Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Добровольський Назар Миколайович

на здобуття ступеня вищої освіти Бакалавра

Система аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітня програма Кібербезпека

Шифр КРБКБ.220106.22.01.05 ПЗ

Виконав студент 4 курсу група КБ-22-1 Назар ДОБРОВОЛЬСЬКИЙ
Керівник канд. техн. наук, доцент Володимир ДЖУЛІЙ
Нормоконтролер д-р філософії Наталія ПЕТЛЯК

До захисту допускаю:
Завідувач кафедри кібербезпеки Юрій КЛЬОЦ
4 06 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Бакалавр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

9 січня 2026 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Добровольському Назару Миколайовичу

1 Тема роботи Система аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій

Керівник роботи канд.техн.наук, доцент Володимир ДЖУЛІЙ

Затверджено наказом ректора університету від 8 січня 2026р. № 7

2 Строк подання студентом кваліфікаційної роботи на кафедру 27 травня 2026р.

3 Вихідні дані до роботи забезпечення автоматизованого збору, обробки, структурування та кореляції розрізнених масивів відкритих даних для формування консолідованих досьє суб'єктів з використанням інструментів OSINT

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Аналіз сучасних OSINT технологій та систем обробки відкритих даних. Проєктування системи аналізу та кореляції відкритих джерел інформації. Розробка системи аналізу та кореляції відкритих джерел інформації з використанням OSINT технологій. Висновки.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Порівняльна матриця існуючих OSINT-рішень

Загальний алгоритм розслідування

Структурна схема модулів системи

6 Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7 Дата видачі завдання 12 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Вибір і затвердження теми кваліфікаційної роботи	Січень-Лютий	
Ознайомлення з предметною областю	Лютий	
Дослідження існуючих рішень	Лютий	
Постановка задачі	Березень	
Визначення загальних принципів рішення задачі	Березень	
Деталізація принципів рішення задачі	Квітень	
Розробка проєктних рішень	Квітень	
Апробація проєктних рішень	Травень	
Оформлення пояснювальної записки згідно вимог	Травень	
Оформлення графічної частини	Травень	
Захист КР	Червень	

Студент Назар ДОБРОВОЛЬСЬКИЙ

Керівник кваліфікаційної роботи Володимир ДЖУЛІЙ

АНОТАЦІЯ

Тема кваліфікаційної роботи: Система аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій

Автор роботи: Добровольський Назар Миколайович

Керівник роботи: канд. техн. наук, доцент Джулій Володимир Миколайович

Пояснювальна записка: 69 с., 20 рисунків, 8 таблиць, 2 додатки, 41 джерел.

Ключові слова: OSINT, відкриті джерела інформації, автоматизований збір даних, веб-скрапінг, кореляція даних, граф знань, інформаційна безпека.

Стрімке зростання обсягів публічно доступних даних та їх розпорошеність у цифровому просторі створюють потребу в ефективних інструментах для OSINT-досліджень, які здатні автоматизовано збирати, структурувати та візуалізувати зв'язки між інформацією.

У кваліфікаційній роботі досліджено сучасні методи та інструменти збору інформації, зокрема взаємодію з відкритими API, статичний та динамічний веб-скрапінг та методи кореляції розрізнених масивів даних. Проведено порівняльний аналіз існуючих комерційних та відкритих OSINT-рішень.

В результаті запропоновано архітектуру та розроблено клієнт-серверну систему аналізу та кореляції відкритих джерел інформації. Серверну частину з використанням асинхронної обробки, та баз даних SQLite. Клієнтську частину виконано у вигляді інтерактивного веб-додатку React для графової візуалізації зв'язків та формування консолідованих досьє суб'єктів.

25.05.2026

ANNOTATION

Theme of qualification work: System for Analysis and Correlation of Open Source Information Using OSINT Technologies

Author of the work: Dobrovolskyi Nazar Mykolayovych

Mentor: Ph.D. Dzhulii Volodymyr Mykolayovych

Explanatory note: 69 pages, 20 figures, 8 tables, 2 appendices, 41 links.

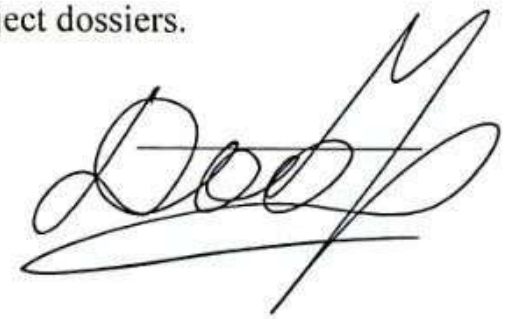
Keywords: OSINT, open source information, automated data collection, web scraping, data correlation, knowledge graph, information security.

The rapid growth and fragmentation of publicly available data in the digital space create a need for effective OSINT research tools capable of automatically collecting, structuring, and visualizing connections between information.

The qualification work investigates modern methods and tools for data collection, specifically interaction with open APIs, static and dynamic web scraping, and methods for correlating disparate datasets. A comparative analysis of existing commercial and open-source OSINT solutions was conducted.

As a result, an architecture was proposed, and a client-server system for the analysis and correlation of open-source information was developed. The server side is built using asynchronous processing and SQLite databases. The client side is implemented as an interactive React web application for the graph visualization of connections and the generation of consolidated subject dossiers.

25.05.2026



ЗМІСТ

ВСТУП

Сучасний інформаційний простір характеризується безпрецедентним зростанням обсягів публічно доступних даних. Значна частина цієї інформації осідає у соціальних мережах, державних реєстрах, новинних агрегаторах та технічних базах даних. Однак паралельно виникла проблема розпорошеності: інформація про один і той самий суб'єкт може одночасно міститися у десятках різнорідних джерел. Без системного підходу до збору та кореляції таких даних ручне опрацювання масивів аналітиком є критично неефективним.

Для вирішення цього завдання застосовується OSINT, дисципліна збору, агрегації та аналітичної обробки публічно доступної інформації. Більшість OSINT-інструментів не забезпечує повністю автоматизованого циклу від збору до аналітичного висновку без участі людини. Крім того, існуючі рішення часто не реалізують повноцінної кореляції, залишаючи розрізнені дані паралельними наборами. Тому розробка інструментів, що здатні автоматизовано збирати, та візуалізувати зв'язки, є надзвичайно актуальною.

Метою даної кваліфікаційної роботи є розробка клієнт-серверної системи аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій. Реалізація даного проєкту дозволить підвищити ефективність та швидкість проведення OSINT розслідувань.

В роботі досліджено сучасні методи та інструменти збору інформації, зокрема взаємодію з відкритими API, статичний та динамічний веб-скрапінг, а також методи кореляції розрізнених масивів даних. Проведено порівняльний аналіз існуючих комерційних та відкритих OSINT-рішень.

В результаті запропоновано архітектуру та розроблено повноцінну програмну систему. Серверну частину побудовано з використанням механізмів асинхронної обробки та бази даних SQLite. Клієнтську частину виконано у вигляді інтерактивного веб-додатку React для графової візуалізації зв'язків та автоматичного формування консолідованих досьє суб'єктів розслідування.

1 АНАЛІЗ СУЧАСНИХ OSINT-ТЕХНОЛОГІЙ ТА СИСТЕМ ОБРОБКИ ВІДКРИТИХ ДАНИХ

1.1 Загальні поняття та класифікація відкритих джерел інформації

Сучасний інформаційний простір характеризується безпрецедентним зростанням обсягів публічно доступних даних. За оцінками дослідників, щодня у відкритому доступі генерується понад 181 екзабайти інформації, значна частина якої осідає у соціальних мережах, державних реєстрах, новинних збирачах та технічних базах даних, рисунок 1.1. Цей феномен створив нові можливості для аналітичної діяльності, зокрема для збору, систематизації та інтерпретації відомостей про суб'єктів, події та явища виключно з відкритих джерел, без застосування методів негласного отримання інформації.

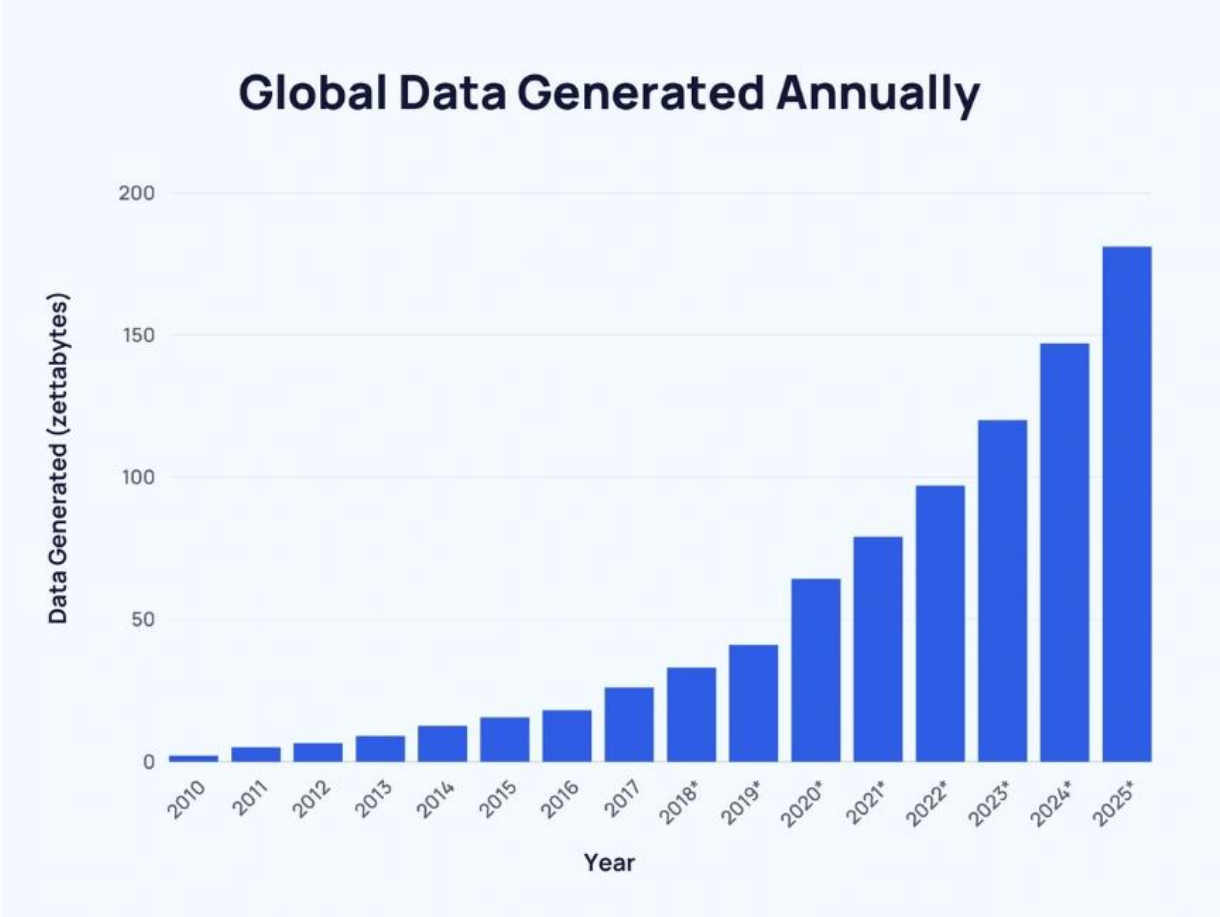


Рисунок 1.1 – Кількість щорічно оброблюваних данних у інтернеті [1]

Паралельно зі зростанням обсягів даних загострилася й інша проблема, данні розпорошені. Інформація про один і той самий суб'єкт може одночасно міститися у десятках різнорідних джерел: профілі в соціальних мережах фіксують соціальні зв'язки, державні реєстри фіксують юридичний статус та майнові відносини, технічні бази даних включають в собі мережеву інфраструктуру, а новинні архіви хронологію подій. Без системного підходу до збору та кореляції таких даних аналітик змушений опрацьовувати масиви вручну, що є критично неефективним в умовах оперативних завдань. Саме ці обставини зумовили актуальність розробки автоматизованих систем аналізу відкритих джерел інформації.

Витоки OSINT сягають періоду Другої світової війни, коли урядові структури США та Великої Британії почали цілеспрямовано слідкувати за іноземними радіопередачами, газетами та офіційними публікаціями з метою отримання розвідувальних даних без залучення агентурних мереж. Сьогодні OSINT охоплює інший масштаб джерел – від публічних профілів у соціальних мережах до метаданих цифрових документів [2].

Ключовими ознаками, що відрізняють OSINT від суміжних розвідувальних дисциплін, є три критерії: публічність джерела, легальність отримання та аналітична цінність. Саме ці критерії відмежовують OSINT від HUMINT, SIGINT, і CYBINT.

Як видно з таблиці 1.1, OSINT вирізняється серед розвідувальних дисциплін повною легальністю, високим потенціалом автоматизації та відсутністю обмежень щодо застосування в цивільному секторі. Саме ця тріада робить OSINT найбільш придатною основою для розробки автоматизованих аналітичних систем у відкритому середовищі, на відміну від SIGINT чи CYBINT, застосування яких жорстко регламентоване або прямо заборонене.

Таблиця 1.1 – Порівняння основних видів розвідки [3]

Дисципліна	Повна назва	Джерело даних	Можливість автоматизації	Правовий статус
OSINT	Open Source Intelligence	Відкриті публічні ресурси	Висока	Легальна
HUMINT	Human Intelligence	Агентурні мережі, контакти	Відсутня	Залежить від контексту
SIGINT	Signals Intelligence	Перехоплення сигналів і комунікацій	Висока	Обмежена законодавством
CYBINT	Cyber Intelligence	Кіберпростір, мережева інфраструктура	Висока	Частково обмежена
GEOINT	Geospatial Intelligence	Супутникові знімки, geodata	Середня	Переважно легальна
FININT	Financial Intelligence	Фінансові транзакції та звітність	Середня	Регульована

Сучасний інтернет не є однорідним середовищем, дослідники традиційно виділяють три різні рівні доступності інформації, що різняться між собою як технічними механізмами доступу, так і характером даних, що на них розміщені. Ці рівні прийнято позначати як Surface Web, Deep Web та Dark Web [4], і кожен із них має різне значення для практики OSINT, таблиця 1.2.

Surface Web, або поверхневий веб, охоплює сукупність веб-ресурсів, що індексуються загальнодоступними пошуковими системами – Google, Bing, DuckDuckGo та їх аналогами [5]. З точки зору OSINT-аналізу, Surface Web є найбільш технічно доступним, проте далеко не найінформативнішим рівнем. Незважаючи на те, що Surface Web є найбільш очевидним об’єктом збору даних, за різними оцінками він становить лише від 4 до 5 відсотків загального обсягу інформації, доступної в мережі Інтернет. До ключових джерел цього рівня належать: офіційні веб-сайти організацій та державних установ, відкриті новинні портали та медіа архіви, публічні профілі у соціальних мережах.

Таблиця 1.2 – Порівняльна характеристика рівнів доступності інформації [8]

Характеристика	Surface Web	Deep Web	Dark Web
Частка від загального обсягу	~4–5%	~90–95%	~1–5%
Індексація пошуковиками	Так	Ні	Ні
Технологія доступу	Стандартний браузер	Стандартний браузер + автентифікація	Tor / I2P / Freenet
Анонімність	Відсутня	Часткова	Висока
Складність автоматизованого збору	Низька	Середня–висока	Дуже висока
Правовий статус моніторингу	Беззаперечно легальний	Легальний	Потребує правової оцінки
Relevance для OSINT	Висока	Дуже висока	Вибіркова

Deep Web – це сукупність веб-ресурсів та баз даних, що не індексуються стандартними пошуковими системами, проте залишаються технічно доступними через стандартні протоколи HTTP/HTTPS за умови наявності прямого посилання або облікових даних [6]. Для OSINT-практики Deep Web є надзвичайно цінним, але технічно складнішим рівнем. За оцінками, Deep Web за обсягом перевищує Surface Web у сотні разів і становить близько 90–96 відсотків усього контенту мережі. Причини невидимості ресурсів для пошукових роботів є різноманітними: динамічна генерація сторінок на основі запитів до баз даних, вимога автентифікації, використання нестандартних форматів файлів, а також свідоме блокування індексації.

Dark Web є технічно відособленим сегментом мережі, доступ до якого потребує спеціалізованого програмного забезпечення, передусім браузера Tor, The Onion Router, що забезпечує анонімність трафіку через багаторівневе шифрування та маршрутизацію через ланцюжок проміжних вузлів [7]. Ресурси Dark Web функціонують у псевдо доменній зоні.onion та не мають маршрутизації через стандартну DNS-інфраструктуру, що унеможливлює їх виявлення та індексацію звичайними засобами. З аналітичної точки зору Dark Web становить

специфічний інтерес для OSINT-досліджень: саме тут концентруються форуми кібер злочинців, нелегальні ринки, канали витоку корпоративних та державних даних, а також комунікаційні платформи, що використовуються суб'єктами, які свідомо уникають виявлення. З правового та етичного боку, моніторинг Dark Web є юридично неоднозначною сферою, що потребує окремого осмислення, збір, зберігання та аналіз даних, отриманих із ресурсів Dark Web, що містять незаконний контент, може кваліфікуватися як участь у незаконній діяльності залежно від юрисдикції та характеру даних.

Порівняльний аналіз трьох рівнів, наведений у таблиці 1.2, демонструє, що максимальна аналітична цінність для цілей OSINT зосереджена на рівні Deep Web, попри технічні складнощі доступу. Саме тому архітектура розроблюваної системи орієнтована передусім на роботу з джерелами цього рівня у поєднанні з повноцінним охопленням Surface Web, тоді як Dark Web залишається поза межами автоматизованого збору з міркувань правової безпеки та операційної доцільності.

Соціальні мережі та платформи користувацького контенту є одними з найбільш інформаційно насичених джерел для OSINT-аналізу [9]. До цієї категорії належать: мережі для блогів, такі як Twitter/X, професійні мережі, LinkedIn, універсальні соціальні мережі, Facebook, Instagram. Аналітична цінність цієї категорії визначається кількома факторами. Соціальні мережі є джерелом даних у реальному часі, події відображаються тут значно швидше, ніж у традиційних медіа, а також дозволяють відстежувати динаміку соціальних зв'язків, хто з ким взаємодіє, які спільноти формуються навколо певних тем або суб'єктів. З технічної точки зору збір даних із соціальних мереж реалізується двома різними способами.

Перший, використання офіційних API, наприклад, Twitter API v2, LinkedIn API, Telegram Bot API, забезпечує структуровані дані у форматі JSON, проте накладає суттєві обмеження на обсяг і швидкість запитів, rate limits, а також на доступні поля даних.

Другий спосіб веб-скрапінг [10], без використання API, він дозволяє отримати значно більший обсяг даних, однак є технічно складнішим через

механізми захисту від автоматизованого доступу та несе певні правові ризики щодо відповідності умовам використання платформ.

На відміну від веб-сайтів, де дані вбудовані в HTML-розмітку і потребують парсингу, API надають структуровані дані безпосередньо у машино зчитуваному форматі, що мінімізує технічні складнощі збору та підвищує надійність інтеграції. До ключових категорій належать: геопросторові API, для збору геоданих, аналізу маршрутів та верифікації локацій, фінансові та економічні API, для отримання курсів валют, макроекономічних показників та фінансової звітності, наукові та академічні API, для моніторингу наукових публікацій, а також спеціалізовані OSINT-орієнтовані API, зокрема VirusTotal API для аналізу загроз, Shodan API для збору даних про мережеву інфраструктуру.

На відміну від реєстрів чи API, що фіксують факти, ЗМІ відображають інтерпретацію подій, суспільний дискурс та динаміку наративів. Це робить медіа моніторинг незамінним інструментом для розуміння контексту, верифікації інформації через перехресне порівняння джерел та відстеження репутаційних змін навколо суб’єктів аналізу. До цієї категорії належать: новинні портали та онлайн-видання з RSS-стрічками, телеграм-канали новинного характеру. На відміну від державних реєстрів, дані яких пройшли офіційну перевірку, медіа матеріали можуть містити неточності, упередженість або свідому дезінформацію [11].

Сканери мережевої інфраструктури, такі як Shodan та Censys є найбільш відомими платформами, що безперервно сканують увесь адресний простір інтернету та індексують відкриті порти, банери сервісів, версії програмного забезпечення та SSL-сертифікати мільярдів пристроїв. Для OSINT-аналізу ці дані дозволяють встановити технологічний стек суб’єкта, виявити незахищені сервіси та встановити зв’язки між різними об’єктами інфраструктури [12].

Різні категорії джерел суттєво відрізняються між собою, за структурою даних, надійністю, частотою оновлення та технічною складністю збору. Саме ця різноманітність є ключовим викликом для проектування автоматизованої

аналітичної системи – і водночас обґрунтуванням необхідності розробки уніфікованого підходу до нормалізації та кореляції різнорідних даних.

У цифровому середовищі будь-яка активність суб’єкта залишає сукупність фіксованих слідів [13] у різних інформаційних системах. Ці сліди виникають як наслідок взаємодії суб’єкта з цифровим середовищем і зберігаються у публічно доступних або напів публічних джерелах незалежно від наміру суб’єкта їх залишити. Таку сукупність прийнято позначати терміном цифровий слід або інформаційний слід.

Ключовими атрибутами інформаційного сліду як аналітичної одиниці є ідентифікатор джерела, тобто звідки отримано данні, часова мітка, коли зафіксовано данні, тип даних. Виділяють два різні типи цифрових слідів залежно від характеру їх виникнення.

Активний інформаційний слід формується внаслідок свідомих дій суб’єкта у цифровому середовищі. До цього типу належать публікації у соціальних мережах, коментарі на форумах, офіційні заяви на корпоративних сайтах, подані до реєстрів документи, зареєстровані доменні імена. Активні сліди відображають те, що суб’єкт свідомо транслює у публічний простір, і тому можуть бути частково сформовані з метою управління репутацією або введення в оману.

Пасивний інформаційний слід виникає без свідомої участі суб’єкта, як побічний продукт технічних процесів або дій третіх сторін. До цього типу належать: метадані документів і фотографій, технічні артефакти мережевої інфраструктури, згадки суб’єкта у публікаціях третіх осіб, записи в агрегованих базах даних. Пасивні сліди, як правило, є більш об’єктивними з аналітичної точки зору, оскільки суб’єкт не контролює їх формування.

З практичної точки зору найбільшу аналітичну цінність має саме поєднання обох типів: активні сліди розкривають декларовану ідентичність суб’єкта, а пасивні підтверджують або спростовують її. Розбіжність між активними та пасивними слідами одного суб’єкта нерідко є індикатором аномалії або прихованої діяльності [14].

1.2 Методи та інструменти автоматизованого збору даних

Основна відмінність від традиційних систем збору інформації полягає в тому, що жоден єдиний метод не є універсально придатним для всього спектру джерел: підхід, оптимальний для роботи з REST API, є непридатним для збору даних із JavaScript-rendered веб-сторінок, а методи статичного парсингу неефективні там, де потрібна обробка потокових даних у реальному часі. Отже, підсистема збору сучасної OSINT-системи не є одним модулем, а оркестрованим набором спеціалізованих технічних компонентів, кожен з яких оптимізований під певний клас джерел.

