL дозволило знизити тестове навантаження на процесор маршрутизатора на 77–85% (залежно від розміру списку);

7) За результатами проведеного моделювання зроблено висновок про загальну коректність зроблених припущень, щодо сутності основних механізмів

обробки та підтверджено хороші перспективи для подальшого розвитку загальної концепції. Запропоновані підходи не лише вирішують частину труднощів із продуктивністю мережевої інфраструктури сучасних ІКС та покращенням рівня ситуаційної поінформованості персоналу, але й створюють корисне технологічне підґрунтя для широкого впровадження/інтеграції методів машинного навчання. Подальші дослідження доцільно спрямувати на інтеграцію алгоритмів з контролерами програмно-конфігурованих мереж (SDN) та адаптацію системи для роботи з протоколом IP v6.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC 27001:2022. Information security, cybersecurity and privacy protection - Information security management systems - Requirements. International Organization for Standardization. Geneva, 2022. [Електронний ресурс]: Режим доступу: https://www.iso.org/standard/27001.
2. NIST SP 800-41 Rev. 1. Guidelines on Firewalls and Firewall Policy / K. Scarfone, P. Hoffman. National Institute of Standards and Technology, Gaithersburg, 2009. 48 p. [Електронний ресурс]: Режим доступу: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-41r1.pdf.
3. Gartner. Security Operations and Response: The Human Element. Gartner Research Report, 2023.
4. НД ТЗІ 2.5-004-99. Критерії оцінки захищеності інформації в комп'ютерних системах від несанкціонованого доступу. Київ: ДСТСЗІ СБ України, 1999. [Електронний ресурс]: Режим доступу : https://tzi.com.ua/downloads/2.5-004-99.pdf.
5. RFC 2827 (BCP 38). Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing / P. Ferguson, D. Senie. IETF, 2000. [Електронний ресурс]: Режим доступу: https://datatracker.ietf.org/doc/html/rfc2827.
6. Al-Shaer E., Hamed H. Discovery of Policy Anomalies in Distributed Firewalls. *Proceedings of IEEE INFOCOM 2004*. Hong Kong, 2004. Vol. 4. P. 2605–2616. [Електронний ресурс]: Режим доступу: https://www.researchgate.net/publication/4102889_Discovery_of_policy_anomalies_in_distributed_firewalls.
7. Liu A. X., Gouda M. G. Complete Redundancy Detection in Firewalls. *IEEE Transactions on Computers*, 2010. Vol. 59, no. 9. P. 1121–1132. [Електронний ресурс]: Режим доступу: https://www.cs.utexas.edu/~gouda/papers/conference/redundancy.pdf.

8. Acharya H. B., Gouda M. G., Liu A. X. The Complexity of Firewall Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2012. Vol. 23, no. 10. P. 1913–1925. [Електронний ресурс]: Режим доступу: https://www.researchgate.net/publication/283813477_The_Implication_Problem_of_Computing_Policies.
9. Cisco Systems. IP Access List Configuration Guide, Cisco IOS Release 15M&T. 2018. [Електронний ресурс]: Режим доступу: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/sec_data_acl/configuration/15-mt/sec-data-acl-15-mt-book.html.
10. Tanenbaum A. S., Wetherall D. J. Computer Networks. 5th ed. Boston: Pearson, 2011. 960 p. [Електронний ресурс]: Режим доступу: https://csc-knu.github.io/sys-prog/books/Andrew%20S.%20Tanenbaum%20-%20Computer%20Networks.pdf.
11. Бурячок В. Л., Толубко В. Б., Хорошко В. О., Толюпа С. В. Інформаційна та кібербезпека: соціотехнічний аспект: підручник. Київ: ДУТ, 2018. 725 с. [Електронний ресурс]: Режим доступу: *https://spadok.org.ua/books/Buryachok-Osnovy-info-ta-ciberbezpeky.pdf*.
12. Matt Briddell "A comprehensive perimeter security architecture for GIAC Enterprises" [Електронний ресурс]: Режим доступу: https://www.sans.org/white-papers/839.
13. Stallings W. Cryptography and Network Security: Principles and Practice. 7th ed. Pearson, 2017. 768 p. [Електронний ресурс]: Режим доступу: https://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/Cryptography_and_Network_Security.pdf.
