Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(освітній рівень)

на тему: "Дослідження методів виявлення та аналізу витоків даних у середовищі Linux"

Виконав: студент(ка) VI курсу, групи СБм – 61

Спеціальності:
125 «Кібербезпека та захист інформації»
(шифр і назва напряму підготовки, спеціальності)
Шарик Олександр Володимирович
підпис (прізвище та ініціали)

Керівник Козак Р.О.
підпис (прізвище та ініціали)

Нормоконтроль Стадник М.А.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

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

Кафедра кібербезпеки
(повна назва кафедри)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

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

Студенту Шарику Олександру Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів виявлення та аналізу витоків даних у середовищі Linux

Керівник роботи Козак Руслан Орестович, к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

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

2. Термін подання студентом завершеної роботи 12.12.2025

3. Вихідні дані до роботи Методи виявлення та аналізу витоків даних у середовищі Linux

4. Зміст роботи (перелік питань, які потрібно розробити)
Сутність проблеми витоку чутливих даних та підходи до її вирішення.
Проектування загальної архітектури утиліти.
Реалізація ключових функціональних модулів.
Проведення тестування та аналіз отриманих результатів.
Охорона праці та безпека в надзвичайних ситуаціях.
Висновки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1.Мета та завдання роботи 2.Основні поняття 3.Cost of a Data Breach Report
4.Канали витоку інформації в середовищі Linux 5.Вибір технологічного стеку
6.Архітектура роботи утиліти 7.Розроблений метод тестування
8.Результати тестування точності класифікації 9.Результати тестування продуктивності
10.Висновки

6. Консультанти розділів роботи

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

7. Дата видачі завдання 19.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	19.09 – 22.09	Виконано
2.	Підбір літературних джерел	22.09 – 30.09	Виконано
3.	Опрацювання джерел в галузі дослідження	02.10 – 09.10	Виконано
4.	Аналіз та вибір програмних засобів розробки	10.10 – 16.10	Виконано
5.	Розроблення програмного коду	17.10 – 28.10	Виконано
6.	Тестування роботи утиліти	29.10 – 01.11	Виконано
7.	Оформлення розділу «Аналітичний огляд методів та засобів виявлення витоків даних»	02.11 – 10.11	Виконано
8.	Оформлення розділу «Розробка архітектури та методів утиліти моніторингу»	11.11 – 15.11	Виконано
9.	Оформлення розділу «Програмна реалізація та експерментальне дослідження утиліти»	16.11 – 23.11	Виконано
10.	Виконання завдання до підрозділу «Охорона праці та безпека в надзвичайних ситуаціях»	24.11 – 30.11	Виконано
11.	Оформлення кваліфікаційної роботи	01.12 – 08.12	Виконано
12.	Нормоконтроль	08.12 – 10.12	Виконано
13.	Перевірка на плагіат	12.12 – 14.12	Виконано
14.	Попередній захист кваліфікаційної роботи	17.12.2025	Виконано
15.	Захист кваліфікаційної роботи	23.12.2025	

Студент _____ Шарик О.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Козак Р.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дослідження методів виявлення та аналізу витоків даних у середовищі Linux // ОР «Магістр» // Шарик Олександр Володимирович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБм-61 // Тернопіль, 2025 // С. 73, рис. – 10, кресл. – 12, додат. – 2.

Ключові слова: Інформаційна безпека, Data Loss Prevention, DLP, Захист даних, Витік даних, Linux, Rust, Моніторинг файлової системи, Inotify, Виявлення чутливих даних, Інсайдерські загрози, Регулярні вирази, Захист кінцевих точок.

Кваліфікаційна робота присвячена дослідженню методів виявлення та попередження витоків чутливих даних в операційних системах сімейства Linux. З метою забезпечення конфіденційності інформації в умовах зростання інсайдерських загроз та використання хмарних інфраструктур, створення спеціалізованих інструментів моніторингу є критично важливим завданням. У роботі розглянуто архітектурні особливості побудови DLP-систем та специфіку обробки файлових подій у Linux. У процесі дослідження було проведено теоретичний аналіз векторів ексфільтрації даних, а також досліджено ефективність різних механізмів перехоплення системних подій. Розроблено програмний засіб, а також проведено експериментальні дослідження для оцінки точності класифікації файлів та вимірювання продуктивності розробленого рішення.

Розроблений підхід та отримані результати можуть бути використані адміністраторами систем та фахівцями з кібербезпеки для посилення захисту серверної інфраструктури та робочих станцій Linux. Кваліфікаційна робота також може бути корисною для дослідників, які займаються питаннями системного програмування та розробкою безпечного програмного забезпечення.

ANNOTATION

Research on methods for detecting and analyzing data leaks in the Linux environment // Master's thesis // Sharyk Oleksandr Volodymyrovych // Ivan Pul'uj Ternopil National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group SBm-61 // Ternopil, 2025 // p. 73, figs. 10, drws. 12, apps. 2.

Keywords: Information security, Data Loss Prevention, DLP, Data protection, Data leakage, Linux, Rust, File system monitoring, Inotify, Sensitive data detection, Insider threats, Regular expressions, Endpoint protection.

This thesis is devoted to the study of methods for detecting and preventing sensitive data leaks in Linux operating systems. In order to ensure information confidentiality in the context of growing insider threats and the use of cloud infrastructures, the creation of specialized monitoring tools is a critically important task. The thesis examines the architectural features of DLP systems and the specifics of file event processing in Linux. The research included a theoretical analysis of data exfiltration vectors and an investigation of the effectiveness of various mechanisms for intercepting system events. A software tool was developed, and experimental studies were conducted to evaluate the accuracy of file classification and measure the performance of the developed solution.

The developed approach and the results obtained can be used by system administrators and cybersecurity specialists to strengthen the protection of server infrastructure and Linux workstations. The thesis may also be useful for researchers involved in system programming and secure software development.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACL – Access Control

DLM – Data Lifecycle Management

DLP – Data Loss Prevention

EDR – Endpoint Detection and Response

FIM – File Integrity Monitoring

GDPR – General Data Protection Regulation

NIST – National Institute of Standards and Technology

NLP – Natural Language Processing

PCI DSS – Payment Card Industry Data Security Standard

PHI – Protected Health Information

PII – Personally Identifiable Information

RAM – Random Access Memory

RegEx – Regular Expressions

SSH – Secure Shell

TOML – Tom's Obvious, Minimal Language

ВСТУП

У сучасному інформаційному просторі, де обсяги обробки даних зростають експоненційно, а хмарні інфраструктури стають стандартом для бізнесу та державного сектору, забезпечення конфіденційності інформації є критично важливим завданням. Операційна система Linux, займаючи домінуючу позицію на ринку серверних платформ та контейнеризації, стає пріоритетною ціллю для зловмисників. Несанкціоноване поширення чутливих даних може призвести до катастрофічних наслідків, включаючи фінансові збитки, юридичну відповідальність згідно з GDPR та непоправну шкоду репутації організації.

Актуальність теми. Існуючі рішення часто стикаються з проблемами масштабованості, високим рівнем хибних спрацювань або надмірним споживанням системних ресурсів. Специфіка архітектури Linux, зокрема різноманітність файлових систем та механізмів контролю доступу, вимагає адаптації підходів до виявлення витоків. Тому дослідження новітніх методів моніторингу та розробка спеціалізованих інструментальних засобів, здатних ефективно виявляти витоки на рівні хоста, є актуальною науково-практичною задачею.

В рамках даної роботи ми прагнемо дослідити архітектурні особливості систем запобігання витокам, проаналізувати вектори загроз у середовищі Linux та розробити програмний засіб, що поєднує високу точність детекції з продуктивністю системного рівня. Це дослідження має на меті створення ефективного механізму захисту конфіденційної інформації від внутрішніх та зовнішніх загроз.

Таким чином, дана робота має велике значення для підвищення рівня захищеності інформаційних систем на базі Linux, сприяючи створенню безпечного цифрового середовища для обробки критично важливих даних.

Мета і задачі дослідження. Метою роботи є підвищення ефективності захисту конфіденційної інформації в автоматизованих системах на базі Linux шляхом удосконалення методів виявлення витоків даних та розробки програмного засобу моніторингу файлової активності.

Завдання дослідження:

– провести аналітичний огляд проблеми витоку даних та проаналізувати життєвий цикл чутливої інформації;

– класифікувати канали витоку даних та вектори загроз, специфічні для операційної системи Linux;

– дослідити існуючі методи ідентифікації контенту;

– розробити архітектуру та програмну реалізацію утиліти для виявлення підозрілої активності з файлами в реальному часі;

– розробити методику та провести експериментальні дослідження для оцінки точності виявлення загроз та продуктивності розробленого засобу.

Предмет дослідження. Методи, алгоритми та програмні засоби виявлення та аналізу витоків чутливих даних в операційних системах сімейства Linux.

Об'єкт дослідження. Процес забезпечення конфіденційності інформації та захисту від інсайдерських загроз в автоматизованих системах на базі Linux.

Наукова новизна одержаних результатів кваліфікаційної роботи. У роботі запропоновано метод для підвищення ефективності захисту конфіденційної інформації в автоматизованих системах на базі Linux шляхом удосконалення засобів виявлення витоків даних, що грунтується на основі процесу моніторингу файлової активності та ідентифікації загроз в режимі реального часу.

Практичне значення одержаних результатів. Результати роботи можуть бути використані системними адміністраторами та спеціалістами з інформаційної безпеки для посилення захисту серверів та робочих станцій під управлінням ОС Linux, а також як база для подальшої розробки комплексних open-source DLP-систем.

Апробація результатів роботи. Основні результати проведених досліджень обговорювались на: XIII науково-технічній конференції «Інформаційні моделі, системи та технології».

Публікації. Основні результати кваліфікаційної роботи опубліковано у працях конференції (див. Додаток А).

1 АНАЛІТИЧНИЙ ОГЛЯД МЕТОДІВ ТА ЗАСОБІВ ВИЯВЛЕННЯ ВИТОКІВ ДАНИХ

1.1 Сутність проблеми витоку чутливих даних та підходи до її вирішення

Чутливі дані є центральним активом, що потребує захисту. У найширшому сенсі, це інформація, несанкціоноване розкриття, зміна або знищення якої може завдати значної шкоди організації або суб'єкту даних. В даному розділі розглянемо термінологічний та концептуальний фундамент для дослідження проблеми витоків даних, визначаючи ключові поняття, їх життєвий цикл, наслідки загроз та базові підходи до їх нейтралізації.

У міжнародній бізнес-практиці та стандартах безпеки використовується для чутливої інформації застосовується досить широка класифікація, вона охоплює будь-яку інформацію, розкриття якої є небажаним, та, як правило, включає:

– Особиста інформація, така як імена, адреси, номери соціального страхування, фінансові деталі.

– Захищена інформація про здоров'я, а саме медичні записи, історії пацієнтів (регулюється такими стандартами, як HIPAA у США).

– Фінансова інформація – номери кредитних карток, банківські реквізити (регулюється PCI DSS).

– Конфіденційна бізнес-інформація – комерційні таємниці, пропрієтарні дані, стратегічні плани.

– Біометричні дані – відбитки пальців, скани сітківки, дані розпізнавання обличчя.

Для коректного впровадження політик безпеки, чутливі дані зазвичай класифікують за рівнями, виходячи з потенційних наслідків їхнього розкриття:

– Публічні – інформація, що не несе ризиків у разі розкриття (наприклад, контент веб-сайту).

– Внутрішні – інформація, недоступна громадськості, але розкриття якої несе мінімальний ризик (наприклад, внутрішні політики).

– Конфіденційні, такі як дані клієнтів або співробітників, розкриття яких становить помірний ризик.

– Обмежені, а саме високочутливі дані (PII, комерційні таємниці), розкриття яких матиме серйозні наслідки.

Чутливі дані повинні залишатися приватними, точними, надійними та доступними, це пов’язує захист чутливих даних із класичною тріадою кібербезпеки (CIA): конфіденційність, цілісність, доступність. Для коректного аналізу методів виявлення та реагування, критично важливо розуміти принципову різницю між втратою та витоком даних, яка полягає у тому, який саме аспект тріади CIA було порушено [1].

Витік даних визначається як несанкціоноване розкриття, розголошення, зміна або знищення конфіденційної інформації. Це, перш за все, порушення конфіденційності. Суть інциденту полягає в тому, що чутлива інформація стає доступною неавторизованим особам, незалежно від того, чи була вона при цьому викрадена, чи просто випадково опублікована. Можливими сценаріями витоку даних є: зовнішні кібератаки, дії інсайдерів, помилки конфігурації хмарних сервісів, випадкове надсилання інформації не тому адресату, необізнаність користувачів. Наслідки несуть за собою прямі фінансові збитки, регуляторні штрафи, втрату довіри клієнтів.

Втрата даних, натомість, означає фізичне або логічне знищення даних, небажане видалення, або ситуацію, коли дані стають недоступними для авторизованих користувачів. Це, перш за все, порушення доступності та/або цілісності. Найпоширенішими сценаріями втрати даних є: апаратні збої (вихід з ладу жорсткого диска), помилки програмного забезпечення, атаки програм-вимагачів, які шифрують дані, випадкове або навмисне видалення даних користувачем, природні катастрофи, що знищують дата-центри. Як наслідок:

втрата доступу до критичної інформації, порушення або повна зупинка ділової діяльності.

Фундаментальна різниця між цими інцидентами полягає у необхідних механізмах реагування. Втрата даних вимагає активації плану аварійного відновлення, основною метою якого є якнайшвидше відновлення даних із резервних копій. Натомість витік даних вимагає активації плану реагування на інциденти, який є набагато складнішим і включає не лише технічні (форензика, стримування загрози), але й юридичні, комплаєнс та комунікаційні заходи. Організація зобов'язана провести розслідування, оцінити збитки та, відповідно до регуляцій, повідомити наглядові органи та постраждалих суб'єктів даних. В даній роботі зосередження саме на проблемі витоку даних, а не їх втраті.

Хоча зовнішні атаки є значною загрозою, аналіз інцидентів свідчить, що інсайдерські загрози є однією з найскладніших та найдорожчих проблем безпеки. Агентство з кібербезпеки та захисту інфраструктури США (CISA) формально визначає інсайдера як особу, яка має або мала авторизований доступ до мереж, систем або даних організації. Загроза полягає у потенціалі використання цього доступу для завдання шкоди, навмисно чи ненавмисно [2]. Розуміння того, що інсайдер не є монолітною загрозою, є критичним для вибору адекватних методів виявлення. Типологія інсайдерських загроз поділяє їх на три основні категорії.

Перша категорія – ненавмисні загрози, це інсайдери, що спричиняють ризик без злого умислу, найчастіше через помилки або недбалість. Недбалий інсайдер – це співробітник, який знає політики безпеки, але свідомо ігнорує їх задля зручності (наприклад, копіює робочі файли на особисту флешку, використовує особисту пошту для пересилання чутливих документів). Випадковий інсайдер – це співробітник, який ненавмисно припускається помилки (наприклад, відправляє лист з чутливими даними не тому адресату) або стає жертвою атак соціальної інженерії (наприклад, клікає на фішингове посилання та компрометує свій пристрій).

Друга категорія – це навмисні загрози. Зловмисний інсайдер, який діє зі злим умислом, мотивований помстою, особистою образою, фінансовою вигодою

або ідеологічними переконаннями. Їхні дії включають крадіжку інтелектуальної власності (IP), саботаж або шпигунство.

Третя категорія – компрометовані загрози. Цей тип загрози є гібридним. Інсайдер (співробітник) не є зловмисником, але його облікові дані (логін та пароль) були викрадені зовнішнім зловмисником (наприклад, через фішинг або шкідливе ПЗ). У цьому випадку зловмисник діє зсередини, використовуючи легітимні права доступу скомпрометованого користувача [3].

Ця типологія має фундаментальне значення для даного дослідження. Вона демонструє, що не існує єдиного методу для боротьби з інсайдерами. Кожен тип вимагає власного підходу до виявлення та запобігання:

Недбалий інсайдер може бути ефективно зупинений за допомогою класичних систем запобігання витокам (DLP), які працюють на основі чітких політик (наприклад, заборонити копіювання на USB").

Зловмисний інсайдер вимагає більш складних технологій, таких як аналіз поведінки користувачів та об'єктів (User and Entity Behavior Analytics, UEBA), які здатні виявляти аномалії, що відхиляються від нормальної поведінки користувача.

Компрометований інсайдер вимагає поєднання сильних засобів автентифікації (наприклад, Multi-Factor Authentication, MFA) для запобігання несанкціонованому входу та засобів моніторингу кінцевих точок для виявлення шкідливої активності на робочій станції.

Таким чином, визначення інсайдера безпосередньо впливає на вибір та архітектуру методів виявлення витоків.

Чутливі дані не є статичним об'єктом; вони динамічно переміщуються в межах організації, проходячи через різні етапи обробки та зберігання. Цей процес відомий як життєвий цикл даних. Розуміння цього циклу є ключовим для ідентифікації точок, де дані є найбільш вразливими.

Національний інститут стандартів і технологій США (NIST) визначає життєвий цикл інформації як етапи, через які проходить інформація, що зазвичай характеризуються як створення або збір, обробка, поширення, використання, зберігання та утилізація, що включає знищення або видалення. Для цілей даного

аналізу ризиків, цю модель доцільно адаптувати до практичної 6-етапної структури, що широко використовується в індустрії безпеки даних: створення, зберігання, використання, передача, архівування та знищення. Кожен з цих етапів має унікальні вразливості, що вимагають специфічних методів контролю.

Отож життєвий цикл даних розпочинається з етапу створення, де інформація надходить через введення користувачами, автоматичний збір сенсорами або імпорт із зовнішніх джерел. Після створення настає етап зберігання, коли дані розміщуються у базах даних, на файлових серверах або локальних дисках. Найбільш динамічним є етап використання, що передбачає активну обробку, аналіз та редагування інформації. Оскільки цей етап орієнтований на людей, він несе найбільші ризики неконтрольованих дій. Під час передачі даних електронною поштою, через API або месенджери, головною вимогою є захист каналів зв’язку. Відсутність шифрування в русі та використання протоколів на кшталт HTTP замість HTTPS дозволяє зловмисникам перехоплювати трафік. Коли дані перестають бути активними, вони переходять на етап архівування для довгострокового зберігання. Завершальним етапом є знищення, яке вимагає безпечного та незворотного видалення інформації. Аналіз життєвого циклу демонструє, що захист даних не може бути забезпечений єдиним рішенням. Він вимагає набору взаємопов'язаних заходів контролю, що застосовуються на кожному з шести етапів.

Витоки чутливих даних призводять до багатогранних збитків, які виходять далеко за межі прямих фінансових витрат. Для повноцінного розуміння проблеми необхідно проаналізувати економічні, репутаційні та юридичні наслідки.

Аналіз останніх звітів "Cost of a Data Breach Report" від IBM демонструє цікаву динаміку. Звіт 2024 року зафіксував зростання глобальної середньої вартості витоку даних на 10% до $4.88 мільйонів [4].

На відміну від економічних збитків, репутаційні важко виміряти, але їхній вплив є довготривалим та руйнівним. Витік даних призводить до двох основних наслідків: втрати довіри та негативного впливу на бренд. Втрата довіри є одним із найсерйозніших наслідків. Коли клієнти, партнери та інвестори втрачають

впевненість у здатності компанії захищати їхні дані, це призводить до прямої відмови від послуг. Новини про фінансові махінації або витоки даних викликають недовіру, що призводить до зменшення кількості замовлень та втрати ринкової позиції. Відновлення репутації та іміджу після такого інциденту вимагає величезних зусиль та ресурсів, причому деякі клієнти можуть не повернутися ніколи.

З впровадженням суворих регуляцій, таких як GDPR в Європі, юридичні наслідки витоків стали однією з найбільших статей витрат. Механізм санкцій GDPR є безпрецедентно жорстким, передбачаючи штрафи до 20 мільйонів євро або 4% від загального річного обороту бізнесу в усьому світі, залежно від того, яка сума є більшою.

Враховуючи складність життєвого циклу даних та серйозність наслідків витоків, організації потребують проактивних технологічних рішень. Основним підходом до вирішення цієї проблеми є Data Loss Prevention (DLP) – системи запобігання витокам (або втраті) даних. DLP – це комплексне рішення, що поєднує технології, процеси та політики, розроблені для ідентифікації, моніторингу та захисту чутливих даних. Фундаментальна мета DLP-системи – забезпечити, щоб чутливі дані не покинули периметр організації та не були використані неавторизованими особами.

Існує декілька ключових завданнь, які вирішують сучасні DLP-системи. Основнлю ціллю є захист персональних даних (PII) та комплаєнс. DLP-системи мають вбудовані шаблони для автоматичного розпізнавання даних, що підпадають під регуляції (GDPR, HIPAA, PCI DSS), та блокування їх несанкціонованої передачі.

Також ключовим завданням є захист інтелектуальної власності та комерційної таємниці. Запобігання крадіжці пропрієтарних даних зловмисними інсайдерами або через зовнішні атаки.

Використовуються системи і для протидії інсайдерським загрозам. DLP є ключовим інструментом протидії як недбалим інсайдерам, так і зловмисним.

А також надання видимості та контролю над спробами співробітників завантажувати корпоративні дані у несанкціоновані хмарні сервіси (особисті Dropbox, Google Drive, ChatGPT) [5].

1.2 Класифікація загроз та каналів витоку даних в середовищі Linux

В умовах стрімкої цифровізації критичної інфраструктури та масової міграції корпоративних обчислювальних потужностей у хмарні середовища, операційні системи сімейства Linux зайняли домінуючу позицію на ринку серверних платформ. Ця повсюдність робить їх пріоритетною ціллю для кіберзлочинців, які розробляють все більш витончені методи проникнення та ексфільтрації даних. Середовище Linux, будучи фундаментом для більшості серверних інфраструктур, хмарних платформ та середовищ розробки, формує унікальний профіль загроз. На відміну від корпоративних Windows-середовищ, де користувачі часто мають обмежені права, адміністратори та розробники в Linux зазвичай володіють привілеями суперкористувача, що робить їхні облікові записи критичною точкою відмови.

Зовнішні загрози еволюціонують у бік складних цільових атак. Зловмисники використовують вразливості нульового дня у веб-сервісах та базах даних для первинного проникнення, після чого закріплюються в системі за допомогою руткітів рівня ядра або шкідливих модулів. Останні дослідження вказують на зростання загрози через ланцюги постачання, коли шкідливий код впроваджується у пакети відкритих репозиторіїв, націлені на ексфільтрацію змінних середовища, SSH-ключів та облікових даних.

Внутрішні загрози в Linux ускладнюються наявністю потужного вбудованого інструментарію, який може бути використаний подвійно (dual-use tools). Утиліти як (netcat, curl, ssh, dd, grep) є стандартними для адміністратора, але водночас ідеальними для зловмисного інсайдера для пошуку, пакування та виведення даних. Детекція такої активності вимагає глибокого розуміння контексту, оскільки просте блокування цих інструментів паралізує роботу легітимних адміністраторів.

Канали витоку в Linux можна систематизувати за рівнем стека технологій та вектором експлуатації. До мережевих каналів слід віднести приховані канали. Використання дозволених протоколів для тунелювання даних. DNS-тунелювання (ексфільтрація через піддомени в DNS-запитах) та ICMP-тунелювання є класичними прикладами, де витік маскується під службовий трафік. Аналіз ентропії пейлоаду пакетів та частотний аналіз запитів є основними методами виявлення таких аномалій [6].