У Pull-моделі система збору виступає активною стороною: вона самостійно ініціює запити до джерела відповідно до заздалегідь визначеного розкладу або логіки. Джерело при цьому є пасивним, воно лише відповідає на вхідні запити, не маючи механізму самостійного сповіщення системи про нові дані. Загальними перевагами Pull-моделі є повний контроль системи над частотою та обсягом запитів, передбачуване навантаження на інфраструктуру та незалежність від наявності у джерела механізмів сповіщення.

У Push-моделі ініціатива передачі даних належить джерелу: воно самостійно надсилає сповіщення або потік даних до системи збору в момент виникнення нової події. Система при цьому відіграє роль пасивного отримувача, що очікує вхідних повідомлень. Ключовими перевагами Push-моделі є мінімальна затримка між подією та її обробкою, відсутність надлишкових запитів та нижче навантаження на джерело.

Жодна з двох базових моделей не є самодостатньою для покриття повного спектру OSINT-джерел. Отже, архітектура підсистеми збору розроблюваної системи ґрунтується на гібридній моделі, де вибір між Push та Pull визначається характеристиками конкретного джерела, зафіксованими у його конфігураційному профілі. Відсутність універсального рішення демонструє таблиця 1.3, кожна модель має чітко виражену спеціалізацію, що підтверджує необхідність гібридного підходу в архітектурі розроблюваної системи [15].

Таблиця 1.3 – Порівняльна характеристика архітектурних моделей збору даних

Характеристика	Polling	Pushing
Ініціатор передачі	Система	Джерело
Затримка отримання даних	Висока	Мінімальна
Надлишкові запити	Багато	Відсутні
Залежність від джерела	Низька	Висока
Складність реалізації	Низька	Середня
Придатність для реєстрів	Висока	Рідко
Придатність для соціальних мереж	Обмежена	Де підтримується
Придатність для ЗМІ	Обмежена	Рідко

Загалом, веб-джерела формують найчисельнішу та технічно найрізноманітнішу категорію серед усіх типів джерел OSINT-системи. Веб-джерела є середовищем, спроектованим передусім для споживання людиною, через це їх автоматизований збір вимагає складнішого технічного підходу. Складність додатково посилюється тим, що сучасні веб-застосунки активно протидіють автоматизованому доступу, застосовуючи різноманітні механізми захисту від ботів [16].

Базовим і найпродуктивнішим методом збору даних з веб-джерел є використання HTTP-клієнтів у поєднанні зі статичним парсингом HTML-розмітки. Цей підхід ґрунтується на прямому відтворенні HTTP-запитів, що їх у звичайному режимі надсилає браузер користувача, з подальшою синтаксичною обробкою отриманої HTML-відповіді для вилучення цільових даних. З технічної точки зору HTTP-клієнт формує запит відповідно до специфікації протоколу, передаючи необхідні заголовки, такі як User-Agent, Accept, Cookie, Referer, тощо, що імітують запит від реального браузера.

Безголовий браузер [17] є повнофункціональним веб-браузером, що виконує всі стандартні операції рендерингу, такі як завантаження ресурсів, виконання JavaScript, побудову DOM, застосування CSS, тощо, однак без графічного інтерфейсу користувача. Програмний інтерфейс безголового браузера дозволяє системі збору керувати його поведінкою: навігувати між сторінками, заповнювати форми, клікати на елементи, очікувати завантаження динамічного контенту та вилучати дані з повністю відрендереного DOM.

Потреба у безголових браузерах виникає у трьох основних сценаріях. По-перше, при роботі з Single Page Applications [18], де весь контент генерується JavaScript-фреймворками, наприклад React, Angular, Vue.js, після початкового завантаження. По-друге, при необхідності взаємодії з інтерактивними елементами, як от з пагінацією, фільтрами, формами пошуку. По-третє, при зборі даних із платформ, що активно застосовують fingerprinting для виявлення автоматизованого доступу і вимагають поведінкової автентичності від клієнта.

Сучасні веб-ресурси застосовують багаторівневі механізми захисту від автоматизованого доступу, мотивацією для яких є захист від надмірного навантаження, запобігання масовому збору даних та дотримання умов ліцензування контенту. Перед розглядом технічних методів важливо окреслити правові межі. Збір публічно доступних даних у відповідності до принципів OSINT є легальним у більшості юрисдикцій – це підтверджено, зокрема, рішенням Апеляційного суду США у справі hiQ Labs v. LinkedIn у 2022 року [19], що визнало збір публічних даних таким, що не порушує Закон про комп’ютерне шахрайство і зловживання “CFAA”.

Rate limiting [20] є найбільш поширеним механізмом захисту, що обмежує кількість запитів від одного клієнта за одиницю часу. Виявляється через HTTP-коди відповіді 429 або 503 з заголовком Retry-After. Базовою стратегією протидії є введення затримок між запитами – як фіксованих, так і рандомізованих у певному діапазоні, що імітує людську поведінку.

Сучасні платформи захисту, такі як Cloudflare, Akamai, DataDome, застосовують багатовимірний fingerprinting [21] для ідентифікації

автоматизованих клієнтів. Аналізується сукупність параметрів: заголовки HTTP-запиту та їх порядок, характеристики TLS-з'єднання, наявність та значення специфічних JavaScript-властивостей браузера, патерни навігації та взаємодії з елементами сторінки, часові характеристики дій.

CAPTCHA є механізмом верифікації людської присутності, що застосовується при виявленні підозрілої активності [22]. Для автоматизованих систем збору існують три підходи до вирішення CAPTCHA: використання спеціалізованих сервісів розпізнавання, що залучають живих операторів або ML-моделі, застосування ML-моделей власної розробки для розпізнавання текстових CAPTCHA, та архітектурне уникнення CAPTCHA через нормалізацію поведінки запитів до порогових значень, що не ініціюють її показ.

З точки зору підсистеми збору OSINT-системи API є пріоритетним методом отримання даних скрізь, де він доступний: структуровані відповіді у форматі JSON або XML не потребують парсингу, контракт взаємодії є стабільним та документованим, а сам факт використання офіційного API знімає переважну більшість правових питань щодо легітимності збору.

Сучасний ландшафт публічних API є технологічно неоднорідним, рисунок 1.2. Різні джерела реалізують різні архітектурні підходи до надання програмного доступу до своїх даних, і підсистема збору має підтримувати кожен із них. REST API є найбільш поширеним архітектурним стилем у сфері публічних API. Відповіді REST API, як правило, надаються у форматі JSON, рідше, у XML або CSV.

Range	Total APIs	URI scheme		Documentation Tool		Output Format			Authentication mechanism		
		REST	RPC	Yes	No	JSON	XML	Both / Other	OAuth	Prop.	Both / Other
1 to 5	82	84%	16%	50%	50%	55%	4%	41%	41%	38%	21%
6-50	44	86%	14%	68%	32%	59%	5%	36%	52%	30%	18%
>50	14	86%	14%	71%	29%	57%	7%	36%	86%	7%	7%

Рисунок 1.2 – Підтверджені використання арі у додатках та основні функції [23]

Rate limiting є механізмом контролю навантаження, що застосовується постачальниками API для захисту інфраструктури від надмірного навантаження та забезпечення справедливого розподілу ресурсів між клієнтами.

Для підсистеми збору OSINT-системи rate limits є одним із ключових операційних обмежень, що безпосередньо впливають на продуктивність та архітектурні рішення. Обмеження можуть специфікуватися на декількох рівнях одночасно: за кількістю запитів на секунду, за кількістю запитів на хвилину або добу, за обсягом переданих даних, за кількістю паралельних з'єднань, а також за специфічними ресурсами API. Наприклад, окремі ендпоінти можуть мати власні, жорсткіші ліміти.

Отримані "сирі" дані є непридатними для безпосереднього аналітичного використання: вони різняться за форматом, кодуванням, схемою та семантикою залежно від джерела, містять дублікати, неповні записи та артефакти збору [24]. Перетворення цих масивів інформації на уніфіковану, узгоджену та аналітично придатну форму є завданням ETL-pipeline – архітектурного патерну, що об'єднує три послідовні фази обробки даних: Extract, Transform та Load [25].

Фаза Extract є точкою входу ETL-pipeline і відповідає за отримання даних від підсистеми збору у вигляді, придатному для подальшої обробки. Фаза Extract є першою лінією обробки збоїв. Фаза Transform є найскладнішою та найважливішою частиною ETL-pipeline, саме тут різні сирі дані з різних джерел перетворюються на уніфіковані записи, придатні для кореляційного аналізу. Фаза Load є завершальним етапом ETL-pipeline і відповідає за збереження нормалізованих та збагачених записів у відповідному сховищі даних з урахуванням їх типу та подальшого аналітичного використання.

Принциповою особливістю OSINT-системи є різноманітність типів даних, що зберігаються: структуровані реєстрові записи, неструктурований текст, графи зв'язків між сутностями, часові ряди активності та бінарні медіаоб'єкти мають принципово різні характеристики доступу та вимоги до зберігання, наведена у таблиці 1.4 архітектура відображає ключовий принцип проектування сховища розроблюваної системи

Таблиця 1.4 – Порівняльна характеристика типів сховищ для зберігання OSINT-даних

Тип сховища	Приклади	Оптимальний тип даних	Модель запитів	Масштабо-ваність	Застосування в системі
Реляційна БД	PostgreSQL, MySQL	Структуровані реєстрові записи	SQL, JOIN-запити	Вертикальна	Профілі суб'єктів
Документо-орієнтована БД	MongoDB	Напівструкту-ровані, текстові дані	Документні запити, повнотексто-вий пошук	Горизон-тальна	Новини, публікації, сирі записи
Графова БД	Neo4j	Зв'язки між сутностями	Граф-обхід (Cypher, Gremlin)	Обмежена	Граф зв'язків між суб'єктами
Колонкова БД	ClickHouse, Apache Cassandra	Часові ряди, аналітичні агрегати	Аналітичні запити, агрегації	Горизон-тальна	Метрики активності, часові ряди
Кеш у пам'яті	Redis	Тимчасові дані, сесії, черги	Ключ-значення	Горизон-тальна	Rate limit стан, сесії воркерів

Замість компромісного вибору єдиного універсального сховища кожен тип даних має зберігається у сховищі, оптимізованому саме під його характеристики доступу та використання.

1.3 Алгоритми аналізу та кореляції розрізнених масивів даних

Між "сирим" масивом записів та аналітично значущими висновками пролягає концептуальна відстань. Запис про юридичну особу у реєстрі, публікація у соціальній мережі та технічний артефакт у базі Shodan є розрізненими фрагментами інформації доти, доки аналітичне ядро системи не встановить між ними змістовні зв'язки, не ідентифікує сутності, яких вони стосуються, та не інтегрує їх у цілісний профіль.

Логіка аналітичного pipeline [26] може бути представлена як модель послідовного збагачення інформації, де кожен рівень підвищує ступінь структурованості та аналітичної цінності даних:

Рівень 1 – Інформаційні сліди. Вхідними даними аналітичного pipeline є

уніфіковані інформаційні сліди, що надходять із ETL-pipeline підсистеми збору. На цьому рівні кожен слід це запис із визначеними атрибутами, наприклад джерелом, часовою міткою, типом даних, але без встановлених зв’язків з іншими записами яких він стосується.

Рівень 2 – Нормалізовані текстові представлення. Текстові поля записів проходять лінгвістичну обробку: токенізацію, лематизацію, видалення шуму та векторизацію. Результатом є структуровані лінгвістичні представлення, придатні для застосування NLP-методів наступного рівня.

Рівень 3 – Іменовані сутності та атрибути. Застосування методів Named Entity Recognition та інформаційної екстракції перетворює неструктурований текст на структурований перелік виявлених сутностей із типами та атрибутами. На цьому рівні система вперше отримує відповідь на питання "про кого або про що йдеться у цьому записі".

Рівень 4 – Розв’язані сутності. Entity Resolution встановлює тотожність між сутностями, виявленими у різних записах: "Добровольський Н. М." у реєстровому записі та "Nazar Dobrovolskyi" у профілі LinkedIn ідентифікуються як одна й та сама особа з певним ступенем достовірності. На цьому рівні формуються канонічні профілі сутностей, що агрегують всі пов’язані з ними записи.

Рівень 5 – Граф знань. Встановлені між сутностями зв’язки формують граф знань, структуру, де вузлами є розв’язані сутності, а ребрами – верифіковані відносини між ними. Граф знань є основним аналітичним артефактом системи, що дозволяє досліджувати зв’язки між сутностями.

Рівень 6 – Розвідувальний профіль. Фінальним продуктом аналітичного pipeline є структурований розвідувальний профіль суб’єкта, що інтегрує атрибути сутності, агреговані дані з усіх джерел, граф прямих та непрямих зв’язків.

Критично важливим архітектурним рішенням є визначення того, які аналітичні операції виконуються у режимі реального часу, потоковій обробці, а які у пакетному режимі, batch processing. Це рішення безпосередньо впливає на архітектуру системи, вибір технологічного стеку та компроміс між актуальністю результатів та їх повнотою [27].

Потокова обробка є обґрунтованою для операцій, результат яких має аналітичну цінність лише при мінімальній затримці: первинна обробка нових записів, базова дедуплікація на основі хешування, сигнали виявлення аномалій у режимі реального часу та оновлення профілів сутностей при надходженні нових слідів.

Пакетна обробка є доцільною для обчислювально інтенсивних операцій, що потребують повного контексту накопичених даних: повноцінний entity resolution між усіма записами сховища, перебудова графу знань після масштабного поповнення даних.

Лямбда-архітектура як компромісне рішення передбачає паралельне існування двох шарів обробки, рисунок 1.3, потоковий для забезпечення актуальних, але попередніх результатів та batch layer для формування точних, але із затримкою результатів. Запити до аналітичного ядра системи обслуговуються з обох шарів одночасно, що дозволяє поєднати переваги обох підходів.

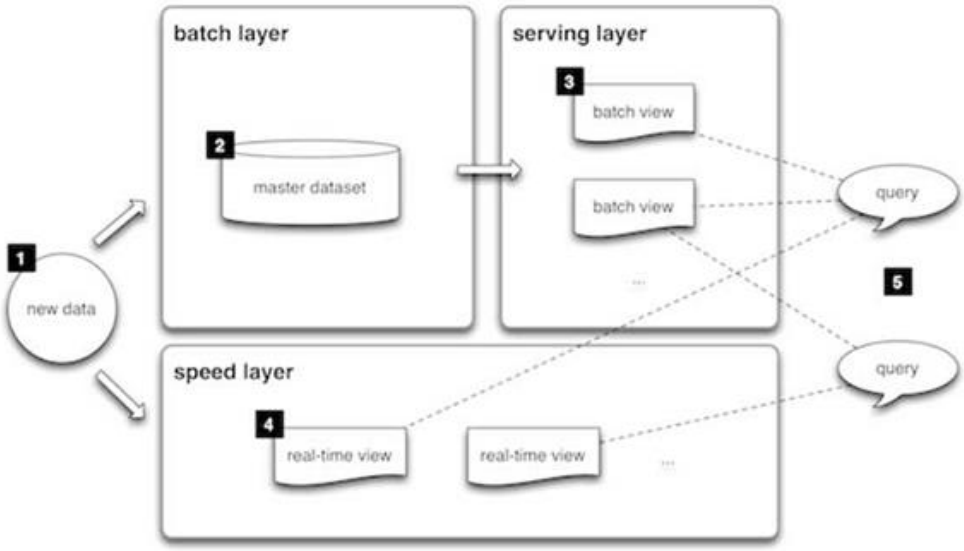


Рисунок 1.3 – Схема лямбда-архітектури [28]

Named Entity Recognition [29], розпізнавання іменованих сутностей, є центральним NLP-інструментом для OSINT-аналізу і першим методом, що перетворює неструктурований текст на структуровані дані про реальні об’єкти

світу. NER вирішує задачу автоматичного виявлення у тексті фрагментів, що відповідають заздалегідь визначеним категоріям сутностей, та їх класифікації за цими категоріями. Стандартний набір категорій сутностей для загальних NER-моделей включає: персони – імена фізичних осіб, організації – назви компаній, установ, органів влади, локації – географічні назви, адреси, дати та часові вирази, грошові суми та фінансові показники, назви продуктів та послуг.

1.4 Аналіз існуючих рішень у сфері OSINT та моніторингу відкритих джерел

Ринок OSINT-інструментів є достатньо зрілим: він налічує десятки комерційних платформ та відкритих інструментів, що пройшли тривалу еволюцію та мають усталені користувацькі бази. Аналіз цих рішень є єдиним коректним способом встановити, та сформулювати конкретні вимоги до системи на основі задокументованих обмежень наявних альтернатив.

Maltego [30] є де-факто стандартом комерційного OSINT-ринку та найбільш широко застосовуваною платформою серед професійних аналітиків і дослідників безпеки. Розроблена компанією Maltego Technologies GmbH, платформа пройшла значну еволюцію та станом на поточну версію є комплексним середовищем граф-орієнтованого аналізу зв’язків. Основою Maltego є модель Transform – атомарна операція, що приймає на вхід сутність одного типу та повертає набір пов’язаних сутностей іншого типу. Наприклад, Transform "Domain to IP" приймає доменне ім’я та повертає пов’язані IP-адреси, "Person to Social Profiles" – ім’я особи та повертає знайдені профілі у соціальних мережах.

З точки зору функціональних можливостей Maltego демонструє ряд незаперечних переваг: зрілий та інтуїтивний граф-орієнтований інтерфейс, що є стандартом для візуалізації зв’язків між сутностями, можливість створення кастомних трансформів через Python SDK, підтримка збереження та спільного доступу до аналітичних графів у корпоративному середовищі.

Однак критичний аналіз виявляє ряд суттєвих системних обмежень, що

унеможливлюють використання Maltego як основи для автоматизованої аналітичної системи [31]. Maltego є виключно інструментом ручного аналізу, тобто кожен трансформ запускається аналітиком вручну, а сама система не має механізму безперервного автоматизованого моніторингу джерел. Концепція Transform є інтерактивною – вона передбачає участь людини у кожному кроці розслідування. Закритий вихідний код унеможливлює кастомізацію ядра системи – зокрема, інтеграцію власних ML-моделей або зміну логіки кореляції.

На відміну від комерційних платформ, відкриті OSINT-інструменти є продуктом спільноти дослідників безпеки та розробників, що розробляють та підтримують їх переважно на волонтерських засадах. Відкритість вихідного коду забезпечує прозорість реалізованих алгоритмів, можливість кастомізації та відсутність ліцензійних обмежень, однак, як правило, ціною нижчої функціональної зрілості, обмеженої підтримки та вузької спеціалізації на конкретних типах задач або джерел.

SpiderFoot [32] є найбільш функціонально повним відкритим OSINT-інструментом та найближчим функціональним аналогом комерційних платформ серед рішень з відкритим вихідним кодом. SpiderFoot станом на поточну версію реалізує модульну архітектуру із понад 200 модулів збору даних, що охоплюють широкий спектр джерел. Архітектурна модель SpiderFoot ґрунтується на концепції цільового об’єкта та модулів. Аналітик визначає цільовий об’єкт, доменне ім’я, IP-адресу, електронну адресу, ім’я особи або назву організації, та вибирає набір модулів для виконання

SpiderFoot підтримує три режими роботи: CLI для інтеграції у скрипти та автоматизовані процеси, веб-інтерфейс для інтерактивного використання, рисунок 1.4, та REST API для програмної взаємодії.

Результати зберігаються у вбудованій базі даних SQLite та можуть експортуватися у формати CSV та JSON для подальшої обробки.

Серед переваг SpiderFoot слід відзначити найширше охоплення джерел серед відкритих рішень, наявність REST API що дозволяє часткову інтеграцію у зовнішні системи, активну спільноту розробників що забезпечує регулярні

оновлення модулів, а також можливість розгортання як серверного застосунку для колективного використання.

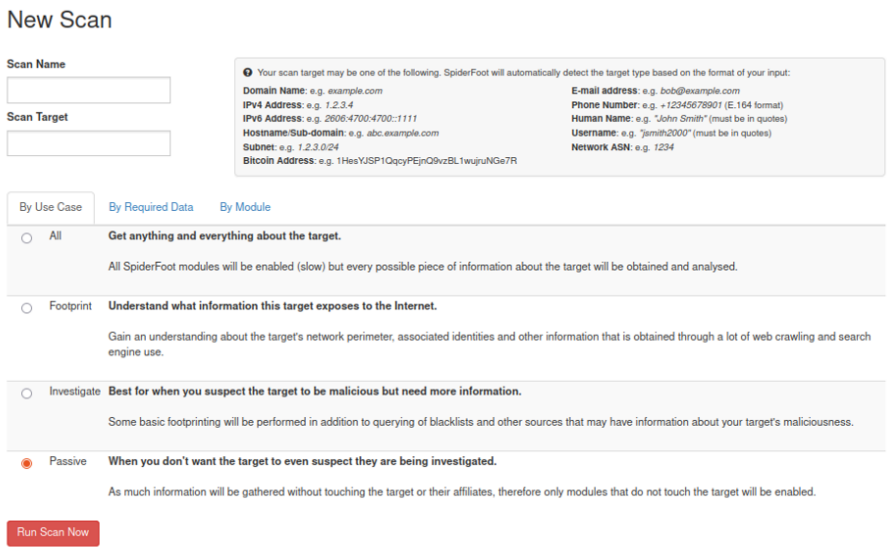


Рисунок 1.4 – Інтерфейс Spiderfoot

При аналізі можна виділити ряд обмежень SpiderFoot. Попри широке охоплення джерел, система фактично не реалізує кореляційного аналізу, рисунок 1.5, результати різних модулів є паралельними наборами даних, що відображаються в єдиному інтерфейсі, однак не проходять через entity resolution.

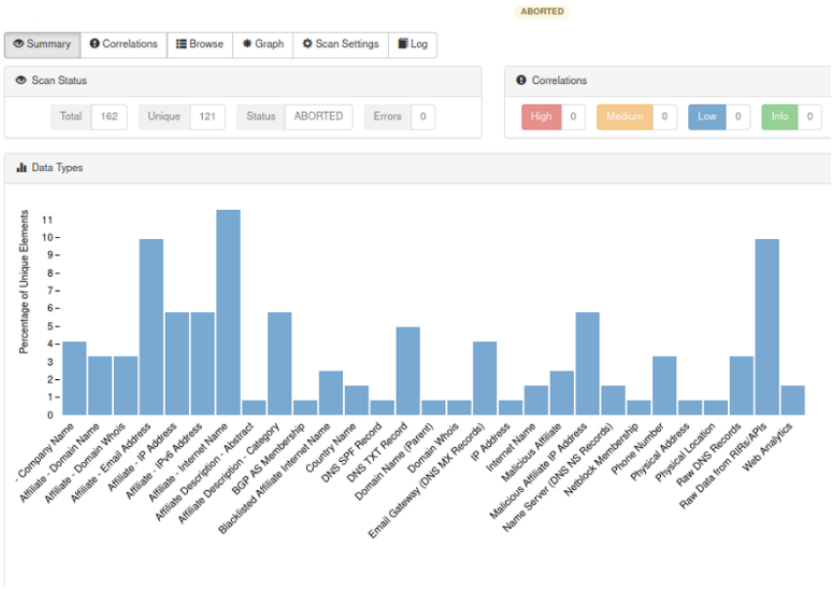


Рисунок 1.5 – Вузіалізація результатів Spiderfoot

Recon-ng є фреймворком для OSINT-розвідки, орієнтований передусім на технічну розвідку [33], а саме збір інформації про доменну інфраструктуру, піддомени, IP-адреси, електронні адреси та контактну інформацію, пов’язану з цільовими технічними об’єктами.