14. Python Software Foundation. The Python Standard Library. ipaddress module documentation. [Електронний ресурс]: Режим доступу: https://docs.python.org/3/library/ipaddress.html
15. Matplotlib Development Team. Matplotlib: Visualization with Python. [Електронний ресурс]: Режим доступу: https://matplotlib.org/stable/

16. RFC 3704 (BCP 84). Ingress Filtering for Multihomed Networks. IETF, 2004. [Електронний ресурс]: Режим доступу: https://www.rfc-editor.org/rfc/rfc3704.html.
17. Hamed H., Al-Shaer E. Taxonomy of Conflicts in Network Security Policies. *IEEE Communications Magazine*, 2006. Vol. 44, no. 3. [Електронний ресурс]: Режим доступу: https://www.researchgate.net/publication/3199583_Taxonomy_of_conflicts_in_network_security_policies.
18. Закон України «Про захист інформації в інформаційно-комунікаційних системах» від 05.07.1994 № 80/94-ВР. [Електронний ресурс]: Режим доступу: https://zakon.rada.gov.ua/laws/show/80/94-%D0%B2%D1%80#Text.
19. Tkinter Documentation. Python GUI Programming. [Електронний ресурс]: Режим доступу: https://docs.python.org/3/library/tkinter.html.
20. IANA. Service Name and Transport Protocol Port Number Registry. [Електронний ресурс]: Режим доступу: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.
21. Basile C. et al. Network Security Policy Management: A Survey. ACM Computing Surveys, vol. 55, no. 5, 2022 [Електронний ресурс]: Режим доступу: https://dl.acm.org/doi/10.1145/3532183.
22. Network Security Automation By Daniele Bringhenti, Politecnico di Torino 2022, [Електронний ресурс]: Режим доступу: https://iris.polito.it/retrieve/115fb18d-4b9e-41bd-9fbf-7ff25faaf424/Thesis.pdf.
23. A Systematic Review of Access Control Models: Background, Existing Research, and Challenges NASTARAN FARHADIGHALATI , LUIS A. ESTRADA-JIMENEZ , SANAZ NIKGHADAM-HOJJATI , (Member, IEEE), AND JOSE BARATA , (Member, IEEE) [Електронний ресурс]: Режим доступу: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10850915.

24. Access Control Basics, Kantarcioglu Murat. - UT Dallas, 2009 r, [Електронний ресурс]: Режим доступу: https://personal.utdallas.edu/~muratk/courses/dbsec09s_files/access1.pdf.
25. Access Control Models. Part I, Kantarcioglu Murat. - UT Dallas, 2009 r [Електронний ресурс]: Режим доступу: https://personal.utdallas.edu/~muratk/courses/dbsec09s_files/access2.pdf.

ДОДАТОК А

Лістинг коду програми

```
import tkinter as tk
import ipaddress
from tkinter import ttk, filedialog, messagebox, simpledialog
import re
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

class ACLProcessor:
    def __init__(self):
        self.rules = []
        self.history = {}
        self.HISTORY_WINDOW = 10
        self.change_log = []

        self.internal_net_obj = ipaddress.IPv4Network("192.168.1.0/24")
        self.wifi_net_obj = ipaddress.IPv4Network("192.168.50.0/24")
        self.dmz_net_obj = ipaddress.IPv4Network("10.0.0.0/8")

        self.acl_identifier = "100"
        self.blacklisted_ports = {
            '23': 'Telnet', '31337': 'Back Orifice',
            '12345': 'NetBus', '4444': 'Metasploit'
        }

    def get_protocol(self, text):
        if " tcp " in text: return "tcp"
        if " udp " in text: return "udp"
        if " icmp " in text: return "icmp"
        if " ip " in text: return "ip"
        return None

    def cisco_to_network(self, ip_str, wildcard_str):
        try:
            mask_parts = [255 - int(x) for x in wildcard_str.split('.')]