Найпоширеніший канали адміністрування (SSH/SCP/SFTP) є водночас і найпоширенішим каналом витоку. Шифрування SSH унеможливлює аналіз вмісту на рівні мережевого шлюзу без розгортання складних проксі-рішень з підміною сертифікатів (Man-in-the-Middle), що часто є технічно складним для реалізації в середовищах розробки.

А також веб-протоколи (HTTP/S).Вивантаження даних на хмарні диски (Google Drive, Dropbox), Pastebin-сервіси або через веб-пошту.

Не менш важливими є локальні та фізичні канали. У середовищі Linux монтування USB-накопичувачів регулюється підсистемою ядра та правилами. Некоректна конфігурація або відсутність блокування на рівні драйверів дозволяє користувачам безперешкодно копіювати дані. Особливу небезпеку становлять пристрої, що емулюють клавіатуру для введення шкідливих команд.

Перехоплення даних з буферу або створення скріншотів може здійснюватися шкідливим ПЗ. Підсистема друку також може бути використана для виведення конфіденційних документів у паперовому вигляді без належного аудиту.

І в решті хмарні та інфраструктурні канали. У хмарних середовищах, що працюють на Linux, найчастішою причиною витоків є логічні помилки конфігурації, зазвичай це залишені відкритими бакети, бази даних без аутентифікації або публічно доступні API-ключі у скриптах.

Для того, щоб виявляти витоки в системах Linux потрібно точно знат, де шукати докази. На відміну від закритих систем, Linux надає повний доступ до своїх конфігураційних файлів та системних журналів. Ці артефакти є головним джерелом для виявлення компрометації.

Зміни в ключових конфігураційних файлах часто є першим і найнадійнішим провідним індикатором компрометації. Зловмисники модифікують ці файли для забезпечення персистентності (закріплення в системі) або створення бекдорів. Тому моніторинг цілісності файлів (File Integrity Monitoring – FIM) для цих файлів є критично важливим [7].

1.3 Аналіз сучасних Data Loss Prevention (DLP) систем

Ефективна протидія витокам даних вимагає впровадження спеціалізованих програмних комплексів, здатних здійснювати глибоку інспекцію контенту та контекстний аналіз операцій з інформацією. Сучасні DLP-системи будуються за модульним принципом, де кожен компонент відповідає за захист даних у певному стані. За точкою застосування контролю DLP-рішення класифікують на три основні категорії: Network DLP, Storage DLP та Endpoint DLP.

Network DLP функціонує безпосередньо на рівні мережевого шлюзу організації, реалізуючись як апаратний пристрій, віртуальна машина або проксі-сервер, через який маршрутизується трафік. Його робота базується на глибокій інспекції пакетів, що дозволяє на льоту аналізувати протоколи SMTP, HTTP/S, FTP та IM. Система автоматично блокує передачу повідомлень або файлів, якщо в них виявлено чутливі дані. Ключовою перевагою такого підходу є простота розгортання та незалежність від операційної системи, що дозволяє захищати Linux, Windows, macOS, а також IoT пристрої. Однак у контексті Linux це рішення має суттєві обмеження: воно неефективне проти зашифрованого трафіку без налаштування складних схем SSL/TLS Inspection і не здатне контролювати внутрішні операції, такі як копіювання на USB, що залишає робочі станції розробників вразливими.

Клас Storage DLP рішень зосереджений на виявленні та класифікації даних, які вже зберігаються в інфраструктурі. Спеціальні сканери періодично обходять файлові сервери, бази даних, сховища та хмарні репозиторії, створюючи інвентар чутливої інформації та перевіряючи права доступу до неї. Для середовища Linux використання Storage DLP є критично важливим, оскільки

Linux-сервери часто виконують роль файлових сховищ або серверів баз даних. Такий моніторинг дозволяє виявити потенційні загрози, які часто ігноруються, наприклад, забуті архіви з бекапами або лог-файли, що містять відкриті паролі.

Endpoint DLP являє собою агентське програмне забезпечення, яке встановлюється безпосередньо на кінцевий пристрій – сервер, робочу станцію або ноутбук. Агент інтегрується в операційну систему на рівні драйверів або модулів ядра, що дозволяє перехоплювати системні виклики до файлової системи, буфера обміну, підсистеми друку та периферійних пристроїв. Для Linux-середовища це найбільш складний, але водночас і найнеобхідніший компонент. Тільки Endpoint DLP здатний запобігти копіюванню даних на флеш-накопичувачі, друку конфіденційних документів або витоку інформації через шифровані SSH-тунелі, оскільки він бачить дані ще до моменту їх шифрування. Втім, саме цей тип DLP вважається найнестабільніший для Linux: через фрагментацію дистрибутивів та різноманітність версій ядра розробка стабільних агентів є значно ускладненою.

Ринок Endpoint DLP для Linux значно вужчий порівняно з Windows. Більшість вендорів надають для Linux урізаний функціонал, часто обмежуючись лише скануванням файлової системи, без можливостей блокування буфера обміну або контролю пристроїв.

Ринок Endpoint DLP для Linux є значно вужчим у порівнянні з екосистемою Windows. Більшість вендорів пропонують для Linux-середовища функціонал з суттєвими обмеженнями, часто зводячи можливості агента лише до сканування файлової системи, ігноруючи при цьому необхідність блокування буфера обміну або контролю периферійних пристроїв. Попри це, комерційні рішення класу Enterprise залишаються затребуваними завдяки централізованому управлінню, наявності готових шаблонів політик безпеки та гарантованій технічній підтримці.

Перший продукт, Symantec зберігає лідерство на ринку завдяки одній із найбільш зрілих технологій. Їхній агент для Linux, орієнтований на Red Hat Enterprise Linux та CentOS, забезпечує сканування локальної файлової системи та моніторинг мережевих з’єднань. Сильною стороною рішення є потужний

двигун контентного аналізу, здатний розпізнавати складні об'єкти, такі як векторні креслення та точні відбитки баз даних, а також єдина консоль управління. Проте впровадження Symantec пов'язане з високими витратами на ліцензування та складністю розгортання. Також критичним недоліком є жорстка прив'язка до версій ядра Linux: оновлення ядра часто призводить до непрацездатності агента.

Продукт, що розвивається під брендом Trellix після злиття McAfee Enterprise та FireEye, використовує модуль Trellix Endpoint Security for Linux. Це рішення вирізняється широкою підтримкою дистрибутивів та надійним контролем пристроїв, що дозволяє блокувати USB-накопичувачі за серійними номерами або класами. Головними перевагами є тісна інтеграція з консоллю ePolicy Orchestrator, яка є корпоративним стандартом у багатьох компаніях, та гнучкість у створенні правил блокування на основі атрибутів файлів. Однак функціонал контентної інспекції на Linux все ще відстає від Windows-версії, а налаштування виключень для системних процесів Linux залишається складним завданням.

Рішення Endpoint Protector (CoSoSys) позиціонується як крос-платформний продукт із акцентом на рівноцінну підтримку всіх операційних систем. Endpoint Protector пропонує найбільш повний набір функцій для Linux серед конкурентів, включаючи повний контроль USB та периферії, глибокий контентний аналіз та сканування даних у спокої. Агент підтримує навіть менш поширені дистрибутиви, такі як OpenSUSE та Linux Mint. Серед ключових переваг – підтримка нових версій ОС та інтуїтивний веб-інтерфейс. Особливу цінність для Linux-середовища становлять спеціалізовані функції захисту вихідного коду, що є критично важливим для робочих станцій розробників [8]. Однак даний продукт фокусується переважно на кінцевій точці, в нього менш розвинута мережева частина.

Сегмент безкоштовних DLP-систем для Linux наразі є досить проблемним. Значна частина проектів, які користувалися популярністю десятиліття тому, сьогодні залишилися без підтримки. Використання такого покинутого програмного забезпечення створює серйозні ризики для безпеки інфраструктури,

оскільки воно не містить актуальних патчів та механізмів захисту від новітніх загроз.

Такі відомі інструменти, як OpenDLP та MyDLP, у сучасних реаліях варто розглядати виключно в історичному контексті або як приклади архітектури, але аж ніяк не як рекомендовані засоби захисту. OpenDLP, написаний на Perl, колись був ефективним інструментом для сканування даних у спокої. Однак проект не підтримується вже понад 10 років, що робить його використання неможливим через несумісність із сучасними системними бібліотеками. MyDLP починався як перспективний проект, що пропонував мережевий шлюз та ендпоінт-агенти. Проте після комерціалізації безкоштовна версія перестала отримувати оновлення сигнатур та підтримку нових ядер Linux, а репозиторії проекту стали неактивними.

Через відсутність на ринку готового Open Source DLP для Linux, сучасна стратегія захисту полягає у комбінуванні спеціалізованих інструментів безпеки. Інженери власноруч будують аналог DLP-системи, використовуючи різні утиліти для закриття конкретних векторів загроз.

Wazuh – це потужна XDR/SIEM платформа, хоч і не є DLP у чистому вигляді, ефективно виконує суміжні функції. Модуль дозволяє в реальному часі відстежувати доступ до критичних файлів та їх модифікацію. Крім того, надає можливість автоматично реагувати на інциденти.

Також популряним є використання спеціалізованого інструменту USBGuard, що реалізує функцію Device Control. Він є ефективною заміною комерційним модулям контролю пристроїв, дозволяючи створювати гнучкі білі та чорні списки USB-накопичувачів.

Для специфічних середовищ використовують Apache Ranger – це рішення забезпечує централізоване управління політиками доступу. Воно виконує роль DLP для баз даних, дозволяючи динамічно застосовувати маскування даних для захисту чутливої інформації.

1.4 Методи ідентифікації чутливих даних

Ефективність будь-якої системи захисту від витоків фундаментально залежить від її здатності точно розпізнавати конфіденційну інформацію серед величезного масиву корпоративних даних. Цей процес, відомий як Data Discovery, є першим і критичним етапом у побудові стратегії безпеки.

Не існує єдиного, універсального методу ідентифікації, який би ефективно охоплював усі типи та стани даних. Кожен метод має свої переваги, недоліки та обчислювальні витрати. Ефективна утиліта повинна комбінувати декілька підходів у гібридній архітектурі. Методи ідентифікації можна класифікувати за чотирма основними стовпами.

Перший – детерміністичні методи. Це найбільш прямий підхід, що використовує чітко визначені правила для пошуку структурованих рядків даних. Яскравим прикладом є регулярні вирази (RegEx). Вони ідеально підходять для даних з відомим форматом.

В свою ж чергу сигнатурні методи порівнюють об'єкти (файли або фрагменти даних) з базою даних відомих чутливих зразків. Вони варіюються від простого хешування файлів до складного фінгерпринтингу документів [9].

Контекстуальні методи – це методи в яких ідентифікація відбувається не на основі вмісту, а на основі попередньо присвоєних властивостей або ярликів (тегів), які зберігаються у метаданих файлу.

Евристичні та ймовірнісні методи: Це найбільш складні методи, що використовують статистичний аналіз, обробку природної мови та машинне навчання для ідентифікації чутливих даних у неструктурованому вигляді та розуміння їх семантичного контексту.

Методи на основі регулярних виразів (RegEx) є найпоширенішим та фундаментальним механізмом для ідентифікації структурованих чутливих даних. Регулярний вираз – це послідовність символів, що визначає шаблон пошуку. Переваги цього методу очевидні: висока швидкість сканування та відносно низькі обчислювальні витрати, що робить його придатним для аналізу

в реальному часі. RegEx є надзвичайно ефективним для пошуку даних, що мають чіткий, стандартизований і незмінний формат [10].

Типові приклади ідентифікаторів, що реалізуються за допомогою RegEx це фінансові дані такф, як номери кредитних карток (напр.,\b(?:\d[-]*?){13,16}\b для карток довжиною 13-16 цифр з можливими пробілами або дефісами). А також національні ідентифікатори. Номери соціального страхування США (SSN) (напр.,\b\d{3}-\d{2}-\d{4}\b). Українські індивідуальні податкові номери (ІПН), які є 10-значними числами. Українські ID-картки (номер документа, РНОКПП). Номери міжнародних паспортів, які часто мають специфічні формати, що поєднують літери та цифри (напр., ^[A-Za-z]{1,2}\d{6,9}$).

Сигнатурний аналіз (також відомий як фінгерпринтинг даних) – це метод ідентифікації, що полягає у порівнянні файлів або фрагментів даних з базою даних попередньо створених сигнатур відомих чутливих документів. Даний метод поділяється на два типи: криптографічне шифрування та фінгерпринтинг документів.

Перший тип полягає у створенні криптографічного хешу для всього файлу. Найпоширенішими алгоритмами є MD5 (128-біт) та SHA-256 (256-біт). Принцип дії заключається в тому, що спершу хеш обчислюється для кожного файлу, що сканується, і порівнюється з чорним списком хешів відомих конфіденційних файлів.

Другий тип – це значно більш просунута та стійка техніка, розроблена для вирішення проблеми крихкості хешування. Фінгерпринтинг документів дозволяє виявляти часткові та похідні збіги, ігноруючи несуттєві зміни. Замість одного хешу для всього файлу, система розбиває документ на безліч малих фрагментів. Потім, за допомогою спеціалізованих алгоритмів, система обирає та хешує лише деякі з цих фрагментів. Це створює набір з десятків або сотень "міні-хешів" для одного документа.

Попередні методи покладаються на аналіз вмісту файлу. На противагу їм, контекстуальний метод – він приймає рішення на основі властивостей файлу, а не його вмісту. Найпотужнішою з цих властивостей є тегування або класифікація.

Класифікація даних – це процес організації даних за категоріями на основі їхньої чутливості, цінності та вимог відповідності. Це не технічний метод виявлення, а скоріше, організаційний процес, результати якого використовуються DLP-системами.

Типова схема класифікації включає 3-4 рівні :

– Public (Загальнодоступні) – інформація, призначена для вільного розповсюдження (напр., маркетингові матеріали).

– Internal (Внутрішні) – Інформація для внутрішнього використання, витік якої не спричинить значної шкоди (напр., внутрішні довідники).

– Confidential (Конфіденційні) – чутливі дані, витік яких може завдати шкоди (напр., списки клієнтів, фінансові плани).

– Restricted / Highly Confidential (Суворо обмежені) – найбільш критичні дані, витік яких матиме катастрофічні наслідки (напр., вихідний код, дані вищого керівництва) [11].

Ці теги можуть присвоюватися або вручну (самим користувачем при створенні документа), або автоматично (спеціалізованими інструментами класифікації).

Поки RegEx та сигнатури ефективні для структурованих та відомих даних, а метадані – для вже класифікованих, залишається величезний пласт даних, де ці методи безсилі. Це неструктуровані дані – електронні листи, чати, юридичні документи у вільній формі, медичні нотатки, коментарі у вихідному коді.

Проблема RegEx полягає у відсутності контексту. Він не може відрізнити номер телефону у списку клієнтів від номера телефону в публічній презентації. Сигнатури IDM не можуть знайти новий чутливий документ, який ще не був проіндексований.

Тут вступають в дію статистичні та лінгвістичні методи, зокрема обробка природної мови (Natural Language Processing, NLP) та машинне навчання (Machine Learning, ML). Ці методи дозволяють не просто шукати рядки, а розуміти семантичне значення та контекст тексту. Наприклад, в охороні здоров'я, більшість даних є неструктурованими, тому NLP є єдиним способом витягти з них значущу інформацію для аналізу.

2 РОЗРОБКА АРХІТЕКТУРИ ТА МЕТОДІВ УТИЛІТИ МОНІТОРИНГУ

2.1 Обґрунтування вибору технологічного стеку

В даному розділі роботі здійснимо огляд архітектури та концепції роботи утиліти для моніторингу та виявленню чутливої інформації Розробка агента моніторингу для середовища Linux вимагає балансу між продуктивністю, безпекою пам'яті та швидкістю розробки. Критично важливим є вибір інструментів, що дозволяють ефективно обробляти великі потоки подій файлової системи з мінімальним впливом на ресурси користувача.

Для реалізації програмного комплексу було обрано мову системного програмування Rust. Цей вибір обґрунтований низкою факторів, що надають суттєві переваги порівняно з традиційними C/C++ та інтерпретованими мовами, такими як Python. Насамперед, ключовою перевагою є гарантії безпеки пам'яті. Оскільки агенти безпеки працюють із високими системними привілеями та постійно обробляють ненадійні вхідні дані, будь-яка вразливість може бути критичною. Rust, завдяки своїй моделі володіння та механізму перевірки позичань, унеможливлює цілий клас помилок, таких як переповнення буфера, та стани гонитви, ще на етапі компіляції. При цьому, на відміну від Java чи Go, це досягається без використання збирача сміття.

Окрім безпеки, Rust забезпечує високу продуктивність завдяки концепції абстракцій з нульовою вартістю. Компіляція у нативний машинний код через інфраструктуру LLVM (Low Level Virtual Machine) дозволяє досягти швидкодії на рівні C++. Важливо, що використання високорівневих конструкцій, таких як ітератори та pattern matching, не створює додаткового навантаження під час виконання програми, що робить код одночасно виразним та ефективним [12].

Суттєву роль у виборі технології відіграли сучасний інструментарій та розвинена екосистема. Наявність пакетного менеджера Cargo значно спрощує процес управління залежностями та збіркою проєкту. Крім того, екосистема Rust пропонує надійні та зрілі бібліотеки, необхідні для проєкту: notify для системного моніторингу, tokio для асинхронного виконання та serde для

серіалізації даних. Це дозволяє розробникам зосередитися на безпосередній логіці виявлення загроз, а не на реалізації низькорівневих системних обгорток.

Notify – це Rust-бібліотека, яка надає уніфікований програмний інтерфейс (API) для моніторингу змін у файловій системі незалежно від операційної системи [13]. Основна мета бібліотеки – приховати платформо-специфічні деталі та надати розробникам єдиний інтерфейс для роботи з подіями файлової системи.

Архітектурно notify складається з трьох основних компонентів. Перший шар – API інтерфейс. Це публічний інтерфейс, який використовують додатки. Він визначає типи подій, методи для створення спостерігачів та налаштування параметрів моніторингу.

Наступний, другий шар – є диспетчером платформ. Цей компонент відповідає за вибір найбільш підходящого backend, в залежності від операційної системи. На етапі компіляції визначається цільова платформа і автоматично підключається відповідна реалізація.

Третій шар – це платформо-специфічний backend. Для кожної операційної системи існує окрема реалізація, яка використовує нативні API операційної системи. На Linux це inotify, на macOS – FSEvents, на Windows – ReadDirectoryChangesW, на BSD-системах – kqueue.

Inotify (inode notify) був введений в Linux kernel версії 2.6.13 у 2005 році як заміна застарілих механізмів dnotify. Основна мета inotify – надати ефективний, масштабований та зручний API для моніторингу подій файлової системі. На відміну від попередника dnotify, inotify працює з file descriptors замість directory descriptors, що робить його більш гнучким та безпечним. Кожен watch descriptor асоціюється з конкретним inode, а не з відкритим файловим дескриптором, що дозволяє відстежувати файли навіть після їх закриття.

Inotify складається з трьох основних компонентів в ядрі Linux. Inotify instance – представлений file descriptor, через який додаток отримує події. Кожен процес може створити декілька instances для різних цілей моніторингу.

Watch descriptor є унікальним ідентифікатором для кожного об'єкту моніторингу (файлу або директорії). Watch descriptor зберігає інформацію про те, які події потрібно відстежувати для конкретного inode.

Event queue – це черга подій в kernel space, де накопичуються події до моменту їх зчитування додатком. Розмір черги обмежений системним параметром max_queued_events [14].

Крім цього Inotify підтримує широкий спектр подій файлової системи:

– IN_ACCESS генерується при читанні файлу. Використовується для аудиту доступу до конфіденційних даних.

– IN_MODIFY генерується при модифікації вмісту файлу. Критична подія для DLP систем, оскільки вказує на потенційну зміну даних.

– IN_ATTRIB генерується при зміні метаданих файлу (права доступу, власник, timestamps). Може вказувати на спробу приховування слідів.

– IN_CLOSE_WRITE генерується при закритті файлу, відкритого для запису. Сигналізує про завершення модифікації файлу.

– IN_CLOSE_NOWRITE генерується при закритті файлу, відкритого тільки для читання.

– IN_OPEN генерується при відкритті файлу. Корисна для моніторингу спроб доступу.

– IN_MOVED_FROM та IN_MOVED_TO генеруються парою при переміщенні файлу. Cookie поле дозволяє зв'язати ці дві події.

– IN_CREATE генерується при створенні нового файлу або директорії в моніторингуваній директорії.

– IN_DELETE генерується при видаленні файлу або директорії.

– IN_DELETE_SELF генерується при видаленні самого об'єкту, що моніториться.

– IN_MOVE_SELF генерується при переміщенні об'єкту, що моніториться.

Ще одним важливим компонентом в архітектурі є Tokio – це асинхронний runtime для мови програмування Rust, який є фундаментальним компонентом системи. Tokio надає інфраструктуру для виконання асинхронного коду, керування операціями введення-виведення та координації паралельних задач.

Без Tokio утиліта система не могла б ефективно обробляти одночасно файлові події, HTTP запити та інші асинхронні операції [15].

Традиційні синхронні програми блокуються під час виконання операцій введення-виведення. Коли програма зчитує файл, очікує на надходження мережевих даних або відповідь від бази даних, центральний процесор (ЦП) перебуває у стані простою. Для обробки великої кількості одночасних операцій можливе створення окремого потоку для кожного завдання, проте потоки споживають значний обсяг оперативної пам'яті та створюють суттєві накладні витрати ресурсів на перемикання контексту. Асинхронне програмування вирішує цю проблему за допомогою механізму кооперативної багатозадачності. Замість блокування на операціях введення-виведення, асинхронна функція призупиняє своє виконання, повертаючи управління планувальнику, що дозволяє іншим завданням виконуватися в межах того ж потоку. Це дає можливість ефективно обробляти тисячі одночасних операцій, використовуючи обмежену кількість системних потоків.

Бібліотека Tokio забезпечує повноцінне середовище виконання (runtime) для асинхронного коду мовою Rust. Це середовище включає планувальник завдань, реактор введення-виведення (що базується на системному виклику epoll у Linux), механізми обробки таймерів та інші компоненти, необхідні для побудови надійних високопродуктивних систем.

У розробці DLP-системи бібліотека Tokio відіграє ключову роль у забезпеченні високої продуктивності та масштабованості. Вона виступає фундаментом для критичних компонентів системи, керуючи виконанням коду, розподілом ресурсів та обробкою подій.

Отже, архітектура програмного комплексу базується на використанні мови Rust, яка унеможливлює критичні помилки пам’яті без втрати швидкодії. Реалізація моніторингу через inotify надає доступ до широкого спектра подій, дозволяючи точно ідентифікувати підозрілу активність. Це робить використане рішення ефективним інструментом для аудиту файлової системи та захисту конфіденційної інформації в середовищі Linux.

2.2 Проектування загальної архітектури утиліти

Архітектура базується на подійно-орієнтованій моделі (Event-Driven Architecture) з використанням асинхронного середовища виконання Tokio. Система складається з незалежних модулів, що комунікують через асинхронні канали, що забезпечує слабку зв'язність та відмовостійкість. Утиліт складається з декількох модулів: модуль конфігурації, модуль сканування, модуль аналізу контенту, модуль журналювання подій, а також додаткових модулів, що контролюють змінні носії та мережу.