Інтерфейс командного рядка є єдиним режимом взаємодії з системою, що робить Recon-ng інструментом виключно для технічно підготовлених користувачів. Recon-ng демонструє ряд переваг для технічно орієнтованих OSINT-задач: відкритий вихідний код з активною підтримкою, CLI-орієнтованість що забезпечує природну інтегрованість у скриптові workflow, чітка організація результатів у реляційній базі даних та низький поріг розробки кастомних модулів завдяки простому Python API.

Обмеження Recon-ng є прямим наслідком його технічної спеціалізації. Фреймворк фактично не підтримує аналіз соціальних мереж, новинних джерел та реєстрових даних у тому обсязі, що є необхідним для повноцінного OSINT-аналізу суб’єктів. Відсутність графового представлення результатів унеможливлює візуальний аналіз зв’язків.

theHarvester є вузькоспеціалізованим інструментом первинної розвідки, орієнтованим на автоматизований збір електронних адрес, імен, піддоменів, IP-адрес та URL-адрес із публічних джерел стосовно визначеного цільового домену або організації [34]. Практична цінність theHarvester полягає у швидкому формуванні первинного переліку технічних та контактних артефактів, пов’язаних із цільовою організацією, на початковому етапі розслідування.

Інструмент є простим у використанні, не потребує складного налаштування та надає результати у структурованому текстовому або XML-форматі. Водночас theHarvester є інструментом виключно первинного збору без жодних аналітичних можливостей: отримані дані не проходять жодної обробки, нормалізації або кореляції. Інструмент не підтримує аналіз фізичних осіб, реєстрових даних або соціальних мереж у будь-якому значущому обсязі. Відсутність механізму збереження стану між запусками та будь-якої бази даних результатів робить theHarvester непридатним для систематичного моніторингу.

Sherlock [35] є інструментом ідентифікації цифрової присутності суб’єкта у соціальних мережах та онлайн-платформах. Sherlock реалізує пошук акаунтів за іменем користувача (username) на понад 300 платформах одночасно: інструмент формує стандартизовані URL-адреси профілів для кожної платформи та перевіряє їх доступність через HTTP-запити.

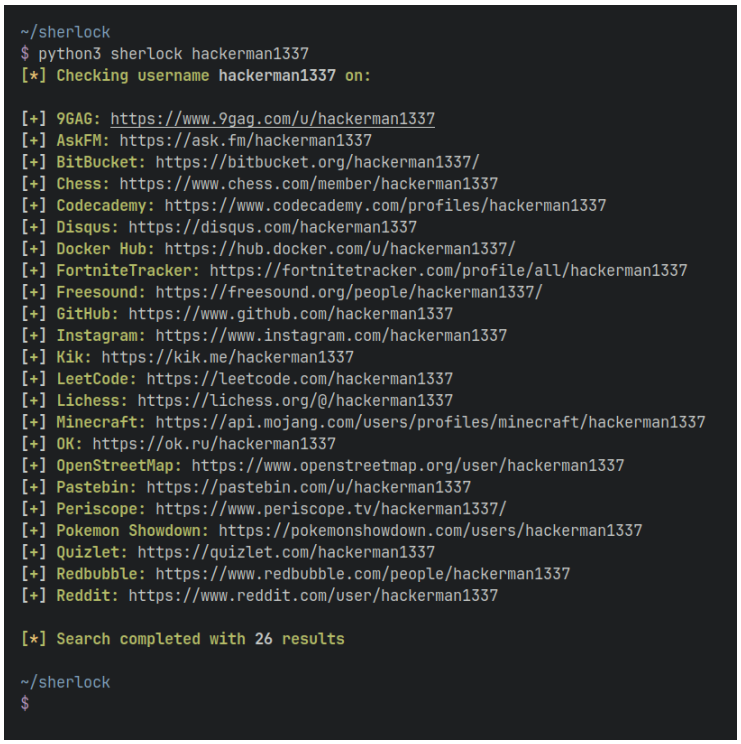


Рисунок 1.6 – Результат роботи утиліти Sherlock

Інструмент має чітко визначену нішу, дозволяючи швидко картографувати цифрову присутність суб’єкта на початковому етапі розслідування, рисунок 1.6. Однак Sherlock виконує єдину функцію та не мають жодних механізмів для збагачення, кореляції або аналізу знайдених акаунтів. Результати є простими переліками URL-адрес без жодної структурованої інформації про самі акаунти. Масштабоване застосування Sherlock до великих списків цільових суб’єктів є технічно складним через відсутність черг задач та обмежену підтримку паралельного виконання.

Найбільш системною прогалиною є відсутність у жодному з розглянутих рішень повністю автоматизованого циклу від збору до аналітичного висновку без

участі людини. Maltego є інтерактивним аналітичним середовищем, де кожен крок розслідування ініціюється аналітиком вручну. SpiderFoot та Recon-ng автоматизують збір даних щодо конкретного цільового об'єкта, однак вимагають ручного запуску для кожного нового об'єкта та не мають механізму безперервного моніторингу. Спеціалізовані інструменти такі як Sherlock, theHarvester, є виключно одноразовими утилітами. Зведена таблиця 1.5 наочно підтверджує системний характер виявлених прогалин.

Таблиця 1.5 – Зведена порівняльна матриця існуючих OSINT-рішень

Критерій	Maltego	SpiderFoot	Recon-ng	theHarvester	Sherlock
Модель доступу	Комерційна	Відкрита / Комерційна	Відкрита	Відкрита	Відкрита
Автоматизований моніторинг	Відсутній	Одиничний запит	Одиничний запит	Відсутній	Відсутній
Охоплення джерел	Широке	Широке	Технічні	Технічні	Вузьке
Entity Resolution	Відсутній	Відсутній	Відсутній	Відсутній	Відсутній
ML-аналітика	Відсутня	Відсутня	Відсутня	Відсутня	Відсутня
Графове представлення	Розвинене	Базове	Відсутнє	Відсутнє	Відсутнє
Підтримка кирилиці	Часткова	Відсутня	Відсутня	Відсутня	Відсутня
Масштабованість	Низька	Обмежена	Обмежена	Відсутня	Відсутня
Відкритий код	Ні	Так	Так	Так	Так
API для інтеграції	Обмежений	Обмежений	Відсутній	Відсутній	Відсутній
Виявлення аномалій	Відсутнє	Відсутнє	Відсутнє	Відсутнє	Відсутнє
Вартість	Від $999/рік	Безкоштовна	Безкоштовна	Безкоштовна	Безкоштовна

Жодне з розглянутих рішень не задовольняє одночасно критеріям повної автоматизації, якісного кореляційного аналізу з entity resolution, підтримки ML-методів, горизонтальної масштабованості та відкритої архітектури. Комерційні

платформи забезпечують вищу функціональну зрілість за рахунок закритості та надвисокої вартості, відкриті інструменти є доступними, але обмеженими.

1.5 Постановка задачі

За результатами проведеного дослідження сучасних методів збору даних та аналізу відкритих джерел, а також огляду наявних програмних рішень, постає задача розробки ефективної системи аналізу та кореляції інформації з використанням OSINT-технологій.

Об’єктом роботи визначено розробку програмної системи, яка забезпечуватиме автоматизований збір, обробку, та наочну візуалізацію даних з відкритих джерел для проведення OSINT-розслідувань.

Система повинна підтримувати виконання пошукових запитів за різними типами вхідних даних. Збір має здійснюватися з використанням спеціалізованих модулів, пошук по відкритій мережі, аналіз соціальних профілів, використання Headless-браузерів для парсингу динамічного контенту.

Зібрана неструктурована інформація має оброблятися для виділення ключових сутностей, таких як імена, геолокації, контактні дані, пов’язані організації.

За результатами аналізу система повинна автоматично формувати комплексний звіт – "досьє", який містить систематизовану інформацію про об’єкт з посиланнями на першоджерела. Збереження історії пошуку, можливість повертатися до попередніх розслідувань, експортувати графи та звіти. Система має бути побудована за клієнт-серверною архітектурою з використанням REST API. Серверна частина повинна бути масштабованою та забезпечувати асинхронне виконання тривалих пошукових запитів.

Клієнтська частина повинна мати сучасний, інтуїтивно зрозумілий інтерфейс. Серверна частина повинна мати плагінну, модульну, структуру. Це необхідно для того, щоб при появі нових відкритих джерел або методів OSINT, новий модуль міг бути інтегрований у систему без зміни її ядра.

2 ПРОЄКТУВАННЯ СИСТЕМИ АНАЛІЗУ ТА КОРЕЛЯЦІЇ ВІДКРИТИХ ДЖЕРЕЛ ІНФОРМАЦІЇ

2.1 Проєктування бази данних OSINT-системи

Розроблювана система побудована за дворівневою клієнт-серверною архітектурою із розмежуванням відповідальності між серверною та клієнтською частинами. Збір та аналіз даних є обчислювально інтенсивними операціями що можуть тривати від кількох секунд до кількох хвилин, та вимагають паралельного запуску десятків мережевих запитів одночасно.

Система складатиметься з декількох рівнів що взаємодіють через визначені інтерфейси та реалізують повний цикл від отримання запиту до відображення результатів у інтерфейсі користувача.

Клієнтський інтерфейс реалізований як React Single Page Application що функціонує у браузері [36]. Клієнт взаємодіє з серверною частиною виключно через REST API [37] запити та не містить жодної бізнес-логіки збору або аналізу даних.

API-шлюз реалізований на основі FastAPI [38] та є єдиною точкою входу для всіх клієнтських запитів. Цей рівень відповідає за маршрутизацію запитів до відповідних обробників, валідацію вхідних даних, захист від надмірного навантаження через rate limiting та формування структурованих відповідей.

Логіка системи містить три незалежні підсистеми що реалізують основну функціональність системи. Підсистема збору даних координує паралельне виконання OSINT-модулів та формування графу сутностей із їх результатів. Кореляційний рушій реалізує алгоритми аналізу графу зв’язків між виявленими сутностями. Підсистема формування досьє будує профілі суб’єктів із агрегованих даних множини модулів та генерує звіти.

Зберігання даних реалізований на основі SQLite та доступний усім компонентам [39]. Цей рівень зберігає всі персистентні дані системи: виявлені сутності, зв’язки між ними, метадані розслідувань та консолідовані досьє.

SQLite обраний через розгортання системи у Docker [40], вона є автономним

застосунком що не потребує зовнішніх сервісів баз даних і повністю функціонує у межах двох контейнерів. Розгортання клієнт-серверної бази данних PostgreSQL або MySQL вимагало б третього контейнера, окремої конфігурації мережевої взаємодії та процедур ініціалізації, що суттєво ускладнює операційне обслуговування. SQLite забезпечує паралельне читання необмеженою кількістю читачів при одночасному записі одного письменника, через що є достатньою моделлю для архітектури з одним серверним процесом.

Схема бази даних складається з чотирьох таблиць що реалізують повну модель даних системи, таблиця 2.1.

Таблиця 2.1 – Схема бази даних OSINT-системи

Назва таблиці	Поле	Тип	Обмеження	Призначення
1	2	3	4	5
entities	Id	TEXT	PRIMARY KEY	UUID-ідентифікатор (8 символів)
	Type	TEXT	NOT NULL	Тип сутності: domain, ip, email, username тощо
	Value	TEXT	NOT NULL	Канонічне значення – ключ дедуплікації
	label	TEXT	–	Відображувана мітка у інтерфейсі
	data	TEXT (JSON)	DEFAULT ‘{}’	Довільні атрибути специфічні для типу
	investigation_id	TEXT	FK → investigations	Прив’язка до розслідування
	created_at / updated_at	TEXT	NOT NULL	Часові мітки у форматі ISO 8601
edges	id	TEXT	PRIMARY KEY	UUID-ідентифікатор ребра
	source_id	TEXT	FK CASCADE DELETE	Вихідний вузол зв’язку
	target_id	TEXT	FK CASCADE DELETE	Цільовий вузол зв’язку
	relation	TEXT	NOT NULL	Тип відношення: resolves_to, owned_by тощо
	weight	REAL	DEFAULT 1.0	Вага ребра для зваженого аналізу

Кінець таблиці 2.1

1	2	3	4	5
investigations	id	TEXT	PRIMARY KEY	UUID розслідування
	target	TEXT	NOT NULL	Цільовий об’єкт розслідування
	target_type	TEXT	NOT NULL	Тип: domain, ip, email, username, phone, person
	status	TEXT	DEFAULT ‘pending’	Статус: pending / completed
	results	TEXT (JSON)	DEFAULT ‘{}’	Агреговані результати модулів
dossiers	id	TEXT	PRIMARY KEY	UUID досьє суб’єкта
	target_name	TEXT	NOT NULL	Ім’я або ідентифікатор суб’єкта
	normalized_data	TEXT (JSON)	DEFAULT ‘{}’	Нормалізований профіль NormalizedPerson
	raw_results	TEXT (JSON)	DEFAULT ‘{}’	Необроблені результати модулів
	risk_score	TEXT	DEFAULT ‘low’	Рівень ризику: low / medium / high / critical

Таблиця entities є центральною таблицею що зберігає вузли графу OSINT-розслідування. Кожна сутність має унікальний восьмисимвольний ідентифікатор що генерується через str(uuid.uuid4())[:8]. Поле investigation_id пов’язує сутність із конкретним розслідуванням та дозволяє фільтрувати граф по розслідуваннях у клієнтському інтерфейсі.

Таблиця edges зберігає ребра графу зв’язків між сутностями. Зовнішні ключі source_id та target_id із директивою ON DELETE CASCADE забезпечують автоматичне видалення ребер при видаленні пов’язаних вузлів. Поле relation зберігає семантичний тип зв’язку у текстовому форматі (resolves_to, owned_by, has_profile, exposed_in тощо) що дозволяє фільтрувати ребра за типом при аналізі. Поле weight із значенням за замовчуванням 1.0 зарезервоване для алгоритмів зваженого аналізу графу.

Таблиця investigations зберігає метадані розслідувань. Статус змінюється з pending на completed після завершення паралельного виконання всіх модулів що

дозволяє відстежувати стан тривалих розслідувань.

Таблиця dossiers зберігає консолідовані профілі суб'єктів із нормалізованими даними, необробленими результатами модулів та оцінкою рівня ризику. Для забезпечення продуктивності запитів на таблицях визначені індекси по полях що найчастіше використовуються у фільтрах: idx_entities_type та idx_entities_value для пошуку сутностей за типом та значенням, idx_entities_investigation для фільтрації по розслідуванню, idx_edges_source та idx_edges_target для навігації по ребрах графу.

Модульність підсистеми збору є ключовим архітектурним принципом що визначає розширюваність системи та її здатність інтегрувати нові джерела даних без модифікації існуючого коду.

Додавання нового джерела даних до системи зводиться до двох кроків: реалізації функції-модуля що відповідає уніфікованому інтерфейсу, та додавання одного рядку до реєстру.

Конфігурація модулів за замовчуванням для кожного типу цільового об'єкта визначається функцією, що повертає список імен модулів оптимальних для даного типу цілі. Наприклад для типу domain запускаються модулі dns, whois, web_scraper, subdomain_enum, threat_intel та tech_fingerprint, для типу email – email, dns, social_scan_email та breach_check.

Валідація та нормалізація вхідних даних реалізована у функції, що застосовує різну логіку обробки залежно від типу цільового об'єкта. Для типу email значення приводиться до нижнього регістру та перевіряється регулярним виразом на відповідність формату адреси електронної пошти. Для типу phone видаляються пробіли, дефіси та дужки, після чого значення нормалізується до формату із обов'язковим префіксом "+". Для типу username видаляється провідний символ "@" та всі символи що не є алфавітно-цифровими, підкресленнями, дефісами або крапками. Для типу domain значення приводиться до нижнього регістру та видаляється протокол.

Розгортання системи реалізоване через Docker. Контейнеризація є умовою відтворюваності середовища виконання: всі залежності системи – інтерпретатор

Python конкретної версії, бібліотеки із зафіксованими версіями у requirements.txt, Node.js та npm-пакети – упаковані у образи та виконуються ідентично незалежно від операційної системи та конфігурації хостової машини.

2.2 Проєктування підсистем збору та кореляції відкритих даних OSINT-системи

Центральним технічним рішенням підсистеми збору є паралельне виконання всіх OSINT-модулів через механізм asyncio, що є стандартним способом конкурентного виконання корутин у Python. При надходженні запиту на розслідування система формує словник для кожного модуля із переліку активних модулів та передає їх до asyncio [41].

При типовому розслідуванні запускаються вісім модулів із середнім часом виконання від 2 до 5 секунд кожен. При послідовному виконанні загальний час становив би від 16 до 40 секунд, що є неприйнятним для інтерактивного аналітичного інструменту. При паралельному виконанні через asyncio загальний час визначається найповільнішим модулем та становить 3–8 секунд. OSINT модулі у системі розбиті по категоріям, таблиця 2.2.

Категорія технічної розвідки містить модулі орієнтовані на збір технічних даних про мережеву інфраструктуру. Реалізовується повний цикл DNS-запитів. Додатково виконується reverse DNS для кожної виявленої IP-адреси через PTR-запити що дозволяє встановити доменні імена пов’язані з IP-адресою. Через бібліотеку python-whois отримується реєстраційна інформація домену, включаючи реєстратора, організацію-власника, email-контакти та дати реєстрації і закінчення. Також реалізовується модуль який визначає географічне розташування IP-адреси через зовнішній геолокаційний API та повертає країну, місто, координати та інтернет-провайдера.

Модулі які виконують перебір піддоменів за словником, отримують всі SSL-сертифікати пов’язані з доменом через відкритий журнал Certificate Transparency

на crt.sh. Модуль, який виявляє технологічний стек веб-ресурсу через аналіз HTTP-заголовків та вмісту сторінки.

Категорія веб-аналізу реалізує збір даних безпосередньо із веб-сторінок. Модуль який виконує веб скрапінг є найбільш функціонально насиченим у цій категорії, він використовує aiohttp для асинхронного HTTP-запиту та BeautifulSoup4 для парсингу HTML та витягує email-адреси та телефонні номери через регулярні вирази, зовнішні посилання через аналіз тегів, посилання на профілі у соціальних мережах через набір платформо-специфічних патернів. Модуль на основі Playwright забезпечує збір із JavaScript-rendered ресурсів де статичний HTTP-запит повертає лише порожній шаблон. Модуль, який витягує метадані із файлів що завантажуються із цільового ресурсу.

Категорія аналізу суб'єктів є найчисельнішою та реалізує збір інформації про конкретних осіб.

Модуль який перевіряє наявність акаунтів із заданим іменем користувача на множині платформ та витягує метадані знайдених профілів включаючи відображуване ім'я, біографію, аватар та геолокацію. Модуль який виконує комплексний аналіз електронної адреси, визначає поштового провайдера, перевіряє MX-записи домену та виявляє пов'язані сервіси. Модуль який використовує бібліотеку phonenumbers для визначення країни, оператора та типу номера. Група модулів, які виконують пошук профілів у соціальних мережах за відповідними ідентифікаторами. Модуль який перевіряє наявність email-адреси у відомих витоках даних.

Категорія загроз та спеціальних модулів реалізує інтеграцію із зовнішніми платформами аналізу загроз та спеціалізовані функції.

Модуль який інтегрується із VirusTotal API та отримує репутаційну оцінку домену або IP-адреси на основі аналізу антивірусними рушіями та даних про фішингові кампанії.

Паралельно із збором даних система автоматично будує граф сутностей із результатів кожного модуля.

Таблиця 2.2 – Перелік OSINT-модулів системи

Категорія	Модуль	Бібліотека	Тип цілі
Технічна розвідка	dns_lookup	Dnspython	domain
	whois_lookup	python-whois	domain, ip
	ip_geolocation	aiohttp + API	ip
	subdomain_enum	Aiohttp	domain
	cert_transparency	aiohttp (crt.sh)	domain
	tech_fingerprint	Aiohttp	domain, url
Веб-аналіз	web_scraper	aiohttp + BS4	domain, url
	metadata_extractor	Aiohttp	url
	headless_browser	Playwright	url
Аналіз суб’єктів	username_checker	Aiohttp	username
	email_analyzer	Aiohttp	email
	phone_lookup	phonenumbers	phone
	social_scan_email	Aiohttp	email
	social_scan_phone	Aiohttp	phone
	social_scan_username	Aiohttp	username
	breach_check	Aiohttp	email
	person_search	Aiohttp	person
	person_dossier_builder	Комплексний	person
Загрози та спеціальні	shodan_search	Shodan API	ip, domain
	threat_intel	VirusTotal API	domain, ip
	image_osint	Pillow	file
	document_analyzer	pypdf, python-docx	file

Підсистема формування досьє будується навколо фізичної особи як суб’єкта аналізу та агрегує всі доступні відомості про неї із множини паралельно запущених модулів у єдиний структурований профіль. Група ідентифікаційних атрибутів містить чотири колекції: names – відомі імена та псевдоніми суб’єкта, emails – виявлені електронні адреси, phones – телефонні номери, usernames – імена користувача на різних платформах.

2.3 Проєктування клієнтської частини

Серверні підсистеми автоматизованого збору, структурування та кореляції інформації реалізують замкнутий цикл, який охоплює етапи витягнення даних із відкритих джерел, їхньої нормалізації, виявлення взаємозв'язків та збереження результуючих сутностей у базі данних.

Клієнтська частина системи інтегрується у загальну архітектуру як відокремлений рівень представлення даних, що виконує функції спеціалізованого графічного інтерфейсу для взаємодії аналітика із розподіленими обчислювальними модулями бекенду.

Взаємодія між компонентами організована через прикладний програмний інтерфейс REST API, що дозволяє клієнтській стороні надсилати асинхронні HTTP-запити, приймати та десеріалізувати структуровані об'єкти у форматі JSON без блокування інтерфейсного потоку.

Отримані дані динамічно трансформуються в інтерактивну топологічну модель графу знань, деталізовані цифрові профілі об'єктів розслідування та зведені звіти, забезпечуючи аналітика верифікованим інструментарієм для аналізу багатовимірних зв'язків та підготовки підсумкової документації.

Реалізована як React Single Page Application клієнтська частина є тонким клієнтом що не містить жодної бізнес-логіки збору або аналізу, вся обчислювальна робота виконується на сервері, клієнт лише відображає результат. Такий підхід гарантує високу продуктивність інтерфейсу, оскільки браузер користувача не перевантажується складними алгоритмами обробки масивів даних, а оперує виключно готовими, серіалізованими JSON-структурами.

