МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ПЕРВАЧУК РОМАН ЮРІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____» _____ 2025 р.

РОЗРОБКА УТИЛІТ ДЛЯ АГРЕГАЦІЇ ТА АНАЛІЗУ СТАНУ ІНТЕРНЕТ-РЕСУРСІВ.

Спеціальність 122 «Комп'ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Антонов Ю.С., кан. фіз.-мат. наук, доцент,
доцент кафедри інформаційних
технологій

_____
(Підпис)

Оцінка_____/_____/_____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(Підпис)

Вінниця 2025

АНОТАЦІЯ

Первачук Р Ю. Розробка багатомодульної системи для агрегації та фільтрації небажаних інтернет-ресурсів. Спеціальність: 122 «Комп'ютерні науки» Донецький національний університет імені Василя Стуса, Вінниця, 2025

У роботі розроблено програмну систему, що автоматизує завантаження, аналіз і формування списків шкідливих інтернет-ресурсів. Реалізовано набір CLI-утиліт для агрегації, парсингу, перевірки та генерації списків блокування у різних форматах. Дані зберігаються в локальній базі SQLite.

Ключові слова: IP-адреса, агрегація, блокування, фільтрація, безпека, Python, DNS

Табл. 6. Рис. 16. Бібліограф.: 40 найм.

Pervachuk R.Y. Development of a multi-module system for aggregation and filtering of unwanted Internet resources. Specialty: 122 "Computer Science" Vasyl Stus Donetsk National University, Vinnytsia, 2025

The work developed a software system that automates the loading, analysis and formation of lists of malicious Internet resources. A set of CLI utilities for aggregation, parsing, verification and generation of blocking lists in various formats was implemented. The data is stored in a local SQLite database.

Keywords: IP address, aggregation, blocking, filtering, security, Python, DNS

Table 6. Fig. 16. Bibliography: 40 refs.

ЗМІСТ

ВСТУП

Актуальність теми дослідження: У сучасних умовах глобального розвитку інформаційних технологій інтернет перетворився на ключове середовище як для поширення знань, так і для загроз інформаційній безпеці. З кожним роком зростає кількість вебресурсів, які використовуються для розповсюдження шкідливого програмного забезпечення, фішингових атак, ботнет-інфраструктур, порнографії, засобів обходу фільтрації, дезінформації, а також контенту, пов’язаного з насильством, тероризмом, торгівлею людьми та наркотиками. Блокування доступу до таких джерел є одним із ключових інструментів захисту кінцевих користувачів, організаційних систем і державних структур.

Попри наявність ряду інструментів фільтрації, велика частина рішень має низку обмежень: закритий вихідний код, відсутність підтримки категоризації, вузьке призначення або потребу в постійному підключенні до хмарних сервісів. Значна частина наявних утиліт не охоплює повного процесу: від збору фідів до генерації списків блокування з можливістю гнучкої фільтрації. Також обмежено представлені рішення, що дозволяють поєднувати декілька джерел, зберігати метадані, перевіряти актуальність записів та експортувати їх у різні формати.

Це визначає актуальність створення автономного, розширюваного, модульного інструменту, здатного агрегувати, обробляти, класифікувати та експортувати списки потенційно небезпечних ресурсів – із урахуванням конфігураційної гнучкості, підтримки кількох форматів та локальної роботи без залежності від сторонніх серверів.

Мета: Розробка концепції та структури автономного програмного інструменту для агрегації та аналізу даних про небезпечні інтернет-ресурси, з подальшою генерацією списків блокування у форматах, придатних для використання в системах захисту мережевого доступу.

Завдання дослідження: Відповідно до поставленої мети, у процесі виконання роботи необхідно:

- виявити типи загроз, пов’язаних з інтернет-ресурсами, та описати джерела таких даних;
- узагальнити існуючі підходи до блокування шкідливих ресурсів, класифікувати існуючі програмні засоби;
- розробити загальну архітектуру системи, визначити її модулі та принципи взаємодії між ними;
- дослідити алгоритми збору, нормалізації та фільтрації мережевих фідів;
- проаналізувати можливі формати представлення вихідних даних та алгоритми їх генерації;
- описати принципи перевірки актуальності ресурсів і зберігання статусу.

Об’єкт дослідження: Об’єктом дослідження є процеси збору, обробки та блокування доступу до небезпечних інтернет-ресурсів.

Предмет дослідження: Предметом дослідження є алгоритми, структури даних та форматні представлення, що використовуються у процесі агрегації, нормалізації та експорту даних для системи фільтрації небезпечних доменів та IP-адрес.

Теоретичне та/або практичне значення: Результати дослідження мають практичне значення, оскільки запропонована структура інструменту може бути застосована для створення автономних систем фільтрації у локальних мережах, державних установах, навчальних закладах або навіть у побутових умовах. Теоретичне значення полягає в аналізі та обґрунтуванні підходів до агрегації, класифікації та структурування даних у контексті інформаційної безпеки. Побудовані моделі та алгоритми можуть бути використані як основа для подальших розробок у сфері захисту від мережевих загроз

Апробація результатів дослідження: Доповідь із публікацією тез на тему «Розробка утиліт для агрегації та аналізу стану інтернет-ресурсів» у VI Всеукраїнській науково-практичній конференції здобувачів, аспірантів та молодих вчених «Прикладні інформаційні технології» (ПІТ-2025)

Структура кваліфікаційної (бакалаврської) роботи: Бакалаврська робота складається із вступу, трьох розділів та висновків до них, списку використаних джерел та двох додатків.

У першому розділі бакалаврської роботи проведено огляд предметної області, та інструментів що дозволяють вирішити поставлену задачу.

У другому розділі наведено методи та технології для реалізації утиліт для агрегації та аналізу інтернет-ресурсів.

У третьому розділі наведено опис конкретних інструментів, використаних для реалізації утиліт, приклади використання робочих програм та опис принципів їх роботи.