В загальному архітектуру утиліти можна представити у вигляді наступної схеми, див. рис. 2.1.

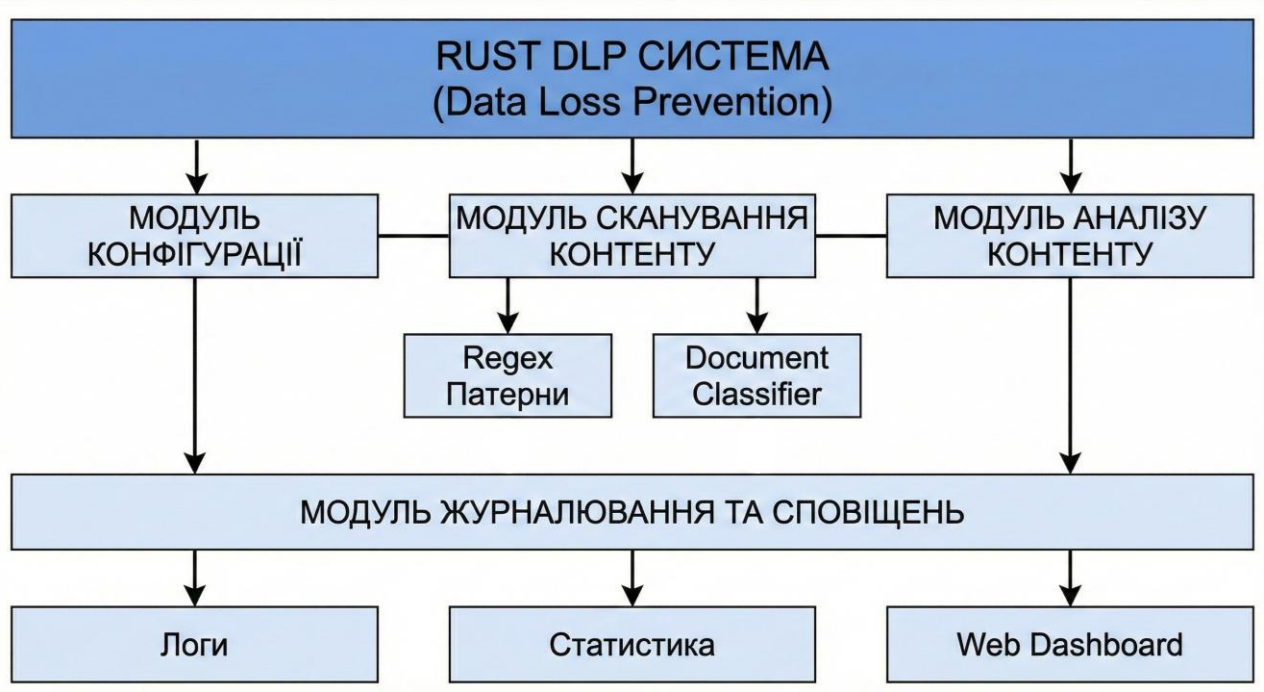

Рисунок 2.1 – Схема архітектури утиліти

Модуль конфігурації виступає ключовим компонентом системи, що забезпечує централізоване керування параметрами функціонування всіх підсистем. Основне призначення даного модуля полягає у завантаженні, перевірці коректності (валідації) та наданні доступу до налаштувань через інтерфейс із суворою типізацією.

Для зберігання конфігураційних даних обрано формат TOML (Tom's Obvious, Minimal Language). Цей формат поєднує зручність сприйняття людиною з простотою автоматичного синтаксичного аналізу. На відміну від формату JSON, TOML підтримує використання коментарів та має більш природний синтаксис для опису ієрархічних структур, що значно спрощує процес адміністрування системи без необхідності залучення спеціалізованих інструментів редагування.

Архітектура модуля базується на принципі суворої типізації. Кожна секція конфігураційного файлу програмно відображається у відповідну структуру даних мови Rust із використанням бібліотеки серіалізації. Такий підхід гарантує коректність структури даних ще на етапі компіляції, що дозволяє виключити цілий клас потенційних помилок, пов'язаних із невідповідністю типів даних або відсутністю обов'язкових полів.

Система забезпечує автоматизоване виявлення широкого спектру конфіденційної інформації за допомогою попередньо налаштованих шаблонів пошуку.

До підтримуваних категорій належать фінансові дані, такі як номери платіжних карток (Visa, Mastercard, American Express), міжнародні номери банківських рахунків (IBAN). Крім цього, автентифікаційні дані: паролі, ключі доступу до API, приватні криптографічні ключі, облікові дані хмарних сервісів (зокрема AWS). А також персональні дані: реквізити українських ідентифікаційних документів (паспорти, реєстраційні номери облікових карток платників податків), медичні записи, адреси електронної пошти та телефонні номери.

Окрім пошуку за шаблонами, реалізовано механізм евристичного аналізу документів. Секція налаштувань дозволяє класифікувати файли на основі частотного аналізу ключових слів. Для кожного класу документів визначається набір термінів, порогове значення мінімальної кількості збігів та рівень критичності. Такий підхід уможливлює виявлення неструктурованих документів, таких як протоколи тендерних комітетів, юридичні договори,

фінансові звіти або медичні висновки, навіть за відсутності в них чітко формалізованих даних, що піддаються пошуку за регулярними виразами.

Процес завантаження конфігурації реалізовано через послідовність чітко визначених етапів. Спершу вміст файлу config.toml повністю завантажується в оперативну пам'ять як текстовий масив у кодуванні UTF-8 за допомогою системного виклику читання.

Після чого текстові дані проходять обробку бібліотекою парсингу TOML, яка трансформує їх у проміжну ієрархічну структуру.

На фінальному етапі відбувається автоматичне відображення проміжної структури на типи даних мови Rust з використанням бібліотеки serde. Цей процес включає сувору валідацію типів та перевірку наявності всіх обов'язкових полів.

Критично важливою архітектурною особливістю є статична ініціалізація налаштувань. Завантаження конфігурації відбувається одноразово під час запуску програми, після чого параметри зберігаються в оперативній пам'яті протягом усього часу роботи системи. Будь-які зміни у конфігураційному файлі не застосовуються динамічно – для їх вступу в силу необхідний повний перезапуск сервісу. Такий підхід гарантує узгодженість поведінки системи та повністю виключає можливість виникнення станів гонитви (race conditions), які могли б виникнути при одночасній зміні правил фільтрації та обробці потоку файлів.

Модуль сканування виступає інтелектуальним ядром системи, що виконує безпосередній аналіз вмісту файлів з метою виявлення конфіденційної інформації. Архітектурно модуль реалізує два взаємодоповнюючі механізми детекції: пошук за шаблонами на основі регулярних виразів та контекстну класифікацію документів за ключовими словами.

Підсистема базується на використанні компільованих регулярних виразів. На етапі ініціалізації модуля всі шаблони, визначені у конфігурації, проходять процес компіляції за допомогою бібліотеки regex. Компіляція трансформує текстове представлення регулярного виразу в оптимізований кінцевий автомат (state machine), який зберігається в оперативній пам'яті. Це є критично важливим

для забезпечення високої продуктивності: пошук за допомогою компільованого автомату виконується на порядки швидше, ніж при використанні інтерпретатора.

Алгоритмічна база механізму містить суттєві оптимізації. Для простих шаблонів застосовується алгоритм Бойєра-Мура або аналогічні методи швидкого пошуку підрядка. Натомість для складних виразів, що містять конструкції попереднього/наступного перегляду та групи захоплення, використовується алгоритм пошуку з поверненням із застосуванням мемоізації проміжних результатів. Це дозволяє уникнути експоненційного зростання часової складності.

Особлива увага приділена підтримці стандарту Unicode. Всі регулярні вирази працюють із текстом у кодуванні UTF-8, що гарантує коректну обробку україномовних документів, зокрема розпізнавання кириличних символів у шаблонах паспортних даних чи податкових номерів [16].

Аналіз вмісту файлу виконується лінійно – за один прохід. Для кожного шаблону застосовується ітеративний метод пошуку збігів. Кожен виявлений інцидент фіксується разом із метаданими:

– Ідентифікатор шаблону – для класифікації типу даних.

– Маскований вміст – знайдений фрагмент тексту у захищеному вигляді.

– Рівень критичності – визначається згідно з налаштуваннями.

– Позиція у файлі – зміщення від початку файлу у байтах.

З міркувань інформаційної безпеки реалізовано обов'язкове маскування чутливих даних перед збереженням у журнал подій . Система ніколи не зберігає повні номери кредитних карток або паролі. Це дозволяє адміністратору ідентифікувати тип витоку без отримання доступу до фактичних конфіденційних значень.

Механізм класифікації реалізує високорівневий аналіз контексту. Замість пошуку жорстко заданих шаблонів, він виявляє ознаки, притаманні певним типам документів. Кожен класифікатор оперує набором характерних термінів та пороговим значенням мінімальної кількості збігів. Алгоритм класифікації складається з наступних кроків:

– Конвертація всього вмісту файлу у нижній регістр для пошуку без урахування регістру символів.

– Перевірка наявності кожного ключового слова зі словника класифікатора.

– Формування списку знайдених термінів.

– Якщо кількість унікальних збігів досягає заданого мінімуму, документ отримує відповідну класифікацію.

– Обчислення відсоткового відношення знайдених термінів до загального обсягу словника класифікатора.

Ключовою перевагою обраного підходу є взаємодоповнюваність методів. Регулярні вирази ефективні для структурованих даних із чітким форматом (номери карток, коди), тоді як класифікатори краще працюють із неструктурованими текстовими документами.

Для економії обчислювальних ресурсів реалізовано стратегію раннього завершення сканування. Метод аналізу критичності перериває обробку файлу, як тільки досягнуто умови блокування (наприклад, знайдено хоча б один збіг рівня "Critical" або "High"). Якщо у файлі вже виявлено номер кредитної картки, подальший пошук інших патернів не є доцільним, оскільки файл вже підлягає ізоляції. Це дозволяє значно знизити навантаження на центральний процесор при обробці файлів великого обсягу.

Модуль аналізу контенту виступає центральним координуючим елементом системи, що забезпечує інтеграцію підсистеми моніторингу файлової системи, механізму сканування та логіки застосування політик безпеки. Цей компонент реалізує повний життєвий цикл обробки інциденту: від моменту детекції файлової операції до прийняття рішення про блокування.

Технологічним фундаментом модуля є інтеграція з бібліотекою notify, яка забезпечує рівень абстракції над специфічними для різних операційних систем механізмами спостереження. В середовищі Linux використовується підсистема ядра inotify, що дозволяє ефективно відстежувати зміни у файловій системі в реальному часі.

Критично важливим аспектом реалізації є підтримка рекурсивного моніторингу. Оскільки нативний механізм inotify не підтримує відстеження

вкладених директорій, бібліотека notify емулює цю функціональність, автоматично обходячи дерево каталогів та реєструючи окремий дескриптор спостереження (watch descriptor) для кожної піддиректорії. Система динамічно додає нові дескриптори при створенні каталогів, мінімізуючи часове вікно стану гонитви (race condition) між створенням папки та початком спостереження за нею.

Перед запуском циклу обробки подій реального часу виконується етап початкового аудиту. Система здійснює рекурсивний обхід файлового дерева цільових директорій. Кожен файл перевіряється на відповідність критеріям аналізу, і у разі відповідності – передається на сканування. Це гарантує виявлення вже існуючих конфіденційних даних при старті системи.

Ядром модуля є цикл обробки подій. Архітектурно він реалізований через асинхронний канал типу MPSC (багато виробників – один споживач). Цей патерн забезпечує потокобезпечну передачу даних між потоком спостерігача, який отримує сигнали від ядра, та основним потоком обробки. Класифікація подій та реакція системи залежать від типу операції. При отриманні сигналу про створення файлу система ініціює примусову затримку тривалістю 500 мілісекунд. Це необхідно для синхронізації з операційною системою: подія створення inode генерується миттєво, проте фактичний запис даних на диск та скидання буферів може займати певний час. Без цієї затримки існує ризик зчитування порожнього або неповного файлу. Особлива увага приділяється операціям перейменування та модифікації, які містять інформацію про вихідний та цільовий шляхи файлу. Це дозволяє реалізувати логіку контролю периметра безпеки [16].

В архітектурі утиліти присутній механізм SafeZone – виділений простір, де дозволена робота з конфіденційними даними. Логіка захисту базується на принципі "нульової довіри" (Zero Trust) при спробі винесення даних за межі периметра логіка роботи наступна:

– При детектуванні переміщення перевіряється належність вихідного та цільового шляхів до захищеної зони.

– Переміщення в межах зони дозволяється.

– Якщо цільовий шлях знаходиться поза межами SafeZone, це кваліфікується як порушення політики.

– Після короткої технологічної затримки (200 мс) файл примусово повертається назад у захищену зону через системний виклик перейменування, незалежно від його вмісту.

Алгоритм аналізу та прийняття рішень базується на процесі перевірки файлів, що в свою чергу включає декілька етапів. На етапі дедуплікації відбувається перевірка, чи файл вже був оброблений, для уникнення повторного сканування при серії однотипних подій.

На етапі зчитуваняя вміст файлу завантажується в оперативну пам'ять як текстовий рядок у кодуванні UTF-8. Файли з некоректним кодуванням або розміром менше 10 байт ігноруються як такі, що не можуть містити значущих патернів.

Після чого на етапі сканування отриманий контент передається модулю пошуку, який повертає перелік виявлених збігів.

Фінальним етапом є аналіз критичності. Якщо серед знайдених збігів присутні такі, що мають рівень "Високий" (High) або "Критичний" (Critical), і файл знаходиться поза межами SafeZone, ініціюється процедура карантину.

Система веде безперервний збір статистики у потокобезпечній структурі, захищеній м'ютексом. Окремий фоновий потік періодично агрегує та зберігає дані про кількість перевірених файлів, виявлені інциденти та заблоковані загрози.

Модуль журналювання забезпечує спостережуваність системи шляхом структурованої реєстрації подій та надання статистичних даних у реальному часі через веб-інтерфейс. Архітектурно модуль розділено на два функціональні компоненти: підсистему ведення журналів та веб-панель моніторингу.

Накопичення метрик реалізовано через глобальну структуру Statistics, доступ до якої захищено м'ютексом для забезпечення потокобезпечності. Структура містить атомарні лічильники для всіх категорій подій. Спільний доступ різних модулів до екземпляра статистики організовано через розумний

вказівник з підрахунком посилань, що дозволяє інкрементувати лічильники з різних потоків без виникнення станів гонитви (race conditions).

М'ютекс гарантує взаємне виключення – одночасно лише один потік може змінювати дані. Накладні витрати на синхронізацію є мінімальними, оскільки критична секція коду є надзвичайно короткою (лише операція інкременту). Окремий потік-репортер періодично зчитує агреговані значення та фіксує їх у журналі, що дозволяє оцінювати активність системи без аналізу "сирих" логів.

Для візуалізації стану системи розроблено веб-інтерфейс на базі асинхронного фреймворку Axum, що функціонує поверх середовища виконання Tokio. Асинхронний підхід дозволяє ефективно обробляти значну кількість одночасних HTTP-з'єднань без необхідності створення окремого потоку для кожного клієнта. При ініціалізації сервер відкриває сокет на порту 8080 та запускає цикл обробки подій. Архітектура API включає три точки доступу.

Точка GET / повертає HTML-сторінку з клієнтським скриптом. Вміст сторінки вбудовується у виконуваний файл програми на етапі компіляції, що спрощує розгортання системи, усуваючи залежність від зовнішніх файлів ресурсів.

В свою ж чергу точка GET /api/stats повертає поточні статистичні дані у форматі JSON. Обробник запиту блокує м'ютекс на мінімальний час, необхідний для копіювання даних, після чого виконує серіалізацію та відправку відповіді.

Тоді як точка GET /api/quarantine надає перелік ізольованих файлів шляхом зчитування директорії карантину та аналізу метаданих.

Клієнтська частина реалізована на чистому JavaScript без використання важковагових фреймворків, що мінімізує споживання ресурсів браузера. Оновлення даних відбувається методом періодичного опитування API кожні 5 секунд. Використання таймера замість постійного з'єднання обумовлено простотою реалізації та достатністю такої частоти оновлення для задач DLP-системи. Також передбачено механізм ручного оновлення для миттєвої перевірки стану після тестових інцидентів.

Крім основних модулів в архітектурі утиліти є додаткові модулі, їх можна представити у вигляді наступної схеми, див. рис. 2.2.

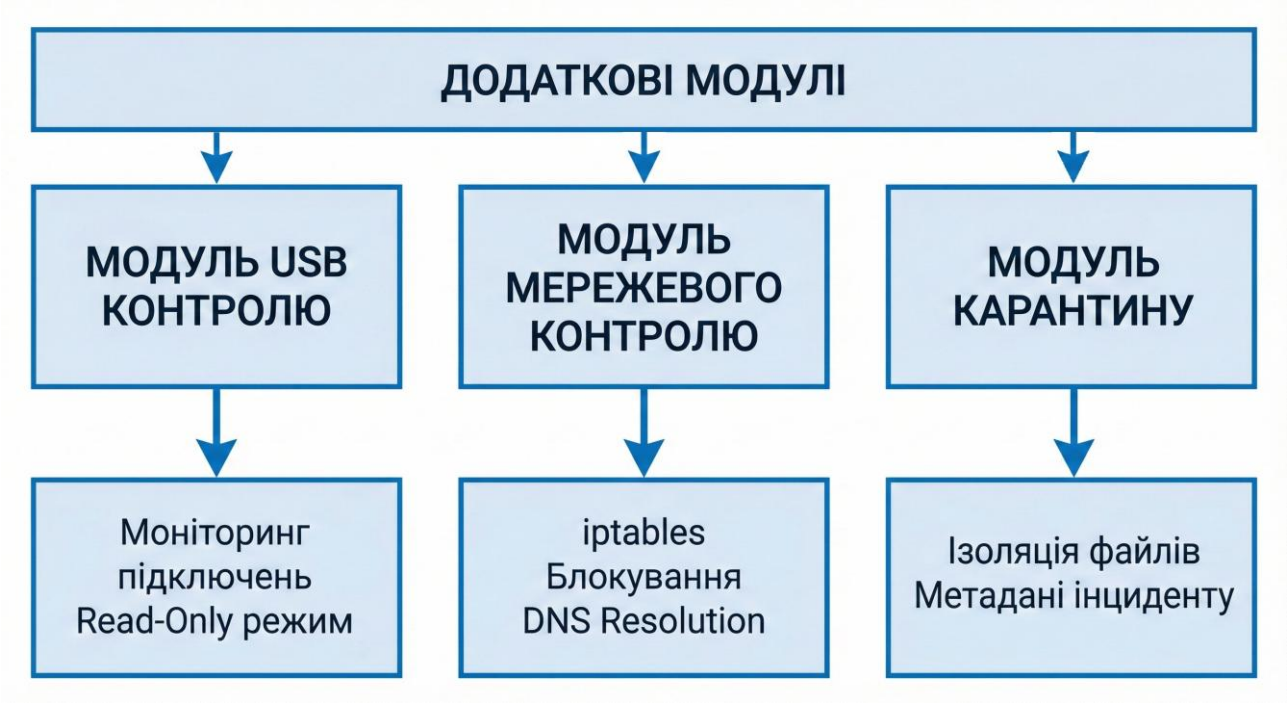

Рисунок 2.2 – Допоміжні модулі утиліти

Одним із ключових каналів витоку конфіденційної інформації є фізичне копіювання даних на зовнішні носії. Для мінімізації цього ризику було розроблено модуль USB. Основною метою даного компонента є забезпечення примусового переведення підключених USB-накопичувачів у режим тільки для читання. Такий підхід дозволяє користувачам переглядати вміст носіїв, необхідний для роботи, але фізично блокує будь-які спроби запису (ексфільтрації) даних на пристрій.

Функціонування модуля базується на безперервному моніторингу віртуальної файлової системи та аналізі шляхів монтування. В операційних системах сімейства Linux, відповідно до стандарту ієрархії файлової системи, існують детерміновані директорії, що використовуються для підключення знімних носіїв. Модуль здійснює нагляд за декількома критичними зонами.

Класична директорія /media, що використовується більшістю дистрибутивів для автоматичного монтування носіїв (USB-флеш-накопичувачів, CD-ROM тощо) від імені користувача.

Традиційна точка монтування /mnt для тимчасово підключених файлових систем, яку часто використовують системні адміністратори для ручних операцій.

А також сучасний стандарт шляху монтування /run/media, що використовується в дистрибутивах з ініціалізацією systemd. Ця директорія є тимчасовою файловою системою (tmpfs), де створюються підкаталоги для кожного користувача, що забезпечує розмежування доступу на рівні сесій.

На низькому рівні взаємодія модуля з ядром операційної системи реалізується через системний виклик mount. Механізм захисту не потребує відмонтування пристрою, що забезпечує прозорість роботи для користувача. Замість блокування доступу, модуль виконує операцію remount (перемонтування) зі зміною атрибутів доступу. Процес виглядає наступним чином:

– Система виявляє подію підключення нового блочного пристрою у відстежуваних директоріях.

– Виконується системний виклик mount із прапорами MS_REMOUNT та MS_RDONLY.

– Ядро Linux оновлює суперблок відповідної файлової системи, змінюючи права доступу на рівні драйвера файлової системи.

Використання прапора MS_REMOUNT є критично важливим, оскільки дозволяє змінити параметри монтування для вже змонтованої файлової системи без переривання активних процесів, які можуть зчитувати дані з носія в момент підключення. Це забезпечує стабільність роботи системи при одночасному дотриманні політик інформаційної безпеки.

Для запобігання витоку конфіденційної інформації через мережеві канали зв'язку розроблено модуль блокування веб-ресурсів. Його основним завданням є блокування доступу до неавторизованих ресурсів, таких як публічні файлообмінні мережі, хмарні сховища даних та підозрілі веб-сервіси. Реалізація фільтрації здійснюється на рівні мережевого стеку операційної системи, що унеможливлює обхід обмежень засобами прикладного рівня (наприклад, через налаштування браузера).

Функціонування модуля базується на використанні фреймворку Netfilter, інтегрованого безпосередньо в ядро Linux. Netfilter забезпечує інфраструктуру

для перехоплення та маніпуляції мережевими пакетами під час їх проходження через мережевий стек [16].

Для керування правилами фільтрації використовується утиліта простору користувача – iptables. Вона дозволяє адміністратору визначати таблиці правил, які ядро використовує для прийняття рішень щодо долі кожного окремого пакета (прийняти, відхилити, модифікувати тощо).

Архітектурно Netfilter організований навколо п'яти стандартних точок перехоплення (hooks), які відповідають етапам проходження пакету через ядро:

– PREROUTING: Обробляє всі вхідні пакети до моменту прийняття рішення про маршрутизацію. Використовується для DNAT (Destination NAT).

– INPUT: Обробляє пакети, призначені безпосередньо для локальної системи (local host).

– FORWARD: Відповідає за транзитні пакети, які проходять через систему, але призначені для іншого вузла мережі (режим маршрутизатора).

– OUTPUT: Обробляє пакети, згенеровані локальними процесами, перед тим як вони будуть передані на мережевий інтерфейс.