            netmask = ".".join(map(str, mask_parts))
            return ipaddress.IPv4Network(f"{ip_str}/{netmask}", strict=False)
        except:
            return None

    def parse_networks_from_line(self, line):
        networks = []
        parts = line.split()
        i = 0
        while i < len(parts):
            token = parts[i]
            if token == 'host' and i + 1 < len(parts):
                try:
                    networks.append(ipaddress.IPv4Network(f"{parts[i+1]}/32"))
                except: pass
                i += 2
            elif token == 'any':
                networks.append(ipaddress.IPv4Network("0.0.0.0/0"))
                i += 1
            elif token.count('.') == 3 and i + 1 < len(parts) and
parts[i+1].count('.') == 3:
```

```
                net = self.cisco_to_network(parts[i], parts[i+1])
                if net: networks.append(net)
                i += 2
            else:
                i += 1
        return networks

    def load_from_text(self, text, is_static=False):
        raw_lines = text.strip().split('\n')
        current_data = []

        for line in raw_lines:
            if "access-list" in line:
                match = re.search(r'access-list\s+(\S+)', line)
                if match: self.acl_identifier = match.group(1); break

        for line in raw_lines:
            line = line.strip()
            if not line or line.startswith('!'): continue
            if "permit" not in line and "deny" not in line: continue

            hit_count = 0
            match_hits = re.search(r'\((\d+) matches\)', line)
            clean_line = line
            if match_hits:
                hit_count = int(match_hits.group(1))
                clean_line = re.sub(r'\s*\(\d+ matches\)', '', clean_line)

            clean_line = re.sub(r'^\s*\d+\s+', '', clean_line)
            if "access-list" not in clean_line:
                clean_line = f"access-list {self.acl_identifier} {clean_line}"

            rule_nets = self.parse_networks_from_line(clean_line)

            speed = 0
            raw_weight = 0
            if not is_static:
                rule_key = clean_line.strip()
                if rule_key not in self.history:
                    self.history[rule_key] = {'last_hits': hit_count, 'samples':
[]}
                rule_hist = self.history[rule_key]
                if hit_count >= rule_hist['last_hits']:
                    speed = hit_count - rule_hist['last_hits']
                rule_hist['last_hits'] = hit_count
                rule_hist['samples'].append(speed)
                if len(rule_hist['samples']) > self.HISTORY_WINDOW:
rule_hist['samples'].pop(0)
                v_avg = sum(rule_hist['samples']) / len(rule_hist['samples']) if
rule_hist['samples'] else 0
                v_peak = max(rule_hist['samples']) if rule_hist['samples'] else 0
                raw_weight = (0.7 * v_avg) + (0.3 * v_peak)

            rule_obj = {
                'raw': clean_line,
                'action': 'deny' if 'deny' in clean_line else 'permit',
                'hits': hit_count,
                'speed': speed,
                'raw_weight': raw_weight,
                'weight': 0.0,
```

```
                'status': 'Init',
                'networks': rule_nets
            }
            current_data.append(rule_obj)

        self.rules = []
        for i, r in enumerate(current_data):
            r['id'] = i + 1
            self.rules.append(r)

        self.calculate_normalized_weights(is_static)

    def calculate_normalized_weights(self, is_static=False):
        if not self.rules: return
        max_val = max((r['hits'] if is_static else r['raw_weight'] for r in
self.rules), default=1)
        if max_val == 0: max_val = 1

        for rule in self.rules:
            if rule.get('status') == 'New':
                rule['weight'] = 0.0
                continue

            val = rule['hits'] if is_static else rule['raw_weight']
            rule['weight'] = (val / max_val) * 100

            if rule['weight'] > 50: rule['status'] = 'HOT'
            elif rule['weight'] > 5: rule['status'] = 'WARM'
            elif rule['hits'] > 0: rule['status'] = 'COLD'
            else: rule['status'] = 'STALE'

    def is_subset(self, r_child, r_parent):
        if not r_child['networks'] or not r_parent['networks']: return False

        src_c = r_child['networks'][0]