Бакалаврська робота включає в себе 62 сторінки, 16 рисунок і список літератури із 40 джерел.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Загрози в інтернеті. Принципи роботи DNS

Інтернет є основним каналом для передачі інформації у сучасному світі, однак разом із корисним контентом у мережі активно поширюються матеріали, що становлять загрозу для користувачів, організацій та національної безпеки. До таких належать як небезпечні ресурси, що безпосередньо пов’язані з шкідливою активністю, так і небажані ресурси, які хоча не є нелегальними, проте можуть мати негативний вплив на поведінку або інформаційне поле користувачів [1].

Розглянемо види різних кібер-загроз:

- Програми-вимагачі (Ransomware)[0] – вид шкідливого ПЗ, котре після зараження пристрою, блокує жертві доступ до нього чи даних на ньому та вимагає викуп для відновлення доступу. Виділяють два основні види: більш поширений – шифрувальний, що шифрує дані жертви, та менш поширений – не-шифрувальний, що блокує доступ до пристрою загалом.
- Викрадачі інформації (Information Stealer) [3] – програми, націлені на збір різних видів фінансових чи персональних даних в заражених системах. Серед них збережені в браузерах паролі, інформація банківських карток, данні крипто-гаманців, різні облікові данні для доступу до критичних сервісів чи дані сесії браузера. Поширюється часто через електрону пошту та рекламні оголошення на сторінках.
- «Трояни» –шкідливе ПЗ, що зазвичай видає себе за офіційне чи безпечне. Ця прихованість в комбінації з необізнаністю чи недбалістю користувачів і породжують проблеми для інформаційних систем. Окрім непомітного поширення з іншим ПЗ «троянські коні» використовують для розповсюдження електронну пошту із зараженими вкладеннями. Деякі з них навіть прославились серед спеціалістів сфери. «Qakbot» [4], до прикладу, активний вже понад півтора десятиліття та еволюціонував до викрадача конфіденційної

банківської інформації зі здатністю поширюватись мережею, збирати дані для входу для подальшого спаму, записувати натискання клавіш. Варто зазначити, що окремо виділяють підвид цієї загрози – RAT[5], «троян віддаленого доступу». Хоч спосіб проникнення в систему жертви та мета залишаються такими ж самими, основна його функція – здобути контроль над зараженим пристроєм.

- APT (загроза підвищеної наполегливості) [6] – складні, чітко спрямовані кібератаки на певні організації чи особистостей. Часто цілями такого типу атак стають урядові структури. Метою таких атак, окрім заволодіння даними, може бути і завдання максимальної шкоди інфраструктурі чи спроба її знищити взагалі. Так, в контексті російсько-Української кібервійни відбувались атаки на «Київстар», «Укрзалізницю».
- Бекдор, (від англ. backdoor, чорний хід)[7] – це метод, з яким користувач без авторизації можуть отримати віддалений доступ до пристрою, системи чи мережі. На етапі розробки чи обслуговування розробники користуються ними щоб легко та зручно тестувати щось усувати несправності. Така легкість доступу приваблює зловмисників.
- Ботнет – [8]це мережа заражених пристроїв, централізовано керованих. «Керівник ботів» – це особа, яка керує інфраструктурою ботнету та використовує скомпрометовані комп’ютери для запуску атак, спрямованих на збій мережі цілі, впровадження шкідливого програмного забезпечення , збору облікових даних або виконання завдань, що ресурсомісткі для процесора. Кожен окремий пристрій у мережі ботнету називається ботом.

Перелік наведених загроз показує різноманіття шкідливих механізмів, які використовуються для ураження інформаційних систем – від масових атак типу ransomware до складних цілеспрямованих кампаній APT. Спільним для

більшості з них є те, що точкою входу до системи жертви часто виступає саме мережевий рівень, а зокрема – система доменних імен (DNS). Саме DNS є першим етапом маршрутизації майже кожного запиту в інтернеті: будь то відкриття веб-сайту, завантаження оновлення чи зв'язок шкідливого ПЗ з командним сервером.

У цьому контексті важливо усвідомлювати, що DNS використовується не лише для легітимної взаємодії з веб-ресурсами, а й як інструмент шкідливої активності – наприклад, для фішингу, ботнет-зв'язку, а також для уникнення виявлення шляхом маскування трафіку. Згідно з дослідження Cisco [9] про активність DNS (рис. 1.1) в різних організаціях з серпня 2023 по березень 2025 найчастіше заблокованими загрозами є викрадення інформації, трояни та програми-вимагачі.