– POSTROUTING: Обробляє всі вихідні пакети після маршрутизації. Використовується для SNAT (Source NAT).

У контексті задачі Data Loss Prevention пріоритетним є контроль даних, що виходять за межі захищеного периметра. Тому модуль фокусує свою роботу на ланцюжку OUTPUT. Саме через точку OUTPUT проходять пакети від веб-браузерів, поштових клієнтів та утиліт командного рядка, запущених користувачем. Алгоритм роботи модуля включає декілька кроків. На першому кроці відбувається ініціалізація правил. При запуску утиліта формує список заборонених IP-адрес та підмереж, що відповідають відомим файлообмінникам та хмарним сервісам.

Після чого розпочинається інтеграція з ядром. За допомогою викликів iptables, модуль додає правила блокування у ланцюжок OUTPUT.

І в решті фільтрація, ядро Linux перевіряє кожен вихідний пакет на відповідність цим правилам. Якщо адреса призначення співпадає із забороненим

списком, пакет негайно відкидається, а з'єднання розривається ще до моменту встановлення.

Такий підхід забезпечує надійне блокування на низькому рівні, мінімізуючи використання ресурсів процесора, оскільки фільтрація відбувається в просторі ядра.

Механізм карантину є критично важливою складовою системи захисту від витоку даних. Його основна мета – негайна ізоляція файлу, що порушує політики безпеки, для запобігання подальшому розповсюдженню інформації та збереження цифрових доказів для подальшого аналізу адміністратором. Процес карантину реалізується шляхом фізичного переміщення файлу з користувацького простору в захищену системну директорію. З технічної точки зору, ця операція в середовищі Linux може виконуватися двома шляхами, залежно від конфігурації файлової системи.

Перший спосіб атомарне переміщення. У більшості випадків, коли вихідна директорія та карантин знаходяться в межах однієї файлової системи, використовується системний виклик rename. Ця операція є атомарною: вона змінює лише метадані файлової системи (шлях до файлу), не зачіпаючи фізичні блоки даних на диску.

Другий спосіб – копіювання з видаленням. Якщо карантинне сховище розташоване на іншому логічному розділі (наприклад, для забезпечення ізоляції місця на диску), системний виклик rename повертає помилку. У такому випадку модуль реалізує резервний алгоритм:

– Повне побітове копіювання файлу у цільову директорію.

– Верифікація цілісності (порівняння хеш-сум або розмірів).

– Примусове видалення оригінального файлу.

Переміщення файлу в карантин супроводжується рядом заходів для нейтралізації потенційної загрози. Спершу змінюються права доступу. Після переміщення файлу його права доступу примусово змінюються (зазвичай на 0600 або 0000). Це унеможливлює виконання коду (якщо файл був шкідливим скриптом) та блокує доступ до нього будь-яким користувачам, окрім root.

Також відбувається зміна власника. Власником файлу стає суперкористувач (root), що запобігає спробам користувача відновити файл або змінити його атрибути.

Для уникнення колізій імен (коли різні користувачі намагаються зберегти файли з однаковими назвами, наприклад report.docx), система генерує унікальні ідентифікатори (UUID) або додає часові мітки (timestamp) до назви файлу в карантині.

Оскільки переміщення файлу руйнує контекст інциденту (де файл знаходився раніше), система карантину створює супровідний файл-маніфест (JSON або XML). У ньому фіксуються наступне:

– Оригінальний шлях до файлу.

– Ім'я користувача (UID/GID), який намагався здійснити операцію.

– Час інциденту.

– Причина блокування (наприклад, спрацювання патерну регулярного виразу на вміст).

Такий підхід дозволяє адміністратору безпеки не лише аналізувати вміст перехопленого файлу, але й розуміти контекст порушення, що є необхідним для розслідування інцидентів.

Всі допоміжні модулі інтегруються з центральним модулем аналізу контенту через єдиний життєвий цикл:

– Ініціалізація – запуск системи, модулі читають конфігурацію та встановлюють необхідні системні обмеження.

– Моніторинг – робота модулів незалежно як фонові процес, паралельно з основним моніторингом директорій.

– Реагування – при виявленні загрози модуль аналізу контенту викликає підсистему карантину для ізоляції файлу.

Така архітектура забезпечує модульність, тестованість та можливість незалежного розвитку компонентів без впливу на основну логіку системи.

2.3 Моделювання процесів роботи утиліти

Отож, як вже було описано розроблена DLP утиліта побудована на основі модульної архітектури, яка складається з п'яти основних компонентів. Модуль конфігурації відповідає за завантаження та валідацію параметрів системи з файлу config.toml, забезпечуючи гнучке налаштування всіх аспектів роботи програми. Модуль сканування реалізує виявлення чутливих даних за допомогою регулярних виразів, що компілюються при ініціалізації системи. Модуль моніторингу здійснює спостереження за файловою системою та координує процес обробки виявлених подій. Модуль типів даних визначає структури даних та рівні критичності, забезпечуючи безпечну роботу всієї системи. Головний модуль координує роботу всіх компонентів системи та керує життєвим циклом програми. Схему роботи утиліти представлено на рисунку 2.3.

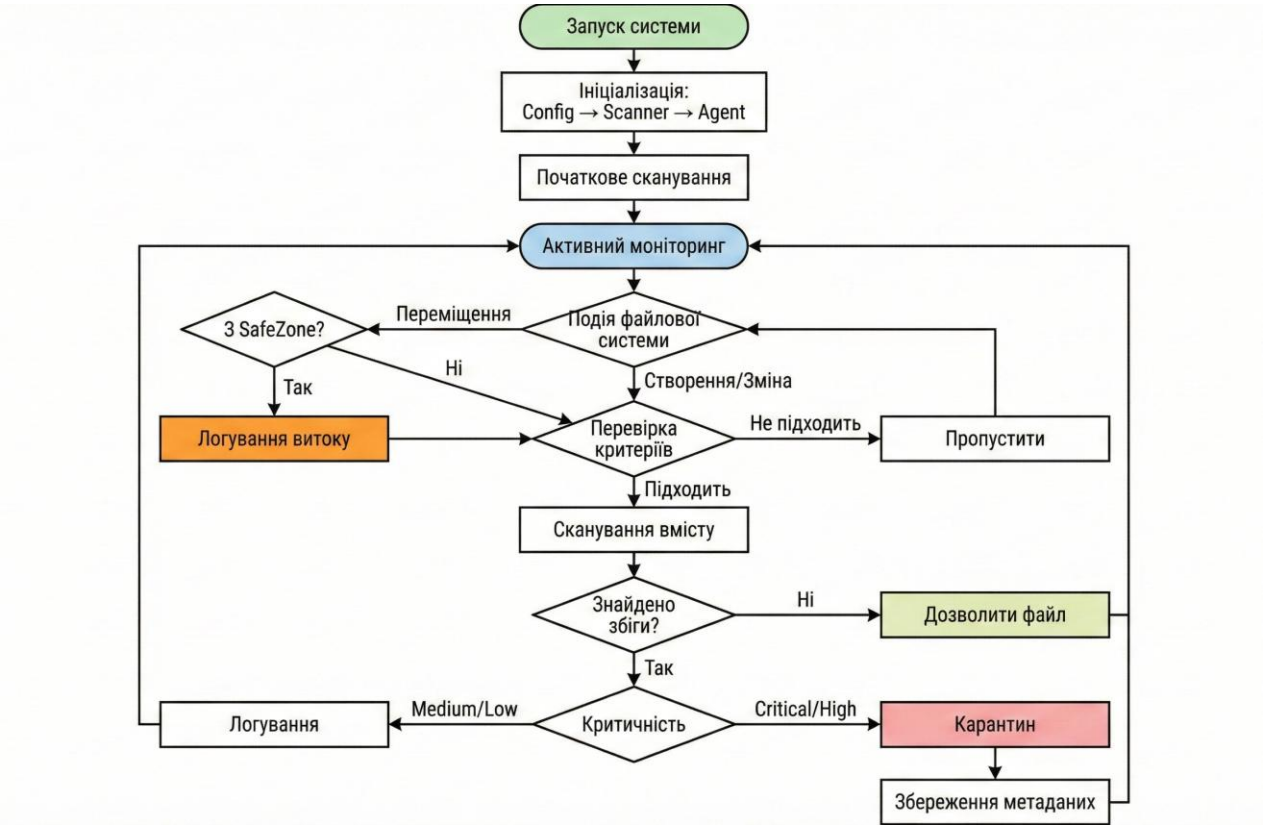

Рисунок 2.3 – Схема логіки роботи утиліти

Процес обробки файлів в системі можна представити у вигляді скінченного автомату з послідовними станами. Початковим станом є створення або модифікація файлу користувачем, що генерує подію в операційній системі. Система моніторингу виявляє цю подію файлової системи та передає

інформацію про файл до наступного етапу обробки. Здійснюється перевірка типу файлу, під час якої система аналізує розширення файлу та його розмір, щоб визначити чи підлягає файл скануванню. Якщо файл не відповідає критеріям, він ігнорується системою і обробка завершується.

Якщо файл пройшов первинну перевірку, відбувається читання його вмісту у текстовому форматі. Далі система виконує сканування вмісту застосовуючи всі налаштовані регулярні вирази для виявлення чутливих даних. За результатами сканування можливі два сценарії розвитку подій. У випадку відсутності збігів з патернами, файл отримує статус дозволеного, інформація про це записується у статистику і обробка завершується успішно. Якщо виявлено збіги, система переходить до аналізу їх критичності.

Аналіз критичності полягає у визначенні найвищого рівня серйозності серед знайдених збігів. Для збігів з рівнем критичності Medium або Low система обмежується логуванням інформації про виявлені дані, дозволяючи файлу залишатися доступним. Якщо виявлено збіги з рівнем Critical або High, система переходить до процедури карантину, блокуючи доступ до файлу та переміщуючи його у спеціальну директорію для ізоляції потенційно небезпечного контенту.

Запуск утиліти відбувається в чітко визначеній послідовності етапів, кожен з яких є критичним для коректної роботи системи. Перший етап полягає у завантаженні конфігурації з файлу config.toml, під час якого система зчитує та парсить всі параметри роботи. До цих параметрів належать шляхи директорій для моніторингу, шлях до безпечної зони, максимальний розмір файлу для сканування який становить десять мегабайт, список допустимих розширень файлів, та набір регулярних виразів для виявлення різних типів чутливих даних. У разі помилки на цьому етапі, робота програми припиняється, оскільки подальше функціонування без коректної конфігурації неможливе.

Другий етап передбачає ініціалізацію сканера контенту, під час якого всі регулярні вирази з конфігурації компілюються у внутрішні структури даних. Система створює патерни для виявлення номерів платіжних карток таких як Visa, Mastercard та American Express, міжнародних банківських номерів рахунків IBAN, паролів та ключів доступу до API, приватних криптографічних ключів,

персональних даних включаючи українські паспорти та ідентифікаційні номери, а також електронних адрес та телефонних номерів. Компіляція регулярних виразів відбувається лише один раз при запуску, що суттєво підвищує продуктивність подальшого сканування.

На третьому етапі створюється агент моніторингу кінцевих точок, який ініціалізує необхідні структури даних для роботи. Система створює директорію карантину якщо вона ще не існує, ініціалізує множину HashSet для відстеження вже просканованих файлів та запускає лічильники статистики для збору метрик роботи. Також на цьому етапі валідуються всі шляхи директорій для моніторингу та виводиться інформація про їх доступність.

Четвертий етап полягає у початковому скануванні існуючих файлів у всіх директоріях що підлягають моніторингу. Система рекурсивно обходить файлову систему до глибини трьох рівнів, виявляючи всі файли що відповідають критеріям сканування за розширенням та розміром. Кожен виявлений файл проходить повну процедуру сканування, що дозволяє виявити вже існуючі файли з чутливими даними ще до початку активного моніторингу змін.

П'ятий завершальний етап активує систему моніторингу файлової системи у реальному часі. Запускається файловий спостерігач на базі бібліотеки notify, який буде відстежувати всі зміни у вказаних директоріях. Одночасно запускається окремий потік для періодичного збору та виведення статистики, який кожні п'ять хвилин формує звіт про роботу системи. Після завершення всіх етапів ініціалізації, система переходить у режим активного моніторингу та готова до обробки подій файлової системи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРМЕНТАЛЬНЕ ДОСЛІДЖЕНННЯ УТИЛІТИ

3.1 Опис середовища розробки та тестування

Ефективність функціонування розробленого DLP-агента критично залежить від характеристик операційного середовища, оскільки утиліта взаємодіє з низькорівневими підсистемами ядра (файлова система, система подій inotify, планувальник процесів). Для забезпечення репрезентативності експериментальних даних та стабільності роботи програми було сформовано уніфіковане середовище розробки та тестування.

В якості базової платформи для розробки та проведення навантажувальних тестів обрано операційну систему Ubuntu Linux (версія 22.04 LTS). на базі гіпервізора Oracle VM VirtualBox Вибір даного дистрибутива зумовлений його поширеністю у корпоративному секторі, стабільністю ядра та широкою підтримкою інструментарію мови Rust. Цей підхід дозволяє емулювати типову станцію Linux, зберігаючи при цьому можливість швидкого відновлення стану системи через механізм знімків.

Характеристики гостьової системи (Guest OS): Для мінімізації впливу сторонніх процесів на результати вимірювань продуктивності та забезпечення сумісності з сучасними механізмами ядра.

Операційна система Ubuntu Linux 24.04 LTS була вибрана в ролі віртуальної машини. Вибір версії зумовлений наявністю ядра Linux 6.8+. Це є критичною умовою для коректної роботи підсистеми асинхронного вводу-виводу. Центральний процесор (vCPU) емулюється архітектура x86_64. Виділено 8 ГБ оперативної памя’яті. Такий обсяг рекомендовано для забезпечення можливості повного завантаження великих файлів у оперативну пам'ять. Віртуальний накопичувач (VDI/VMDK) розміщено на фізичному SSD-диску хост-системи для уникнення вузьких місць I/O.

Розробка велася з використанням стабільної гілки компілятора Rust (Stable Channel). Система збірки Cargo використовувалася для керування графом залежностей та запуску бенчмарків.

Стандартні налаштування Ubuntu розраховані на звичайне використання і встановлюють консервативні ліміти на кількість об'єктів спостереження inotify. Для коректної роботи DLP-агента, який має відстежувати тисячі файлів у реальному часі, було внесено зміни до конфігурації ядра, а саме збільшено ліміту спостерігачів та екземплярів.

3.2 Реалізація ключових функціональних модулів

Розроблена утиліта працює на комп'ютері користувача і слідкує за тим, які файли створюються або змінюються. Система працює локально, тобто всі дані залишаються на комп'ютері користувача і нікуди не передаються. Це важливо для приватності.

Модуль config.rs відповідає за читання налаштувань з файлу config.toml. Коли програма запускається, вона першим ділом звертається до цього модуля. Модуль відкриває файл config.toml, читає його як звичайний текст, а потім перетворює цей текст в структуровані дані, з якими програма вміє працювати через бібліотеку serde та toml парсер.

Структура Config є головною структурою даних модуля і містить всі налаштування системи. Вона включає п'ять основних секцій які покривають різні аспекти роботи утиліти. Секція MonitoringConfig визначає цільові директорії (watch_paths), опціональну "безпечну зону" для роботи з конфіденційними даними та фільтри сканування: обмеження розміру файлу до 10 МБ (для запобігання зависанню) та білий список текстових розширень. Секція QuarantineConfig задає шлях для ізоляції небезпечних файлів.

Ядром системи є секція patterns – хеш-мапа правил пошуку. Кожен запис містить назву, регулярний вираз та рівень серйозності. Правила охоплюють:

– Critical (блокування) – фінансові дані (картки Visa/Mastercard/Amex, IBAN) та аутентифікаційні дані (паролі, ключі API/AWS, приватні ключі).

– High (блокування) – персональні дані (паспорти, ІПН, SSN, медичні записи).

– Medium/Low (логування) – контактні дані (email, телефони), IP-адреси.

Функціонал розширюється опціональними модулями: UsbConfig переводить USB-накопичувачі в режим "тільки читання", WebConfig блокує доступ до хмарних сховищ на рівні мережі, а document_classes дозволяє класифікувати документи за набором ключових слів (наприклад, протоколи тендерів). Метод Config::load гарантує надійну обробку помилок при читанні або парсингу файлу. Код модуля наведено в додатку Б.

Модуль types.rs схожий на словник термінів для всієї програми. Він визначає всі типи даних які використовуються в системі та забезпечує типобезпеку на рівні компілятора через систему типів Rust.

Найважливіший тип даних тут – це enum Severity який визначає наскільки серйозною є знайдена інформація. Enum це перелічуваний тип даних у Rust який може мати одне з фіксованого набору значень. Severity має чотири варіанти: Low, Medium, High, Critical. Кожен варіант є просто іменованою константою без асоційованих даних. Система використовує декілька основних структур. Структура Pattern описує правило пошуку. Містить назву, рівень серйозності та, що критично важливо, скомпільований regex::Regex. Це гарантує високу продуктивність, оскільки складний процес компіляції виразу відбувається лише один раз при старті, а не при скануванні кожного файлу.

Структура Match фіксує деталі знайденого збігу – назву правила, замаскований текст, рівень серйозності та позицію. Позиція зберігається як індекс байту, що забезпечує точність навігації у файлах з UTF-8 кодуванням (де символи мають змінну довжину).

QuarantineEvent – це структура для формування детального звіту у форматі JSON, що зберігається поруч із заблокованим файлом. Вона містить оригінальний та карантинний шляхи, час блокування (RFC3339), SHA256 хеш файлу для контролю цілісності та список усіх виявлених загроз.

Для моніторингу роботи використовується структура Statistics, яка через поля типу u64 (що унеможливлює переповнення) підраховує кількість

сканованих, заблокованих та дозволених файлів, а також час аптайму. У багатопотоковому середовищі ця структура використовується через Arc<Mutex<>>, що гарантує безпечний доступ до лічильників з різних потоків. Код модуля подано в додатку Б.

Модуль scanner.rs – це серце системи детекції конфіденційних даних. Його головна задача – взяти текстовий вміст файлу і знайти в ньому всі чутливі дані згідно з правилами з конфігурації використовуючи як регулярні вирази так і класифікацію за ключовими словами.

Процес ініціалізації модуля через метод from_config починається з компіляції текстових патернів у оптимізовані скінченні автомати. Використання бібліотеки regex є критичним архітектурним рішенням, оскільки воно гарантує лінійний час виконання, що ефективно захищає систему від ReDoS-атак та зависань на складних виразах. Паралельно, якщо у конфігурації визначені специфічні класи документів, ініціалізується структура DocumentClassifier з відповідним набором маркерних слів.

Основна логіка сканування реалізована в методі scan, який обробляє вхідний текст у два послідовні етапи. Спершу відбувається перевірка регулярними виразами. При виявленні збігу система викликає функцію mask_sensitive, яка адаптує спосіб приховування даних залежно від їх довжини: короткі фрагменти маскуються повністю, тоді як у довших записах залишаються відкритими перші та останні чотири символи для ідентифікації типу даних. При цьому система фіксує точну позицію знахідки за індексом байта, що важливо для коректної роботи з UTF-8.

На другому етапі, якщо це передбачено налаштуваннями, виконується класифікація документа в цілому. Алгоритм перевіряє текст на наявність заданих груп слів, розраховує відсоток "впевненості" (confidence) та додає результат до загального списку загроз. Для швидкого прийняття рішень модуль надає допоміжний метод has_critical_matches, який дозволяє агенту миттєво з'ясувати, чи містить файл загрози рівня Critical або High, що є прямим сигналом для ізоляції об'єкта в карантин. Код модуля наведено в додатку Б.

Модуль document_classifier.rs реалізує класифікацію документів на основі наявності ключових слів на відміну від сканування регулярними виразами. Цей підхід дозволяє виявляти цілі типи конфіденційних документів навіть якщо вони не містять специфічних номерів чи кодів але мають характерну термінологію.

Основою конфігурації є структура DocumentClass, яка визначає параметри пошуку для кожної категорії: назву, набір характерних ключових слів, рівень серйозності та поріг спрацювання min_matches. Останній параметр є критично важливим, оскільки він визначає мінімальну кількість унікальних ключових слів, які повинні зустрітися в тексті, щоб документ був віднесений до цього класу. Наприклад, для ідентифікації тендерного протоколу можна задати поріг у два співпадіння зі списку слів "протокол", "закупівля", "учасник", що дозволяє відсіяти випадкові згадки окремих слів у безпечному контексті.

Логіка сканування зосереджена в методі classify. Для забезпечення ефективності та ігнорування регістру символів, метод спершу виконує конвертацію всього вмісту документа у нижній регістр. Це оптимізаційне рішення дозволяє виконувати пошук ключових слів незалежно від їх написання (наприклад, "Протокол" та "ПРОТОКОЛ") без необхідності багаторазового перетворення тексту. Алгоритм перевіряє наявність кожного ключового слова з налаштованих класів і, якщо кількість знайдених збігів досягає або перевищує min_matches, фіксує результат.

Результатом роботи є створення об'єктів DocumentMatch, які містять назву класу, список фактично знайдених слів та рівень загрози. Особливістю реалізації є метод розрахунку впевненості (confidence), який визначає відсоткову відповідність документа профілю класу, базуючись на співвідношенні знайдених слів до загальної кількості слів у словнику цього класу. Надійність роботи алгоритму забезпечується набором модульних тестів, які верифікують коректність детекції на реалістичних зразках тексту. Код модуля наведено в додатку Б.

Модуль endpoint.rs – це найскладніша і найбільша частина системи. Він відповідає за все: моніторинг файлової системи, координацію сканування,

прийняття рішень про блокування, управління карантином, роботу з безпечною зоною, та збір детальної статистики.

Процес ініціалізації агента включає підготовку інфраструктури, зокрема створення директорій для карантину та валідацію шляхів моніторингу. Після успішного старту система виконує первинне сканування наявних файлів, після чого переходить у режим активного спостереження. Моніторинг реалізовано на базі бібліотеки notify, яка використовує нативні API операційної системи (inotify, FSEvents) для миттєвої реакції на створення або модифікацію файлів. Для запобігання помилкам читання під час запису файлу сторонніми програмами агент застосовує адаптивну затримку перед початком сканування.

Ключовим елементом захисту є механізм контролю "безпечної зони" (SafeZone). Система реалізує специфічну логіку обробки подій переміщення файлів, відстежуючи спроби винесення даних за межі захищеного периметра. У разі виявлення несанкціонованого переміщення агент автоматично повертає файл у вихідну директорію та фіксує спробу порушення політики безпеки, дозволяючи при цьому вільну роботу з конфіденційними даними всередині самої зони.

Процедура ізоляції загроз активується, коли сканер виявляє поза безпечною зоною дані з рівнем загрози Critical або High. Процес карантину спроєктований як атомарна послідовність дій: файл миттєво переміщується в захищене сховище зі зміною назви, обчислюється його контрольна сума SHA256 для гарантії цілісності, а на оригінальному місці створюється текстова заглушка для інформування користувача. Паралельно генерується структурований JSON-звіт (QuarantineEvent), що містить всі метадані інциденту для подальшого аудиту. Фоновий потік статистики регулярно звітує про стан системи, забезпечуючи прозорість роботи захисних механізмів. Код модуля подано в додатку Б.