Кореневим компонентом застосунку є App.jsx що реалізує централізоване управління навігаційним станом та координацію між основними компонентами. Навігація між п'ятьма основними вікнами реалізована через стан activeView що приймає одне з п'яти значень: dashboard, investigate, graph, report, settings. Перелік навігаційних елементів визначений декларативно через константу, що містить ідентифікатор, мітку та іконку кожного вікна. Додавання нового вікна зводиться

до одного запису у цьому переліку та відповідного умовного рендерингу у React. Декларативний підхід до маршрутизації забезпечує високу швидкість перемикання контексту, оскільки алгоритм узгодження ефективно оновлює лише змінені вузли віртуального DOM-дерева без повного перезавантаження сторінки.

Компонент реалізує дворівневий макет, бічну панель зліва та область основного вмісту справа, що займає весь залишковий простір. Бічна панель підтримує два стани: розгорнутий із відображенням повних міток навігаційних елементів та згорнутий, де відображаються лише іконки.

Кожне основне вікно обгорнуте компонентом ErrorBoundary із атрибутом “key={activeView}”, при зміні активного вікна React створює новий екземпляр ErrorBoundary що скидає стан попередньої помилки. Без цього атрибута помилка у одному компоненті залишала б ErrorBoundary у стані помилки навіть після переходу до іншого вікна, що унеможливлювало б подальше використання застосунку без перезавантаження сторінки.

Dashboard є стартовим вікном системи що відображається при початковому завантаженні застосунку. Компонент отримує статистику системи та відображає три ключові показники: загальну кількість сутностей у графі, кількість зв’язків між ними та кількість проведених розслідувань. Навігаційні картки надають швидкий доступ до основних функцій системи.

Investigation є основним робочим вікном для запуску нових OSINT-розслідувань. Компонент реалізує форму із двома обов’язковими полями: вибір типу цільового об’єкта із переліку підтримуваних типів, domain, ip, email, username, phone, url, person, та поле введення значення цільового об’єкта.

GraphView інтерактивно візуалізаціє граф сутностей, використовуючи можливості Canvas API для апаратного прискорення рендерингу. Компонент реалізує механізми фільтрації що застосовуються по графу одночасно. Операції фільтрації вузлів за типом або належністю до конкретного розслідування виконуються за константний час, забезпечуючи миттєву реакцію інтерфейсу на дії аналітика.

EntityDetail є слайдовою панеллю що відображається поверх основного

вмісту при кліку на будь-який вузол графу або сутність у Dashboard. Компонент завантажує повні дані та відображає всі її атрибути включаючи тип, значення, мітку, JSON-дані специфічні типам та часові мітки.

Report відображає список збережених досьє та розслідувань із можливістю відкрити деталі кожного.

Компонент DossierReport відображає повне досьє суб'єкта організоване у секціях, секція identity із переліком імен, email-адрес, телефонів та username, секція digital footprint із таблицею знайдених профілів у соціальних мережах та показниками достовірності кожного, секція breach exposure із переліком виявлених витоків даних, секція timeline із хронологією подій, секція correlations із виявленими крос-джерельними зв'язками.

Модуль api.js є централізованим API-клієнтом що реалізує всі методи взаємодії React-застосунку з REST API серверної частини. Базовий URL сервера визначений в одному місці як константа http://localhost:8000/api, зміна адреси або порту сервера потребує модифікації лише одного рядка коду незалежно від кількості компонентів що використовують API. Всі HTTP-запити реалізовані через стандартний Fetch API.

Кожен метод API-клієнта реалізований як асинхронна функція із обробкою помилок через try/catch. При виникненні мережевої помилки або помилки HTTP-відповіді функція логує деталі через console.error та повертає null або порожню структуру даних відповідного типу. Це рішення є свідомим компромісом, воно дозволяє інтерфейсу продовжувати роботу частково, відображаючи порожні стани замість критичного падіння всього застосунку при мікро-відмовах бекенду.

2.4 Висновки

У цьому розділі було здійснено комплексне проєктування системи аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій, архітектура якої базується на сучасному дворівневому клієнт-серверному підході.

Обрана архітектурна модель дозволила чітко розмежувати відповідальність між обчислювально інтенсивною серверною частиною та інтерактивним інтерфейсом користувача.

Серверна частина реалізована за допомогою API-шлюзу на базі фреймворку FastAPI. Використання бази даних SQLite з контейнеризацією через Docker забезпечило максимальну портативність та автономність застосунку. Таке рішення дозволило уникнути надмірної інфраструктурної складності, надаючи при цьому надійне збереження графових структур, метаданих розслідувань та досьє.

Ключовим технічним рішенням підсистеми збору стало використання механізму паралельного виконання розроблених OSINT-модулів, що дозволило зменшити загальний час обробки розслідування до кількох секунд. Було спроєктовано модулі для чотирьох основних категорій: технічної розвідки, веб-аналізу, аналізу суб'єктів та оцінки загроз. Результати їхньої роботи автоматично формують загальний граф сутностей та зв'язків, після чого підсистема формування досьє агрегує всі виявлені відомості у єдиний структурований профіль фізичної особи чи цільового об'єкта.

Клієнтська частина системи розроблена як клієнт у вигляді інтерактивного веб-додатку на базі React, який взаємодіє із сервером виключно через REST API. Інтерфейс забезпечує повну взаємодію аналітика із системою: від панелі із загальною статистикою та форми для ініціалізації нових розслідувань, до компонентів інтерактивної візуалізації графу.

Процес кореляції інформації базується на автоматичній побудові топологічної моделі. Система нормалізує отримані сирі дані, дедуплікує їх та зв'язує у єдиний граф сутностей: домени, IP-адреси, email-адреси, профілі у соціальних мережах. На основі цього графу підсистема формування досьє автоматично агрегує виявлені відомості у єдиний структурований профіль цільового суб'єкта з оцінкою рівня ризику та хронологією цифрових слідів.

3 РОЗРОБКА СИСТЕМИ АНАЛІЗУ ТА КОРЕЛЯЦІЇ ВІДКРИТИХ ДЖЕРЕЛ ІНФОРМАЦІЇ З ВИКОРИСТАННЯМ OSINT-ТЕХНОЛОГІЙ

3.1 Проєктування алгоритму роботи OSINT-системи

Загальний алгоритм роботи системи визначає послідовність обробки аналітичного запиту від моменту його надходження через REST API до формування структурованої відповіді клієнту.

Алгоритм реалізований у файлі server/routes.py та координує взаємодію між усіма підсистемами.

На першому кроці система отримує HTTP POST запит на ендпоінт /api/investigate що містить об’єкт InvestigateRequest із трьома полями: target, type та опціональне поле modules. Бібліотека Pydantic автоматично виконує валідацію типів та структури вхідного об’єкта на рівні FastAPI фреймворку до передачі керування обробнику запиту.

На другому кроці алгоритм створює запис розслідування у базі даних із початковим статусом pending та зберігає ціль і її тип. Одночасно виконується нормалізація значення цільового об’єкта. Після нормалізації створюється головна сутність розслідування що стає кореневим вузлом графу.

На третьому кроці алгоритм визначає перелік модулів для виконання. Якщо клієнт передав явний перелік у полі modules, використовується він, в іншому випадку викликається окрема функція що повертає перелік модулів за замовчуванням для даного типу цілі. Наприклад для типу domain за замовчуванням запускаються модулі dns, whois, web_scraper, metadata, subdomain_enum, threat_intel, tech_fingerprint, для типу email – email, dns, social_scan_email та breach_check.

Четвертий крок – паралельний запуск усіх визначених модулів, результати виконання кожного модуля накопичуються у словнику де ключем є символьне ім’я модуля, а значенням структурований словник результатів або об’єкт Exception при збої.

На п’ятому кроці алгоритм послідовно обробляє результати кожного модуля

перетворюючи специфічні для кожного модуля результати на вузли та ребра графу і зберігає їх через EntityStore.

На шостому кроці статус розслідування оновлюється з pending на completed із збереженням агрегованих результатів усіх модулів у форматі JSON.

Останнім кроком формується та повертається клієнту структурована відповідь що містить ідентифікатор розслідування, цільовий об’єкт, статус виконання та повні результати всіх модулів, рисунок 3.1.

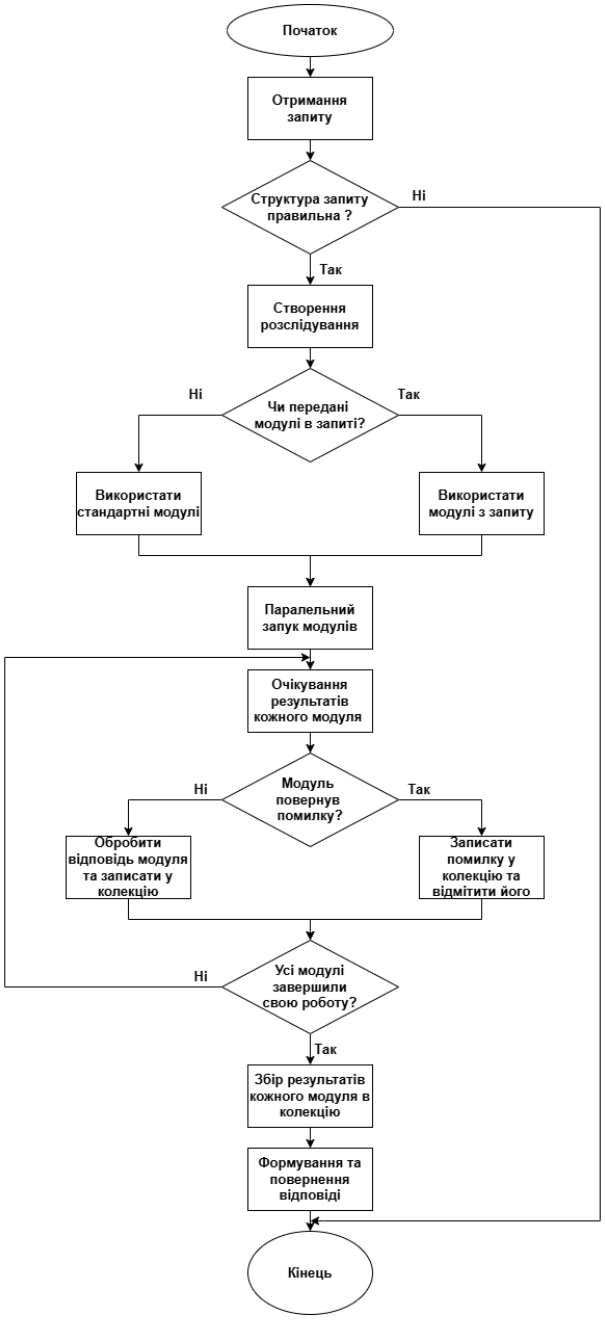


Рисунок 3.1 – Загальний алгоритм розслідування

Обробка виняткових ситуацій реалізована на двох рівнях. На рівні окремого модуля функція-обгортка _run_module перехоплює будь-які винятки через блок try/except та повертає словник із полем error замість нормального результату. Це гарантує що збій одного модуля не перериває виконання всіх інших модулів і не призводить до повної відмови розслідування. На рівні API rate limiter через бібліотеку slowapi обмежує частоту запитів десятьма за хвилину та автоматично повертає HTTP статус 429 при перевищенні що захищає систему від надмірного навантаження.

Алгоритм паралельного виконання модулів збору вирішує ключову проблему продуктивності підсистеми збору. При послідовному виконанні восьми модулів із середнім часом відповіді три секунди кожен загальний час розслідування становив би двадцять чотири секунди що є неприйнятним для інтерактивного аналітичного інструменту. Паралельне виконання через asyncio.gather скорочує цей час до часу виконання найповільнішого модуля, зазвичай три-п’ять секунд. Алгоритм паралельного виконання складається з трьох послідовних фаз.

У першій фазі формується словник задач. Для кожного імені модуля із переліку активних модулів викликається функція _run_module що повертає корутину без її негайного виконання. Ключем словника є символьне ім’я модуля, значенням – корутина що очікує на виконання. Розділення списку імен та списку корутин на два окремих списки module_names та coroutines необхідне для подальшого зіставлення результатів із назвами модулів.

У другій фазі всі корутини передаються до asyncio.gather як розпакований список.

У третій фазі алгоритм зіставляє результати з іменами модулів через паралельну ітерацію по module_names та gathered_results через функцію zip. Для кожного результату виконується перевірка типу через isinstance(result, Exception), при позитивному результаті формується словник помилки із полями error та module, при негативному, зберігається оригінальний результат модуля.

Фінальний словник module_results передається до наступного кроку

алгоритму для формування графу сутностей. Функція-обгортка _run_module виконує три допоміжні функції: пошук функції-обробника у реєстрі модулів за символьним іменем, логування початку та завершення виконання із вимірюванням часу через time.time(), та перехоплення виняткових ситуацій через блок try/except із формуванням стандартизованої відповіді про помилку.

Реєстр модулів реалізований як словник що відображає символьні імена на відповідні функції-обробники: dns на dns_lookup, whois на whois_lookup, web_scraper на web_scrape, тощо. При надходженні невідомого імені модуля функція повертає словник помилки без генерації винятку. Алгоритм нормалізації вхідних даних реалізований у функції _validate_target та застосовується до значення цільового об'єкта відповідно до його типу перед передачею у модулі збору.

Для типу email значення приводиться до нижнього регістру через метод lower та перевіряється регулярним виразом r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$' – при невідповідності виводиться попередження у журнал без блокування виконання. Для типу phone видаляються пробіли, дефіси, дужки та крапки через регулярний вираз r'[\s\-\(\)\.]', після чого значення доповнюється префіксом "+" якщо він відсутній – що приводить номер до міжнародного формату E.164.

Для username видаляється провідний символ "@" через метод lstrip та застосовується регулярний вираз для видалення символів що не є алфавітно-цифровими, підкресленнями, дефісами або крапками. Для типу domain значення приводиться до нижнього регістру та видаляється протокол через регулярний вираз r'^https?://' із наступним видаленням завершального символу "/".

Автоматичне формування графу сутностей є ключовим алгоритмом що забезпечує перетворення неструктурованих результатів модулів збору на зв'язану графову структуру придатну для кореляційного аналізу.

Алгоритм реалізований у функції _create_entities_from_result файлу server/routes.py та викликається послідовно для результату кожного модуля після завершення паралельного збору.

На першому кроці алгоритм виконує попередню перевірку результату: якщо словник результату є порожнім або містить поле error – функція завершується без жодних дій. Це запобігає створенню порожніх або некоректних вузлів у графі при збоях модулів та забезпечує цілісність графової структури. Центральним механізмом алгоритму є дедуплікація сутностей що реалізована у методі add_entity класу EntityStore. Перед виконанням операції INSERT метод виконує SELECT-запит за комбінацією полів type та value що є унікальним ключем сутності. При виявленні існуючого запису метод оновлює лише поле data шляхом злиття нових атрибутів із наявними через словникову операцію update та повертає існуючу сутність без створення дублікату. При відсутності існуючого запису виконується INSERT нового запису із генерацією восьмисимвольного UUID-ідентифікатора. Завдяки цьому механізму один email виявлений одночасно модулями web_scraper та whois буде представлений єдиним вузлом у графі із ребрами від обох джерел – що є коректним графовим представленням кореляції між джерелами.

Логіка формування графу є специфічною для кожного модуля та відображає семантичну природу даних що він повертає. Для модуля dns алгоритм ітерує по списку resolved_ips та для кожної IP-адреси створює вузол типу ip із ребром resolves_to від головного вузла домену, для кожного MX-запису із records.MX створює вузол типу domain із ребром mail_handled_by, для кожного NS-запису із records.NS створює вузол типу nameserver із ребром nameserver.

Для модуля whois алгоритм витягує поля registrar, org та emails із підсловника data та створює відповідні вузли типів registrar, organization та email із ребрами registered_via, owned_by та contact_email.

Для модуля username алгоритм ітерує по списку found_on та для кожного знайденого профілю створює вузол типу social_profile із ребром has_profile, додатково аналізує поля extracted_emails та extracted_usernames із метаданих профілю та створює вторинні вузли із ребрами mentioned_in_bio. Аналогічна логіка витягування вторинних сутностей із біографії профілю реалізована для модулів social_scan_email, social_scan_phone, social_scan_username та

person_search.

Для модуля breach_check кожен виявлений витік даних стає вузлом типу breach із ребром exposed_in та атрибутами дати витоку та переліку скомпрометованих типів даних. Метод add_edge EntityStore також реалізує дедуплікацію ребер: перед INSERT виконується перевірка існуючого ребра за комбінацією source_id, target_id та relation. При виявленні існуючого ребра метод повертає його без повторного створення що запобігає дублюванню зв’язків при повторних запусках розслідувань щодо того самого цільового об’єкта.

Алгоритм кореляційного аналізу реалізований у класі CorrelationEngine файлу server/correlation.py та використовує бібліотеку NetworkX для побудови та аналізу графу зв’язків між виявленими сутностями, рисунок 3.2.

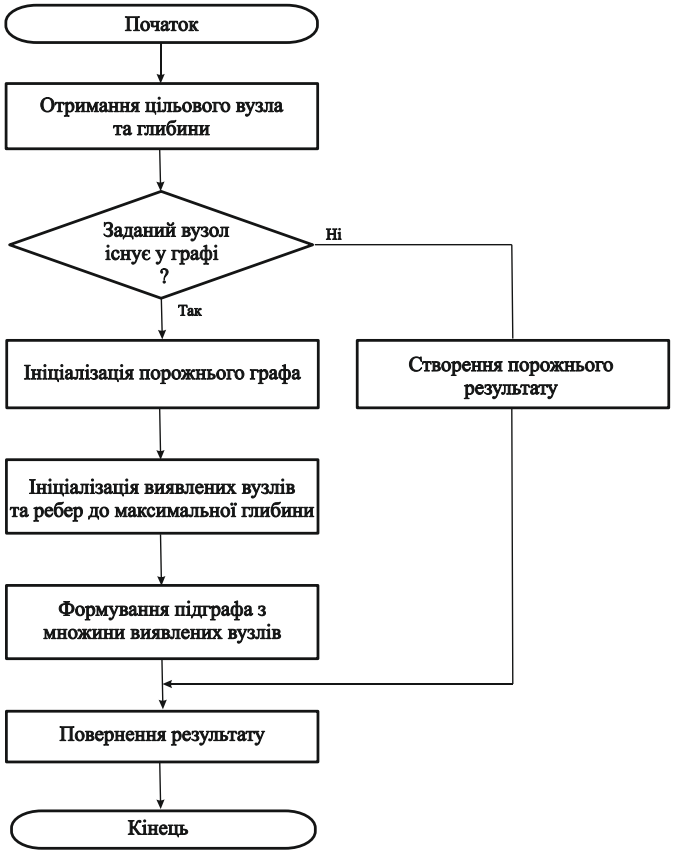


Рисунок 3.2 – Алгоритм побудови графу кореляційного аналізу

Базовим методом що використовується всіма аналітичними алгоритмами є build_graph що будує об’єкт nx.Graph із поточного стану EntityStore при кожному

виклику. Метод отримує повний знімок графу через store.get_graph() що повертає словник із списками вузлів та ребер, після чого додає вузли до NetworkX-графу із атрибутами типу, значення та мітки, а ребра, із атрибутом типу відношення та вагою.

Алгоритм виявлення крос-розслідувальних зв’язків find_cross_investigation_links групує всі вузли графу за нормалізованим ключем у форматі type:value де значення приводиться до нижнього регістру та обрізається від зайвих пробілів. Групування виконується лише для вузлів типів email, username, phone та person, ідентифікаторів що можуть однозначно вказувати на одного суб’єкта у різних розслідуваннях. Групи де кількість вузлів перевищує один включаються до результату із обчисленим показником достовірності за формулою min(0.6 + len(nodes) * 0.15, 0.95) – що забезпечує вищу впевненість при більшій кількості збігів із верхньою межею 0.95.

Алгоритм побудови досьє суб’єкта реалізований у функції build_person_dossier файлу server/modules/person_dossier_builder.py, він будує консолідований профіль фізичної особи із множини паралельно запущених модулів збору, рисунок 3.3.

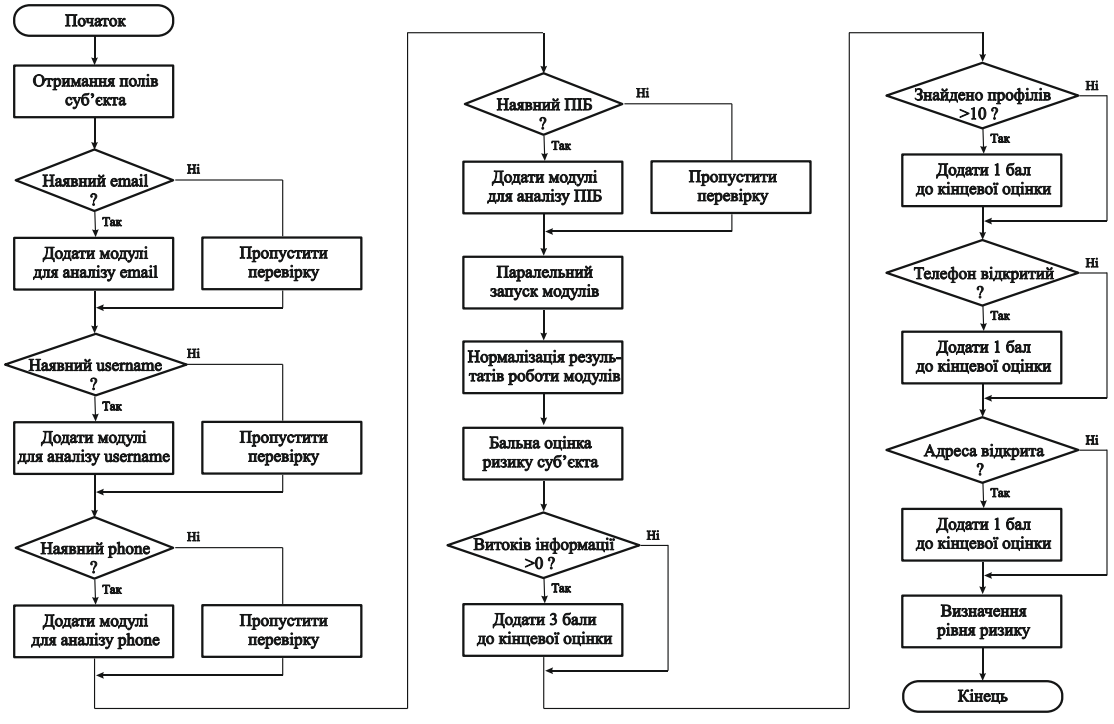


Рисунок 3.3 – Алгоритм побудови досьє

На першому кроці алгоритм отримує словник полів суб'єкта fields що може містити довільну комбінацію ідентифікаторів: ім'я, прізвище, email-адреси, телефонні номери, імена користувача на різних платформах. Залежно від набору переданих полів алгоритм формує перелік релевантних модулів збору: при наявності email запускаються email_analyzer, social_scan_email та breach_check, при наявності username – username_checker та social_scan_username, при наявності phone – phone_lookup та social_scan_phone, при наявності імені – person_search та name_search.