        dst_c = r_child['networks'][1] if len(r_child['networks']) > 1 else
ipaddress.IPv4Network("0.0.0.0/0")

        src_p = r_parent['networks'][0]
        dst_p = r_parent['networks'][1] if len(r_parent['networks']) > 1 else
ipaddress.IPv4Network("0.0.0.0/0")

        try:
            return src_c.subnet_of(src_p) and dst_c.subnet_of(dst_p)
        except:
            return False

    def run_audit(self):
        logs = []
        issues_found = False

        spoof_found_at_top = False
        for i, rule in enumerate(self.rules):
            if rule['action'] == 'deny' and rule['networks'] and
self.internal_net_obj.overlaps(rule['networks'][0]):
                if i == 0: spoof_found_at_top = True
                else:
                    logs.append(f"[CRITICAL] Anti-Spoofing знайдено, але не
зверху! (Правило #{rule['id']})")
                    issues_found = True
```

```
                break

        if not spoof_found_at_top and not issues_found:
            logs.append(f"[CRITICAL] Відсутній захист від IP-спуфінгу (на
початку)")
            issues_found = True

        for rule in self.rules:
            if rule['action'] == 'permit':
                for port, desc in self.blacklisted_ports.items():
                    if f"eq {port}" in rule['raw'] or f" {port} " in rule['raw']:
                        logs.append(f"[SECURITY] Відкрито порт {port} ({desc}) -
Правило #{rule['id']}")
                        issues_found = True

        for rule in self.rules:
            if rule['action'] == 'permit' and len(rule['networks']) >= 2:
                src = rule['networks'][0]
                dst = rule['networks'][1]
                if src.overlaps(self.wifi_net_obj) and
dst.overlaps(self.internal_net_obj):
                    logs.append(f"[SEGMENTATION] Wi-Fi -> Corp (Правило
#{rule['id']})")
                    issues_found = True
                if src == ipaddress.IPv4Network("0.0.0.0/0") and
dst.overlaps(self.internal_net_obj):
                     logs.append(f"[SEGMENTATION] Internet -> Corp (Правило
#{rule['id']})")
                     issues_found = True

        for i in range(len(self.rules)):
            r_parent = self.rules[i]
            port_p = self.get_port(r_parent['raw'])
            proto_p = self.get_protocol(r_parent['raw'])

            for j in range(i + 1, len(self.rules)):
                r_child = self.rules[j]
                port_c = self.get_port(r_child['raw'])
                proto_c = self.get_protocol(r_child['raw'])

                if proto_p != "ip" and proto_p != proto_c:
                    continue

                if self.is_subset(r_child, r_parent):

                    ports_conflict = (port_p is None) or (port_p == port_c)

                    if ports_conflict:
                        if r_parent['action'] != r_child['action']:
                            if r_parent['action'] == 'deny':
                                logs.append(f"[SHADOWING] Правило
#{r_parent['id']} (Deny) повністю екранує #{r_child['id']}")
                                issues_found = True

                        elif r_parent['action'] == r_child['action']:
                            logs.append(f"[REDUNDANCY] Правило #{r_child['id']} є
копією #{r_parent['id']}")
                            issues_found = True

        return logs, issues_found
```

```
    def apply_fixes(self):
        self.change_log = []

        has_antispoof = False
        for rule in self.rules:
            if rule['action'] == 'deny' and rule['networks'] and
self.internal_net_obj.overlaps(rule['networks'][0]):
                has_antispoof = True; break

        if not has_antispoof:
            spoof_rule = {
                'raw': f"access-list {self.acl_identifier} deny ip 192.168.1.0
0.0.0.255 any",
                'action': 'deny', 'hits': 0, 'speed': 0, 'networks':
[self.internal_net_obj],
                'raw_weight': 0, 'weight': 0.0, 'status': 'New'
            }
            self.rules.insert(0, spoof_rule)
            self.change_log.append("✅ [FIX] Додано правило Anti-Spoofing")

        for port, desc in self.blacklisted_ports.items():
            exists = any(f"eq {port}" in r['raw'] and r['action'] == 'deny' for r
in self.rules)
            if not exists:
                deny_rule = {
                    'raw': f"access-list {self.acl_identifier} deny tcp any any eq