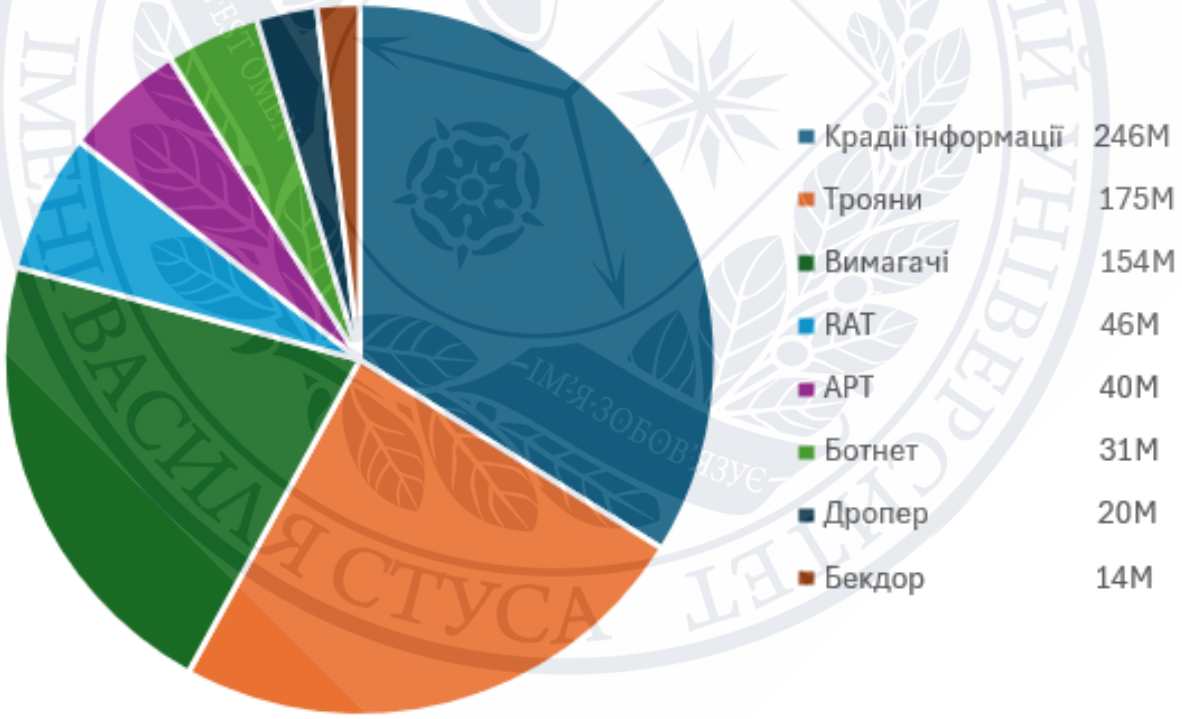

Рисунок 1.1 – Графік кількості доменів, заблокованих з серпня 2023 року по березень 2024 року

Саме тому аналіз DNS-запитів, виявлення та блокування небезпечних ресурсів є пріоритетом для кібербезпеки. У державному контексті це може

стосуватися реалізації політики інформаційного суверенітету, у корпоративному – захисту інфраструктури та персональних даних співробітників, у побутовому – безпеки домашніх мереж та дітей.

На цьому етапі варто ґрунтовніше розібрати, що таке DNS та як воно працює. DNS – це протокол, який «перекладає» зрозумілі для людей адреси, наприклад example.com, у зрозумілу для машини IP-адресу, наприклад 192.168.1.1. Розглянемо принцип роботи DNS [11] за схемою нижче (рис 1.2).

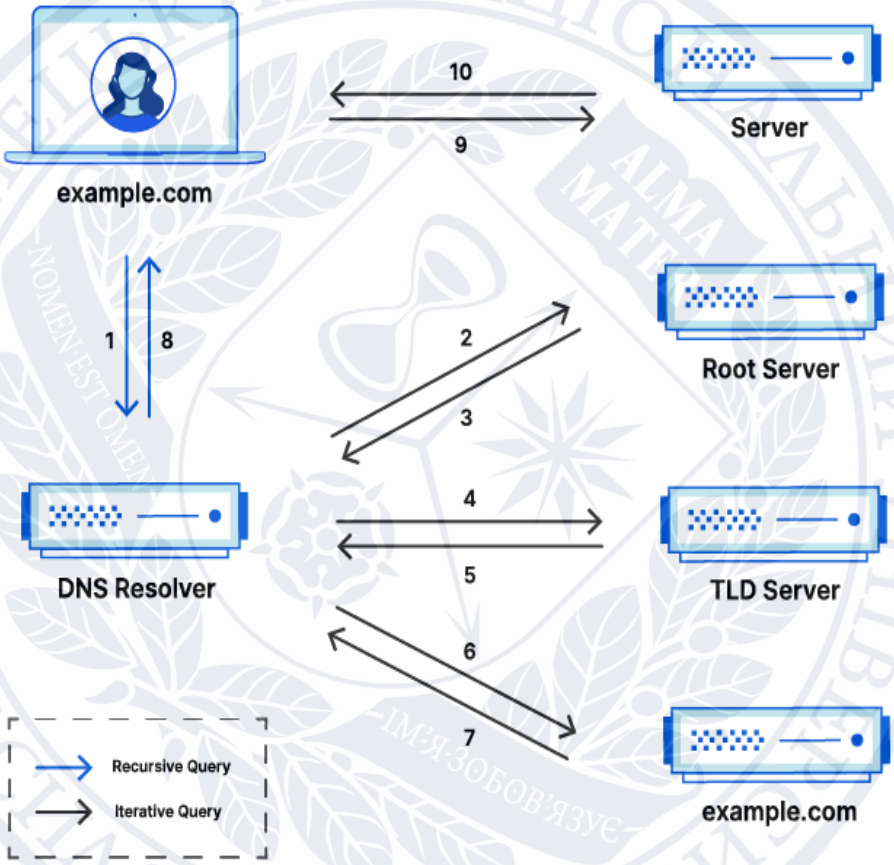

Рисунок 1.2 – Схема запитів для пошуку IP з DNS [11]

1. Користувач звертається через браузер до «example.com» і цей запит обробляється рекурсивним DNS-резолвером – сервером, що виступає посередником між клієнтом і системою доменних імен.
2. Резолвер звертається до кореневого DNS-серверу імен (позначеного символом .).

3. У відповідь він отримує адресу сервера домену верхнього рівня (TLD), наприклад .com або .net, залежно від того, до якого домену належить запитуване ім’я. У нашому прикладі – .com TLD.
4. Резолвер робить запит до .com TLD.
5. Потім сервер TLD відповідає IP-адресою сервера імен домену, example.com
6. Нарешті, рекурсивний резолвер надсилає запит на сервер імен домену.
7. IP-адреса example.com потім повертається резолверу з сервера імен.
8. Потім DNS-резолвер відповідає веб-браузеру IP-адресою домену, який запитувався спочатку.

1.2 Агрегація даних

Агрегація – це процес збору, об’єднання та первинної обробки інформації з різних джерел з метою її подальшого аналізу та використання. Через невпинне перенасичення інтернету контентом такі інструменти та сервіси допомагають організувати та полегшити пошук необхідної інформації[10].