На другому кроці модулі запускаються паралельно через механізм asyncio.gather.

На третьому кроці алгоритм нормалізації агрегує результати всіх модулів у структуру NormalizedPerson. Нормалізація виконується по шести категоріях атрибутів: names, emails, phones, usernames, locations та social_profiles. Для кожної категорії алгоритм ітерує по результатах відповідних модулів та витягує значення із різних полів залежно від типу модуля – наприклад email-адреси можуть надходити із модулів email_analyzer, social_scan_email та breach_check.

Четвертий крок реалізує дедуплікацію в межах кожної категорії атрибутів: для кожного нового елементу перевіряється чи не існує вже елемент із таким самим значенням поля value у відповідній колекції. При виявленні дублікату зберігається елемент із вищим показником confidence або більш інформативним полем source. Це забезпечує що один email виявлений через два різних модулі буде представлений єдиним записом у профілі.

П'ятий крок реалізує алгоритм бальної оцінки ризику суб'єкта. Алгоритм послідовно перевіряє набір індикаторів небезпеки та накопичує бальний рахунок: наявність записів у базах витоків даних додає два-три бали залежно від кількості виявлених витоків, більше десяти профілів у соціальних мережах додає один бал як індикатор широкої цифрової присутності, наявність телефону у відкритому доступі додає один бал, наявність персональної адреси у відкритому доступі додає один бал. Кожен спрацьований індикатор фіксується як текстовий опис у колекції risk_factors. На основі накопиченого балу визначається рівень ризику: від нуля до

двох балів – low, від трьох до п’яти – medium, від шести до дев’яти – high, десять і більше – critical.

3.2 Реалізація серверної частини OSINT-системи

Серверна частина системи реалізована як FastAPI-застосунок у файлі server/app.py. До застосунку підключається CORS middleware бібліотеки fastapi.middleware.cors із параметрами “allow_origins=["*"]”, “allow_credentials=True”, “allow_methods=["*"]” та “allow_headers=["*"]” що забезпечує крос-доменні запити від клієнтського React-додатку розгорнутого на порту 5173 до серверного API на порту 8000.

Rate limiter інтегрується через бібліотеку slowapi: об’єкт limiter підключається до стану застосунку через “app.state.limiter”, а обробник помилок “_rate_limit_exceeded_handler” реєструється для виключень типу RateLimitExceeded що автоматично генерує HTTP-відповідь 429 Too Many Requests при перевищенні визначених лімітів.

Усі API-ендпоінти підключаються через роутер із префіксом /api через виклик “app.include_router(router, prefix=”/api”)” що забезпечує єдиний простір імен для всіх ендпоінтів системи.

Подія shutdown реалізує коректне завершення роботи: при наявності модуля безголового браузера Playwright викликається функція “shutdown_browser()” для звільнення ресурсів браузерного процесу.

При запуску сервера подія startup ініціалізує базу даних SQLite, яка створює необхідні таблиці та індекси.. Усі API-ендпоінти підключаються через роутер із префіксом /api, рисунок 3.4.

Архітектурне рішення розділити точку входу app.py та бізнес-логіку маршрутів routes.py відповідає принципу єдиної відповідальності та спрощує тестування окремих компонентів.

Рисунок 3.4 – Реалізація FastAPI застосунку server/app.py

Підсистема збору реалізована у файлі server/routes.py та складається з двох ключових компонентів: функції _run_module() для виконання окремого модуля та механізму паралельного запуску через asyncio.gather.

Функція _run_module() реалізує єдиний уніфікований інтерфейс виконання будь-якого OSINT-модуля. Вона містить централізований реєстр із сімнадцяти модулів у вигляді словника де ключем є символьне ім’я модуля а значенням – відповідна асинхронна функція-обробник: dns, whois, ip_geo, email, username, web_scraper, shodan, threat_intel, metadata, subdomain_enum, cert_transparency,

tech_fingerprint, social_scan_email, social_scan_phone, social_scan_username, person_search та person_dossier_builder.

При виклику функція фіксує час початку виконання через time.time(), виконує корутину модуля та після завершення обчислює і логує час виконання у форматі “[module_name] Completed in X.Xs”, де X.X це час виконання. При виникненні будь-якого винятку всередині модуля функція перехоплює його через блок “except Exception as e”, логує повідомлення про помилку із часом виконання та повертає стандартизований словник помилки із полями error та module, що ізолює збій одного модуля та гарантує продовження виконання всіх інших.

Паралельне виконання реалізоване в ендпоінті /api/investigate: для кожного модуля із переліку all_modules через виклик _run_module(name, target, target_type) створюється корутина без її негайного запуску, після чого всі корутини передаються до asyncio.gather(*coroutines, return_exceptions=True) як розпакований список. Без параметру “return_exceptions=True” перший виняток у будь-якій корутині негайно скасовував би виконання всіх інших через механізм скасування asyncio, тоді як із цим параметром виняток повертається як звичайний результат у вигляді об’єкта Exception.

Після завершення asyncio.gather алгоритм ітерує по парах (module_name, result) через функцію zip та формує словник module_results де кожному імені модуля відповідає або структурований результат або словник помилки.

При тестуванні розслідування домену ‘gmail.com’ із запуском 8 модулів загальний час виконання становив 17.4 секунди при паралельному виконанні. Без механізму asyncio.gather цей час становив би орієнтовно 40–60 секунд.

Модуль dns_lookup реалізований у файлі server/modules/dns_lookup.py та забезпечує комплексний DNS-аналіз цільового домену. Використовує бібліотеку dnspython для резолвінгу семи типів DNS-записів: A, AAAA, MX, NS, TXT, CNAME та SOA. Для кожної розв’язаної IP-адреси виконується зворотний DNS-запит через dns.reversename. У випадку якщо стандартний DNS не повернув A-записів, модуль використовує fallback через socket.gethostbyname.

Кожен тип DNS-запису обробляється в окремому блоці try/except що

забезпечує стійкість модуля до часткових збоїв.

Для записів типу A та AAAA результатом є список IP-адрес що зберігається у полі resolved_ips – ці адреси потім автоматично перетворюються на вузли типу ip у графі сутностей через функцію _create_entities_from_result. Для MX-записів модуль витягує ім’я поштового сервера та його пріоритет, для NS-записів – імена серверів імен, для TXT-записів – текстовий вміст що нерідко містить записи SPF, DKIM та DMARC корисні для аналізу поштової інфраструктури цілі.

Опісля, для кожної знайденої IP-адреси виконується зворотний DNS-запит через dns.reversename.from_address() із подальшим резолвінгом PTR-запису що дозволяє встановити доменне ім’я пов’язане із IP-адресою та виявити хостингового провайдера або CDN. У випадку якщо стандартний DNS не повернув A-записів модуль використовує резервний метод через “socket.gethostbyname()”, що забезпечує отримання хоча б однієї IP-адреси навіть при нетипових DNS-конфігураціях.

Результат модуля містить поля target, resolved_ips, reverse_dns та records, де records є словником із ключами типів записів та відповідними значеннями.

Модуль web_scraper реалізований у файлі server/modules/web_scraper.py та здійснює комплексний аналіз веб-сторінки, рисунок 3.5. Використовує асинхронну бібліотеку aiohttp для HTTP-запитів із імітацією User-Agent браузера Chrome та бібліотеку BeautifulSoup4 для парсингу HTML.

Модуль витягує комплексний набір даних із веб-сторінки цільового ресурсу. Заголовок сторінки отримується через пошук тегу <title> та мета-теги через словник, де ключем є атрибут name тегу <meta> – що дозволяє отримати опис сторінки, ключові слова та Open Graph метадані.

Email-адреси виявляються через регулярний вираз що охоплює стандартні формати адрес як у текстовому вмісті сторінки так і у значеннях атрибутів href посилань типу “mailto:”. Телефонні номери виявляються через окремий регулярний вираз що розпізнає формати із дужками, дефісами та пробілами. Зовнішні посилання збираються через перебір усіх тегів <a> із перевіркою що href починається із http та не містить домен цільового ресурсу – що відфільтровує

внутрішню навігацію та залишає лише зовнішні зв’язки.

Дедуплікація посилань виконується через перетворення списку на множину set(). Посилання на соціальні мережі виявляються через словник патернів де кожній платформі – Twitter, LinkedIn, Facebook, Instagram, YouTube, GitHub, Telegram – відповідає підрядок домену для перевірки.

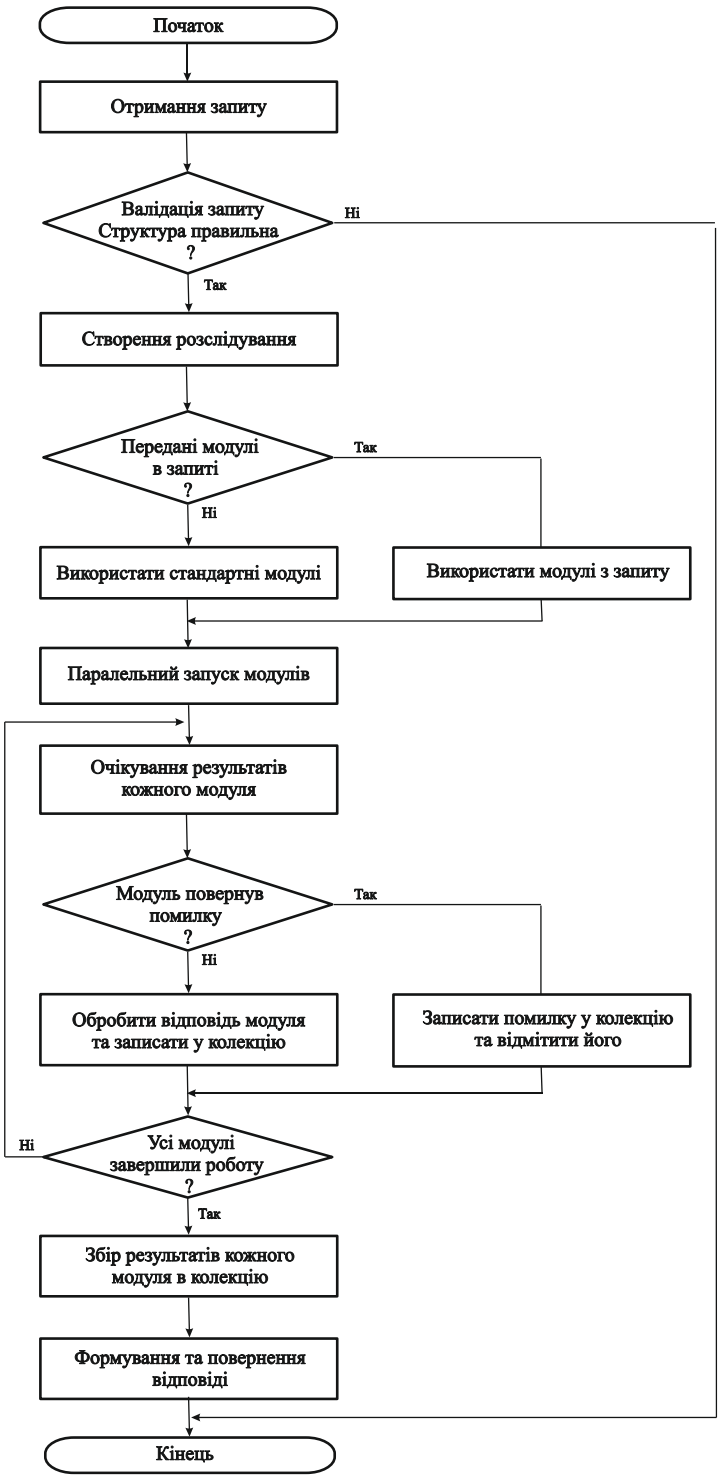


Рисунок 3.5 – Алгоритм реалізації модуля web_scraper

Функція _detect_technologies() аналізує HTTP-заголовки відповіді та вміст сторінки для виявлення технологічного стеку: перевіряє заголовок X-Powered-By, шукає характерні маркери фреймворків у HTML, наприклад wp-content для WordPress, Drupal.settings для Drupal, та аналізує заголовок Server для визначення веб-сервера.

Весь процес збору виконується асинхронно із заголовком User-Agent що імітує браузер Chrome для мінімізації відмов від захисту від ботів.

Підсистема зберігання реалізована у файлі server/entity_store.py та використовує SQLite як вбудовану реляційну базу даних. Вибір SQLite обґрунтований відсутністю потреби у мережевому сервері БД для аналітичного інструменту, що спрощує розгортання.

Функція init_db() виконується при кожному старті сервера через подію startup та створює чотири таблиці та шість індексів якщо вони ще не існують через директиву CREATE TABLE IF NOT EXISTS.

Перед створенням таблиць функція виконує налаштування WAL-режиму через PRAGMA journal_mode=WAL та вмикає підтримку зовнішніх ключів через PRAGMA foreign_keys=ON.

Таблиця entities зберігає вузли графу із такими полями: id (TEXT PRIMARY KEY) – восьмисимвольний UUID-ідентифікатор що генерується через str(uuid.uuid4())[:8], type (TEXT NOT NULL) – тип сутності із визначеного словника, value (TEXT NOT NULL) – значення що є ключем дедуплікації, label (TEXT) – відображувана мітка у інтерфейсі, data (TEXT DEFAULT ‘{}’) – довільні атрибути у форматі JSON що дозволяє зберігати специфічні для типу поля без зміни схеми таблиці, investigation_id (TEXT) – зовнішній ключ до таблиці investigations, а також created_at та updated_at.

На таблиці entities визначено три індекси: idx_entities_type по полю type, idx_entities_value по полю value та idx_entities_investigation по полю investigation_id вони забезпечують продуктивність запитів при фільтрації та пошуку.

Таблиця edges зберігає ребра графу із зовнішніми ключами source_id та

target_id що посилаються на таблицю entities із директивою ON DELETE CASCADE – що автоматично видаляє ребра при видаленні вузлів та гарантує структурну цілісність графу без додаткової логіки у застосунку. Поле relation зберігає семантичний тип зв'язку у текстовому форматі, поле weight із значенням за замовчуванням 1.0 зарезервоване для майбутніх алгоритмів зваженого аналізу. Таблиці investigations та dossiers зберігають метадані розслідувань із полем status, pending або completed, та консолідовані профілі суб'єктів із полем risk_score відповідно.

Таблиця edges зберігає ребра із зовнішніми ключами та директивою ON DELETE CASCADE. Таблиці investigations та dossiers зберігають метадані розслідувань та консолідовані профілі відповідно. Метод add_entity(), рисунок 3.6, реалізує дедуплікацію: перед створенням нової сутності виконується пошук за комбінацією 'type + value', і якщо сутність вже існує, оновлюються лише її метадані.

Рисунок 3.6 – Алгоритм реалізації методу add_entity

Кореляційний механізм реалізований у файлі server/correlation.py як клас CorrelationEngine, та використовує бібліотеку NetworkX для побудови та аналізу графу зв’язків між виявленими сутностями.

Метод build_graph() є базовим методом що викликається усіма аналітичними алгоритмами: він отримує повний знімок графу через store.get_graph() що повертає словник із списками вузлів та ребер, після чого послідовно додає вузли до nx.Graph через G.add_node(node_id, **attrs) із атрибутами типу, значення та мітки, та ребра через G.add_edge(src, tgt, relation=rel, weight=w).

Метод find_connections() реалізує BFS-обхід для знаходження сусідів у межах depth кроків та повертає підграф досяжних вузлів. Метод detect_anomalies() обчислює статистичні характеристики розподілу ступенів вузлів та виявляє чотири типи аномалій: hub_entity при degree > avg + 2*std, bridge_entity при betweenness > avg*3, isolated_risk для breach-вузлів із мінімальною кількістю зв’язків та orphan_cluster для ізольованих компонент із двох або менше вузлів.

Метод find_cross_investigation_links() виявляє сутності що зустрічаються у кількох незалежних розслідуваннях шляхом групування вузлів за нормалізованим ключем type:value та обчислює показник достовірності зв’язку за формулою min(0.6 + len(nodes) * 0.15, 0.95).

Усі методи класу будують граф заново при кожному виклику що гарантує актуальність аналітичних результатів відносно поточного стану сховища.

Головний компонент App.jsx реалізує бічну панель навігації із логотипом системи та мініатюрною статистикою, рисунок 3.7, кількість сутностей та зв’язків, яка оновлюється автоматично. Кожне вікно обгорнуте компонентом ErrorBoundary для ізоляції помилок рендерингу.

Компонент Investigation є основним робочим простором для ініціалізації OSINT-розслідувань. Він підтримує вісім типів цілей: домени, IP-адреси, email, імена користувачів, телефони, URL, файли та комплексний пошук по особі.

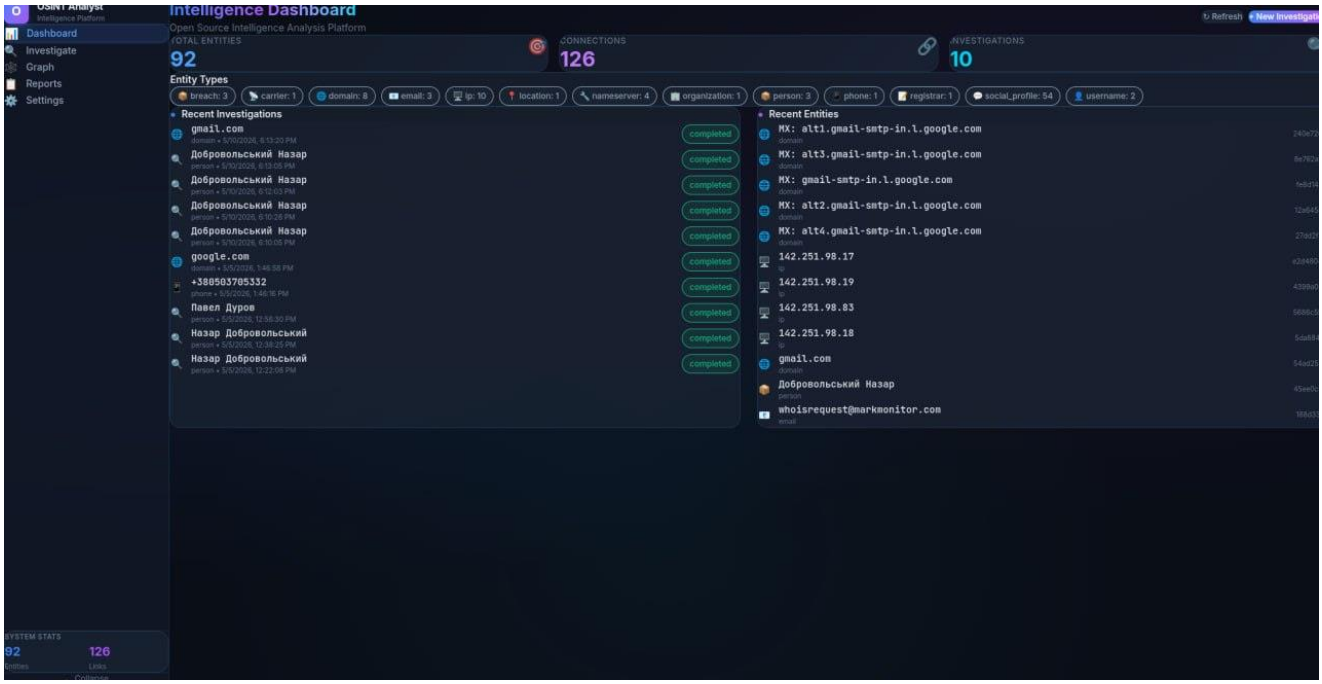


Рисунок 3.7 – Скріншот головного вікна Dashboard

Інтерфейс динамічно адаптується залежно від обраного типу цілі. Для комплексного розслідування суб'єкта, реалізовано розширену форму із можливістю додавання множинних ідентифікаторів: декілька email-адрес, номерів телефону, нікнеймів, зазначення локації та додаткового контексту.

Запуск збору генерує асинхронний запит до API-шлюзу. Для зручності аналітика компонент забезпечує інтерактивну індикацію процесу обробки, а після завершення виконання модулів, автоматично рендерить досьє суб'єкта або деталізовані результати технічного аналізу.

Компонент Graph.jsx використовує бібліотеку vis-network для інтерактивної візуалізації графу сутностей. Для кожного з 14 типів сутностей визначено унікальний колір та форму: домени відображаються синіми колами, IP-адреси – зеленими ромбами, email – жовтими квадратами, імена користувачів – фіолетовими трикутниками, рисунок 3.9.

Зв'язки між сутностями візуалізуються напрямленими згладженими кривими з маркуванням типу відношення. Інтерактивна складова охоплює масштабування полотна, виведення ідентифікаторів та значень в інформаційну панель за одинарним кліком, а також виклик тригера переходу до повного досьє за подвійним кліком.

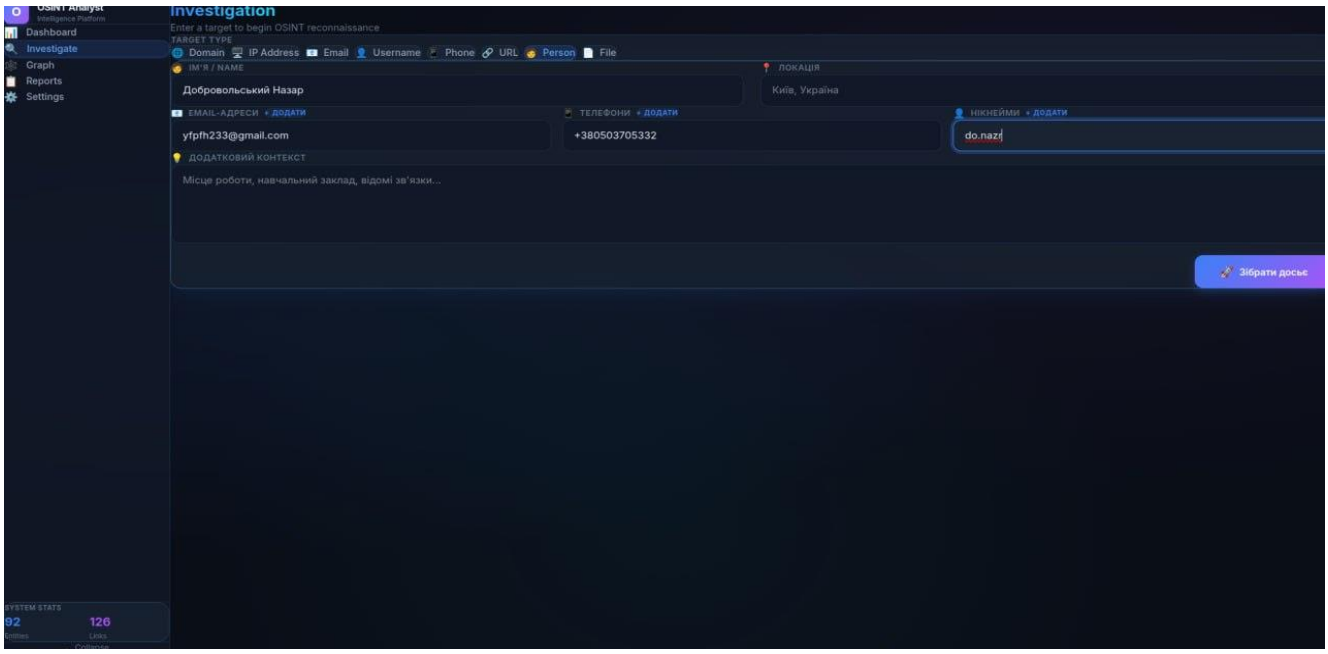


Рисунок 3.8 – Скріншот вікна Investigation із заповненою формою пошуку

Система фільтрації забезпечує ізоляцію підграфів за ідентифікатором конкретного розслідування або за обраним типом сутності у константний час. Панель керування надає інструменти гарячого перезавантаження масиву даних, центрування графу та перманентного очищення бази даних із запобіжним підтвердженням.