Модуль usb_blocker.rs реалізує захист від копіювання конфіденційних файлів на USB накопичувачі шляхом переведення їх у режим тільки-читання. Це додатковий рівень захисту який запобігає витоку даних навіть якщо файли не були заблоковані основною системою сканування наприклад бо вони зашифровані або упаковані.

Ініціалізація компонента базується на стандартах Linux-систем, де структура UsbBlocker формує список цільових шляхів для моніторингу, включаючи /media, /mnt та /run/media. Метод активації start працює за гібридним сценарієм: спершу він синхронно обробляє вже підключені накопичувачі, застосовуючи обмеження до наявних точок монтування, а потім запускає фоновий потік для відстеження нових пристроїв. Для моніторингу використовується бібліотека notify з інтервалом опитування у дві секунди та нерекурсивним режимом, оскільки систему цікавить лише факт появи нової директорії (точки монтування), а не зміни файлів у ній.

Реалізація обмежень у методі make_readonly спирається на системні утиліти Linux. Пріоритетним способом блокування є виконання команди mount з аргументами -o remount,ro. Це дозволяє перемонтувати файлову систему накопичувача в режим Read-Only "на льоту", гарантуючи повну заборону запису на рівні драйвера файлової системи. Такий підхід є найбільш надійним, проте вимагає наявності прав суперкористувача (root) у процесу агента.

На випадок відсутності необхідних привілеїв або помилки при монтуванні передбачено механізм відмовостійкості (fallback). Система автоматично переходить до методу apply_chmod_readonly, який рекурсивно виконує команду chmod -R a-w, знімаючи права на запис для всіх користувачів на рівні атрибутів файлів. Хоча цей метод є менш жорстким, ніж монтування, він дозволяє створити суттєві перешкоди для ексфільтрації даних навіть у середовищі з обмеженими правами доступу. Код модуля подано в додатку Б.

Модуль web_blocker.rs реалізує захист від завантаження конфіденційних файлів на хмарні сховища та файлообмінники шляхом блокування мережевого доступу до цих сервісів через правила iptables на рівні операційної системи.

Архітектура компонента базується на структурі WebBlocker, яка оперує вектором blocked_domains. При ініціалізації цей список автоматично наповнюється широким спектром цільових ресурсів, охоплюючи глобальні хмарні платформи (Dropbox, Google Drive, OneDrive, Box), сервіси швидкої передачі файлів (WeTransfer, Mega, MediaFire), а також ресурси для тимчасового хостингу (file.io), текстові pastebin-сервіси та навіть веб-версію Telegram. Такий

комплексний підхід дозволяє перекрити більшість популярних каналів несанкціонованого виводу інформації.

Процес активації захисту, реалізований у методі start, починається з перевірки сумісності операційної системи, оскільки механізм жорстко прив'язаний до ядра Linux. Основна логіка імплементації зосереджена у функції apply_iptables_blocks, яка спершу верифікує наявність встановленої утиліти iptables. За умови успішної перевірки система ініціює створення нового, ізольованого ланцюжка правил з назвою DLP_BLOCK (через команду з прапорцем -N). Використання окремого ланцюжка є важливим архітектурним рішенням, яке дозволяє сегментувати правила DLP від інших налаштувань брандмауера та спрощує управління мережевою безпекою.

3.3 Розробка методики експериментальних досліджень

Для об'єктивної оцінки ефективності розробленого програмного засобу необхідно сформувати комплексну методику, що охоплює як якісні показники (здатність виявляти загрози), так і кількісні (вплив на ресурси системи). Експериментальні дослідження поділяються на три ключові етапи: оцінка точності класифікації даних (Detection Accuracy), аналіз споживання системних ресурсів (Performance Profiling) та вимірювання часових затримок реагування (Latency Analysis).

Оскільки основною функцією агента є бінарна класифікація файлових об'єктів на "безпечні" та "чутливі", для верифікації ефективності використовується апарат математичної статистики. Формування тестового датасету: Для проведення експерименту формується еталонний набір даних (Ground Truth), що складається з двох контрольних груп. Перша група – позитивний набір, а самк файли, що містять реальні або синтетичні зразки чутливих даних (PII, номери кредитних карток, фрагменти приватних ключів, файли з високою ентропією).

Друга група в свою ж чергу це негативний набір, легітимні системні файли, вихідний код open-source проектів, текстові документи без конфіденційної інформації.

Результати сканування класифікується за матрицею помилок (Confusion Matrix):

– True Positive (TP): Агент коректно ідентифікував файл із чутливими даними та згенерував алерт.

– False Positive (FP): Агент помилково позначив безпечний файл як чутливий (хибне спрацювання).

– True Negative (TN): Агент коректно проігнорував безпечний файл.

– False Negative (FN): Агент пропустив файл, що містив чутливі дані.

На основі цих значень розраховуються ключові метрики ефективності, такі як: повнота, точність, F1-міра.

Повнота (Recall/Sensitivity) відображає здатність системи виявляти всі фактичні загрози. Для DLP-систем це критичний показник, оскільки пропуск навіть одного файлу (FN) може призвести до витоку [18]. Для обислення використовується наступна формула (3.1):

$$Recall = \frac{TP}{TP + FN} \times 100\% \quad (3.1)$$

Точність (Precision) відображає рівень довіри до алертів системи. Низька точність призводить до явища "втоми від сповіщень" (alert fatigue), коли адміністратори починають ігнорувати попередження через велику кількість шуму (FP), використовується наступна формула (3.2):

$$Precision = \frac{TP}{TP + FP} \times 100\% \quad (3.2)$$

F1-міра (F1-Score): Гармонічне середнє, що дозволяє знайти баланс між агресивністю блокування та зручністю використання, використовується наступна формула (3.3):

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3.3)$$

Враховуючи, що DLP-агент працює у фоновому режимі (daemon), критично важливим є мінімальний вплив на продуктивність хоста. Для оцінки використовуються системні утиліти Linux (ps, top, pidstat) для збору метрик процесу агента.

Швидкодія системи визначається як часовий інтервал між фізичним завершенням запису даних на диск та моментом генерації сповіщення про інцидент.

3.4 Проведення тестування та аналіз отриманих результатів

Перевіримо роботу утиліти, для цього створимо декілька файлів з різними даними в тестовій директорії та запустимо утиліту, для того, щоб перевірити чи зможе утиліта виявити чутливу інформацію в цих файлах, див. рис. 3.1.

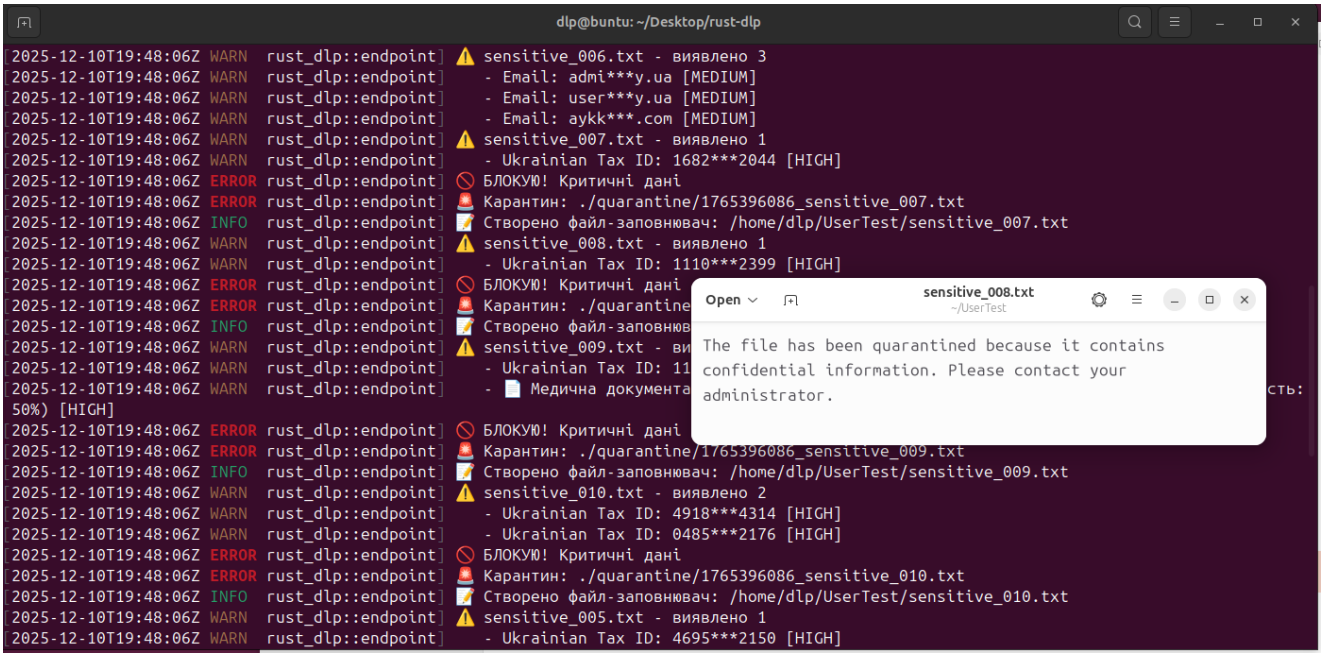

Рисунок 3.1 – Результат сканування утиліти

Згідно результатів, утиліта успішно виявила файли з критичною інформацією та перемістила їх в карантин. Далі перевіримо поведінку утиліти в

режимі реально часу, та реакцію на модифікування файлу для цього запустимо утиліту та під час її роботи створимо файли з звичайним текстом, а потім добавимо туди чутливу інформацію, див. рис. 3.2.

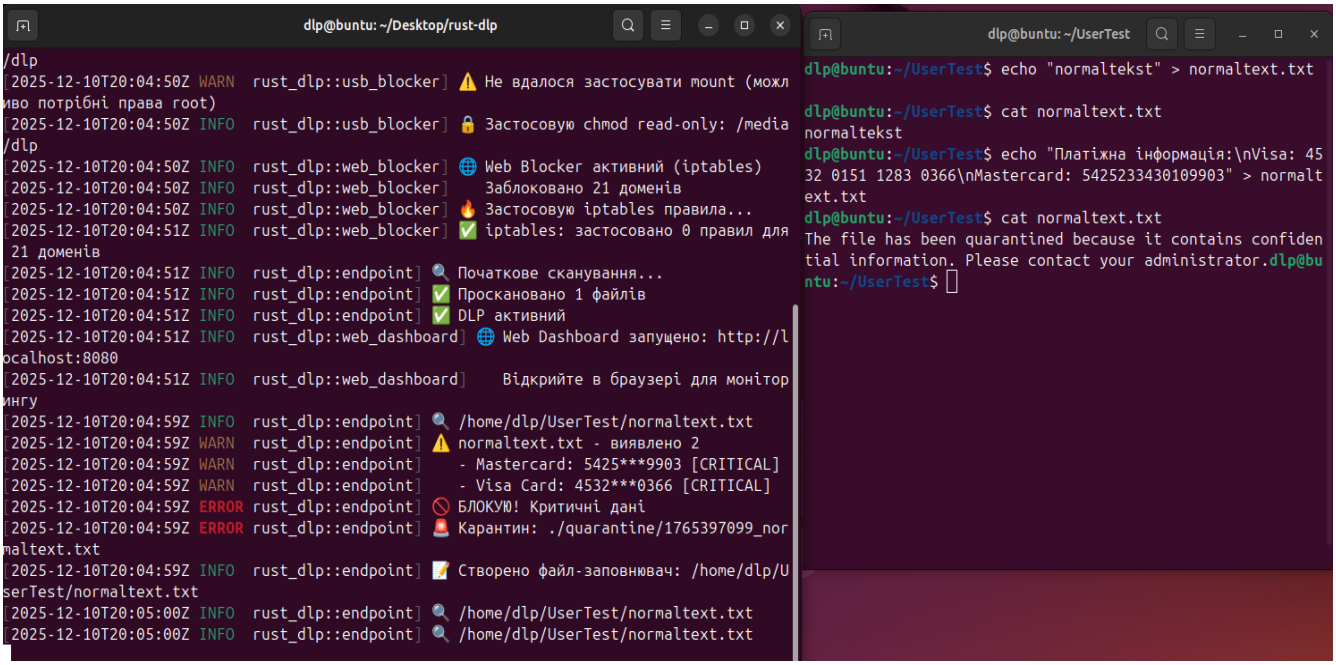

Рисунок 3.2 – Результат роботи утиліти в реальному часі

За результатами роботи можна зробити висновки, що утиліта успішно виконує дане завдання. Ще одним важливим функціоналом є наявність SafeZone, спеціальної директорії де є можливість створювати та модифікувати файли з чутливою інформацією, однак заборонено переміщати та копіювати їх за межі директорії, як зображено на рисунку 3.3.

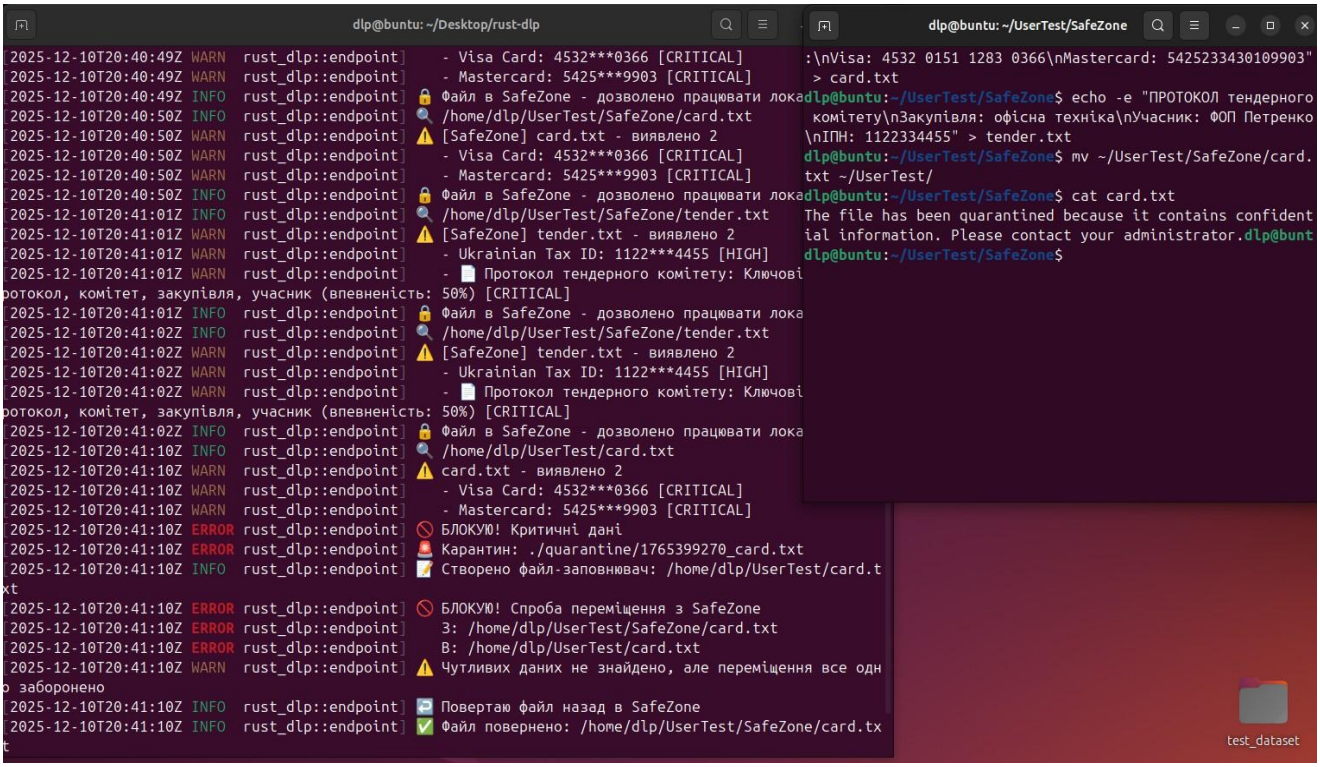

Рисунок 3.3 – Робота в SafeZone

Переглянемо описаний додатковий функціонал утиліти, а саме блокування веб-ресурсів через iptables, див. рис. 3.4.

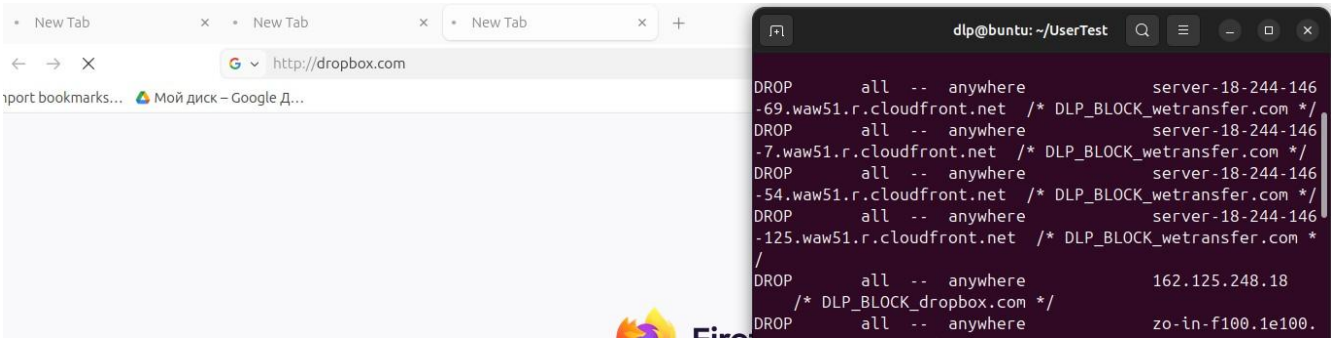

Рисунок 3.4 – Модуль блокування веб-ресурсів

Результат роботи модуля позитивний, модуль успішно блокує ресурси налаштовані в конфігураційному файлі утиліти.

Ще одним важливим функціоналом є логування усіх подій файлів, що потрапили на карантин, події зберігаються в лог-файлах, а також мають зручне представлення у вигляді статистики у веб-інтерфейсі з регулярним оновленням даних в режимі реального часу, див. рис. 3.5.

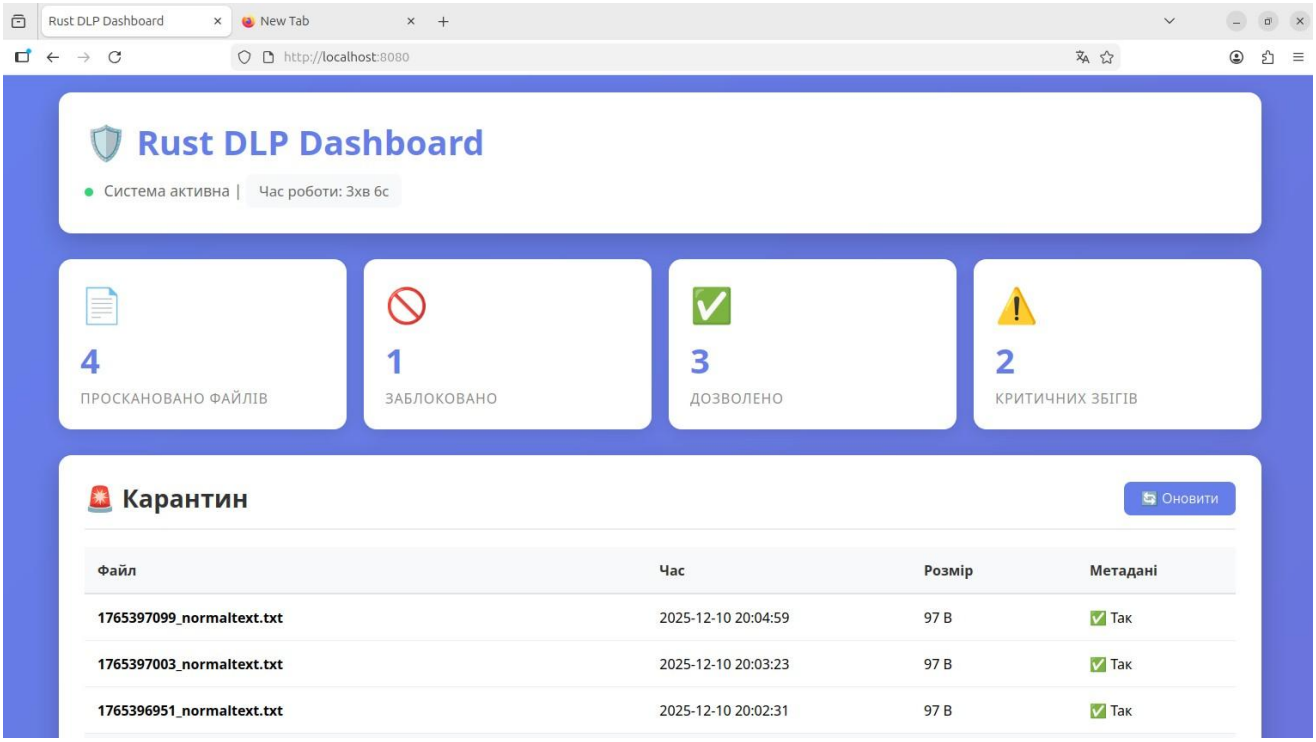

Рисунок 3.5 – Веб-інтерфейс модуля логування подій

Система тестування DLP утиліти побудована так, щоб тестування поведінки системи відбувалося без втручання в її внутрішню логіку. Тестування охоплює три критичні аспекти роботи системи захисту від витоку даних: точність виявлення конфіденційної інформації, продуктивність під час роботи та швидкість реакції на потенційні загрози. Кожен з цих аспектів вимагає окремої методології та інструментарію для об'єктивної оцінки.

Система тестування складається з п'яти незалежних модулів, кожен з яких відповідає за конкретний етап перевірки. Генератор тестових даних створює еталонний набір файлів з відомим змістом, що дозволяє об'єктивно оцінити роботу DLP утиліти. Модуль тестування точності порівнює рішення системи з еталонною розміткою та обчислює метрики якості класифікації. Модуль тестування продуктивності безперервно моніторить використання системних ресурсів процесом DLP. Модуль тестування швидкодії вимірює час реакції системи на файлові операції. Модуль візуалізації агрегує результати всіх тестів та створює графічні представлення для аналізу.

Тестування точності базується на методології supervised testing з наперед відомою розміткою даних. Спочатку генератор створює сто файлів трьох

категорій. Сорок файлів містять справжні конфіденційні дані, ці дані генеруються псевдовипадково з дотриманням реальних форматів, що робить їх невідрізнимими від справжніх документів. Ще сорок файлів містять звичайні дані без будь-якої конфіденційної інформації. Останні двадцять файлів представляють граничні випадки, які особливо складні для класифікації.

Граничні випадки включають файли з частковими збігами патернів, де присутні послідовності символів схожі на конфіденційні дані, але не є ними повністю. Наприклад, номер замовлення може виглядати як ідентифікаційний номер, також тестуються маскові дані, де частина інформації прихована зірочками, та контекстуально неоднозначні дані, такі як дата народження без супутньої персональної інформації.

Після створення тестового набору файли копіюються в директорію, яку моніторить DLP утиліта. Система сканує кожен файл та приймає рішення про наявність чутливих даних. Файли з виявленими конфіденційними даними переміщуються до карантину, а безпечні залишаються на місці. Модуль тестування точності аналізує ці рішення, порівнюючи їх з еталонною розміткою.