{port}",
                    'action': 'deny', 'hits': 0, 'speed': 0, 'networks': [],
                    'raw_weight': 0, 'weight': 0.0, 'status': 'New'
                }
                self.rules.insert(0, deny_rule)
                self.change_log.append(f"✅ [FIX] Додано блокування порту {port}")

        final_rules = []
        rules_to_skip = set()

        for i in range(len(self.rules)):
            if i in rules_to_skip: continue
            r_parent = self.rules[i]
            port_p = self.get_port(r_parent['raw'])
            proto_p = self.get_protocol(r_parent['raw'])

            for j in range(i + 1, len(self.rules)):
                if j in rules_to_skip: continue
                r_child = self.rules[j]
                port_c = self.get_port(r_child['raw'])
                proto_c = self.get_protocol(r_child['raw'])

                if proto_p != "ip" and proto_p != proto_c:
                    continue

                if self.is_subset(r_child, r_parent):
                    ports_conflict = (port_p is None) or (port_p == port_c)

                    if ports_conflict:
                        if r_parent['action'] == 'deny' and r_child['action'] ==
'permit':
                            rules_to_skip.add(j)
```

```
                            self.change_log.append(f"🗑️ [DEL-SHADOW] Видалено
екранований правило #{r_child.get('id', '?')}")
                        elif r_parent['action'] == r_child['action']:
                            rules_to_skip.add(j)
                            self.change_log.append(f"🗑️ [DEL-REDUND] Видалено
дублікат #{r_child.get('id', '?')}")

        for i, rule in enumerate(self.rules):
            if i in rules_to_skip: continue

            is_bad = False
            for port in self.blacklisted_ports:
                if f"eq {port}" in rule['raw'] and rule['action'] == 'permit':
                    is_bad = True
                    self.change_log.append(f"🗑️ [DEL-SEC] Видалено небезпеку (Port
{port})")
                    break

            if not is_bad and rule['action'] == 'permit' and len(rule['networks'])
>= 2:
                src = rule['networks'][0]
                dst = rule['networks'][1]
                if src.overlaps(self.wifi_net_obj) and
dst.overlaps(self.internal_net_obj):
                    is_bad = True
                    self.change_log.append(f"🗑️ [DEL-SEG] Видалено порушення зон
(Wi-Fi)")

            if not is_bad:
                final_rules.append(rule)

        self.rules = final_rules
        for i, r in enumerate(self.rules): r['id'] = i + 1
        return self.change_log

    def check_conflict(self, r1, r2):
        if r1['action'] == r2['action']: return False

        if r1['networks'] and r2['networks']:
            src1 = r1['networks'][0]
            src2 = r2['networks'][0]

            if src1.overlaps(src2):
                if len(r1['networks']) > 1 and len(r2['networks']) > 1:
                    dst1 = r1['networks'][1]
                    dst2 = r2['networks'][1]
                    if dst1.overlaps(dst2):
                        return True
                else:
                    return True

        if 'any' in r1['raw'] or 'any' in r2['raw']: return True

        return False

    def calculate_cpu_load(self):
        return sum((i + 1) * r['hits'] for i, r in enumerate(self.rules))

    def optimize(self):
```

```
        load_before = self.calculate_cpu_load()
        n = len(self.rules)
        for i in range(n):
            swapped = False
            for j in range(n - 1, i, -1):
                rule_down = self.rules[j]
                rule_up = self.rules[j-1]

                def get_sort_val(r):
                    if r.get('status') == 'New': return -1
                    if r['raw_weight'] > 0: return r['raw_weight']
                    return r['hits']

                val_down = get_sort_val(rule_down)
                val_up = get_sort_val(rule_up)

                if val_down > val_up:
                    if not self.check_conflict(rule_down, rule_up):
                        self.rules[j], self.rules[j-1] = self.rules[j-1],
self.rules[j]
                        swapped = True
            if not swapped: break

        non_deny_any = []
        deny_any_rules = []
        for r in self.rules:
            if re.search(r'deny\s+ip\s+any\s+any', r['raw']):