Агрегація є початковим і критично важливим етапом у системі фільтрації загроз, адже саме вона формує базу даних, на основі якої будуються подальші механізми блокування. Якщо дані, що надходять на вхід, є неякісними, неповними або застарілими, то й система фільтрації виявиться неефективною або, що ще гірше, хибно позитивною.

Агрегація передбачає кілька ключових кроків:

- Завантаження даних з кількох джерел.
- Нормалізація форматів, тобто приведення даних до єдиного внутрішнього стандарту.
- Видалення дублікатів та записів із недійсними структурами.
- Фільтрація та попередня класифікація – наприклад, відокремлення phishing-доменів від botnet-адрес чи C2-серверів.
- Збереження у внутрішнє сховище – локальну базу даних (наприклад, SQLite), яке використовується наступними етапами системи.

Важливо підкреслити, що агрегація – це не просто механічне збирання даних, а також і їхня початкова аналітична обробка. На цьому етапі відбувається відсів недостовірних або малозначущих записів, а також визначення, які саме ресурси дійсно становлять загрозу.

Таким чином, агрегація є першою ланкою в логічному ланцюгу протидії інтернет-загрозам. Саме з цього процесу починається формування надійної системи фільтрації, яка дозволяє виявляти та блокувати потенційно шкідливу активність ще до того, як вона завдасть шкоди.

1.3 Аналіз аналогів та готових рішень

На сьогодні існує кілька підходів до побудови рішень для фільтрації небажаного та небезпечного інтернет-трафіку. Усі вони мають спільну мету – запобігти комунікації з ресурсами, які можуть нести загрозу. Водночас вони суттєво відрізняються за своєю архітектурою, принципом дії, гнучкістю та технологічними основами. Можна виділити такі основні типи:

1. Локальні системи DNS-фільтрації – встановлюються на маршрутизатор або в локальній мережі користувача; обробляють DNS-запити і блокують шкідливі домени до моменту встановлення з’єднання.
2. Хмарні DNS-платформи – забезпечують фільтрацію через зовнішній DNS-релевантний сервер, часто із застосуванням машинного навчання та поведінкової аналітики.
3. Автономні утиліти агрегації – скрипти або модулі, що збирають, обробляють та структурують інформацію з фідів (threat feeds), готуючи її для подальшого використання в системах блокування.

1.3.1 Локальні системи DNS-фільтрації

Pi-hole [12] – це одне з найвідоміших рішень у цій категорії. Програма функціонує як локальний DNS-сервер, що працює на пристрої в мережі (наприклад, Raspberry Pi або віртуальній машині). Коли клієнт запитує IP-адресу домену, Pi-hole перевіряє її у списках заблокованих і, якщо відповідність знайдена, повертає некоректну адресу (0.0.0.0) або зовсім не відповідає. Таким

чином, запобігається встановленню з’єднання з потенційно небезпечним або небажаним ресурсом.

Система підтримує підключення численних фідів блокування – списки доменів, які регулярно оновлюються спільнотою (наприклад, StevenBlack’s hosts). Програмне ядро написане на Bash та PHP, адміністрування здійснюється через зручну веб-панель.

AdGuard Home [13] є альтернативою Pi-hole, проте має дещо інші можливості. Він розроблений на мові Go і, окрім DNS-фільтрації, підтримує блокування HTTPS-запитів, фільтрацію за контентом (в тому числі рекламу, аналітику, шкідливе ПЗ) та локальний проксі-сервер. Особливо важливою є підтримка протоколів DNS-over-HTTPS (DoH) та DNS-over-TLS (DoT), що дозволяє не лише фільтрувати, а й шифрувати DNS-запити користувача. На відміну від Pi-hole, AdGuard має вбудовані списки та фільтри, подібні до тих, що використовуються у браузерних розширеннях.

1.3.2 Хмарні платформи DNS-фільтрації

DNSFilter [14] – це комерційна SaaS-платформа, яка забезпечує фільтрацію DNS-запитів на зовнішньому рівні. Вона обробляє понад 200 мільйонів загроз щодня, використовуючи інструменти машинного навчання для аналізу поведінки доменів. DNSFilter автоматично класифікує домени за категоріями (phishing, malware, botnet тощо), аналізує нові домени з моменту їх реєстрації, а також ідентифікує зловживання дешевими TLD.

Архітектурно, платформа працює як DNS-resolver з централізованим керуванням політиками, журналюванням і API-доступом. Однією з її переваг є інтеграція слабко контрольованих моделей ML, які дозволяють виявляти “поведінкові аномалії” – домени, що не мають репутації, але діють як шкідливі.

1.3.3 Автономні утиліти агрегації

Існує окрема категорія інструментів – це невеликі утиліти або скрипти, які виконують функцію агрегації списків загроз з різних джерел. Вони, зазвичай, реалізовані на Python, Go або Bash і працюють у командному рядку. До них належать:

- hBlock – об’єднує блокувальні списки в єдиний hosts-файл.
- Власні скрипти – що завантажують IoC-фіди з Abuse.ch, Phishtank, ThreatFox, URLhaus тощо.

Ці інструменти забезпечують гнучкість і автономність, дозволяючи користувачеві створювати власні фільтри відповідно до специфіки середовища. Проте вони потребують певного рівня технічної компетенції для розгортання й підтримки.

Огляд наявних програмних рішень показує, що підходи до фільтрації інтернет-ресурсів та обробки даних про кіберзагрози суттєво різняться залежно від цільового середовища, глибини аналізу й технічних вимог. Локальні DNS-фільтратори, як-от Pi-hole та AdGuard Home, ефективні для базового блокування доменів, проте мають обмежені можливості в плані класифікації загроз, агрегації фідів та поглибленої обробки даних.

Хмарні рішення на зразок DNSFilter демонструють високу точність, машинне навчання й поведінкову аналітику, однак вони залишаються закритими системами з обмеженою кастомізацією, що створює додаткові ризики в контексті конфіденційності. Їх інтеграція в систему потребує постійного доступу до Інтернету й платної підписки.