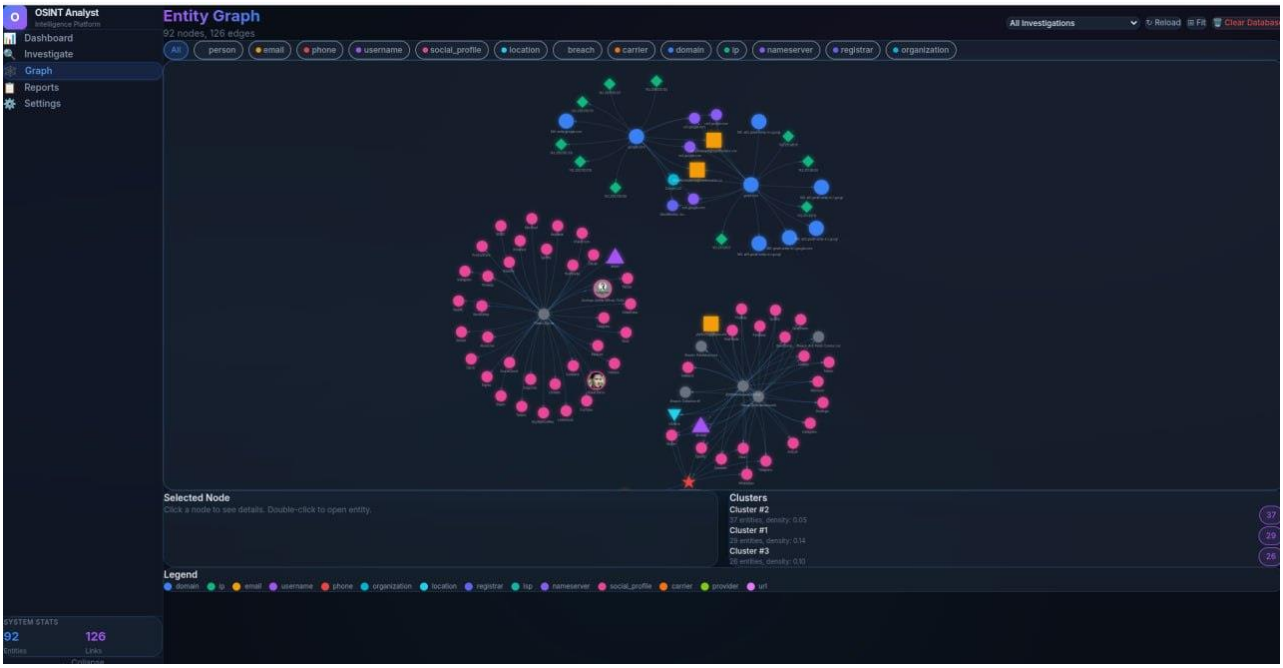


Рисунок 3.9 – Скріншот графу зв’язків між сутностями

Компонент DossierReport.jsx є збирає результати паралельного збору даних та відображає комплексне досьє цільового суб’єкта, організоване у сім функціональних вкладок, рисунок 3.10.

Інтерфейс компонента оснащений панеллю керування, що підтримує генерацію розширеного HTML-звіту та прямий експорт досьє у формат PDF для подальшого використання. Базова інформація акумулюється у профільній картці, яка динамічно підвантажує аватар із виявлених соціальних мереж та автоматично присвоює об'єкту маркер рівня ризику із відповідною індикацією: зелений для низького, жовтий для середнього, червоний для високого та пульсуючий червоний для критичного рівня.

Вкладка Summary містить профільну картку із аватаром суб’єкта що завантажується із знайдених соціальних профілів, badge рівня ризику із кольоровою індикацією, зелений для low, жовтий для medium, помаранчевий для high, червоний для critical, зведену статистику кількості акаунтів, витоків даних та виявлених кореляцій, а також перелік всіх виявлених ідентифікаторів суб’єкта: імен, email-адрес, телефонів та username.

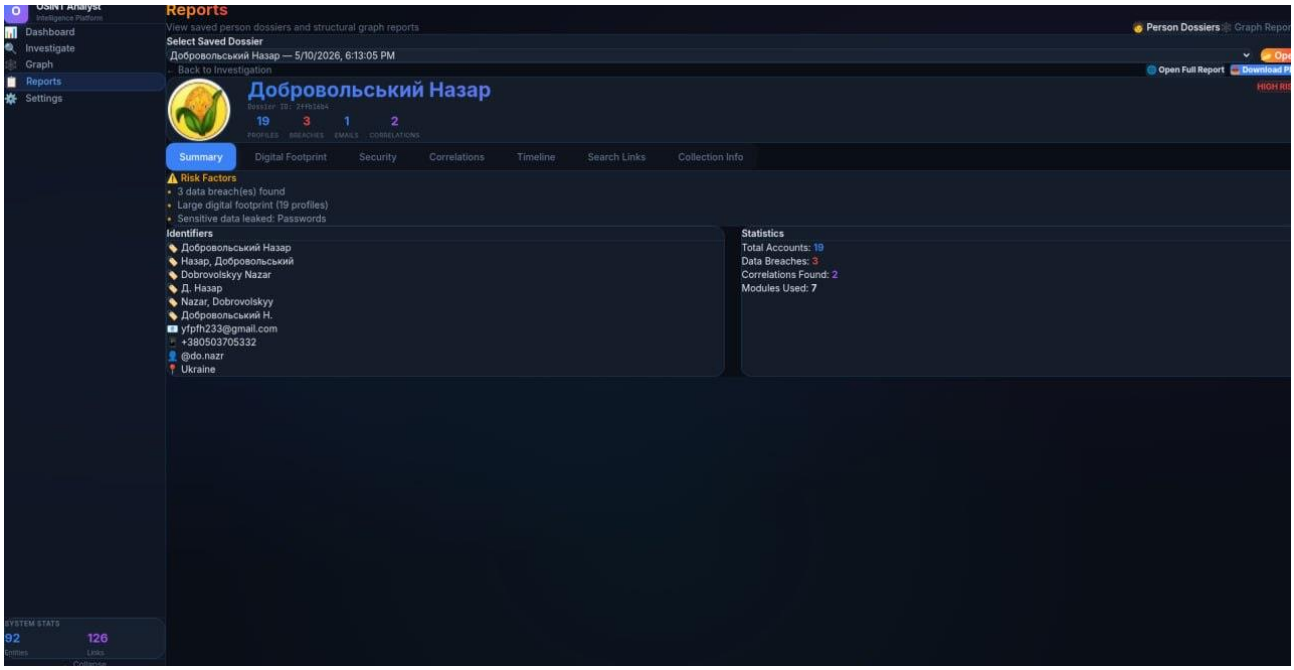


Рисунок 3.10 – Скріншот досьє суб’єкта

Вкладка Digital Footprint відображає таблицю знайдених профілів у соціальних мережах та онлайн-платформах із полями платформи, URL, відображуваного імені та показника достовірності. Вкладка Security відображає перелік виявлених витоків даних із назвою витоку, датою та переліком скомпрометованих типів даних, а також перелік факторів ризику risk_factors що обґрунтовують обчислений рівень. Вкладка Correlations відображає виявлені крос-джерельні зв’язки між різними ідентифікаторами суб’єкта.

Вкладка Timeline відображає хронологічну шкалу подій пов’язаних із суб’єктом із типами info та danger. Вкладка Collection Info відображає метадані процесу збору: які модулі були використані, які завершились із помилкою та час збору. Кнопки "Open Full Report" та "Graph Dossier" у шапці компонента ініціюють відкриття повного HTML-звіту у новій вкладці браузера та перехід до графового представлення досьє відповідно.

3.3 Налаштування роботи OSINT-системи

Розгортання системи реалізоване через Docker Compose – інструмент оркестрації контейнерів, що дозволяє описати повну конфігурацію багатоконтейнерного застосунку у декларативному файлі. Система складається з двох сервісів: backend на базі Python 3.11 із FastAPI та frontend на базі Node.js 20 із Vite.

Backend-контейнер, рисунок 3.11 базується на базовому образі python:3.11-slim, що дозволяє суттєво зменшити кінцевий обсяг дискового простору та оптимізувати швидкість розгортання в ізольованому середовищі. Оскільки базове середовище за замовчуванням позбавлене компонентів графічної підсистеми, виникає потреба у додатковому ручному завантаженні залежностей, які є необхідними для функціонування Playwright Chromium.

Для стабільного запуску та безперебійної роботи браузера у безголовому headless режимі у Dockerfile.backend виконується попереднє оновлення індексів

пакетів та примусове встановлення розширеного переліку графічних залежностей, системних шрифтів, а також бібліотек для рендерингу та міжпроцесної взаємодії, зокрема libglib, libnss, libatk, libpango та libcairo.

Після підготовки середовища всередині контейнера завантажуються залежності Python, які прописані у файлі вимог requirements.txt, такі як: веб-фреймворк FastAPI, асинхронний сервер Uvicorn та інші спеціалізовані модулі для мережевого аналізу та збору інформації. Інсталяція бібліотек виконується через менеджер пакетів pip із відключенням локального кешування, що запобігає збереженню тимчасових завантажених архівів та надає максимальну чистоту і компактність сформованих шарів Docker-образу.

Dockerfile.backend

```
FROM python:3.11-slim

WORKDIR /app

# System deps required by playwright chromium
RUN apt-get update && apt-get install -y --no-install-recommends \
    libglib2.0-0 libnss3 libnspr4 libdbus-1-3 libatk1.0-0 \
    libatk-bridge2.0-0 libcups2 libdrm2 libxkbcommon0 libxcomposite1 \
    libxdamage1 libxfixes3 libxrandr2 libgbm1 libpango-1.0-0 \
    libcairo2 libasound2 libatspi2.0-0 libxshmfence1 \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Install Chromium for Playwright
RUN playwright install chromium

COPY server/ ./server/
RUN mkdir -p /app/data

EXPOSE 8000

CMD ["uvicorn", "server.app:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

Рисунок 3.11 – Конфігурація Dockerfile.backend

Завершальним етапом конфігурування серверного середовища є безпосереднє розгортання виконавчих файлів браузера Chromium через внутрішні

команди утиліти Playwright, а також створення необхідної інфраструктурної структури внутрішніх директорій для тривалого збереження результатів роботи.

Frontend-контейнер побудований на образі node:20-slim з Vite dev-сервером. Використання даної модифікації дозволяє виключити надлишкові системні пакети та допоміжні інструменти адміністрування, що суттєво зменшує підсумкову вагу Docker-образу, оптимізує використання дискового простору та скорочує час розгортання ініціалізації сервісу в ізольованому інфраструктурному середовищі.

Внутрішня конфігурація інструкцій Dockerfile передбачає призначення робочої директорії та покрокове попереднє копіювання файлів проектних залежностей перед перенесенням усього масиву вихідного коду клієнтської частини, рисунок 3.12.

Рисунок 3.12 – Конфігурація Dockerfile.frontend

Такий розподіл операцій побудови образу забезпечує ефективне задіяння вбудованого механізму кешування шарів Docker, оскільки повторна інсталяція сторонніх модулів через менеджер пакетів запускається виключно у випадку безпосередньої модифікації конфігурації залежностей, а не при поточній зміні візуальних компонентів інтерфейсу.

Для забезпечення мережевої доступності у конфігурації контейнера декларується експорт порту 5173.

Фінальна директива запуску ініціює сервер Vite із передаванням параметра прив’язки до нульової мережевої адреси, що змушує внутрішній процес Node.js прослуховувати всі наявні мережеві інтерфейси всередині ізольованого контейнера і гарантує коректну маршрутизацію вхідного трафіку з хост-машини

Для підтримки режиму автоматичного перенесення змін у реальному часі без необхідності повторного тривалого збирання інфраструктурних образів або повного перезапуску сервісів налаштовано динамічне зв’язування файлових систем.

Для серверної частини застосовується монтування локального каталогу з вихідним кодом сервера у внутрішню цільову директорію контейнера, що дозволяє вбудованому асинхронному серверу миттєво фіксувати будь-які модифікації у файлах бізнес-логіки, модулях збору даних чи алгоритмах кореляції та застосовувати їх без переривання поточних процесів.

Аналогічний підхід динамічного відображення файлів використано і для клієнтського компонента системи, де локальна папка з вихідними кодами зв’язується з робочим простором контейнера Node.js, завдяки чому інструмент швидкого збирання Vite забезпечує миттєву заміну модулів у браузері розробника при зміні візуальних елементів або логіки.

Окрему увагу приділено підсистемі збереження даних, оскільки за замовчуванням внутрішня файлова система контейнерів повністю очищується при їхній зупинці або перезбиранні.

Для забезпечення властивості системи зберігати розвідувальної інформації та результатів аналізу локальний файл реляційної бази даних SQLite винесено за межі ізольованого середовища виконання контейнера шляхом монтування локального каталогу даних хост-машини у внутрішню цільову папку сервера, що повністю нівелює ризик втрати або пошкодження накопичених сутностей, зв’язків, системних логів та згенерованих досьє суб’єктів незалежно від етапів життєвого циклу сервісів.

Маршрутизація трафіку та взаємодія між розгорнутими компонентами організована через механізм явного прокидання портів на інтерфейси хост-машини. Прикладний програмний інтерфейс серверної частини стає доступним для обробки зовнішніх асинхронних запитів через виділений мережевий порт 8000, тоді як інтерфейсна частина користувача, яка відповідає за візуалізацію аналітичних панелей та взаємодію з інтерактивним графом, функціонує та приймає вхідні з'єднання на порту 5173.

Повна інфраструктурна взаємозв'язку між зазначеними сервісами, конфігурація їхніх мережевих інтерфейсів, логіка проксіювання та особливості розподілу дискового простору для тривалого збереження інформації детально відображені на рисунку 3.13.

docker-compose.yml

```yaml
version: "3.9"

services:
  backend:
    build:
      context: .
      dockerfile: Dockerfile.backend
    ports:
      - "8000:8000"
    volumes:
      - ./server:/app/server
      - ./data:/app/data
    environment:
      - PYTHONUNBUFFERED=1
    restart: unless-stopped