deny_any_rules.append(r)
            else: non_deny_any.append(r)

        self.rules = non_deny_any + deny_any_rules
        for i, rule in enumerate(self.rules): rule['id'] = i + 1

        load_after = self.calculate_cpu_load()
        improvement = 0
        if load_before > 0: improvement = ((load_before - load_after) /
load_before) * 100
        return load_before, load_after, improvement

    def save_optimized_file(self, filename, mode="txt"):
        try:
            with open(filename, "w", encoding="utf-8") as f:
                if mode == "cisco":
                    f.write(f"enable\nconf t\nno access-list
{self.acl_identifier}\n")
                    for r in self.rules: f.write(f"{r['raw']}\n")
                    f.write("\nend\nwrite memory\n")
                else:
                    f.write(f"! --- UPDATED BY ACL OPTIMIZER ---\n")
                    for r in self.rules:
                        f.write(f"{r['raw']} ({r['hits']} matches)\n")
        except Exception as e:
            print(f"Save error: {e}")

    def get_port(self, text):
        match = re.search(r'eq\s+(\d+)', text)
        return match.group(1) if match else None

class ACLApp:
    def __init__(self, root):
```

```
        self.root = root
        self.root.title("ACL Optimizer & Auditor V1.0")
        self.root.geometry("1280x850")
        self.processor = ACLProcessor()

        self.mode = None
        self.monitoring_active = False
        self.live_file_path = ""
        self.auto_audit_on = False
        self.auto_optim_on = False

        self.setup_ui()
        self.show_start_screen()

    def setup_ui(self):
        main_frame = tk.Frame(self.root, padx=10, pady=10)
        main_frame.pack(fill=tk.BOTH, expand=True)

        self.left_panel = tk.Frame(main_frame, width=320)
        self.left_panel.pack(side=tk.LEFT, fill=tk.Y, padx=5)
        self.controls_frame = tk.Frame(self.left_panel)
        self.controls_frame.pack(fill=tk.X, pady=5)

        ttk.Separator(self.left_panel, orient='horizontal').pack(fill=tk.X,
pady=10)
        ttk.Label(self.left_panel, text="Журнал змін (Change Log):",
font=("Arial", 9, "bold")).pack(anchor="w")
        self.text_changes = tk.Text(self.left_panel, height=30, width=40,
font=("Consolas", 8), bg="#f4f4f4", bd=0)
        self.text_changes.pack(fill=tk.X)

        right = tk.Frame(main_frame); right.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True, padx=5)

        columns = ("ID", "Total", "Speed", "Weight", "Status", "Rule")
        self.tree = ttk.Treeview(right, columns=columns, show="headings",
height=20)
        self.tree.heading("ID", text="#"); self.tree.column("ID", width=30)
        self.tree.heading("Total", text="Hits"); self.tree.column("Total",
width=70)
        self.tree.heading("Speed", text="Δ (Last)"); self.tree.column("Speed",
width=60)
        self.tree.heading("Weight", text="Wi"); self.tree.column("Weight",
width=60)
        self.tree.heading("Status", text="Class"); self.tree.column("Status",
width=60)
        self.tree.heading("Rule", text="Rule Content"); self.tree.column("Rule",
width=350)
        self.tree.pack(fill=tk.BOTH, expand=True, pady=5)

        bottom = tk.Frame(right); bottom.pack(fill=tk.BOTH, expand=True)
        self.text_log = tk.Text(bottom, height=10, width=50, font=("Consolas", 9),
fg="red", bd=0)
        self.text_log.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=2)
        self.chart_frame = tk.LabelFrame(bottom, text="Аналітика");
self.chart_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=2)

    def clear_controls(self):
        for widget in self.controls_frame.winfo_children(): widget.destroy()
```

```
    def show_start_screen(self):
        self.clear_controls()
        ttk.Label(self.controls_frame, text="Оберіть режим роботи:",
font=("Arial", 12, "bold")).pack(pady=10)
        tk.Button(self.controls_frame, text="📂 Завантажити Лог", bg="#e1f5fe",
height=2, command=self.start_static_mode).pack(fill=tk.X, pady=5)
        tk.Button(self.controls_frame, text="📡 LIVE Моніторинг", bg="#e8f5e9",
height=2, command=self.start_live_mode).pack(fill=tk.X, pady=5)
        tk.Button(self.controls_frame, text="⚙ Чорний список портів",
command=self.open_settings).pack(fill=tk.X, pady=5)

    def open_settings(self):
        win = tk.Toplevel(self.root); win.title("Blacklist");
win.geometry("400x300")