Автономні утиліти агрегації, як-от hBlock або кастомні Python-скрипти, забезпечують більшу гнучкість та можливість адаптації до конкретних вимог. Однак такі рішення здебільшого зосереджені на одному етапі – зборі або генерації списку, часто без класифікації чи автоматизованого тестування.

Висновок до розділу 1

У першому розділі було здійснено теоретичне обґрунтування актуальності дослідження, проаналізовано природу та типологію загроз, що походять від небезпечних або небажаних інтернет-ресурсів, розкрито роль системи доменних імен (DNS) у їх поширенні та виявленні, а також окреслено методи первинної протидії – зокрема, через фільтрацію DNS-запитів.

Проведено детальний огляд процесу агрегації як початкового етапу боротьби з загрозами, а також розглянуто типи та приклади існуючих технічних

рішень, зокрема локальні фільтратори (Pi-hole, AdGuard Home), хмарні платформи (DNSFilter) та автономні утиліти агрегації. У результаті порівняльного аналізу виявлено, що жоден із розглянутих підходів не забезпечує повної функціональної автономності, модульності та можливості класифікації загроз у межах єдиної відкритої системи. Це дозволяє обґрунтовано сформулювати новизну пропонованого програмного рішення.

На основі викладеного сформульовано такі завдання для наступного етапу дослідження:

- розробити архітектуру програмного комплексу на основі функціональних блоків (агрегація, парсинг, класифікація, генерація, тестування);
- створити конфігураційні файли, які задають структуру джерел та правила обробки;
- реалізувати модуль завантаження й збереження фідів у базу даних;
- розробити механізми попередньої обробки та фільтрації зібраних даних;
- реалізувати функціональність генерації вихідних списків блокування у різних форматах;
- розробити утиліту перевірки ресурсів (ping, DNS-запити) для підвищення точності;
- здійснити тестування працездатності системи та аналіз її ефективності.

РОЗДІЛ 2 РОЗРОБКА МЕТОДІВ, АЛГОРИТМІВ І СТРУКТУР ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМИ

2.1 Формалізація задачі та загальна модель системи

Задача фільтрації небезпечних інтернет-ресурсів належить до класу задач інформаційної безпеки, у яких вирішується проблема обмеження доступу до джерел, що несуть потенційну загрозу користувачам, організаційним системам чи державній інфраструктурі. Такі джерела можуть включати фішингові сайти, сторінки з розповсюдженням шкідливого ПЗ, ботнет-інфраструктури, анонімайзери, ресурси з протиправним контентом тощо.

З практичної точки зору, найбільш ефективною є фільтрація на рівні DNS-запитів або на рівні системного доступу. Для цього необхідно сформувати списки доменів або IP-адрес, що мають бути заблоковані. Такі списки повинні відповідати наступним характеристикам:

- структурованість;
- актуальність;
- сформовані на основі кількох зовнішніх джерел;
- готовими до використання у кількох форматах (залежно від цільової системи).

Таким чином, задача, яка вирішується в рамках цієї роботи, формалізується як створення системи агрегації, обробки та генерації даних про небезпечні інтернет-ресурси, яка б забезпечувала повний ланцюг обробки:

1. збір
2. уніфікація
3. зберігання
4. перевірка
5. генерація списків.

В основі побудови архітектури системи закладено принцип Unix Philosophy[15], а саме – концепція "одна програма – одна функція". Згідно з цією філософією, кожна утиліта має виконувати одне конкретне завдання, робити це

добре, і забезпечувати простий інтерфейс взаємодії з іншими частинами системи. Це дозволяє:

- створювати модульну структуру, де кожен компонент є незалежним;
- легко тестувати та масштабувати частини системи;
- запускати утиліти вручну або автоматизовано (через cron, скрипти);
- використовувати утиліти в інших системах або переносити незалежно.

На цій основі система поділяється на окремі логічні модулі:

- утиліта для завантаження (агрегації) фідів із зовнішніх джерел;
- утиліта для парсингу та нормалізації завантажених даних;
- утиліта для перевірки доступності ресурсів;
- утиліта для генерації списків у заданих форматах.

Всі ці компоненти взаємодіють через спільні структуровані формати –база даних, що виступає єдиним джерелом зберігання, та формат конфігураційного керування параметрами роботи утиліт. Це забезпечує гнучкість, простоту зміни налаштувань і повну автономність системи.

Логічна модель системи ґрунтується на ідеї послідовної обробки даних, наближеній до моделі ETL (Extract-Transform-Load)[17][18]:

- Extract – витягування даних із фідів (текст, JSON, CSV);
- Transform – очищення, фільтрація, уніфікація;
- Load – збереження в базу для подальшої генерації й аналізу.

Схематично цю модель можна подати у вигляді потокової діаграми що зображена на рисунку 2.1, де кожен блок відповідає одному з модулів системи, а між ними передається структурована інформація – або у вигляді проміжних файлів, або безпосередньо через БД.