  frontend:
    build:
      context: .
      dockerfile: Dockerfile.frontend
    ports:
      - "5173:5173"
    volumes:
      - ./src:/app/src
      - ./index.html:/app/index.html
    depends_on:
      - backend
    restart: unless-stopped
```

Рисунок 3.13 – Конфігурація docker-compose.yml

Керування розгорнутою інфраструктурою OSINT-інструменту здійснюється за допомогою стандартного CLI-інтерфейсу Docker Compose. Для адміністрування системи визначens такі базові команди: Використовується при першому запуск або після внесення змін у конфігураційні файли, наприклад, requirements.txt або Dockerfile, команда “docker compose up --build”, ініціюючи повне перезбирання шарів образу. Команда “docker compose up -d” запускає всі сервіси у фоні, звільняючи термінал для подальшої роботи користувача. Команда “docker-compose logs -f backend” виводить логи серверної частини в реальному часі, що є може бути важливим для відстеження процесу виконання асинхронних OSINT-задач та фіксації можливих помилок. Команда “docker compose down” зупиняє роботу всіх активованих контейнерів, ізольованих віртуальних мереж, призупиняючи процеси без втрати даних, збережених у змонтованих томах.

Усі задіяні OSINT-модулі відпрацювали успішно та з високою швидкістю, таблиця 3.1, виконавши комплексний збір даних по цільовому домену, електронній пошті, нікнейму та номеру телефону.

Модуль username_checker виявив 14 активних акаунтів на різних платформах, включаючи GitHub та Telegram. Аналізатор breach_check зафіксував 2 критичні витоки паролів, пов’язані із вказаною електронною адресою. Компонент phone_lookup ідентифікував прив’язку номера телефону до оператора Vodafone та регіону реєстрації. Інструмент social_scanner знайшов цифрові сліди суб'єкта у відкритих реєстрах та професійних спільнотах. Усі отримані вузли та зв'язки між ними були миттєво відображені у веб-інтерфейсі системи. Інтерактивна візуалізація дозволила аналітику чітко простежити кореляцію між нікнеймом та реальними даними особи.

Загальний час формування повного досьє за другим сценарієм склав 22.8 секунди, таблиця 3.2. Висока швидкість обробки підтвердила ефективність архітектури паралельного запуску ізольованих контейнерів Docker.Результати тестування довели спроможність платформи автоматизувати рутинні операції збору первинної інформації.Розроблений OSINT-інструмент продемонстрував стабільність роботи під навантаженням та високу точність аналізу.

Таблиця 3.1 – Результати функціонального тестування модулів

Модуль	Ціль	Результат	Час, с
dns_lookup	gmail.com	7 типів записів, 4 IP-адреси, 5 MX-серверів, 4 NS-сервери	1.2
whois_lookup	gmail.com	Реєстратор MarkMonitor, організація Google LLC	0.8
web_scraper	gmail.com	Заголовок сторінки, meta-теги, 0 email, визначено технології	2.1
tech_fingerprint	gmail.com	Виявлено Google Analytics, reCAPTCHA	1.5
subdomain_enum	gmail.com	Перелік піддоменів	3.2
metadata_extractor	gmail.com	Метадані HTTP-відповіді	1.1
social_scanner	Тестова пошта	19 зареєстрованих сервісів, метадані профілів	4.5
breach_check	Тестова пошта	3 витоки даних виявлено	1.8
username_checker	Тестова нікнейм	Профілі на платформах з аватарами та біографіями	5.1
phone_lookup	Тестовий номер телефону	Оператор Vodafone, країна Ukraine	0.3
correlation	Результат граф з 92 вузлами	3 кластери, 6 аномалій, centrality-аналіз	0.2

Зібрана розвідувальна інформація, включаючи витоки даних, технічну інфраструктуру та соціальні сліди, була ефективно зведена у кореляційний граф для фінального аналізу.

Для комплексної демонстрації роботи системи проведено два сценарії тестування.

При розслідуванні домену ‘gmail.com’ система паралельно запустила 8 OSINT-модулів. Загальний час виконання становив 17.4 секунди. Було виявлено 4 IP-адреси, 5 MX-серверів, 4 NS-сервери, реєстратора MarkMonitor та організацію Google LLC. Усі виявлені сутності автоматично додані до графу із відповідними зв’язками.

При побудові досьє суб’єкта “Добровольський Назар” (email: yfpfh233@gmail.com, username: do.nazr, телефон: +380503705332), рисунок 3.14, система запустила модулі social_scanner, breach_check, username_checker, phone_lookup та name_resolver паралельно.

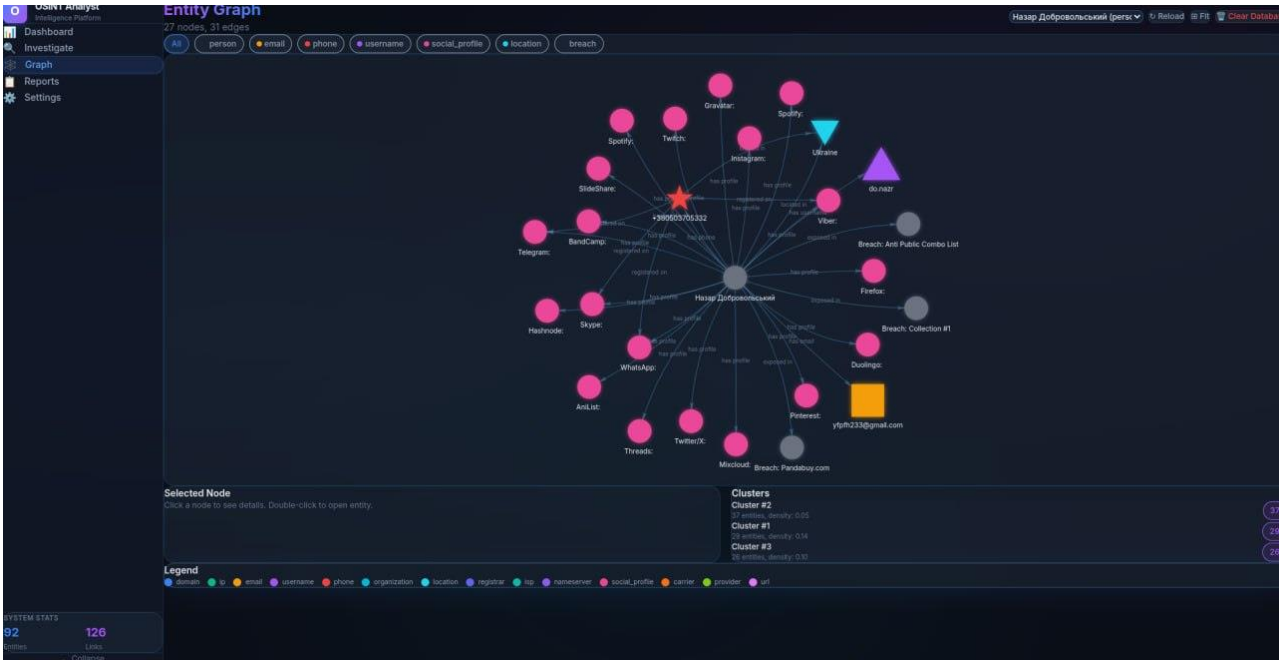


Рисунок 3.14 – Графова візуалізація досьє

Час виконання – 7.3 секунди. Результати: виявлено 19 соціальних профілів на різних платформах, знайдено 3 витоки даних із компрометованими класами даних, визначено рівень ризику: HIGH, побудовано 6 варіантів імені

(транслітерація, скорочення), виявлено локацію: Ukraine, отримано аватар через Gravatar.

3.4 Висновки

У третьому розділі було здійснено повну програмну реалізацію системи аналізу та кореляції відкритих джерел інформації з використанням OSINT-технологій. Розроблено серверну частину на базі FastAPI із 22 OSINT-модулями для збору даних з різнорідних джерел, підсистему зберігання EntityStore на базі SQLite із механізмами дедуплікації та індексації, кореляційний механізм CorrelationEngine на базі бібліотеки NetworkX для аналізу графу зв'язків, та інтерактивний веб-інтерфейс на React із візуалізацією графу через vis-network.

Система складається з підсистеми збору: dns_lookup, web_scraper, social_scanner, breach_check, username_checker, phone_lookup та 16 інших модулів, підсистеми зберігання із 4 таблиці SQLite з 6 індексами та автоматичною дедуплікацією, підсистеми кореляції із кластеризацією через connected_components, centrality-аналіз, виявлення 4 типів аномалій, пошук cross-investigation links, та клієнтської частини із 5 вікон інтерфейсу: Dashboard, Investigation, Graph, Reports, Settings із 7-вкладковим звітом досьє.

Функціональне тестування на реальних об'єктах підтвердило коректність роботи всіх модулів та відповідність системи функціональним вимогам. Паралельне виконання модулів через механізм asyncio.gather забезпечило виконання розслідування з 8 модулями за 17.4 секунди та побудову досьє за 7.3 секунди. Кореляційний аналіз графу із 92 вузлами та 126 зв'язками виконується за 0.2 секунди та виявляє hub-сутності, bridge-сутності та ізольовані кластери. Система розгорнута у контейнеризованому середовищі Docker Compose із двома сервісами, що гарантує відтворюваність та переносимість середовища виконання.

ВИСНОВКИ

В рамках підготовки до кваліфікаційної роботи були виконані наступні завдання:

– дослідження сучасного стану технології верифікації інформації та аналіз існуючих рішень у сфері OSINT;

– проєктування архітектури системи аналізу та підсистем збору й кореляції даних;

– програмна розробка алгоритмів збору даних;

– аналіз отриманих результатів та тестування системи.

В результаті виконання кваліфікаційної роботи було розроблено повноцінну систему аналізу та відкритих джерел інформації з використанням OSINT-технологій. Було проаналізовано принципи обробки відкритих даних і розроблено серверну частину, модулі для збору даних з різнорідних джерел. Серед ключових реалізованих компонентів системи є:

– підсистема зберігання данних із механізмами дедуплікації та індексації;

– кореляційний механізм для аналізу графу зв’язків, кластеризації та виявлення аномалій;

– клієнтський інтерактивний веб-інтерфейс із візуалізацією графу досьє;

– модуль автоматичного генерування консолідованих досьє, що містять інформацію про цифрові сліди, витоки даних, та показник ризику.

І як результат, було суттєво скорочено час проведення розслідувань. Розроблена система була розгорнута у контейнеризованому середовищі, що гарантує її переносимість, відтворюваність та готовність до практичного використання в експлуатації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Duarte F. Amount of Data Created Daily. Exploding Topics. 2026. URL: https://explodingtopics.com/blog/data-generated-per-day (дата звернення: 18.03.2026).
2. Ludo Block. The long history of OSINT. 2023. URL: https://www.tandfonline.com/doi/full/10.1080/16161262.2023.2224091 (дата звернення: 18.03.2026)
3. Susan Henrico. Dries Putter. Intelligence Collection Disciplines a Systematic Review. 2025. URL: https://www.academia.edu/114712000/Intelligence_Collection_Disciplines_A_Systematic_Review (дата звернення: 18.03.2026)
4. Dario Adriano Bermudez Villalva. Understanding the difference in malicious activity between Surface Web and Dark Web. 2022. URL: https://discovery.ucl.ac.uk/id/eprint/10147915/2/Thesis_Final.pdf (дата звернення: 18.03.2026)
5. Kevin Chen-Chuan Chang. Junghoo Cho. Accessing the Web: From Search to Integration..URL: https://dl.acm.org/doi/epdf/10.1145/1142473.1142601 (дата звернення 22.03.2026)
6. Cyble Editorial. What is the Deep Web. 2024. URL: https://cyble.com/knowledge-hub/what-is-the-deep-web/ (дата звернення 22.03.2026)
7. Stanley Okyere-Agyei. The Dark Web A Review. 2022. URL: https://web.archive.org/web/20220727021130id_/https://www.isteams.net/_files/ugd/185b0a_1f9aed314a0346b997fbb79dd75c7c19.pdf (дата звернення 22.03.2026)
8. Jon Clay. Dark web vs deep web vs surface web. 2025. URL: https://www.trendmicro.com/en/what-is/dark-web/deep-web-vs-dark-web.html (дата звернення 22.03.2026)
9. Javier Pastor-Galindo. Pantaleone Nespoli. The Not Yet Exploited Goldmine of OSINT: Opportunities, Open Challenges and Future Trends. 2020. URL: https://ieeexplore.ieee.org/abstract/document/8954668 (дата звернення 24.03.2026)

10. Moaiad Ahmad Khder. Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application. 2021. URL: https://www.i-csrs.org/Volumes/ijasca/2021.3.11.pdf (дата звернення 25.03.2026)

11. Sergio Mauricio. The way to find the real information source through OSINT. 2021. URL: https://www.sciencedirect.com/science/article/pii/S0020025521005673 (дата звернення 28.03.2026)

12. Beyond Google: Navigating the Hidden Internet with Shodan and Censys. 2025. URL: https://medium.com/@himadrisingh061/beyond-google-navigating-the-hidden-internet-with-shodan-and-censys-2cf5015f1b57 (дата звернення 28.03.2026)

13. Andrea Armstrong. Jo Briggs. Everyday digital traces. 2023. URL: https://journals.sagepub.com/doi/pdf/10.1177/20539517231213827 (дата звернення 29.03.2026)

14. Graeme Horsman. Understanding and comparing digital traces. 2024. URL: https://www.tandfonline.com/doi/full/10.1080/00450618.2024.2381535 (дата звернення 29.03.2026)

15. Anubhav Singhal. Push vs Pull API Architecture. 2022. URL: https://dev.to/anubhavitis/push-vs-pull-api-architecture-1djo (дата звернення 29.03.2026)

16. Calin Ioan Julan. Mihai Togan. Methodologies for Retrieving and Processing Information from Open Sources (OSINT). 2023. URL: https://jmiltechnol.mta.ro/11/5_C.%20Julan,%20M.%20Togan.pdf (дата звернення 30.03.2026)

17. Adelina Kiskyte. What Is a Headless Browser and Where Is It Used. 2026. URL: https://oxylabs.io/blog/what-is-headless-browser (дата звернення 30.03.2026)

18. Single-page application. MDN. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (дата звернення 04.04.2026)

19. Битва за дані користувачів або Чи законно парсити соцмережі? VKP. 2020. URL: https://vkp.ua/publication/bitva-za-dani-koristuvachiv-abo-chi-zakonno-parsiti-sotsmerezhi (дата звернення 04.04.2026)

20. What is rate limiting. Rate limiting and bots. CloudFlare. URL:

https://www.cloudflare.com/learning/bots/what-is-rate-limiting/ (дата звернення 04.04.2026)

21. Multi-Device Fingerprinting. Multilogin. URL: https://multilogin.com/glossary/multi-device-fingerprinting/ (дата звернення 08.04.2026)

22. How CAPTCHAs work. What does CAPTCHA mean. Cloudflare. URL: https://www.cloudflare.com/learning/bots/how-captchas-work/ (дата звернення 08.04.2026)

23. Andy Neumann. Nuno Laranjeiro. Jorge Bernardino. An Analysis of Public REST Web Service APIs. 2018. URL: https://eden.dei.uc.pt/~cnl/selected-research/2018-tsc-rest.pdf (дата звернення 08.04.2026)

24. Acquisition and Preparation of Data for OSINT Investigations: http://www.ir.harambeeuniversity.edu.et/bitstream/handle/123456789/862/Babak%20Akhgar.pdf?sequence=1&isAllowed=y#page=76 (дата звернення 08.04.2026)

25. ETL Pipeline. Qlik. URL: https://www.qlik.com/us/etl/etl-pipeline

26. Reproducible Analytical Pipelines. URL: https://analysisfunction.civilservice.gov.uk/support/reproducible-analytical-pipelines/ (дата звернення 18.04.2026)

27. David Meador. Single-threaded and Multi-threaded Processes. 2026. URL: https://www.tutorialspoint.com/article/single-threaded-and-multi-threaded-processes (дата звернення 18.04.2026)

28. What is Lambda Architecture. Databricks. URL: https://www.databricks.com/blog/what-is-lambda-architecture (дата звернення 18.04.2026)

29. What is name entity recognition. IBM. URL: https://www.ibm.com/think/topics/named-entity-recognition (дата звернення 18.04.2026)

30. Igor Sikorsky Kyiv Polytechnic Institute, Ukraine. OPEN SOURCE INTELLIGENCE (OSINT) 2022. URL: https://conferenc-journal.its.kpi.ua/article/view/256894/253850 (дата звернення 18.04.2026)

31. Maltego documentation. Maltego. URL: https://docs.maltego.com/en/support/home (дата звернення 13.05.2026)

32. Spiderfoot documentation. Github. URL: https://github.com/smicallef/spiderfoot (дата звернення 13.05.2026)

33. Recon-ng tool documentation. Kali. URL: https://www.kali.org/tools/recon-ng/#tool-documentation (дата звернення 13.05.2026)

34. Theharvester documentation. Kali. URL: https://www.kali.org/tools/theharvester/ (дата звернення 13.05.2026)

35. Sherlock documentation. Gitlub. URL: https://github.com/sherlock-project/sherlock (дата звернення 13.05.2026)

36. React documentation. React. URL: https://react.dev/learn/creating-a-react-app (дата звернення 13.05.2026)

37. What is REST API. Redhat. 2020. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api (дата звернення 13.05.2026)

38. FastAPI documentation. FastAPI Tiangolo. URL: https://fastapi.tiangolo.com/

39. SQLite documentation. SQLite. URL: https://sqlite.org/docs.html (дата звернення 13.05.2026)

40. Docker documentation. Docker. URL: https://docs.docker.com/ (дата звернення 13.05.2026)

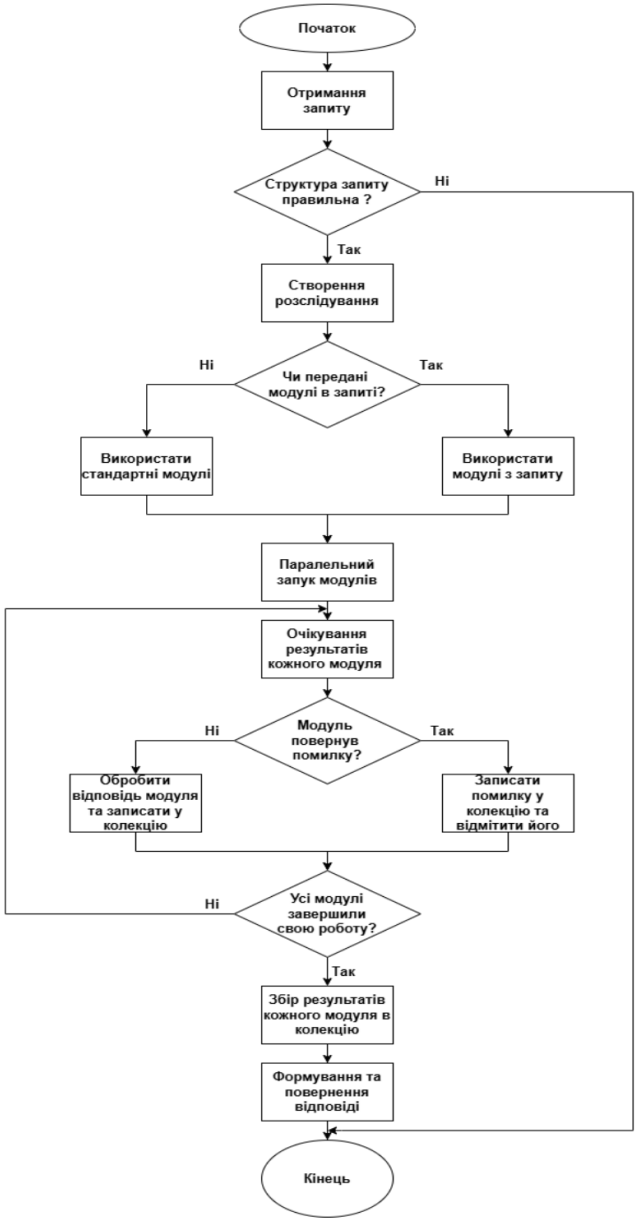
41. Asyncio python library documentation. Python docs. URL: https://docs.python.org/3/library/asyncio.html (дата звернення 13.05.2026)

Додаток А
Копія графічної частини

Критерій	Maltego	SpiderFoot	Recon-ng	theHarvester	Sherlock
Модель доступу	Комерційна	Відкрита / Комерційна	Відкрита	Відкрита	Відкрита
Автоматизований моніторинг	Відсутній	Одиничний запит	Одиничний запит	Відсутній	Відсутній
Охоплення джерел	Широке	Широке	Технічні	Технічні	Вузьке
Entity Resolution	Відсутній	Відсутній	Відсутній	Відсутній	Відсутній
ML-аналітика	Відсутня	Відсутня	Відсутня	Відсутня	Відсутня
Графове представлення	Розвинене	Базове	Відсутнє	Відсутнє	Відсутнє
Підтримка кирилиці	Часткова	Відсутня	Відсутня	Відсутня	Відсутня
Масштабованість	Низька	Обмежена	Обмежена	Відсутня	Відсутня
Відкритий код	Ні	Так	Так	Так	Так
API для інтеграції	Обмежений	Обмежений	Відсутній	Відсутній	Відсутній
Виявлення аномалій	Відсутнє	Відсутнє	Відсутнє	Відсутнє	Відсутнє
Вартість	Від $999/рік	Безкоштовна	Безкоштовна	Безкоштовна	Безкоштовна

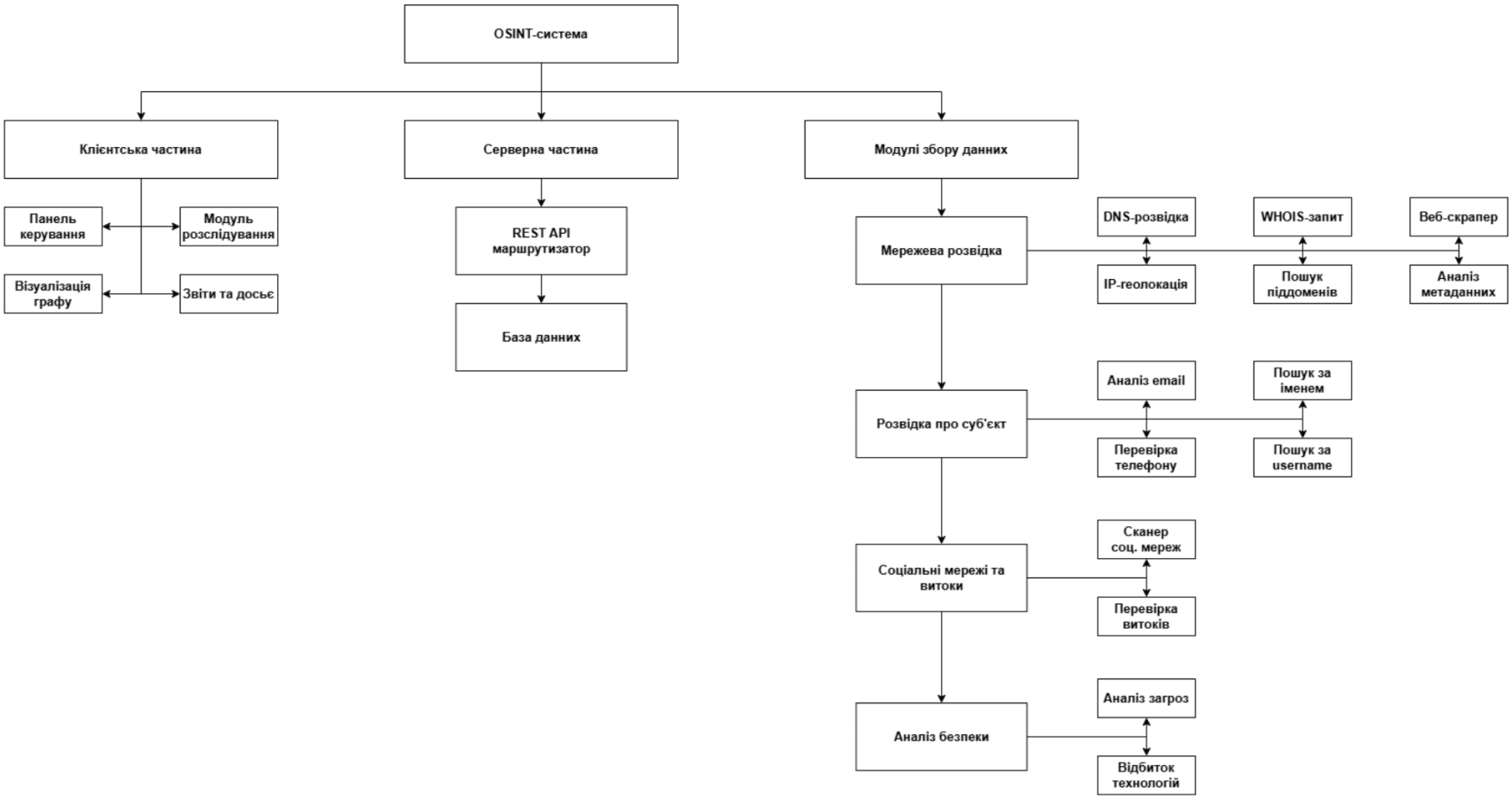
КРБКБ.220106.22.01.05 Е8

Зм.	Арк.	№докум.	Підпис	Дата	КРБКБ.220106.22.01.05 Е8			
Розроб.		Добровольський			Порівняльна матриця існуючих OSINT-рішень	Літ	Маса	Масштаб
Перевір.		Чешун В.М.				У		
Т. контр.						Аркуш	Аркушів	1
Н. контр.		Петляк Н.С.				ХНУ, КБ-22-1		
Затверд.		Клюєв О.П.						



						КРБКБ.220106.22.01.05 Е8	
Зм.	Арк.	№докум.	Підпис	Дата	Загальний алгоритм розслідування	Літ	Маса / Масштаб
Розроб.		Добровольський Н.М.				У	
Перевір.		Чешун В.М.				Аркуш	Аркушів 1
Т.контр.							
Н.контр.		Петляк Н.С.				ХНУ,КБ-22-1	
Затверд.		КльоцЮ.П.					

КРБКБ.220106.22.01.05 Е8



					КРБКБ.220106.22.01.05 Е8		
Зм.	Арк.	№докум.	Підпис	Дата	Структурна схема модулів системи	Літ	Маса / Масштаб
Розроб.		Добровольський Н.М				У	
Перевір.		Чешун В.М.				Аркуш	Аркушів 1
Т.контр.							
Н.контр.		Петляк Н.С.				ХНУ,КБ-22-1	
Затверд.		КльоцЮ.П.					

Додаток Б
Фрагмент коду реалізації системи

```
"""
API Routes — all endpoints for the OSINT system.
"""
from fastapi.responses import Response
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from typing import Optional
import asyncio
import json
import logging
import time
import re
from datetime import datetime

from server.entity_store import store
from server.correlation import correlation_engine
from server.modules.dns_lookup import dns_lookup
from server.modules.whois_lookup import whois_lookup
from server.modules.ip_geolocation import ip_geolocation
from server.modules.email_analyzer import email_analysis
from server.modules.username_checker import username_check
from server.modules.web_scraper import web_scrape
from server.modules.phone_lookup import phone_lookup
from server.modules.metadata_extractor import metadata_extract
from server.modules.person_search import person_search
from server.modules.social_scanner import social_scan_email, social_scan_phone, social_scan_username
from server.modules.breach_check import check_breaches
from server.modules.name_resolver import name_search
from server.modules.dossier import generate_website_dossier
from server.modules.person_dossier_builder import build_person_dossier
from server.models.person import NormalizedPerson
from server.modules.subdomain_enum import enumerate_subdomains
from server.modules.threat_intel import vt_domain_report
from server.modules.shodan_search import shodan_host_search
from server.modules.tech_fingerprint import tech_fingerprint
from server.modules.image_osint import analyze_image
from server.modules.document_analyzer import analyze_document
from server.api_keys import load_api_keys, save_api_keys
from server.limiter import limiter

router = APIRouter()


class InvestigateRequest(BaseModel):
    target: str
    type: str  # domain, ip, email, username, phone, url, person
    modules: Optional[list] = None


@router.post("/investigate")
@limiter.limit("10/minute")
async def investigate(request: Request, req: InvestigateRequest):
    """Run OSINT investigation on a target."""
    inv = store.create_investigation(req.target, req.type)
    target = _validate_target(req.target, req.type)

    main_entity = store.add_entity(
        entity_type=req.type,
```

```
        value=target,
        label=req.target,
        investigation_id=inv["id"],
    )

    results = {}
    all_modules = req.modules or _get_default_modules(req.type)

    tasks = {}
    for module_name in all_modules:
        task = _run_module(module_name, req.target, req.type)
        tasks[module_name] = task

    module_names = list(tasks.keys())
    coroutines = list(tasks.values())
    gathered_results = await asyncio.gather(*coroutines, return_exceptions=True)

    module_results = {}
    for name, result in zip(module_names, gathered_results):
        if isinstance(result, Exception):
            module_results[name] = {"error": str(result), "module": name}
        else:
            module_results[name] = result

    for module_name, result in module_results.items():
        results[module_name] = result
        _create_entities_from_result(
            module_name, result, main_entity, inv["id"]
        )

    store.update_investigation(inv["id"], status="completed", results=results)

    return {
        "investigation_id": inv["id"],
        "target": req.target,
        "type": req.type,
        "status": "completed",
        "results": results,
        "entity_id": main_entity["id"],
    }


def _get_default_modules(target_type: str) -> list:
    """Get default modules for a target type."""
    defaults = {
        "domain": ["dns", "whois", "web_scraper", "metadata",
                   "subdomain_enum", "threat_intel", "tech_fingerprint"],
        "ip": ["ip_geo", "whois", "metadata", "shodan", "threat_intel"],
        "email": ["email", "dns", "social_scan_email", "breach_check"],
        "username": ["username"],
        "phone": ["phone", "social_scan_phone"],
        "url": ["web_scraper", "metadata", "threat_intel", "tech_fingerprint"],
        "person": ["person_search"],
    }
    return defaults.get(target_type, ["dns", "whois"])


async def _run_module(module_name: str, target: str, target_type: str):
    """Run a single OSINT module with logging and timing."""
    modules = {
        "dns": dns_lookup,
        "whois": whois_lookup,
        "ip_geo": ip_geolocation,
```