Результати класифікації представляються у вигляді матриці помилок, яка містить чотири категорії рішень. True Positive представляє ситуацію, коли файл справді містить конфіденційні дані і DLP правильно визначила його як загрозу, відправивши до карантину. True Negative означає, що звичайний файл без чутливих даних був правильно ідентифікований як безпечний і залишився доступним для роботи.

False Positive являє собою помилку першого роду, коли безпечний файл помилково класифікується як загроза. Це призводить до необґрунтованого блокування легітимних робочих документів, що створює незручності для користувачів та знижує продуктивність праці. False Negative є значно небезпечнішою помилкою другого роду, коли файл з конфіденційними даними не виявляється системою, що дозволяє витік інформації. На основі матриці помилок обчислюються ключові метрики якості класифікації.

Тестування продуктивності фокусується на вимірюванні впливу DLP утиліти на системні ресурси під час роботи. Система моніторингу використовує

бібліотеку psutil для доступу до низькорівневих системних показників через псевдофайлову систему proc у Linux. Моніторинг виконується з інтервалом півсекунди, що дозволяє виявити короткочасні піки навантаження, які можуть бути пропущені при більших інтервалах.

Використання процесора вимірюється як відсоток часу, який процес DLP споживає від загального процесорного часу. Система відстежує як користувацький час, так і системний час, витрачений на обробку системних викликів. Низьке споживання процесора критично важливе, оскільки DLP система має працювати постійно у фоновому режимі, не заважаючи основній роботі користувача. Значення нижче 10% вважається відмінним результатом, оскільки практично не впливає на продуктивність інших програм.

Використання оперативної пам'яті аналізується через кілька показників. Resident Set Size представляє фактичну фізичну пам'ять, яку процес займає в RAM, а Virtual Memory Size показує загальний обсяг віртуальної адресної простору. Для DLP системи більш важливим є RSS, оскільки він безпосередньо впливає на доступність пам'яті для інших програм. Ефективна DLP система має споживати менше 50Мб для забезпечення комфортної роботи навіть на системах з обмеженою пам'яттю.

Кількість потоків виконання показує, наскільки ефективно система використовує багатопотоковість для паралельної обробки файлів. Надмірна кількість потоків може призвести до overhead від перемикання контексту, тоді як недостатня кількість не використовує повну потужність багатоядерних процесорів. Моніторинг потоків також виявляє потенційні витоки ресурсів, коли потоки створюються але не звільняються після завершення роботи.

Всі зібрані дані зберігаються як часовий ряд та експортуються у CSV формат для подальшого аналізу. Для кожного показника обчислюються статистичні характеристики включаючи середнє значення, мінімум, максимум та стандартне відхилення. Середнє значення показує типове навантаження під час нормальної роботи. Максимум виявляє пікові навантаження при обробці складних файлів або масових операцій. Стандартне відхилення характеризує

стабільність системи, де низькі значення означають передбачувану та рівномірну роботу.

Тестування швидкодії вимірює час реакції DLP системи на потенційні загрози. Для систем захисту від витоку даних швидкість виявлення критично важлива.

Перший тест вимірює затримку виявлення при створенні нових файлів. Система генерує файл з конфіденційними даними та фіксує точний час створення. Потім починається очікування появи цього файлу в директорії карантину. Різниця між часом створення та часом карантину становить латентність виявлення. Тест повторюється десять разів для отримання статистично значущих результатів. Кожна ітерація використовує унікальне ім'я файлу з часовою міткою для запобігання кешуванню результатів.

Другий тест перевіряє швидкість реакції на модифікацію існуючих файлів. Спочатку створюється файл з безпечним вмістом, який проходить початкове сканування. Після паузи у дві секунди до файлу додаються конфіденційні дані. Система вимірює час від моменту модифікації до виявлення загрози. Цей сценарій імітує ситуацію, коли користувач додає чутливу інформацію до існуючого документу під час редагування.

Третій тест оцінює пропускну здатність системи при масовому створенні файлів. П'ятдесят файлів створюються одночасно, половина з яких містить конфіденційні дані. Вимірюється загальний час обробки всього пакету та обчислюється метрика throughput як кількість файлів на секунду. Цей тест виявляє здатність системи справлятися з піковими навантаженнями та ефективність черг обробки подій.

Для кожного типу тесту обчислюються статистичні характеристики розподілу затримок. Середнє значення показує очікуваний час відгуку в типових умовах. Медіана є більш стійкою до викидів та краще відображає типову продуктивність, особливо якщо траплялися аномальні затримки через зовнішні фактори системи. Мінімальне значення показує найкращий можливий час при оптимальних умовах. Максимальне значення виявляє найгірший випадок, що важливо для планування часових обмежень. Стандартне відхилення

характеризує передбачуваність системи, де низькі значення означають стабільну та надійну роботу.

Тестова система взаємодіє з DLP утилітою виключно через зовнішні інтерфейси без модифікації її коду. Файлова система слугує основним каналом комунікації, де тести створюють файли в моніторингових директоріях, а DLP система реагує на ці події через механізм inotify ядра Linux. Результати роботи DLP проявляються у вигляді переміщених файлів до карантину та створених JSON файлів з метаданими.

Кожен заблокований файл супроводжується детальними метаданими, що включають оригінальний шлях, час карантину, знайдені патерни та криптографічний хеш файлу. Ці метадані використовуються модулем тестування точності для визначення, які саме файли були виявлені системою. Метадані також дозволяють аналізувати, які конкретно патерни спрацювали, що цінно для налаштування правил виявлення.

Розроблена система тестування забезпечує повну відтворюваність результатів завдяки фіксованим вхідним даним та автоматизації всього процесу. Кожен запуск тестів використовує ідентичний набір файлів з еталонною розміткою, що дозволяє коректно порівнювати результати різних конфігурацій системи або різних версій DLP утиліти. Автоматизація виключає можливість людської помилки при виконанні тестів та забезпечує консистентність методології між різними запусками.

За результатами тестів, матриця помилок розкриває високу ефективність механізмів класифікації. З 40 тестових файлів, що містили конфіденційні дані, система правильно ідентифікувала 39, що становить повноту виявлення на рівні 97.5%. Це означає, що лише 2,5% потенційних загроз можуть пройти непоміченими, що є виключно високим показником захищеності. Результати зображені на рисунках 3.6 – 3.7.

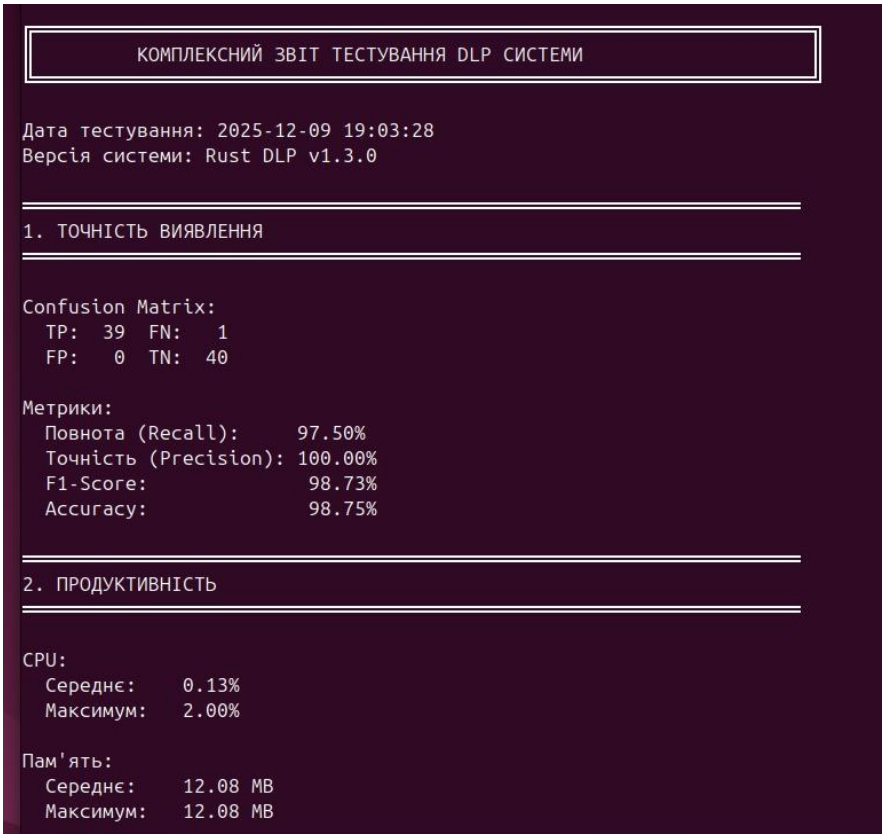

Рисунок 3.6 – Результати проведення тестів

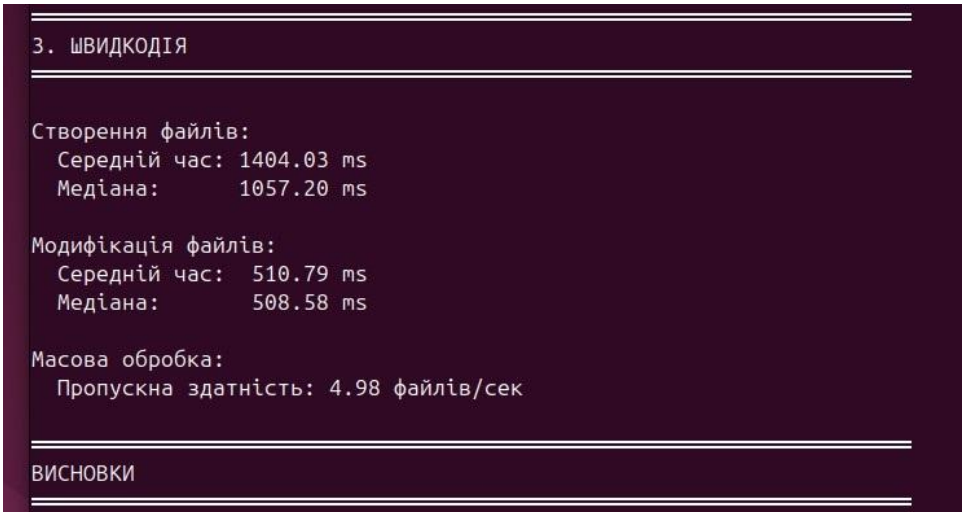

Рисунок 3.7 – Результати проведення тестів

Непоганим результатом є повна відсутність хибних спрацювань, що представлені категорією False Positive. Усі сорок файлів без конфіденційної інформації були правильно ідентифіковані як безпечні та залишилися доступними для роботи. Це критично важливо з точки зору практичного використання системи, оскільки хибні спрацювання створюють значні незручності для користувачів.

Точність класифікації на 100% означає, що кожне рішення системи про блокування файлу було обґрунтованим. Цей показник обчислюється як відношення правильно виявлених чутливих файлів до загальної кількості заблокованих файлів. У випадку розробленої системи всі 39 заблокованих файлів справді містили конфіденційну інформацію, що підтверджує високу якість використаних механізмів детекції та відсутність надмірної чутливості системи.

F1-Score як гармонічне середнє між точністю та повнотою демонструє виключно збалансовану роботу системи. Значення 98.73% вказує на те, що система однаково ефективна як у виявленні загроз, так і в уникненні хибних тривог. Цей показник особливо цінний для наукового аналізу, оскільки дозволяє порівнювати різні DLP системи або різні конфігурації однієї системи через єдину числову метрику. Отримане значення значно перевищує поріг у дев'яносто відсотків, який прийнятий у індустрії як критерій відмінної роботи.

Загальна точність системи на рівні 98.75% підтверджує високу якість класифікації через простіше обчислення відношення всіх правильних рішень до загальної кількості файлів. Сімдесят дев'ять правильних класифікацій з вісімдесяти можливих створюють міцну основу для довіри до системи як з боку адміністраторів безпеки, так і з боку кінцевих користувачів.

Тестування швидкодії виявило різні показники залежно від типу операції з файлами. Створення нових файлів з конфіденційними даними призводить до їх виявлення та карантину у середньому за 1404.03 ms, що становить приблизно півтори секунди. Цей результат потрапляє у категорію доброї швидкодії, хоча і не досягає критерію відмінної роботи у менше однієї секунди. Медіана часу реакції на рівні 1057.20 ms, що є нижчою за середнє значення, вказує на наявність декількох аномально повільних випадків, які підвищили середню затримку. Типова затримка фактично знаходиться ближче до одної секунди, що є більш репрезентативним показником реальної швидкодії системи.

Час реакції на модифікацію існуючих файлів виявився значно коротшим та склав 510.79 ms у середньому. Це майже втричі швидше за виявлення нових файлів та попадає у категорію відмінної швидкодії. Медіана на рівні 508.58 ms практично співпадає із середнім значенням, що свідчить про стабільність часу

обробки без значних викидів. Швидша обробка модифікацій може пояснюватися тим, що файл вже присутній у системі та його метадані можуть бути кешовані.

Тест масової обробки файлів показав пропускну здатність на рівні чотири цілих дев'яносто вісім сотих файлів за секунду. При одночасному створенні п'ятдесяти файлів система витратила приблизно десять секунд на їх повну обробку. Цей показник відображає здатність системи справлятися з піковими навантаженнями, коли користувач виконує масові операції копіювання або завантаження. Хоча значення близько п'яти файлів за секунду може здаватися не дуже високим, для DLP системи це є прийнятним результатом з огляду на те, що кожен файл проходить складну обробку.

Виключно низьке споживання процесора підтверджує ефективність використання механізму inotify замість періодичного сканування. Традиційні підходи до моніторингу файлів через polling вимагають регулярної перевірки стану файлової системи, що створює постійне навантаження на процесор навіть за відсутності змін. Модель дозволяє системі залишатися неактивною до моменту реальної зміни, споживаючи ресурси лише для обробки актуальних подій. Комбінація з асинхронним виконанням через tokio runtime забезпечує ефективне використання потоків без блокування на операціях вводу-виводу.

Стабільність споживання пам'яті свідчить про відсутність типових проблем довготривалої роботи програм, таких як витоки пам'яті або фрагментація. Система власності Rust на рівні компілятора гарантує, що кожен виділений блок пам'яті має чіткого власника та буде звільнений автоматично при виході змінної з області видимості.

Отже розроблена DLP система продемонструвала виключно високі результати у всіх трьох категоріях тестування. Інтегральний показник F1-Score на рівні 98.73% свідчить про готовність системи до промислової експлуатації. Комбінація високої точності виявлення загроз, мінімального впливу на продуктивність робочої станції та швидкої реакції на потенційні витоки створює збалансоване рішення для захисту конфіденційної інформації.

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Проектування та розробка утиліти для виявлення витоків даних вимагає постійної роботи за персональним комп’ютером, тому важливо правильно організувати робоче місце оператора ПК. Правильно оптимізоване робоче місце зменшує шкідливий вплив на здоров’я, підвищує комфорт та продуктивність операторів, які проводять значну кількість часу в роботі за комп’ютером.

Організація робочого місця користувача комп'ютера базується на вимогах ДСТУ 8604:2015, який регламентує відповідність технічних засобів та просторової організації робочої зони психофізіологічним і антропометричним характеристикам людини. Головною метою ергономічного проектування є створення комфортних умов праці, що забезпечують збереження здоров'я, стійку працездатність та ефективність виконання виробничих завдань. Робоче місце розглядається як ергатична система, в якій взаємне розташування елементів (стіл, крісло, дисплей, засоби введення) повинно відповідати розмірам тіла людини та характеру виконуваних рухів.

Конструкція робочого столу має забезпечувати можливість зручного розміщення обладнання та документів з урахуванням зон досяжності рук оператора. Оптимальна висота робочої поверхні зазвичай становить 725 мм, проте для забезпечення максимальної ергономічності рекомендовано використовувати столи з регульованою висотою в межах 680-800 мм. Важливою умовою є наявність достатнього простору для ніг під столом, який повинен мати висоту не менше 600 мм, ширину – не менше 500 мм та глибину на рівні колін – не менше 450 мм. Поверхня столу повинна бути матовою, щоб уникнути появи світлових відблисків, які створюють додаткове навантаження на зір [22].

Робоче крісло є ключовим елементом, що забезпечує фізіологічно раціональну робочу позу та запобігає розвитку захворювань опорно-рухового апарату. Згідно з ергономічними стандартами, крісло повинно бути підйомно-

поворотним, регульованим за висотою та кутами нахилу сидіння і спинки. Конструкція спинки має обов'язково мати вигин у поперековій зоні для підтримки хребта, що дозволяє зменшити статичне навантаження на м'язи спини. Поверхня сидіння та спинки повинна бути напівм'якою, з нековзним, повітропроникним покриттям, що легко піддається очищенню. Для зняття напруги з м'язів плечового поясу крісло має бути обладнане стаціонарними або регульованими підлокітниками.

Розміщення монітора (відеотермінала) здійснюється з урахуванням характеристик зорового аналізатора людини. Екран дисплея повинен знаходитися на відстані витягнутої руки, що становить приблизно 600–700 мм від очей користувача (але не ближче 500 мм). Верхній край активної області екрана рекомендується встановлювати на рівні очей або трохи нижче, щоб кут погляду на центр екрана становив 15–20 градусів униз від горизонтальної лінії. Таке розташування забезпечує природне положення шиї та мінімізує напруження очних м'язів. Також необхідно передбачити можливість регулювання кута нахилу екрана для усунення відблисків від джерел загального освітлення [22].

Клавіатура та маніпулятор «миша» повинні розташовуватися на робочій поверхні (або спеціальній висувній полиці) на рівні ліктів, щоб забезпечити кут у ліктьовому суглобі 90–100 градусів. Клавіатуру слід розміщувати на відстані 100–300 мм від краю столу, що дає змогу використовувати вільну зону для опори передпліч. Це дозволяє уникнути статичного напруження м'язів рук і плечей. У разі інтенсивної роботи з текстом рекомендується використання спеціальних підставок для зап'ясть, що допомагає утримувати кисті в рівному положенні та запобігає стисненню нервових закінчень у зап'ястному каналі [22].

Окремим важливим аспектом ергономічної організації робочого місця є розташування паперових носіїв інформації, які використовуються в процесі роботи. Часте переведення погляду з екрана монітора на документи, що лежать горизонтально на поверхні столу, викликає значне навантаження на шийний відділ хребта через необхідність постійних нахилів голови. Крім того, різниця у відстані до об'єктів (екран – 60–70 см, папір на столі – 30–40 см) змушує

кришталик ока постійно змінювати фокусну відстань, що прискорює втому зорового аналізатора.

Для усунення цього негативного фактору рекомендується використання спеціальних утримувачів документів (пюпітрів). Пюпітр повинен бути встановлений у вертикальній площині на одній висоті та на одній відстані з екраном дисплея. Таке розташування забезпечує перебування обох об'єктів у межах одного візуального поля, що мінімізує амплітуду рухів голови та очей, стабілізує акомодацію і підвищує продуктивність обробки інформації.

Дотримання вищезазначених вимог при організації робочого місця дозволяє сформувати оптимальну робочу зону, що суттєво знижує вплив негативних виробничих факторів на функціональний стан оператора та сприяє підвищенню ефективності праці.

4.2 Фактори, що впливають на функціональний стан користувачів комп'ютерів

Діяльність користувача комп'ютерних систем характеризується специфічними умовами праці, які формують складний комплекс навантажень на організм людини. Функціональний стан оператора залежить від сукупності ергономічних, фізичних та психофізіологічних чинників. Тривала дія цих факторів без належної регламентації режимів праці та відпочинку призводить до накопичення втоми, зниження продуктивності та підвищує ризик виникнення професійних захворювань. До основних груп шкідливих чинників відносять навантаження на зоровий аналізатор, напруження опорно-рухового апарату, нервово-емоційне навантаження, а також фізичні фактори виробничого середовища.

Зорове напруження є домінуючим фактором, що визначає функціональний стан користувача ПК, оскільки сприйняття інформації з екрана дисплея принципово відрізняється від читання з паперових носіїв. Зображення на моніторі є самосвітним, дискретним (складається з пікселів) та може мати певний рівень мерехтіння, що створює додаткове навантаження на очі.

Специфіка роботи вимагає від користувача тривалої фіксації погляду на близькій відстані з частим переведенням фокусу між екраном, клавіатурою та паперовими документами. Це викликає інтенсивну роботу м'язів ока, що відповідають за акомодацію та конвергенцію. Наслідком тривалого зорового напруження є розвиток так званого «комп'ютерного зорового синдрому» (Computer Vision Syndrome), симптомами якого є печіння в очах, тимчасове погіршення гостроти зору, почервоніння склер та головний біль [23].

Робота за комп’ютером класифікується як праця, пов'язана з вимушеною робочою позою та локальними м'язовими навантаженнями. Тривале перебування у положенні сидячи призводить до гіпокінезії та статичного напруження м'язів шиї, плечового поясу і спини. Це, у свою чергу, спричиняє погіршення кровообігу, застійні явища в органах малого тазу та порушення трофіки міжхребцевих дисків.

Окрім статичного навантаження, робота з пристроями введення (клавіатурою та мишею) вимагає виконання великої кількості стереотипних дрібних рухів кистями рук. Постійна напруга м'язів передпліччя без періодичного розслаблення суттєво підвищує ризик виникнення тунельного синдрому зап'ястка та інших захворювань суглобово-зв'язкового апарату верхніх кінцівок.

Інтелектуальна діяльність користувача пов'язана з обробкою значних обсягів інформації, що вимагає високої концентрації уваги та активізації процесів пам'яті. Особливістю такої праці є поєднання монотонності виконуваних операцій з високим рівнем сенсорного навантаження, що призводить до швидкого розвитку нервово-емоційної втоми. Додатковими стресогенними факторами виступають необхідність прийняття рішень в умовах дефіциту часу, очікування реакції програмного забезпечення, а також відповідальність за збереження даних. Такий режим роботи може викликати передчасне виснаження нервової системи та зниження загальної резистентності організму [23].

Комп'ютерна техніка виступає джерелом фізичних полів, які змінюють параметри виробничого середовища. Системні блоки та периферійне обладнання

генерують складний спектр електромагнітних випромінювань низької та наднизької частоти. Хоча сучасні дисплеї мають високий ступінь захисту, сумарний вплив офісної техніки створює специфічний електромагнітний фон.

Крім того, наявність електростатичного поля на поверхнях обладнання призводить до деіонізації повітря та зміни його іонного складу (зниження кількості легких аероіонів), що негативно впливає на дихальну систему та імунітет. Додатковим несприятливим чинником є постійний монотонний акустичний шум від систем охолодження комп'ютерів, який, навіть при невисоких рівнях, сприяє швидшому настанню втоми.

Проведений аналіз демонструє, що робота користувача ПК супроводжується комплексним негативним впливом на зорову, опорно-рухову та нервову системи. Забезпечення стабільного функціонального стану оператора можливе лише за умови дотримання ергономічних вимог до організації робочого місця та впровадження раціональних режимів праці й відпочинку.

ВИСНОВКИ

У кваліфікаційній роботі було проведено комплексне дослідження проблеми витоку чутливих даних в операційних системах сімейства Linux, а також здійснено аналіз методів їх виявлення та попередження. Метою дослідження було підвищення ефективності захисту конфіденційної інформації в автоматизованих системах на базі Linux шляхом удосконалення методів виявлення витоків даних та розробки програмного засобу, здатного ефективно моніторити файлову активність та ідентифікувати загрози в режимі реального часу. Для досягнення цієї мети було детально проаналізовано життєвий цикл даних, класифіковано специфічні для Linux вектори загроз, такі як інсайдерська діяльність та використання легітимних адміністративних інструментів для ексфільтрації інформації.