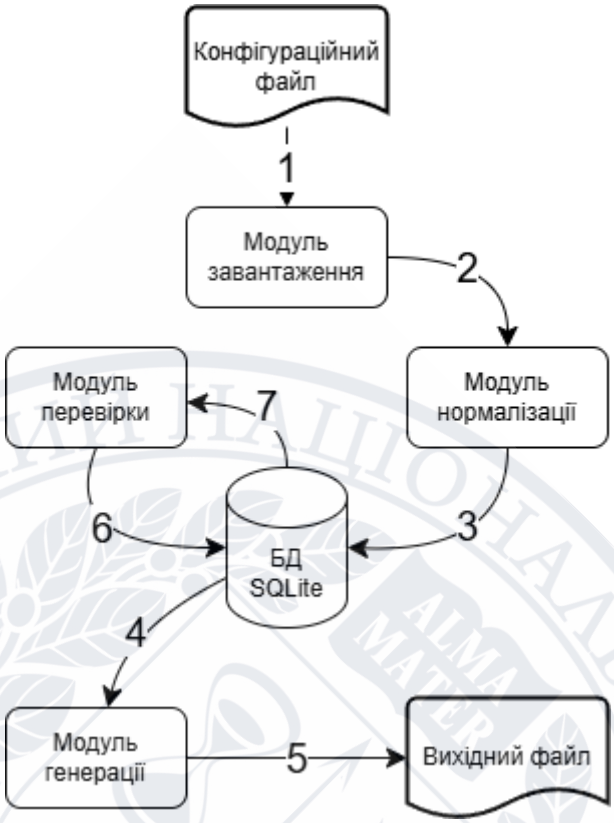

Рисунок 2.1 – data-flow діаграма роботи системи

1. Система починає роботу зі зчитування конфігураційного файлу, у якому міститься перелік джерел даних (URL або локальні файли), тип форматів, категорії загроз і параметри генерації. Цю конфігурацію читає модуль завантаження, який на її основі ініціює подальшу обробку.
2. Отримані сирі дані передаються з модуля агрегації до модуля нормалізації, де виконується попередня обробка. На цьому етапі відбувається уніфікація форматів, видалення дублікатів, технічних записів, а також підготовка до збереження у внутрішньому форматі.
3. Нормалізовані записи (домен/IP, категорія, джерело, дата, тощо) зберігаються у локальну базу даних. Саме вона виступає основним сховищем і джерелом даних для всіх наступних етапів. Записи можуть доповнюватися метаданими або статусами з інших модулів (наприклад, результатами перевірки).

4. Модуль генерації запитує з бази тільки актуальні, унікальні та валідні записи – залежно від обраних критеріїв у конфігурації (категорії загроз, активність, джерело). Ці дані використовуються для побудови списків блокування.
5. На цьому етапі система формує фінальні вихідні списки у заданих форматах: hosts, dnsmasq.conf, json, sqlite, unbound.conf тощо. Ці файли готові до інтеграції з ОС, маршрутизаторами або DNS-серверами.
6. Модуль перевірки (наприклад, ping, nslookup, dns.resolve) читає з бази записи, які ще не були перевірені або потребують повторної перевірки. Він тестує домени на доступність та актуальність і повертає результати у вигляді оновлених статусів.
7. Оскільки модуль перевірки працює незалежно від нормалізації, він самостійно отримує з бази необхідні записи. Це дозволяє перевіряти дані як у фоновому, так і у ручному режимі.

Цей підхід дозволяє створити систему, яка не лише вирішує поставлену задачу в рамках фільтрації загроз, а й є відкритою до розширення – наприклад, додавання нових типів фідів, підтримки нових форматів, інтерфейсів адміністрування тощо.

У наступному підрозділі буде розглянуто загальні підходи до агрегації, обробки та нормалізації напівструктурованих даних, які лягають в основу роботи із зовнішніми джерелами інформації про загрози.

2.2 Структура зберігання даних: моделювання, нормалізація, обґрунтування

Одним із ключових компонентів системи обробки та фільтрації шкідливих інтернет-ресурсів є внутрішнє сховище даних. Його призначення полягає у централізованому збереженні оброблених записів: доменів і IP-адрес, що потенційно становлять загрозу, разом із супровідною інформацією – категоріями, джерелами, датами додавання та статусами перевірки.

Щоб забезпечити стабільну роботу системи, її адаптивність до нових джерел, ефективність запитів та уникнення надлишковості, потрібно спиратися на принципи нормалізації даних, відомі в реляційній моделі. Нормалізація – це процес організації структури БД для зменшення надмірності та забезпечення логічної цілісності. У межах цієї роботи застосовано принципи трьох нормальних форм [19]:

- Перша нормальна форма: у записах таблиць груп даних що повторюються.
- Друга нормальна форма: перша нормальна форма та відсутність у записах не ключових даних, що залежать від частини первинного ключа таблиці.
- Третя нормальна форма: друга нормальна форма та відсутність у записах таблиць атрибутів, що залежать від інших не ключових атрибутів

Відповідно до цього, база даних моделюється як структура з трьох таблиць:

- threats – основна таблиця, що містить всі домени/IP та пов’язану інформацію;
- categories – довідник категорій загроз (наприклад, malware, phishing, anonymizer);
- types – довідник типів ресурсів (наприклад, domain, ipv4, ipv6).

Таке розділення забезпечує:

- уникнення дублювання текстових значень у кожному записі;
- стабільність структури при додаванні нових типів/категорій;
- спрощення фільтрації та сортування через числові ідентифікатори;
- логічну простоту подальших запитів.

Отже, зважаючи на вищеперелічене, потрібно побудувати загальну інформаційну структуру через чотири сутності(див. табл. 2.1)

Таблиця 2.1 План інформаційної структури

Сутність	Властивості
Конфігураційний файл	Список джерел (назва, URL, тип фіду), категорія для кожного джерела, формат вхідних даних (csv, txt…), параметри генерації (формат виводу, фільтрація, статус актуальності), режими перевірки (тип перевірки, періодичність), параметри логування або каталогів збереження.
Категорії загроз	Унікальний ідентифікатор категорії, назва категорії (наприклад, malware, phishing, anonymizer, botnet), короткий опис або позначення джерела класифікації (опціонально).
Типи ресурсів	Унікальний ідентифікатор типу, назва типу (domain, ipv4, ipv6), можлива форма валідації або приклади, логічне застосування (локальне, мережеве).
Запис про ресурс	Доменне ім’я або IP-адреса, тип (ідентифікатор з таблиці типів), категорія (ідентифікатор з таблиці категорій), джерело (назва або мітка), дата додавання, дата останньої перевірки, статус активності (так/ні).

Таким чином, структурна модель системи базується на принципах реляційного моделювання, нормалізована до третьої нормальної форми, та забезпечує логічну цілісність і розширюваність.