```
        "email": email_analysis,
        "username": username_check,
        "web_scraper": web_scrape,
        "phone": phone_lookup,
        "metadata": metadata_extract,
        "person_search": person_search,
        "social_scan_email": social_scan_email,
        "social_scan_phone": social_scan_phone,
        "social_scan_username": social_scan_username,
        "breach_check": lambda t: check_breaches(t, "email"),
        "name_search": name_search,
        "subdomain_enum": enumerate_subdomains,
        "shodan": shodan_host_search,
        "threat_intel": vt_domain_report,
        "tech_fingerprint": tech_fingerprint,
    }
    module_fn = modules.get(module_name)
    if not module_fn:
        return {"error": f"Unknown module: {module_name}", "module": module_name}

    start = time.time()
    try:
        result = await module_fn(target)
        elapsed = time.time() - start
        return result
    except Exception as e:
        elapsed = time.time() - start
        return {"error": str(e), "module": module_name}


def _validate_target(target: str, target_type: str) -> str:
    target = target.strip()

    if target_type == "email":
        target = target.lower().strip()
        if not re.match(
            r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$', target
        ):
            pass

    elif target_type == "phone":
        target = re.sub(r'[\s\-\(\)\.]', '', target)
        if target and not target.startswith('+'):
            target = '+' + target

    elif target_type == "username":
        target = target.lstrip('@').strip()
        cleaned = re.sub(r'[^a-zA-Z0-9_\-.]', '', target)
        if cleaned != target:
            target = cleaned

    elif target_type == "domain":
        target = target.lower().strip()
        target = re.sub(r'^https?://', '', target)
        target = target.rstrip('/')

    elif target_type == "url":
        if not target.startswith(('http://', 'https://')):
            target = 'https://' + target

    return target

"""
```

```
DNS Lookup Module — resolves all DNS record types using dnspython.
"""
import dns.resolver
import dns.reversename
import socket
from typing import Optional


async def dns_lookup(target: str) -> dict:
    """Perform comprehensive DNS lookup for a domain."""
    results = {
        "module": "dns",
        "target": target,
        "records": {},
        "resolved_ips": [],
    }

    record_types = ["A", "AAAA", "MX", "NS", "TXT", "CNAME", "SOA"]

    for rtype in record_types:
        try:
            answers = dns.resolver.resolve(target, rtype)
            records = []
            for rdata in answers:
                record_str = rdata.to_text()
                records.append(record_str)
                if rtype == "A":
                    results["resolved_ips"].append(record_str)
            results["records"][rtype] = records
        except (dns.resolver.NoAnswer, dns.resolver.NXDOMAIN,
                dns.resolver.NoNameservers, dns.exception.Timeout):
            pass
        except Exception:
            pass

    results["reverse_dns"] = {}
    for ip in results["resolved_ips"]:
        try:
            rev_name = dns.reversename.from_address(ip)
            answers = dns.resolver.resolve(rev_name, "PTR")
            results["reverse_dns"][ip] = [r.to_text() for r in answers]
        except Exception:
            pass

    if not results["resolved_ips"]:
        try:
            ip = socket.gethostbyname(target)
            results["resolved_ips"].append(ip)
        except Exception:
            pass

    results["total_records"] = sum(
        len(v) for v in results["records"].values()
    )
    return results

"""
WHOIS Lookup Module — queries WHOIS data for domains and IPs.
"""
import whois
from datetime import datetime
```

```
def _serialize_dates(obj):
    """Convert date objects to ISO strings for JSON serialization."""
    if isinstance(obj, datetime):
        return obj.isoformat()
    if isinstance(obj, list):
        return [_serialize_dates(item) for item in obj]
    return obj


async def whois_lookup(target: str) -> dict:
    """Perform WHOIS lookup for a domain or IP."""
    results = {
        "module": "whois",
        "target": target,
        "data": {},
    }

    try:
        w = whois.whois(target)

        results["data"] = {
            "domain_name": w.domain_name if hasattr(w, 'domain_name') else None,
            "registrar": w.registrar if hasattr(w, 'registrar') else None,
            "whois_server": w.whois_server if hasattr(w, 'whois_server') else None,
            "creation_date": _serialize_dates(w.creation_date)
                if hasattr(w, 'creation_date') else None,
            "expiration_date": _serialize_dates(w.expiration_date)
                if hasattr(w, 'expiration_date') else None,
            "updated_date": _serialize_dates(w.updated_date)
                if hasattr(w, 'updated_date') else None,
            "name_servers": w.name_servers if hasattr(w, 'name_servers') else None,
            "status": w.status if hasattr(w, 'status') else None,
            "emails": w.emails if hasattr(w, 'emails') else None,
            "org": w.org if hasattr(w, 'org') else None,
            "country": w.country if hasattr(w, 'country') else None,
            "state": w.state if hasattr(w, 'state') else None,
            "city": w.city if hasattr(w, 'city') else None,
            "address": w.address if hasattr(w, 'address') else None,
            "registrant": w.get("name") if hasattr(w, 'get') else None,
            "dnssec": w.dnssec if hasattr(w, 'dnssec') else None,
        }

        results["data"] = {
            k: v for k, v in results["data"].items() if v is not None
        }
        results["success"] = True

    except Exception as e:
        results["error"] = str(e)
        results["success"] = False

    return results

"""
Email Analysis Module — analyzes email addresses for OSINT data.
Checks MX records, Gravatar, disposable email detection, domain analysis.
"""
import dns.resolver
import hashlib
import aiohttp
import re

DISPOSABLE_DOMAINS = {
```

```
    "mailinator.com", "guerrillamail.com", "tempmail.com",
    "throwaway.email", "yopmail.com", "dispostable.com",
    "sharklasers.com", "guerrillamailblock.com", "grr.la",
    "10minutemail.com", "trashmail.com", "fakeinbox.com",
    "temp-mail.org", "getnada.com", "mohmal.com",
    "burpcollaborator.net", "maildrop.cc", "discard.email",
    "mailnesia.com", "tempr.email",
}


async def email_analysis(email: str) -> dict:
    """Analyze an email address for OSINT information."""
    results = {
        "module": "email",
        "target": email,
        "data": {},
    }

    if not re.match(
        r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$", email
    ):
        results["error"] = "Invalid email format"
        results["success"] = False
        return results

    local_part, domain = email.split("@", 1)

    results["data"]["local_part"] = local_part
    results["data"]["domain"] = domain
    results["data"]["is_disposable"] = domain.lower() in DISPOSABLE_DOMAINS

    try:
        mx_records = dns.resolver.resolve(domain, "MX")
        results["data"]["mx_records"] = [
            {"priority": r.preference, "server": r.exchange.to_text()}
            for r in mx_records
        ]
        results["data"]["has_mx"] = True
        results["data"]["mail_provider"] = _detect_provider(
            [r.exchange.to_text() for r in mx_records]
        )
    except Exception:
        results["data"]["has_mx"] = False
        results["data"]["mx_records"] = []

    email_hash = hashlib.md5(
        email.lower().strip().encode()
    ).hexdigest()
    results["data"]["gravatar_hash"] = email_hash
    results["data"]["gravatar_url"] = (
        f"https://www.gravatar.com/avatar/{email_hash}"
    )

    try:
        async with aiohttp.ClientSession() as session:
            url = f"https://www.gravatar.com/avatar/{email_hash}?d=404"
            async with session.head(
                url, timeout=aiohttp.ClientTimeout(total=5)
            ) as resp:
                results["data"]["has_gravatar"] = resp.status == 200
    except Exception:
        results["data"]["has_gravatar"] = False
```

```
    results["data"]["domain_info"] = domain
    results["data"]["patterns"] = _analyze_patterns(local_part)
    results["success"] = True
    return results


def _detect_provider(mx_servers: list) -> str:
    """Detect the email provider from MX servers."""
    mx_text = " ".join(mx_servers).lower()
    providers = {
        "google": "Google Workspace / Gmail",
        "outlook": "Microsoft 365 / Outlook",
        "protonmail": "ProtonMail",
        "zoho": "Zoho Mail",
        "yandex": "Yandex Mail",
        "mail.ru": "Mail.ru",
        "icloud": "Apple iCloud",
        "fastmail": "FastMail",
        "tutanota": "Tutanota",
        "migadu": "Migadu",
    }
    for key, name in providers.items():
        if key in mx_text:
            return name
    return "Custom / Unknown"


def _analyze_patterns(local_part: str) -> dict:
    """Analyze the email local part for patterns."""
    patterns = {
        "contains_name": not bool(re.match(r'^[a-z0-9]+$', local_part)),
        "contains_numbers": bool(re.search(r'\d', local_part)),
        "contains_dots": '.' in local_part,
        "contains_plus": '+' in local_part,
        "length": len(local_part),
    }

    if '.' in local_part:
        parts = local_part.split('.')
        if len(parts) == 2 and parts[0].isalpha() and parts[1].isalpha():
            patterns["possible_name"] = (
                f"{parts[0].capitalize()} {parts[1].capitalize()}"
            )

    return patterns

"""
Phone Lookup Module — parses and analyzes phone numbers
using the phonenumbers library.
"""
import phonenumbers
from phonenumbers import geocoder, carrier, timezone


async def phone_lookup(phone: str) -> dict:
    """Analyze a phone number for OSINT data."""
    results = {
        "module": "phone",
        "target": phone,
        "data": {},
    }

    try:
```

```
        parsed = None
        for region in [None, "US", "GB", "DE", "UA", "PL", "FR"]:
            try:
                parsed = phonenumbers.parse(phone, region)
                if phonenumbers.is_valid_number(parsed):
                    break
            except Exception:
                continue

        if not parsed or not phonenumbers.is_valid_number(parsed):
            results["error"] = "Invalid phone number"
            results["success"] = False
            return results

        results["data"] = {
            "formatted_international": phonenumbers.format_number(
                parsed, phonenumbers.PhoneNumberFormat.INTERNATIONAL
            ),
            "formatted_national": phonenumbers.format_number(
                parsed, phonenumbers.PhoneNumberFormat.NATIONAL
            ),
            "formatted_e164": phonenumbers.format_number(
                parsed, phonenumbers.PhoneNumberFormat.E164
            ),
            "country_code": parsed.country_code,
            "national_number": parsed.national_number,
            "country": geocoder.description_for_number(parsed, "en"),
            "carrier": carrier.name_for_number(parsed, "en"),
            "timezones": list(timezone.time_zones_for_number(parsed)),
            "is_valid": phonenumbers.is_valid_number(parsed),
            "is_possible": phonenumbers.is_possible_number(parsed),
            "number_type": _get_number_type(
                phonenumbers.number_type(parsed)
            ),
            "region_code": phonenumbers.region_code_for_number(parsed),
        }
        results["success"] = True

    except Exception as e:
        results["error"] = str(e)
        results["success"] = False

    return results


def _get_number_type(num_type) -> str:
    """Convert phonenumbers type to string."""
    types = {
        phonenumbers.PhoneNumberType.FIXED_LINE: "Fixed Line",
        phonenumbers.PhoneNumberType.MOBILE: "Mobile",
        phonenumbers.PhoneNumberType.FIXED_LINE_OR_MOBILE:
            "Fixed Line or Mobile",
        phonenumbers.PhoneNumberType.TOLL_FREE: "Toll Free",
        phonenumbers.PhoneNumberType.PREMIUM_RATE: "Premium Rate",
        phonenumbers.PhoneNumberType.SHARED_COST: "Shared Cost",
        phonenumbers.PhoneNumberType.VOIP: "VoIP",
        phonenumbers.PhoneNumberType.PERSONAL_NUMBER: "Personal",
        phonenumbers.PhoneNumberType.PAGER: "Pager",
        phonenumbers.PhoneNumberType.UAN: "UAN",
        phonenumbers.PhoneNumberType.VOICEMAIL: "Voicemail",
    }
    return types.get(num_type, "Unknown")
```

```
"""
Breach Check Module — checks if email/username appears in known data breaches.
Uses the Have I Been Pwned API pattern and additional free breach databases.
"""
import aiohttp
import hashlib

HIBP_API_URL = "https://haveibeenpwned.com/api/v3"

KNOWN_BREACH_DB = {
    "gmail.com": {"likely_breaches": ["Collection #1", "Anti Public Combo List"]},
    "yahoo.com": {"likely_breaches": ["Yahoo 2013", "Yahoo 2014"]},
    "hotmail.com": {"likely_breaches": ["Collection #1"]},
    "linkedin.com": {"likely_breaches": ["LinkedIn 2012", "LinkedIn 2021"]},
    "adobe.com": {"likely_breaches": ["Adobe 2013"]},
    "dropbox.com": {"likely_breaches": ["Dropbox 2012"]},
}

PWNED_PASSWORDS_API = "https://api.pwnedpasswords.com/range/{}"


async def check_breaches(identifier: str, id_type: str = "email") -> dict:
    """Check if an identifier appears in known data breaches."""
    results = {
        "module": "breach_check",
        "target": identifier,
        "type": id_type,
        "breaches": [],
        "total_breaches": 0,
        "sources_checked": [],
        "risk_level": "unknown",
    }

    headers = {
        "User-Agent": "OSINT-Analyst-Platform/1.0",
        "Accept": "application/json",
    }

    connector = aiohttp.TCPConnector(limit=5, ssl=False)
    async with aiohttp.ClientSession(
        headers=headers, connector=connector
    ) as session:

        # 1. HIBP API
        try:
            hibp_url = (
                f"https://haveibeenpwned.com/api/v3/"
                f"breachedaccount/{identifier}?truncateResponse=false"
            )
            async with session.get(
                hibp_url,
                timeout=aiohttp.ClientTimeout(total=10),
                headers={
                    "hibp-api-key": "",
                    "User-Agent": "OSINT-Analyst-Platform"
                },
                ssl=False,
            ) as resp:
                results["sources_checked"].append("Have I Been Pwned")
                if resp.status == 200:
                    breaches = await resp.json()
                    for breach in breaches:
                        results["breaches"].append({
```

```
                    "name": breach.get("Name", "Unknown"),
                    "domain": breach.get("Domain", ""),
                    "date": breach.get("BreachDate", ""),
                    "description":
                        breach.get("Description", "")[:200],
                    "data_classes":
                        breach.get("DataClasses", []),
                    "is_verified":
                        breach.get("IsVerified", False),
                    "pwncount": breach.get("PwnCount", 0),
                    "source": "HIBP",
                })
    except Exception:
        results["sources_checked"].append(
            "Have I Been Pwned (error)"
        )

    # 2. LeakCheck free tier
    try:
        leak_url = (
            f"https://leakcheck.io/api/public?check={identifier}"
        )
        async with session.get(
            leak_url,
            timeout=aiohttp.ClientTimeout(total=10),
            ssl=False,
        ) as resp:
            results["sources_checked"].append("LeakCheck")
            if resp.status == 200:
                data = await resp.json()
                if data.get("found"):
                    for source in data.get("sources", []):
                        results["breaches"].append({
                            "name": source.get("name", "Unknown"),
                            "date": source.get("date", ""),
                            "data_classes": [],
                            "source": "LeakCheck",
                        })
    except Exception:
        results["sources_checked"].append("LeakCheck (error)")

    # 3. EmailRep
    try:
        rep_url = f"https://emailrep.io/{identifier}"
        async with session.get(
            rep_url,
            timeout=aiohttp.ClientTimeout(total=8),
            headers={
                "Key": "",
                "User-Agent": "OSINT-Analyst-Platform"
            },
            ssl=False,
        ) as resp:
            results["sources_checked"].append("EmailRep")
            if resp.status == 200:
                data = await resp.json()
                results["reputation"] = {
                    "reputation":
                        data.get("reputation", "unknown"),
                    "suspicious":
                        data.get("suspicious", False),
                    "references":
                        data.get("references", 0),
```

```
                    "details": data.get("details", {}),
                    "profiles":
                        data.get("details", {}).get("profiles", []),
                    "data_breach":
                        data.get("details", {}).get(
                            "data_breach", False
                        ),
                    "credentials_leaked":
                        data.get("details", {}).get(
                            "credentials_leaked", False
                        ),
                }
        except Exception:
            results["sources_checked"].append("EmailRep (error)")

        # 4. Pwned Passwords (k-anonymity)
        if id_type == "password":
            try:
                sha1 = hashlib.sha1(
                    identifier.encode()
                ).hexdigest().upper()
                prefix = sha1[:5]
                suffix = sha1[5:]
                pwd_url = PWNED_PASSWORDS_API.format(prefix)
                async with session.get(
                    pwd_url,
                    timeout=aiohttp.ClientTimeout(total=5),
                    ssl=False
                ) as resp:
                    results["sources_checked"].append("Pwned Passwords")
                    if resp.status == 200:
                        text = await resp.text()
                        for line in text.splitlines():
                            parts = line.split(":")
                            if parts[0] == suffix:
                                results["password_pwned"] = True
                                results["password_occurrences"] = int(
                                    parts[1]
                                )
                                break
                        else:
                            results["password_pwned"] = False
            except Exception:
                pass

    # 5. Domain-based breach intelligence (fallback)
    if id_type == "email" and "@" in identifier:
        domain = identifier.split("@")[1].lower()
        if domain in KNOWN_BREACH_DB:
            db_entry = KNOWN_BREACH_DB[domain]
            for breach_name in db_entry["likely_breaches"]:
                if not any(
                    b["name"] == breach_name
                    for b in results["breaches"]
                ):
                    results["breaches"].append({
                        "name": breach_name,
                        "domain": domain,
                        "data_classes": [
                            "Email addresses", "Passwords"
                        ],
                        "source": "Domain Intelligence",
                    })
```

```
    # Deduplicate
    seen = set()
    unique_breaches = []
    for b in results["breaches"]:
        key = b["name"]
        if key not in seen:
            seen.add(key)
            unique_breaches.append(b)
    results["breaches"] = unique_breaches
    results["total_breaches"] = len(unique_breaches)
    results["risk_level"] = _assess_risk(results)
    results["success"] = True
    return results


def _assess_risk(results: dict) -> str:
    """Assess risk level based on breach data."""
    total = results["total_breaches"]
    has_passwords = any(
        "Passwords" in b.get("data_classes", [])
        for b in results["breaches"]
    )
    rep = results.get("reputation", {})
    is_suspicious = rep.get("suspicious", False)
    creds_leaked = rep.get("credentials_leaked", False)

    if total >= 5 or (has_passwords and total >= 2) or creds_leaked:
        return "critical"
    elif total >= 3 or has_passwords or is_suspicious:
        return "high"
    elif total >= 1:
        return "medium"
    elif total == 0:
        return "low"
    return "unknown"

"""
Web Scraper Module — extracts metadata, links, emails, phones,
and technologies from web pages.
"""
import aiohttp
from bs4 import BeautifulSoup
import re
from urllib.parse import urljoin, urlparse


async def web_scrape(url: str) -> dict:
    """Scrape a web page for OSINT-relevant data."""
    if not url.startswith(("http://", "https://")):
        url = f"https://{url}"

    results = {
        "module": "web_scraper",
        "target": url,
        "data": {},
    }

    headers = {
        "User-Agent": (
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
            "AppleWebKit/537.36 (KHTML, like Gecko) "
            "Chrome/120.0.0.0 Safari/537.36"
```

```
    )
}

try:
    async with aiohttp.ClientSession(headers=headers) as session:
        async with session.get(
            url,
            timeout=aiohttp.ClientTimeout(total=15),
            allow_redirects=True,
            ssl=False
        ) as resp:
            html = await resp.text()
            final_url = str(resp.url)

            results["data"]["status_code"] = resp.status
            results["data"]["final_url"] = final_url
            results["data"]["content_type"] = resp.content_type

            results["data"]["headers"] = {
                "server": resp.headers.get("Server"),
                "x_powered_by":
                    resp.headers.get("X-Powered-By"),
                "content_security_policy": bool(
                    resp.headers.get("Content-Security-Policy")
                ),
                "strict_transport_security": bool(
                    resp.headers.get("Strict-Transport-Security")
                ),
                "x_frame_options":
                    resp.headers.get("X-Frame-Options"),
                "x_content_type_options":
                    resp.headers.get("X-Content-Type-Options"),
            }

            soup = BeautifulSoup(html, "html.parser")

            title_tag = soup.find("title")
            results["data"]["title"] = (
                title_tag.get_text(strip=True) if title_tag else None
            )

            meta_data = {}
            for meta in soup.find_all("meta"):
                name = meta.get("name", meta.get("property", ""))
                content = meta.get("content", "")
                if name and content:
                    meta_data[name] = content
            results["data"]["meta"] = meta_data

            email_pattern = (
                r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
            )
            emails = list(set(re.findall(email_pattern, html)))
            results["data"]["emails"] = emails[:20]

            phone_pattern = (
                r'[\+]?[(]?[0-9]{1,4}[)]?[-\s\./0-9]{7,15}'
            )
            phones = list(set(re.findall(phone_pattern, html)))
            phones = [
                p.strip() for p in phones
                if len(re.sub(r'\D', '', p)) >= 7
            ]
```

```
            results["data"]["phones"] = phones[:20]

            links = []
            domain = urlparse(final_url).netloc
            for a in soup.find_all("a", href=True):
                href = urljoin(final_url, a["href"])
                parsed = urlparse(href)
                if (parsed.netloc and parsed.netloc != domain
                        and parsed.scheme in ("http", "https")):
                    links.append({
                        "url": href,
                        "text": a.get_text(strip=True)[:100],
                        "domain": parsed.netloc,
                    })
            seen = set()
            unique_links = []
            for link in links:
                if link["url"] not in seen:
                    seen.add(link["url"])
                    unique_links.append(link)
            results["data"]["external_links"] = unique_links[:50]

            social_patterns = {
                "facebook": r'facebook\.com/[\w.]+',
                "twitter": r'(twitter\.com|x\.com)/[\w]+',
                "instagram": r'instagram\.com/[\w.]+',
                "linkedin":
                    r'linkedin\.com/(in|company)/[\w-]+',
                "youtube":
                    r'youtube\.com/(c|channel|user|@)[\w-]+',
                "github": r'github\.com/[\w-]+',
                "telegram": r't\.me/[\w]+',
                "tiktok": r'tiktok\.com/@[\w.]+',
            }
            social_links = {}
            for platform, pattern in social_patterns.items():
                matches = re.findall(
                    pattern, html, re.IGNORECASE
                )
                if matches:
                    social_links[platform] = (
                        matches[0]
                        if isinstance(matches[0], str)
                        else matches[0][0]
                    )
            results["data"]["social_links"] = social_links
            results["data"]["technologies"] = (
                _detect_technologies(html, resp.headers)
            )
            results["success"] = True

    except Exception as e:
        results["error"] = str(e)
        results["success"] = False

    return results


def _detect_technologies(html: str, headers) -> list:
    """Detect web technologies used by the page."""
    techs = []
    html_lower = html.lower()
```

```
detections = {
    "jQuery": "jquery" in html_lower,
    "React": "react" in html_lower
        or "_reactroot" in html_lower,
    "Vue.js": "vue" in html_lower
        and ("vue.js" in html_lower
             or "vue.min" in html_lower
             or "__vue" in html_lower),
    "Angular": "ng-app" in html_lower
        or "angular" in html_lower,
    "Bootstrap": "bootstrap" in html_lower,
    "WordPress": "wp-content" in html_lower
        or "wordpress" in html_lower,
    "Google Analytics": "google-analytics" in html_lower
        or "gtag" in html_lower,
    "Cloudflare": headers.get("cf-ray") is not None,
    "Nginx": "nginx" in str(
        headers.get("server", "")
    ).lower(),
    "Apache": "apache" in str(
        headers.get("server", "")
    ).lower(),
    "Next.js": "__next" in html_lower,
    "Nuxt.js": "__nuxt" in html_lower,
}

for tech, detected in detections.items():
    if detected:
        techs.append(tech)

return techs
```