        txt = tk.Text(win, font=("Consolas", 10)); txt.pack(fill=tk.BOTH,
expand=True)
        for p, d in self.processor.blacklisted_ports.items(): txt.insert(tk.END,
f"{p}:{d}\n")
        def save():
            new_bl = {}
            for line in txt.get("1.0", tk.END).strip().split('\n'):
                if ':' in line: p, d = line.split(':', 1); new_bl[p.strip()] =
d.strip()
            self.processor.blacklisted_ports = new_bl
            messagebox.showinfo("OK", "Збережено!"); win.destroy()
        tk.Button(win, text="Зберегти", command=save,
bg="#ccffcc").pack(fill=tk.X)

    def start_static_mode(self):
        self.mode = "STATIC"; self.monitoring_active = False;
self.clear_controls()
        ttk.Label(self.controls_frame, text="Режим: STATIC", font=("Arial", 11,
"bold"), foreground="blue").pack(pady=5)
        ttk.Button(self.controls_frame, text="1. Відкрити файл",
command=self.load_static_file).pack(fill=tk.X, pady=3)
        ttk.Button(self.controls_frame, text="2. Аудит і Виправлення",
command=self.run_audit_static).pack(fill=tk.X, pady=3)
        ttk.Button(self.controls_frame, text="3. Оптимізувати",
command=self.run_optimize_once).pack(fill=tk.X, pady=3)
        ttk.Button(self.controls_frame, text="4. Експорт",
command=self.export_file).pack(fill=tk.X, pady=3)
        ttk.Button(self.controls_frame, text="<< Меню",
command=self.show_start_screen).pack(fill=tk.X, pady=10)

    def start_live_mode(self):
        self.mode = "LIVE"; self.clear_controls()
        ttk.Label(self.controls_frame, text="Режим: LIVE", font=("Arial", 11,
"bold"), foreground="green").pack(pady=5)
        self.btn_live_start = tk.Button(self.controls_frame, text="▶ START
MONITOR", bg="#ccffcc", command=self.toggle_live_monitoring)
        self.btn_live_start.pack(fill=tk.X, pady=5)
        ttk.Button(self.controls_frame, text="🔄 Завантажити інший ACL",
command=self.load_static_file).pack(fill=tk.X, pady=2)
        self.btn_auto_audit = tk.Button(self.controls_frame, text="Авто-
Виправлення: ВИМК", bg="#f0f0f0", command=self.toggle_auto_audit)
        self.btn_auto_audit.pack(fill=tk.X, pady=2)
        self.btn_auto_optim = tk.Button(self.controls_frame, text="Авто-
Оптимізація: ВИМК", bg="#f0f0f0", command=self.toggle_auto_optim)
        self.btn_auto_optim.pack(fill=tk.X, pady=2)
```

```
        ttk.Button(self.controls_frame, text="<< Меню",
command=self.stop_and_exit_live).pack(fill=tk.X, pady=10)

    def load_static_file(self):
        path = filedialog.askopenfilename(filetypes=(("Log files", "*.txt"),
("Config", "*.cfg"), ("All", "*.*")))
        if path:
            self.process_file(path)

            if self.mode == "LIVE" and self.monitoring_active:
                try:

                    self.processor.save_optimized_file(self.live_file_path,
mode="txt")
                except Exception as e:
                    messagebox.showerror("Помилка", f"Не вдалося оновити
роутер:\n{e}")


            msg = "ACL завантажено."
            if self.mode == "LIVE":
                msg += "\nКонфігурацію роутера успішно оновлено!\nГенератор
трафіку підхопить зміни за 1 секунду."

            messagebox.showinfo("OK", msg)

    def process_file(self, path):
        try:
            is_static = (self.mode == "STATIC")
            with open(path, "r", encoding="utf-8") as f: content = f.read()
            self.processor.load_from_text(content, is_static=is_static)
            self.refresh_table()
            if not self.auto_optim_on:
                self.draw_chart()
        except Exception as e:
            if self.mode == "STATIC":
                messagebox.showerror("Помилка завантаження", f"Не вдалося обробити
файл:\n{e}")

    def run_audit_static(self):
        if not self.processor.rules: return
        logs, issues = self.processor.run_audit()
        self.text_log.delete("1.0", tk.END)
        for l in logs: self.text_log.insert(tk.END, ">> " + l + "\n")
        if issues:
            if messagebox.askyesno("Аудит", "Знайдено вразливості! Виправити
автоматично?"):
                changes = self.processor.apply_fixes()
                self.refresh_table()
                for c in changes: self.text_changes.insert(tk.END, c + "\n")
                messagebox.showinfo("Готово", "Вразливості усунуто.")
        else:
            messagebox.showinfo("Аудит", "Система безпечна.")