2.3 Вибір технологій для реалізації інформаційної структури

SQL-системи (Structured Query Language) базуються на реляційній моделі даних. Дані організовані у вигляді таблиць з чітко визначеною структурою: кожен стовпчик має фіксований тип, записи (рядки) мають однакову кількість полів, використовуються первинні та зовнішні ключі. Зв’язки між таблицями формалізуються, що забезпечує високу цілісність, передбачуваність і стабільність структури. SQL-системи підтримують складні операції: об’єднання таблиць, агрегування, вкладені запити, транзакції, сувору типізацію. Вони

ідеально підходять для систем, де критично важливими є узгодженість, жорстка схема та контроль над структурою даних. Прикладами є SQLite, PostgreSQL, MySQL.

NoSQL-системи (Not Only SQL) орієнтовані на гнучкість, масштабованість і здатність обробляти великі обсяги даних із динамічно змінною структурою. У таких системах схема може бути відсутня або визначатися на рівні кожного окремого документа. NoSQL охоплює кілька підходів: документоорієнтовані бази (MongoDB), графові (Neo4j), колоночні (Cassandra), сховища типу ключ-значення (Redis)[21]. Дані зазвичай подаються у форматах, які зручно обробляти як машинам, так і людям – зокрема, JSON, YAML, BSON, які підтримують вкладені структури, списки, словники. Coursera [20] зазначає, що ці формати стали основою сучасного NoSQL-підходу, дозволяючи відмовитися від суворої схеми та ефективно працювати з напівструктурованими або різнорідними даними.

Отож розглянемо та порівняємо ці формати:

- PostgreSQL [22] – потужна серверна СКБД, що підтримує транзакції, розширення, паралелізм і велику кількість одночасних користувачів. Призначена для складних багатокористувацьких або розподілених систем. Вимагає розгортання серверного середовища та адміністрування.
- SQLite [23] – це вбудована реляційна система керування базами даних, яка зберігає всі таблиці в одному локальному файлі. Вона не потребує запуску окремого сервера, працює автономно та має повну підтримку SQL. Її перевага – у простоті використання, відсутності залежностей та високій швидкодії при роботі з невеликими або середніми обсягами даних.
- MySQL [24] – ще одна популярна серверна реляційна СКБД. Відома своєю швидкодією, але вимагає окремого сервера та не підходить для локальної автономної роботи.

- MongoDB [25] – документоорієнтована NoSQL-база, яка зберігає дані у вигляді JSON-подібних об’єктів. Відрізняється гнучкою схемою, не вимагає заздалегідь визначеної структури. Придатна для динамічних даних, але менш контрольована у плані зв’язків між сутностями.
- JSON [26] – стандарт для обміну даними, який також використовується для конфігурацій. Є формалізованим і компактним. Часто використовується у веб-сервісах. Проте не має коментарів та не є дуже читаблельним.
- XML [27] – теговий формат для опису структурованих даних. Дає змогу суворо описати схеми, але є занадто громіздким і складним для ручного редагування у невеликих системах.
- INI [28] – формат конфігурацій, що дозволяє зберігати прості ключ-значення. Придатний для базових налаштувань, але не підтримує вкладені структури, що є критичним недоліком у контексті складної конфігурації фідів.
- YAML [29] – мова опису даних, яка добре підходить для конфігурацій. Підтримує вкладені структури, списки, словники, зручна для читання й ручного редагування, може біти повільнішою через текстовий формат. Використовується для керування поведінкою систем через конфігураційні файли.

Так, для реалізації легкого локального реляційного сховища обирано SQLite. Крім того для збереження конфігурації використовуватиметься YAML за його читабельність. Хоч він може працювати повільніше через формат записів, розмір планованого конфігураційного файлу та відносна рідкість використання в ході програми дозволяє нехтувати цією швидкістю.

2.4 Теоретичні засади агрегації та обробки неструктурованих даних

У побудові сучасних систем аналітики ключовим етапом є робота з даними, що надходять із численних неструктурованих або напівструктурованих джерел. У випадку системи фільтрації інтернет-ресурсів це – публічні фіди

(feeds), які містять перелік доменів або IP-адрес, що пов’язані з певними типами загроз. Дані з таких джерел потребують попередньої обробки, структурування та адаптації до внутрішньої моделі, перш ніж можуть бути збережені й використані для прийняття рішень.

Формалізованою моделлю такої обробки є ETL-процес – Extract (витягування), Transform (перетворення), Load (завантаження). У даному контексті:

- Extract – На цьому етапі система звертається до зовнішніх джерел даних — фідів, які містять списки потенційно небезпечних доменів або IP-адрес. Джерела можуть бути представлені у форматах .txt, .csv, JSON API тощо. Список джерел фіксується в конфігураційному файлі, зчитується відповідною утилітою, після чого вміст завантажується локально у відповідну директорію. У цьому контексті витягування є пасивним, але часто реалізується з періодичністю, що робить його частиною потокової ETL-парадигми (streaming ETL) при подальшому автоматичному розгортанні.
- Transform – це процес нормалізації, фільтрації, категоризації та приведення даних до внутрішньої уніфікованої структури. Дані, отримані на попередньому етапі, можуть мати нерівномірну структуру, дублікати, зайву інформацію або навіть некоректні записи. Утиліта parser виконує:
    - очищення (усунення коментарів, порожніх рядків, шкідливих символів);
    - нормалізацію (визначення типу ресурсу, категорії, джерела);
    - перетворення формату в уніфікований вигляд;
    - фільтрацію на основі регулярних виразів або логіки;
    - перевірку на дублікати та узгодження з первинними/зовнішніми ключами в БД.

Цей етап забезпечує приведення неструктурованих або напівструктурованих потоків до структурованої реляційної моделі, що відповідає вимогам внутрішнього сховища.