На основі теоретичного аналізу було вибрано технологічний стек, що включає мову системного програмування Rust та бібліотеку notify для взаємодії з механізмом ядра inotify. В рамках практичної частини було спроектовано та реалізовано архітектуру DLP-агента, яка базується на асинхронній моделі та використовує гібридний підхід до аналізу контенту.

Проведені експериментальні дослідження у середовищі підтвердили працездатність та ефективність запропонованого рішення. Результати тестування продемонстрували високу точність виявлення чутливої інформації при збереженні низького рівня хибних спрацювань.

Отримані результати мають важливе практичне значення для системних адміністраторів та фахівців з інформаційної безпеки, надаючи їм дієвий інструмент для аудиту та захисту кінцевих точок. Розроблений підхід дозволяє не лише фіксувати факти порушення політик безпеки, але й виявляти підготовку до витоку даних на ранніх стадіях. Подальший розвиток системи вбачається у переході до проактивних методів блокування загроз та інтеграції з моделями машинного навчання для покращення контекстного аналізу даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. General Data Protection Regulation (GDPR). Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data. Official Journal of the European Union. 2016. Vol. L119. P. 1–88.

2. NIST SP 800-37 Rev. 2. Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy. National Institute of Standards and Technology. 2018. URL: https://csrc.nist.gov/publications/detail/sp/800-37/rev-2/final (дата звернення: 10.11.2025).

3. CISA Insider Threat Mitigation Guide. Cybersecurity and Infrastructure Security Agency. 2020. URL: https://www.cisa.gov/publication/insider-threat-mitigation-guide (дата звернення: 10.11.2025).

4. Cost of a Data Breach Report 2024. IBM Security. 2024. 62 p. URL: https://www.ibm.com/reports/data-breach (дата звернення: 10.11.2025).

5. Gartner Market Guide for Data Loss Prevention. Gartner, Inc. 2023. URL: https://www.gartner.com/en/documents/4005676 (дата звернення: 10.11.2025).

6. MITRE ATT&CK® for Enterprise. Exfiltration Techniques (TA0010). URL: https://attack.mitre.org/tactics/TA0010/ (дата звернення: 10.11.2025).

7. GTFOBins. Curated list of Unix binaries that can be used to bypass local security restrictions. URL: https://gtfobins.github.io/ (дата звернення: 10.11.2025).

8. CoSoSys Endpoint Protector. Data Loss Prevention for Linux. CoSoSys. URL: https://www.endpointprotector.com/solutions/data-loss-prevention-DLP-for-Linux (дата звернення: 10.11.2025).

9. Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital Investigation. 2006. Vol. 3. P. 91–97.

10. Oliver J., Cheng C., Chen Y. TLSH – A Locality Sensitive Hash. 4th Cybercrime and Trustworthy Computing Workshop. Sydney, 2013. (дата звернення: 14.11.2025).

11. Microsoft Presidio. PII Detection and Anonymization. Microsoft Documentation. URL: https://microsoft.github.io/presidio/ (дата звернення: 14.11.2025).

12.Klabnik S., Nichols C. The Rust Programming Language. 2nd ed. San Francisco: No Starch Press, 2023. 560 p.

13. Blandy J., Orendorff J., Tindall L. Programming Rust: Fast, Safe Systems Development. 2nd ed. O'Reilly Media, 2021. 734 p.

14. Notify Crate Documentation. Cross-platform filesystem notification library for Rust. Docs.rs. URL: https://docs.rs/notify/latest/notify/ (дата звернення: 23.11.2025).

15. Turon A. Fearless Concurrency with Rust. Rust Blog. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html (дата звернення: 23.11.2025).

16. Kerrisk M. The Linux Programming Interface: A Linux and UNIX System Programmer's Handbook. San Francisco: No Starch Press, 2010. 1552 p.

17. Powers D. M. W. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. Journal of Machine Learning Technologies. 2011. Vol. 2, no. 1. P. 37–63.

18. King C. I. stress-ng: A tool to load and stress a computer system. Ubuntu Manpages. URL: https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html (дата звернення: 27.11.2025).

19. Nethercote N. The Rust Performance Book. 2024. URL: https://nnethercote.github.io/perf-book/ (дата звернення: 27.11.2025).

20. Microsoft Presidio. Context aware, pluggable and customizable PII anonymization service. GitHub Repository. URL: https://github.com/microsoft/presidio (дата звернення: 27.11.2025).

21. Інжиніринг систем. Вимоги до ергономіки та технічної естетики : ДСТУ 8604:2015. [Чинний від 2016-01-01]. Київ : ДП «УкрНДНЦ», 2016. 26 с.

22. Желібо Є. П., Заверуха Н. М., Зацарний В. В. Безпека життєдіяльності : навчальний посібник / за ред. Є. П. Желібо. — 6-те вид. — Київ : Каравела, 2009. — 344 с.

23. Derkach, M., Matiuk, D., Skarga-Bandurova, I., & Zagorodna, N. (2025). CrypticWave: A zero-persistence ephemeral messaging system with client-side encryption.

24. Lypa B., Horyn I., Zagorodna N., Tymoshchuk D., Lechachenko T. Comparison of feature extraction tools for network traffic data. CEUR Workshop Proceedings. 2024. vol. 3896. P. 1-11.

25. Микитишин А. Г., Митник М. М., Стухляк П. Д. Телекомунікаційні системи та мережі. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2017. 384 с.

26. Nedzelskyi, D., Derkach, M., Tatarchenko, Y., Safonova, S., Shumova, L., & Kardashuk, V. (2019, August). Research of efficiency of multi-core computers with shared memory. In 2019 7th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW) (pp. 111-114). IEEE.

27. Muzh, V., & Lechachenko, T. (2024). Computer technologies as an object and source of forensic knowledge: challenges and prospects of development. Вісник Тернопільського національного технічного університету, 115(3), 17-22.

28. Skarga-Bandurova, I., Kotsiuba, I., & Velasco, E. R. (2021). Cyber Hygiene Maturity Assessment Framework for Smart Grid Scenarios. Front. Comput. Sci. 3: 614337. doi: 10.3389/fcomp.

29. Деркач М. В., Хомишин В. Г., Гудзенко В. О. Тестування безпеки вебресурсу на базі інструментів для сканування та виявлення вразливостей. Наукові вісті Далівського університету. 2023. №25.

30. Zagorodna, N., Skorenkyy, Y., Kunanets, N., Baran, I., & Stadnyk, M. (2022). Augmented Reality Enhanced Learning Tools Development for Cybersecurity Major. In ITTAP (pp. 25-32).

31. Revniuk, O., Zagorodna, N., Kozak, R., & Yavorskyy, B. (2025). Development of an information system for the quantitative assessment of web application security based on the OWASP ASVS standard. Вісник Тернопільського національного технічного університету, 118(2), 56-65.

ДОДАТОК А Публікація

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

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

УДК 004.056.53
О. Шарик; Р. Козак, к. т. н., доц.
(Тернопільський національний технічний університет імені Івана Пулюя)

МЕТОДИ ТА ЗАСОБИ ВИЯВЛЕННЯ ВИТОКІВ ДАНИХ У СЕРЕДОВИЩІ ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX

UDC 004.056.53
O. Sharyk; R. Kozak, PhD., Assoc. Prof.

METHODS AND TOOLS FOR DATA LEAK DETECTION IN THE LINUX OPERATING SYSTEM ENVIRONMENT

Стрімка цифровізація бізнес-процесів та масова міграція критичної інфраструктури у хмарні середовища актуалізують проблематику захисту конфіденційної інформації в операційних системах сімейства Linux. На відміну від традиційних загроз, пов'язаних із втратою даних внаслідок технічних збоїв, проблема витоку інформації характеризується порушенням конфіденційності через дії інсайдерів або некоректну конфігурацію прав доступу [1]. Існуючі комерційні рішення класу DLP (Data Loss Prevention) часто орієнтовані на екосистему Windows або ж демонструють надмірне споживання системних ресурсів. У рамках кваліфікаційної роботи проведено дослідження методів виявлення аномальної файлової активності та розроблено спеціалізований програмний засіб моніторингу, що поєднує високу продуктивність із точністю детекції.

У ході роботи було проаналізовано вектори загроз у середовищі Linux та встановлено, що ефективний захист вимагає комбінації сигнатурних та евристичних методів аналізу. Для реалізації програмного агента обрано мову системного програмування Rust, яка забезпечує гарантії безпеки пам'яті на етапі компіляції без використання збирача сміття, що дозволяє уникнути критичних вразливостей та забезпечити передбачувану продуктивність. Архітектура розробленого рішення базується на використанні підсистеми ядра Linux inotify через бібліотеку notify для перехоплення подій файлової системи в режимі реального часу. Для забезпечення високої пропускної здатності та обробки пікових навантажень, відомих як «шторми подій», застосовано асинхронне середовище виконання Tokio та патерн проектування Producer-Consumer, що дозволяє ефективно розпаралелити процеси моніторингу та глибокого аналізу контенту [2, 3].

Розроблений програмний засіб реалізує гібридний підхід до ідентифікації чутливих даних. Для виявлення персональної інформації (PII), такої як номери банківських карток або паспортні дані, застосовуються оптимізовані регулярні вирази з алгоритмічною валідацією, що мінімізує кількість хибних спрацювань. Експериментальні дослідження, проведені у віртуалізованому середовищі Ubuntu Linux, підтвердили, що запропонований підхід дозволяє досягти високих показників точності виявлення при мінімальному навантаженні на центральний процесор та оперативну пам'ять. Отримані результати свідчать про перспективність використання мови Rust для створення безпечних системних утиліт та можуть бути використані для підвищення рівня захищеності інформаційних систем підприємств.

Література

1. Cost of a Data Breach Report 2024. IBM Security. URL: https://www.ibm.com/reports/data-breach (дата звернення: 08.12.2024).
2. Klabnik S., Nichols C. The Rust Programming Language. San Francisco: No Starch Press, 2023. 560 p.
3. Love R. Linux Kernel Development. 3rd ed. Addison-Wesley Professional, 2010. 480 p.

ДОДАТОК Б Код реалізованої DLP утиліти

```
use anyhow::{Context, Result};
use serde::Deserialize;
use std::collections::HashMap;
use std::fs;
#[derive(Debug, Deserialize)]
pub struct Config {
    pub monitoring: MonitoringConfig,
    pub quarantine: QuarantineConfig,
    pub patterns: HashMap<String, PatternConfig>,
    pub usb: Option<UsbConfig>,
    pub web: Option<WebConfig>,
    pub document_classes: Option<Vec<DocumentClassConfig>>,
}
#[derive(Debug, Deserialize)]
pub struct MonitoringConfig {
    pub watch_paths: Vec<String>,
    pub safe_zone_path: Option<String>,
    pub max_file_size: u64,
    pub scannable_extensions: Vec<String>,}
#[derive(Debug, Deserialize)]
pub struct QuarantineConfig {
    pub path: String,
}
#[derive(Debug, Deserialize, Clone)]
pub struct PatternConfig {
    pub name: String,
    pub regex: String,
    pub severity: String,}
#[derive(Debug, Deserialize)]
pub struct UsbConfig {
    pub block_write: bool,}
#[derive(Debug, Deserialize)]
pub struct WebConfig {
    pub block_file_sharing: bool,}
#[derive(Debug, Deserialize, Clone)]
pub struct DocumentClassConfig {
    pub name: String,
    pub keywords: Vec<String>,
    pub min_matches: usize,
    pub severity: String,
}
impl Config {
    pub fn load(path: &str) -> Result<Self> {
        let content = fs::read_to_string(path)

            .with_context(|| format!("Failed to read {}", path))?;
        toml::from_str(&content).context("Failed to parse
config.toml")
    }
}
use serde::Deserialize;
```

```
#[derive(Debug, Clone, Deserialize)]
pub struct DocumentClass {
    pub name: String,
    pub keywords: Vec<String>,
    pub min_matches: usize, // Мінімальна кількість ключових слів
для спрацювання
    pub severity: String,
}

pub struct DocumentClassifier {
    classes: Vec<DocumentClass>,
}

impl DocumentClassifier {
    pub fn new(classes: Vec<DocumentClass>) -> Self {
        Self { classes }
    }

    pub fn classify(&self, content: &str) -> Vec<DocumentMatch> {
        let content_lower = content.to_lowercase();
        let mut matches = Vec::new();

        for class in &self.classes {
            let mut matched_keywords = Vec::new();

            // Шукаємо всі ключові слова в тексті
            for keyword in &class.keywords {
                let keyword_lower = keyword.to_lowercase();
                if content_lower.contains(&keyword_lower) {
                    matched_keywords.push(keyword.clone());
                }
            }

            // Якщо знайдено достатньо ключових слів - документ
класифіковано
            if matched_keywords.len() >= class.min_matches {
                matches.push(DocumentMatch {
                    class_name: class.name.clone(),
                    matched_keywords,
                    total_keywords: class.keywords.len(),
                    severity: class.severity.clone(),
                });
            }
        }

        matches
    }
}

#[derive(Debug, Clone)]
pub struct DocumentMatch {
    pub class_name: String,
    pub matched_keywords: Vec<String>,
```

```
    pub total_keywords: usize,
    pub severity: String,
}

impl DocumentMatch {
    pub fn confidence(&self) -> f32 {
        (self.matched_keywords.len() as f32) /
(self.total_keywords as f32) * 100.0
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_tender_protocol_detection() {
        let class = DocumentClass {
            name: "Протокол тендерного комітету".to_string(),
            keywords: vec![
                "протокол".to_string(),
                "тендерний комітет".to_string(),
                "закупівля".to_string(),
                "учасник".to_string(),
            ],
            min_matches: 2,
            severity: "Critical".to_string(),
        };

        let classifier = DocumentClassifier::new(vec![class]);

        let content = "Протокол засідання тендерного комітету від
23.11.2024. Розглянуто пропозиції учасників.";
        let matches = classifier.classify(content);

        assert_eq!(matches.len(), 1);
        assert_eq!(matches[0].class_name, "Протокол тендерного
комітету");
        assert!(matches[0].matched_keywords.len() >= 2);
    }
}
use crate::config::Config;
use crate::scanner::ContentScanner;
use crate::types::{Match, MatchInfo, QuarantineEvent, Severity,
Statistics};
use anyhow::{Context, Result};
use notify::{Event, EventKind, RecommendedWatcher, RecursiveMode,
Watcher};
use sha2::{Digest, Sha256};
use std::collections::HashSet;

use std::fs;
use std::path::{Path, PathBuf};
```

```
use std::sync::{Arc, Mutex};
use std::time::{Duration, SystemTime};
use walkdir::WalkDir;

pub struct EndpointAgent {
    scanner: Arc<ContentScanner>,
    watched_paths: Vec<PathBuf>,
    safe_zone: Option<PathBuf>,
    quarantine_dir: PathBuf,
    scanned_files: Arc<Mutex<HashSet<PathBuf>>>,
    statistics: Arc<Mutex<Statistics>>,
    max_file_size: u64,
    scannable_extensions: Vec<String>,
    start_time: SystemTime,
}

impl EndpointAgent {
    pub fn from_config(config: &Config, scanner:
Arc<ContentScanner>) -> Result<Self> {
        let quarantine_dir =
PathBuf::from(&config.quarantine.path);
        fs::create_dir_all(&quarantine_dir)?;

        let watched_paths: Vec<PathBuf> =
config.monitoring.watch_paths
            .iter()
            .map(PathBuf::from)
            .collect();

        let safe_zone = config.monitoring.safe_zone_path
            .as_ref()
            .map(|p| {
                let path = PathBuf::from(p);
                fs::create_dir_all(&path).ok();
                path
            });

        log::info!("📂 Директорії для моніторингу:");
        for path in &watched_paths {
            if path.exists() {
                log::info!("   ✓ {}", path.display());
            } else {
                log::warn!("   ⚠ {} (не існує)",
path.display());
            }
        }

        if let Some(ref sz) = safe_zone {
            log::info!("🔒 Безпечна зона: {}", sz.display());
            log::info!("   Файли тут не блокуються, але не можна
копіювати/переміщати");
        }
```

```
        Ok(Self {
            scanner,
            watched_paths,
            safe_zone,
            quarantine_dir,
            scanned_files: Arc::new(Mutex::new(HashSet::new())),
            statistics: Arc::new(Mutex::new(Statistics::new())),
            max_file_size: config.monitoring.max_file_size,
            scannable_extensions:
config.monitoring.scannable_extensions.clone(),
            start_time: SystemTime::now(),
        })
    }

    pub fn start(self: Arc<Self>) -> Result<()> {
        log::info!("🔍 Початкове сканування...");
        self.scan_existing_files()?;

        let agent_clone = Arc::clone(&self);
        std::thread::spawn(move || {
            if let Err(e) = agent_clone.start_file_watcher() {
                log::error!("❌ Watcher error: {}", e);
            }
        });

        let agent_clone = Arc::clone(&self);
        std::thread::spawn(move || {
            agent_clone.statistics_reporter();
        });

        log::info!("✅ DLP активний");
        Ok(())
    }

    fn statistics_reporter(&self) {
        loop {
            std::thread::sleep(Duration::from_secs(300));
            let stats = self.statistics.lock().unwrap();
            let uptime = SystemTime::now()
                .duration_since(self.start_time)
                .unwrap()
                .as_secs();
            log::info!("📊 Статистика: ⏱️ {}хв | 📄 {} | 🚫 {} | ✅
{}",
                uptime / 60, stats.files_scanned,
stats.files_blocked, stats.files_allowed);
        }
    }
    fn scan_existing_files(&self) -> Result<()> {
        let mut scanned = 0;
        for watch_path in &self.watched_paths {
```

```
            if !watch_path.exists() { continue; }
```



```
            for entry in
WalkDir::new(watch_path).max_depth(3).into_iter().filter_map(|e|
e.ok()) {
                let path = entry.path();
                if path.is_file() && self.should_scan_file(path) {
                    scanned += 1;
                    let _ = self.scan_file(path, false);
                }
            }
        }
        log::info!("✅ Просканировано {} файлів", scanned);
        Ok(())
    }

    fn start_file_watcher(&self) -> Result<()> {
        let (tx, rx) = std::sync::mpsc::channel();
        let mut watcher: RecommendedWatcher = Watcher::new(
            tx,

notify::Config::default().with_poll_interval(Duration::from_secs(1
)),
        )?;

        for path in &self.watched_paths {
            if path.exists() {
                watcher.watch(path, RecursiveMode::Recursive)
                    .context(format!("Failed to watch {:?}",
path))?;
            }
        }

        loop {
            match rx.recv() {
                Ok(Ok(event)) => self.handle_fs_event(event),
                Ok(Err(e)) => log::error!("❌ Watcher error: {}",
e),
                Err(e) => {
                    log::error!("❌ Channel error: {}", e);
                    break;
                }
            }
        }
        Ok(())
    }

    fn handle_fs_event(&self, event: Event) {
        match event.kind {
            EventKind::Create(_) => {
                for path in event.paths {
                    if path.is_file() &&
self.should_scan_file(&path) {
```

```
std::thread::sleep(Duration::from_millis(500));
                        let _ = self.scan_file(&path, true);
                    }
                }
            }
            // Відстежуємо переміщення файлів

EventKind::Modify(notify::event::ModifyKind::Name(rename_mode)) =>
{
                use notify::event::RenameMode;
                match rename_mode {
                    RenameMode::Both => {
                        if event.paths.len() >= 2 {
                            let from = &event.paths[0];
                            let to = &event.paths[1];

                            // Перевіряємо SafeZone violation
                            if let Some(ref safe_zone) =
self.safe_zone {
                                if from.starts_with(safe_zone) &&
!to.starts_with(safe_zone) {

self.check_safezone_violation(from, to);
                                    return;
                                }
                            }

                            // Звичайне переміщення - сканумо
                            if to.is_file() &&
self.should_scan_file(to) {

std::thread::sleep(Duration::from_millis(500));
                                let _ = self.scan_file(to, true);
                            }
                        }
                    }
                    RenameMode::To => {
                        for path in &event.paths {
                            if path.is_file() &&
self.should_scan_file(path) {

std::thread::sleep(Duration::from_millis(500));
                                let _ = self.scan_file(path,
true);
                            }
                        }
                    }
                    _ => {}
                }
            }
            EventKind::Modify(_) => {
                for path in event.paths {
```

```
                    if path.is_file() &&
self.should_scan_file(&path) {

std::thread::sleep(Duration::from_millis(500));
                        let _ = self.scan_file(&path, true);
                    }
                }
            }
            _ => {}
        }
    }

    fn check_safezone_violation(&self, from: &Path, to: &Path) {
        if let Some(ref safe_zone) = self.safe_zone {
            if from.starts_with(safe_zone) &&
!to.starts_with(safe_zone) {
                log::error!("⛔ БЛОКУЮ! Спроба переміщення з
SafeZone");
                log::error!("   З: {}", from.display());
                log::error!("   В: {}", to.display());

                std::thread::sleep(Duration::from_millis(200));

                if to.exists() && to.is_file() {
                    match fs::read_to_string(to) {
                        Ok(content) => {
                            let matches =
self.scanner.scan(&content);

                            if !matches.is_empty() {
                                log::error!("🔍 ЗНАЙДЕНО ЧУТЛИВІ
ДАНІ У ФАЙЛІ:");
                                for m in &matches {
                                    log::error!("   - {}: {}
[{}]",
                                        m.pattern_name,
m.matched_text, m.severity);
                                }
                            } else {
                                log::warn!("⚠️  Чутливих даних не
знайдено, але переміщення все одно заборонено");
                            }
                        }
                        Err(e) => {
                            log::error!("❌ Не вдалося прочитати
файл: {}", e);
                        }
                    }

                    // Повертаємо файл назад
```

```
                    if let Err(e) = self.return_to_safezone(to,
from) {
```

Продовження додатку Б

```
                        log::error!("✗ Не вдалося повернути файл:
{}", e);
                    }
                }
            }
        }
    }

    fn return_to_safezone(&self, current_path: &Path,
original_path: &Path) -> Result<()> {
        log::info!("↩  Повертаю файл назад в SafeZone");
        fs::rename(current_path, original_path)?;
        log::info!("✓ Файл повернено: {}",
original_path.display());
        Ok(())
    }

    fn is_in_safe_zone(&self, path: &Path) -> bool {
        if let Some(ref safe_zone) = self.safe_zone {
            path.starts_with(safe_zone)
        } else {
            false
        }
    }

    fn should_scan_file(&self, path: &Path) -> bool {
        if path.starts_with(&self.quarantine_dir) {
            return false;
        }

        let extension = path.extension()
            .and_then(|e| e.to_str())
            .unwrap_or("")
            .to_lowercase();

        if !self.scannable_extensions.contains(&extension) {
            return false;
        }

        if let Ok(metadata) = fs::metadata(path) {
            if metadata.len() > self.max_file_size {
                return false;
            }
        }

        true
    }
    fn scan_file(&self, path: &Path, is_new_file: bool) ->
Result<()> {
        {
```

```
            let mut scanned = self.scanned_files.lock().unwrap();
            if scanned.contains(path) && !is_new_file {
```