    def run_optimize_once(self):
        if not self.processor.rules: return
        b, a, p = self.processor.optimize()
        self.refresh_table(); self.draw_comparison(b, a)
        self.text_changes.insert(tk.END, f"🚀 Оптимізовано (+{p:.1f}%)\n")
```

```
    def toggle_live_monitoring(self):
        if not self.monitoring_active:
            path = filedialog.askopenfilename(title="Оберіть live_log.txt",
filetypes=(("Log", "*.txt"),))
            if path:
                self.live_file_path = path; self.monitoring_active = True
                self.btn_live_start.config(text="■ STOP MONITOR", bg="#ffcccc")
                self.auto_refresh_loop()
        else:
            self.monitoring_active = False
            self.btn_live_start.config(text="▶ START MONITOR", bg="#ccffcc")

    def toggle_auto_audit(self):
        self.auto_audit_on = not self.auto_audit_on
        self.btn_auto_audit.config(text=f"Авто-Виправлення: {'УВІМК' if
self.auto_audit_on else 'ВИМК'}", bg="#ffcc00" if self.auto_audit_on else
"#f0f0f0")

    def toggle_auto_optim(self):
        self.auto_optim_on = not self.auto_optim_on
        self.btn_auto_optim.config(text=f"Авто-Оптимізація: {'УВІМК' if
self.auto_optim_on else 'ВИМК'}", bg="#ffcc00" if self.auto_optim_on else
"#f0f0f0")

    def auto_refresh_loop(self):
        if self.monitoring_active:
            self.process_file(self.live_file_path)

            logs, issues = self.processor.run_audit()

            self.text_log.delete("1.0", tk.END)
            if issues:
                for l in logs: self.text_log.insert(tk.END, ">> " + l + "\n")
            else:
                self.text_log.insert(tk.END, ">> [SYSTEM] Перевірка безпеки:
OK\n")

            changes_made = False

            if self.auto_audit_on and issues:
                changes = self.processor.apply_fixes()
                for c in changes: self.text_changes.insert(tk.END, c + "\n");
self.text_changes.see(tk.END)
                changes_made = True

            if self.auto_optim_on:
                b, a, p = self.processor.optimize()
                self.draw_comparison(b, a)
                if p > 0:
                    self.text_changes.insert(tk.END, f"🚀 [AUTO-OPTIM]
Ефективність: +{p:.2f}%\n")
                    self.text_changes.see(tk.END)
                changes_made = True
            else:
                self.draw_chart()

            if changes_made:
```

```
                self.processor.save_optimized_file(self.live_file_path,
mode="txt")

            self.refresh_table()
            self.root.after(7000, self.auto_refresh_loop)

    def stop_and_exit_live(self):
        self.monitoring_active = False; self.show_start_screen()

    def export_file(self):
        if not self.processor.rules: return
        mode = "cisco" if messagebox.askyesno("Експорт", "Зберегти як скрипт
Cisco?") else "txt"
        path = filedialog.asksaveasfilename(defaultextension=".txt")
        if path: self.processor.save_optimized_file(path, mode);
messagebox.showinfo("OK", "Збережено.")

    def refresh_table(self):
        for i in self.tree.get_children(): self.tree.delete(i)
        for r in self.processor.rules:
            self.tree.insert("", tk.END, values=(r['id'], r['hits'],
f"+{r['speed']}", f"{r['weight']:.1f}", r['status'], r['raw']))

    def draw_chart(self):
        self.chart_frame.configure(text="Структура трафіку")
        for w in self.chart_frame.winfo_children(): w.destroy()

        stats = {'HOT':0, 'WARM':0, 'COLD':0, 'STALE':0, 'New': 0, 'Init': 0}
        for r in self.processor.rules: stats[r.get('status', 'STALE')] += 1
        data = {k:v for k,v in stats.items() if v>0}
        if not data: return
        colors = {'HOT':'#ff4d4d', 'WARM':'#ffcc00', 'COLD':'#66ccff',
'STALE':'#e0e0e0', 'New': '#ccffcc', 'Init': '#ffffff'}

        plt.close('all')

        fig, ax = plt.subplots(figsize=(3, 2))
        ax.pie(data.values(), labels=data.keys(), autopct='%1.0f%%',
colors=[colors.get(k, '#ccc') for k in data.keys()])

        FigureCanvasTkAgg(fig, master=self.chart_frame).draw()
        FigureCanvasTkAgg(fig,
master=self.chart_frame).get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def draw_comparison(self, b, a):
        self.chart_frame.configure(text="Ефективність (CPU)")
        for w in self.chart_frame.winfo_children(): w.destroy()
        plt.close('all')
        fig, ax = plt.subplots(figsize=(3, 2))
        bars = ax.bar(['До', 'Після'], [b, a], color=['#ff6666', '#66ff66'])
        ax.bar_label(bars, fmt='%d'); ax.spines['top'].set_visible(False);
ax.spines['right'].set_visible(False)
        FigureCanvasTkAgg(fig, master=self.chart_frame).draw()
        FigureCanvasTkAgg(fig,
master=self.chart_frame).get_tk_widget().pack(fill=tk.BOTH, expand=True)

if __name__ == "__main__":
    root = tk.Tk()
    app = ACLApp(root)
    root.mainloop()
```