Продовження додатку Б

```
                return Ok(());
            }
            scanned.insert(path.to_path_buf());
        }

        if is_new_file {
            log::info!("🔍 {}", path.display());
        }

        {
            let mut stats = self.statistics.lock().unwrap();
            stats.file_scanned();
        }

        let content = match fs::read_to_string(path) {
            Ok(c) => c,
            Err(_) => return Ok(()),
        };

        if content.len() < 10 {
            return Ok(());
        }

        let matches = self.scanner.scan(&content);

        if matches.is_empty() {
            let mut stats = self.statistics.lock().unwrap();
            stats.file_allowed();
            return Ok(());
        }

        let in_safezone = self.is_in_safe_zone(path);

        if in_safezone {
            log::warn!("⚠️  [SafeZone] {} - виявлено {}",
                path.file_name().unwrap().to_string_lossy(),
matches.len());
        } else {
            log::warn!("⚠️  {} - виявлено {}",
                path.file_name().unwrap().to_string_lossy(),
matches.len());
        }

        for m in &matches {
            log::warn!("   - {}: {} [{}]", m.pattern_name,
m.matched_text, m.severity);
        }
        {
            let mut stats = self.statistics.lock().unwrap();
            stats.matches_found(matches.len());
```

```
        }

        // Якщо файл в SafeZone - НЕ блокувати
        if in_safezone {
            log::info!("🔒 Файл в SafeZone - дозволено працювати
локально");
            return Ok(());
        }

        // Файл НЕ в SafeZone і має критичні дані - блокуємо
        if self.scanner.has_critical_matches(&matches) {
            log::error!("⛔ БЛОКУЮ! Критичні дані");
            let critical_count = matches.iter()
                .filter(|m| matches!(m.severity,
Severity::Critical | Severity::High))
                .count();
            {
                let mut stats = self.statistics.lock().unwrap();
                stats.file_blocked(critical_count);
            }
            self.quarantine_file(path, &matches)?;
        }

        Ok(())
    }

    fn quarantine_file(&self, path: &Path, matches: &[Match]) ->
Result<()> {
        let file_name = path.file_name()
            .ok_or_else(|| anyhow::anyhow!("No filename"))?;

        let file_size = fs::metadata(path)?.len();
        let file_hash = Self::calculate_file_hash(path)?;

        let timestamp = SystemTime::now()
            .duration_since(SystemTime::UNIX_EPOCH)?
            .as_secs();

        let quarantine_name = format!("{}_{}", timestamp,
file_name.to_string_lossy());
        let quarantine_path =
self.quarantine_dir.join(&quarantine_name);

        fs::rename(path, &quarantine_path)?;

        log::error!("🔐 Карантин: {}", quarantine_path.display());
        let placeholder_message = "The file has been quarantined
because it contains confidential information. Please contact your
administrator.";
        fs::write(path, placeholder_message)?;
        log::info!("📝 Створено файл-заповнювач: {}",
path.display());
```

```
        let metadata_path =
quarantine_path.with_extension("json");
        let event = QuarantineEvent {
```

Продовження додатку Б

```
            original_path: path.to_string_lossy().to_string(),
            quarantine_path:
quarantine_path.to_string_lossy().to_string(),
            quarantine_time: chrono::Local::now().to_rfc3339(),
            matches: matches.iter().map(|m| MatchInfo {
                pattern: m.pattern_name.clone(),
                severity: format!("{}", m.severity),
                text: m.matched_text.clone(),
            }).collect(),
            file_size,
            file_hash,
        };

        fs::write(metadata_path,
serde_json::to_string_pretty(&event)?)?;
        Ok(())
    }

    fn calculate_file_hash(path: &Path) -> Result<String> {
        let content = fs::read(path)?;
        let mut hasher = Sha256::new();
        hasher.update(&content);
        Ok(format!("{:x}", hasher.finalize()))
    }

    pub fn get_statistics(&self) -> Statistics {
        let stats = self.statistics.lock().unwrap();
        let mut stats_copy = Statistics {
            files_scanned: stats.files_scanned,
            files_blocked: stats.files_blocked,
            files_allowed: stats.files_allowed,
            matches_found: stats.matches_found,
            critical_matches: stats.critical_matches,
            uptime_seconds: 0,
        };
        stats_copy.uptime_seconds = SystemTime::now()
            .duration_since(self.start_time)
            .unwrap()
            .as_secs();
        stats_copy
    }
}

mod config;
mod endpoint;
mod scanner;
mod types;
mod usb_blocker;
mod web_blocker;
mod document_classifier;
```

```
mod web_dashboard;

use anyhow::Result;
```

Продовження додатку Б

```
use config::Config;
use endpoint::EndpointAgent;
use scanner::ContentScanner;
use usb_blocker::UsbBlocker;
use web_blocker::WebBlocker;
use web_dashboard::start_dashboard;
use std::path::PathBuf;
use std::sync::Arc;

#[tokio::main]
async fn main() -> Result<()> {

env_logger::Builder::from_env(env_logger::Env::default().default_f
ilter_or("info")).init();

    println!("🛡️  Rust DLP v1.3.0\n");

    let config = Config::load("config.toml")?;
    println!("✅ Конфіг завантажено");

    // Ініціалізуємо основні компоненти
    let scanner = Arc::new(ContentScanner::from_config(&config)?);
    let endpoint = Arc::new(EndpointAgent::from_config(&config,
Arc::clone(&scanner))?);

    // USB блокування
    let _usb_blocker = if let Some(usb_config) = &config.usb {
        if usb_config.block_write {
            let usb_blocker = Arc::new(UsbBlocker::new());
            let usb_clone = Arc::clone(&usb_blocker);
            usb_clone.start()?;
            Some(usb_blocker)
        } else {
            None
        }
    } else {
        None
    };

    // Web блокування
    let _web_blocker = if let Some(web_config) = &config.web {
        if web_config.block_file_sharing {
            let web_blocker = Arc::new(WebBlocker::new());
            let web_clone = Arc::clone(&web_blocker);
            web_clone.start()?;
            Some(web_blocker)
        } else {
            None
        }
    } else {
```

```
        None
    };
```

Продовження додатку Б

```
    // Запускаємо основний DLP агент
    let endpoint_clone = Arc::clone(&endpoint);
    endpoint_clone.start()?;

    // Запускаємо Web Dashboard
    let quarantine_dir = PathBuf::from(&config.quarantine.path);
    let dashboard_endpoint = Arc::clone(&endpoint);

    tokio::spawn(async move {
        if let Err(e) = start_dashboard(dashboard_endpoint,
quarantine_dir, 8080).await {
            log::error!("❌ Dashboard error: {}", e);
        }
    });

    // Головний цикл
    loop {

tokio::time::sleep(tokio::time::Duration::from_secs(60)).await;
        let stats = endpoint.get_statistics();
        if stats.files_scanned > 0 {
            log::info!("📊 {} файлів, {} блок, {} ок",
                stats.files_scanned, stats.files_blocked,
stats.files_allowed);
        }
    }
}

use crate::config::Config;
use crate::document_classifier::{DocumentClass,
DocumentClassifier};
use crate::types::{Match, Pattern, Severity};
use anyhow::Result;
use regex::Regex;

pub struct ContentScanner {
    patterns: Vec<Pattern>,
    document_classifier: Option<DocumentClassifier>,
}
impl ContentScanner {
    pub fn from_config(config: &Config) -> Result<Self> {
        let mut patterns = Vec::new();

        for (_, pattern_config) in &config.patterns {
            let regex = Regex::new(&pattern_config.regex)?;
            patterns.push(Pattern {
                name: pattern_config.name.clone(),
                regex,
                severity:
Severity::from_str(&pattern_config.severity),
```

```
            });
        }
```

Продовження додатку Б

```
        log::info!("📋 Завантажено {} regex patterns",
patterns.len());

        // Завантажуємо класифікатор документів
        let document_classifier = if let Some(doc_classes) =
&config.document_classes {
            let classes: Vec<DocumentClass> = doc_classes
                .iter()
                .map(|c| DocumentClass {
                    name: c.name.clone(),
                    keywords: c.keywords.clone(),
                    min_matches: c.min_matches,
                    severity: c.severity.clone(),
                })
                .collect();

            if !classes.is_empty() {
                log::info!("📚 Завантажено {} класифікаторів
документів", classes.len());
                Some(DocumentClassifier::new(classes))
            } else {
                None
            }
        } else {
            None
        };

        Ok(Self {
            patterns,
            document_classifier,
        })
    }

    pub fn scan(&self, content: &str) -> Vec<Match> {
        let mut matches = Vec::new();

        // 1. Сканування через regex патерни
        for pattern in &self.patterns {
            for capture in pattern.regex.captures_iter(content) {

                if let Some(matched) = capture.get(0) {
                    let text = matched.as_str();
                    let masked_text = Self::mask_sensitive(text);

                    matches.push(Match {
                        pattern_name: pattern.name.clone(),
                        matched_text: masked_text,
                        severity: pattern.severity,
                        position: matched.start(),
                    });
```

```
                }
            }
        }
```

Продовження додатку Б

```
        if let Some(classifier) = &self.document_classifier {
            let doc_matches = classifier.classify(content);

            for doc_match in doc_matches {
                let severity =
Severity::from_str(&doc_match.severity);
                let keywords_str =
doc_match.matched_keywords.join(", ");
                let confidence = doc_match.confidence();

                matches.push(Match {
                    pattern_name: format!("📄 {}",
doc_match.class_name),
                    matched_text: format!("Ключові слова: {}
(впевненість: {:.0}%)", keywords_str, confidence),
                    severity,
                    position: 0,
                });
            }
        }
        matches
    }
    pub fn has_critical_matches(&self, matches: &[Match]) -> bool
{
        matches
            .iter()
            .any(|m| matches!(m.severity, Severity::Critical |
Severity::High))
    }
    fn mask_sensitive(text: &str) -> String {
        if text.len() <= 8 {
            return "*".repeat(text.len());
        }
        let start = &text[..4];
        let end = &text[text.len() - 4..];
        format!("{}***{}", start, end)
    }
}

use serde::{Deserialize, Serialize};
use std::fmt;

#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize,
Deserialize)]
pub enum Severity {
    Low,
    Medium,
    High,
    Critical,
}
```

```
impl fmt::Display for Severity {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
```



```
        match self {
            Severity::Low => write!(f, "LOW"),
            Severity::Medium => write!(f, "MEDIUM"),
            Severity::High => write!(f, "HIGH"),
            Severity::Critical => write!(f, "CRITICAL"),
        }
    }
}

impl Severity {
    pub fn from_str(s: &str) -> Self {
        match s.to_lowercase().as_str() {
            "critical" => Severity::Critical,
            "high" => Severity::High,
            "medium" => Severity::Medium,
            _ => Severity::Low,
        }
    }
}

#[derive(Clone)]
pub struct Pattern {
    pub name: String,
    pub regex: regex::Regex,
    pub severity: Severity,
}

impl fmt::Debug for Pattern {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Pattern")
            .field("name", &self.name)
            .field("severity", &self.severity)
            .finish()
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Match {
    pub pattern_name: String,
    pub matched_text: String,
    pub severity: Severity,
    pub position: usize,
}

#[derive(Debug, Serialize, Deserialize)]
pub struct QuarantineEvent {
    pub original_path: String,
    pub quarantine_path: String,
    pub quarantine_time: String,
    pub matches: Vec<MatchInfo>,
```

```
    pub file_size: u64,
    pub file_hash: String,
}
```

Продовження додатку Б

```
#[derive(Debug, Serialize, Deserialize)]
pub struct MatchInfo {
    pub pattern: String,
    pub severity: String,
    pub text: String,
}

#[derive(Debug, Default, Serialize, Deserialize)]
pub struct Statistics {
    pub files_scanned: u64,
    pub files_blocked: u64,
    pub files_allowed: u64,
    pub matches_found: u64,
    pub critical_matches: u64,
    pub uptime_seconds: u64,
}

impl Statistics {
    pub fn new() -> Self {
        Self::default()
    }

    pub fn file_scanned(&mut self) {
        self.files_scanned += 1;
    }

    pub fn file_blocked(&mut self, critical_count: usize) {
        self.files_blocked += 1;
        self.critical_matches += critical_count as u64;
    }

    pub fn file_allowed(&mut self) {
        self.files_allowed += 1;
    }
    pub fn matches_found(&mut self, count: usize) {
        self.matches_found += count as u64;
    }
}

use anyhow::Result;
use notify::{RecommendedWatcher, RecursiveMode, Watcher, Event,
EventKind};
use std::fs;
use std::path::{Path, PathBuf};
use std::sync::Arc;
use std::time::Duration;

pub struct UsbBlocker {
    media_paths: Vec<PathBuf>,}
impl UsbBlocker {
```

```
    pub fn new() -> Self {
        // Типові точки монтування для Linux
        let media_paths = vec![
```

Продовження додатку Б

```
            PathBuf::from("/media"),
            PathBuf::from("/mnt"),
            PathBuf::from("/run/media"),
        ];

        Self { media_paths }
    }
    pub fn start(self: Arc<Self>) -> Result<()> {
        log::info!("🖫 USB Blocker активний");
        log::info!("   Режим: тільки читання з USB");

        // Застосовуємо read-only до вже підключених USB
        self.apply_readonly_to_existing()?;

        // Відстежуємо нові підключення
        let blocker_clone = Arc::clone(&self);
        std::thread::spawn(move || {
            if let Err(e) = blocker_clone.watch_usb_mounts() {
                log::error!("✗ USB Watcher error: {}", e);
            }
        });
        Ok(())
    }
    fn apply_readonly_to_existing(&self) -> Result<()> {
        for media_path in &self.media_paths {
            if !media_path.exists() {
                continue;
            }
            if let Ok(entries) = fs::read_dir(media_path) {
                for entry in entries.filter_map(|e| e.ok()) {
                    let path = entry.path();
                    if path.is_dir() {
                        self.make_readonly(&path)?;
                    }
                }
            }
        }
        Ok(())
    }

    fn watch_usb_mounts(&self) -> Result<()> {
        let (tx, rx) = std::sync::mpsc::channel();
        let mut watcher: RecommendedWatcher = Watcher::new(
            tx,

notify::Config::default().with_poll_interval(Duration::from_secs(2
)),
        )?;

        // Відстежуємо /media та /mnt
```

```
        for path in &self.media_paths {
            if path.exists() {
```

Продовження додатку Б

```
                let _ = watcher.watch(path,
RecursiveMode::NonRecursive);
            }
        }

        loop {
            match rx.recv() {
                Ok(Ok(event)) => self.handle_mount_event(event),
                Ok(Err(e)) => log::error!("❌ USB Watcher error:
{}", e),
                Err(e) => {
                    log::error!("❌ Channel error: {}", e);
                    break;
                }
            }
        }
        Ok(())
    }

    fn handle_mount_event(&self, event: Event) {
        match event.kind {
            EventKind::Create(_) => {
                for path in event.paths {
                    if path.is_dir() {
                        log::warn!("💾 Виявлено нове USB
підключення: {}", path.display());
                        if let Err(e) = self.make_readonly(&path)
{
                            log::error!("❌ Не вдалося застосувати
read-only: {}", e);
                        }
                    }
                }
            }
            _ => {}
        }
    }

    fn make_readonly(&self, path: &Path) -> Result<()> {
        // На Linux використовуємо mount -o remount,ro

        #[cfg(target_os = "linux")]
        {
            use std::process::Command;

            log::info!("🔒 Застосовую read-only режим: {}",
path.display());

            let output = Command::new("mount")
```

```
                .args(&["-o", "remount,ro",
path.to_str().unwrap()])
                .output();
```

Продовження додатку Б

```
            match output {
                Ok(result) => {
                    if result.status.success() {
                        log::info!("✅ USB тепер в режимі read-
only");
                    } else {
                        log::warn!("⚠️  Не вдалося застосувати
mount (можливо потрібні права root)");
                        // Альтернатива: застосовуємо chmod
recursively
                        self.apply_chmod_readonly(path)?;
                    }
                }
                Err(e) => {
                    log::warn!("⚠️  mount не доступний: {}", e);
                    self.apply_chmod_readonly(path)?;
                }
            }
        }
        #[cfg(not(target_os = "linux"))]
        {
            log::warn!("⚠️  USB блокування не підтримується на цій
ОС");
        }

        Ok(())
    }

    fn apply_chmod_readonly(&self, path: &Path) -> Result<()> {
        use std::process::Command;

        log::info!("🔒 Застосовую chmod read-only: {}",
path.display());

        let _ = Command::new("chmod")
            .args(&["-R", "a-w", path.to_str().unwrap()])
            .output();

        Ok(())
    }
}

use anyhow::Result;
use std::process::Command;
use std::sync::Arc;

pub struct WebBlocker {
    blocked_domains: Vec<String>,
}
```

```
impl WebBlocker {
    pub fn new() -> Self {
```

Продовження додатку Б

```
        let blocked_domains = vec![
            // Файлообмінники
            "wetransfer.com".to_string(),
            "dropbox.com".to_string(),
            "drive.google.com".to_string(),
            "docs.google.com".to_string(),
            "mega.nz".to_string(),
            "mediafire.com".to_string(),
            "sendspace.com".to_string(),
            "zippyshare.com".to_string(),
            "filemail.com".to_string(),

            // Хмарні сховища
            "onedrive.live.com".to_string(),
            "box.com".to_string(),
            "pcloud.com".to_string(),
            "sync.com".to_string(),

            // Тимчасові файлообмінники
            "file.io".to_string(),
            "transfer.sh".to_string(),
            "0x0.st".to_string(),
            "tmpfiles.org".to_string(),

            // Pastebin та схожі
            "pastebin.com".to_string(),
            "paste.ee".to_string(),
            "privatebin.net".to_string(),

            // Соц мережі (опціонально)
            "web.telegram.org".to_string(),
        ];

        Self { blocked_domains }
    }

    pub fn start(self: Arc<Self>) -> Result<()> {
        log::info!("🌐 Web Blocker активний (iptables)");
        log::info!("   Заблоковано {} доменів",
self.blocked_domains.len());

        #[cfg(target_os = "linux")]
        {

            self.apply_iptables_blocks()?;
        }

        #[cfg(not(target_os = "linux"))]
        {
```

```
            log::warn!("⚠️  iptables підтримується тільки на
Linux");
        }
```

Продовження додатку Б

```
        Ok(())
    }

    #[cfg(target_os = "linux")]
    fn apply_iptables_blocks(&self) -> Result<()> {
        log::info!("🔥 Застосовую iptables правила...");

        // Перевіряємо чи iptables доступний
        let check = Command::new("iptables").arg("--
version").output();

        if check.is_err() {
            log::error!("❌ iptables не знайдено! Встановіть: sudo
apt install iptables");
            return Ok(());
        }

        // Створюємо новий ланцюг для DLP
        let _ = Command::new("iptables")
            .args(&["-N", "DLP_BLOCK"])
            .output();

        let mut blocked_count = 0;

        // Для кожного домену резолвимо IP та блокуємо
        for domain in &self.blocked_domains {
            if let Ok(ips) = self.resolve_domain(domain) {
                for ip in ips {
                    // Блокуємо вихідні з'єднання до цього IP
                    let output = Command::new("iptables")
                        .args(&["-A", "OUTPUT", "-d", &ip, "-j",
"DROP", "-m", "comment", "--comment", &format!("DLP_BLOCK_{}",
domain)])
                        .output();

                    match output {
                        Ok(result) if result.status.success() => {
                            blocked_count += 1;
                        }
                        Ok(result) => {
                            let stderr =
String::from_utf8_lossy(&result.stderr);

                            if !stderr.contains("already exists")
{
                                log::debug!("⚠️  Правило для {}
({}): {}", domain, ip, stderr);
                            }
```

```
                        }
                        Err(e) => {
                            log::warn!("⚠️  Не вдалося застосувати
iptables для {}: {}", domain, e);
```

Продовження додатку Б

```
                        }
                    }
                }
            }
        }

        log::info!("✅ iptables: застосовано {} правил для {}
доменів", blocked_count, self.blocked_domains.len());

        Ok(())
    }

    #[cfg(not(target_os = "linux"))]
    fn apply_iptables_blocks(&self) -> Result<()> {
        Ok(())
    }

    fn resolve_domain(&self, domain: &str) -> Result<Vec<String>>
{
        use std::net::ToSocketAddrs;

        let mut ips = Vec::new();

        // Пробуємо резолвити домен через стандартний DNS
        let addr = format!("{}:443", domain);
        if let Ok(addrs) = addr.to_socket_addrs() {
            for addr in addrs {
                let ip = addr.ip().to_string();
                if !ips.contains(&ip) {
                    ips.push(ip);
                }
            }
        }

        // Також пробуємо через nslookup як альтернативу
        if ips.is_empty() {
            if let Ok(output) =
Command::new("nslookup").arg(domain).output() {
                let stdout =
String::from_utf8_lossy(&output.stdout);
                for line in stdout.lines() {
                    if line.contains("Address:") {

                        if let Some(ip) =
line.split_whitespace().last() {
                            if ip.contains('.') ||
ip.contains(':') {
                                if !ips.contains(&ip.to_string())
{
```

```
                                    ips.push(ip.to_string());
                                }
                            }
                        }
```

Продовження додатку Б

```
                    }
                }
            }
        }

        Ok(ips)
    }

    pub fn remove_blocks(&self) -> Result<()> {
        #[cfg(target_os = "linux")]
        {
            self.remove_iptables_blocks()?;
        }

        Ok(())
    }

    #[cfg(target_os = "linux")]
    fn remove_iptables_blocks(&self) -> Result<()> {
        log::info!("🔥 Видаляю iptables правила...");

        // Отримуємо список всіх OUTPUT правил
        let output = Command::new("iptables")
            .args(&["-L", "OUTPUT", "-n", "--line-numbers"])
            .output();
        if let Ok(result) = output {
            let stdout = String::from_utf8_lossy(&result.stdout);
            let mut lines_to_delete = Vec::new();

            // Шукаємо правила з коментарем DLP_BLOCK
            for line in stdout.lines() {
                if line.contains("DLP_BLOCK") ||
(line.contains("DROP") && line.split_whitespace().next().map(|n|
n.chars().all(|c| c.is_numeric())).unwrap_or(false)) {
                    if let Some(num) =
line.split_whitespace().next() {
                        if num.chars().all(|c| c.is_numeric()) {

lines_to_delete.push(num.parse::<usize>().unwrap());
                        }
                    }
                }
            }

            // Видаляємо в зворотному порядку щоб номери не
зміщувались
            lines_to_delete.sort();
            lines_to_delete.reverse();
```

```
            for num in lines_to_delete {
                let _ = Command::new("iptables")
                    .args(&["-D", "OUTPUT", &num.to_string()])
                    .output();
```

Продовження додатку Б

```
                log::debug!("✅ Видалено правило #{}", num);
            }
        }

        // Видаляємо ланцюг DLP_BLOCK
        let _ = Command::new("iptables")
            .args(&["-F", "DLP_BLOCK"])
            .output();

        let _ = Command::new("iptables")
            .args(&["-X", "DLP_BLOCK"])
            .output();

        log::info!("✅ iptables правила видалено");

        Ok(())
    }

    #[cfg(not(target_os = "linux"))]
    fn remove_iptables_blocks(&self) -> Result<()> {
        Ok(())
    }
}

impl Drop for WebBlocker {
    fn drop(&mut self) {
        log::info!("🧹 Очищення Web Blocker...");
        let _ = self.remove_blocks();
    }
}
```