- Load – збереження даних у централізоване сховище для подальшої генерації списків блокування або аналітики. Після обробки дані зберігаються у внутрішню базу SQLite, яка функціонує як центральне сховище для подальшої роботи утиліт. У процесі збереження дотримуються принципів нормалізації: зовнішні ключі, окремі таблиці, обмеження тощо. Навантаження реалізується вставкою з перевіркою, що гарантує консистентність даних і зменшує надмірність.

Фіди загроз, з якими працює система, здебільшого мають напівструктурований або неструктурований характер. Вони надходять у різних форматах, з неоднорідною структурою, нерідко без жорстко заданої схеми. При цьому їх можна умовно класифікувати за форматом:

- TXT – прості списки, де в кожному рядку міститься один домен або IP. Часто містять службові символи, коментарі, або обфусковані домени (example[.]com, hxxp://...).
- CSV – файли з фіксованими колонками, де один зі стовпців містить домен або IP. Додаткові колонки можуть містити дату, категорію, джерело, опис.
- JSON – більш формалізований формат, часто використовується при доступі до API. Дані можуть мати вкладену структуру, типу { "data": [{...}, {...}] }.

Такі джерела часто є змінними в часі: структура записів може змінюватися, формати можуть бути оновлені, деякі фіди – неповні або містити зайву інформацію.

Оскільки дані з різних джерел не відповідають єдиному стандарту, для їх обробки використовуються адаптивні методи парсингу, що залежать від типу вхідного формату:

- для TXT – рядки проходять через фільтри: видаляються коментарі (#), порожні рядки, службові символи, виконується розпізнавання невизначених форм;
- для CSV – дані обробляються за іменами колонок або індексами. Необхідне поле (наприклад, url, host, ip) витягується, інші – ігноруються або використовуються для метаданих;
- для JSON – застосовується рекурсивне обходження структури: витягуються вкладені поля, визначаються відповідні значення через ключі.

Наступним етапом є нормалізація – приведення даних до уніфікованого представлення. У нашому випадку, для кожного запису визначаються такі атрибути:

- значення (домен або IP);
- тип (domain, ipv4, ipv6);
- категорія (malware, phishing, anonymizer тощо);
- джерело (назва фіду або API);
- дата отримання.

Для досягнення внутрішньої узгодженості даних застосовуються методи:

- шаблонного вилучення значень – наприклад, регулярні вирази[30] для виділення доменів у складених рядках;
- rule-based cleanup – набір правил, які дозволяють перетворити обфусковані записи на нормальну форму;
- перевірки формату – для валідації IP-адрес використовується синтаксичне приведення; для доменів – перевірка довжини, допустимих символів і TLD;
- категоризації за джерелом або ознаками – якщо у фіді присутнє поле threat_type або category, воно зчитується напряму. Якщо ні – категорія може бути задана вручну через конфігурацію.

Такі методи дозволяють перетворити нерівномірні вхідні потоки у стабільний потік структурованих записів, придатних до зберігання в реляційній базі.

Після нормалізації всі записи передаються до бази даних – спільного структурного сховища, яке:

- забезпечує єдине джерело істини;
- дозволяє фільтрувати, сортувати та оновлювати записи;
- підтримує зв’язки типу «багато до одного» з довідниками (типів, категорій);
- фіксує метадані (дата додавання, перевірка, джерело).

Таким чином, процес агрегації та нормалізації в системі спирається на сучасні підходи ETL та принципи обробки напівструктурованих даних. Це дозволяє перетворити неузгоджену інформацію з відкритих джерел на внутрішньо цілісну, контрольовану й готову до подальшої обробки – зокрема, перевірки та генерації у прикладні формати.

Висновок до розділу 2

У цьому розділі було здійснено деталізовану розробку архітектури, структур, алгоритмів та принципів роботи програмного комплексу, призначеного для агрегації, обробки та фільтрації небезпечних інтернет-ресурсів. Сформовано цілісну модель взаємодії між модулями системи, з акцентом на модульність, автономність та узгодженість логіки обробки на кожному етапі.

Першим етапом стало формалізоване подання загальної логіки роботи системи, відображене у вигляді схеми потоків даних. Далі було проаналізовано алгоритми збору даних з відкритих джерел, зокрема способи роботи з різними форматами фідів та методи нормалізації вхідної інформації. Запропоновано комбінований підхід до агрегації та уніфікації даних, який дозволяє забезпечити їх структурованість, валідацію та збереження в узагальненому форматі.

Визначено та обґрунтовано вибір реляційної моделі зберігання на основі SQLite. Структура бази даних описана як цілісна модель із фіксованими полями,

унікальними обмеженнями та підтримкою запитів, оновлення й фільтрації, необхідних для наступних етапів обробки. У таблиці зберігаються всі ключові параметри загроз, включно з типом ресурсу, категорією, джерелом, статусом активності та часовими мітками.

Алгоритм генерації вихідних даних розглянуто з урахуванням різних форматів представлення: від простих hosts-файлів до конфігурацій для dnsmasq та табличного вигляду у csv. Проаналізовано переваги й обмеження кожного формату, а також їхню відповідність до цільового середовища застосування.

Останнім елементом функціонального циклу стала перевірка актуальності ресурсів. Розглянуто алгоритми, які дозволяють за допомогою ping, DNS-запитів та інших методів автоматично визначати, чи є ресурс активним, і в залежності від цього приймати рішення щодо включення його до кінцевого списку.

У результаті проведеної роботи було розроблено повну логічну модель системи: визначено складові модулі, структуру даних, алгоритми обробки, взаємодії та генерації. Ця модель лягає в основу програмної реалізації, яка буде виконана на наступному етапі дослідження.

РОЗДІЛ 3 РОЗРОБКА УТИЛІТ

3.1 Організація проєкту та структури утиліт

Розроблений програмний продукт реалізовано як багатомодульну систему на мові програмування Python. Архітектурно він побудований за принципами Unix-філософії: кожна утиліта виконує окрему, чітко визначену функцію. Такий підхід дозволяє забезпечити максимальну автономність компонентів, простоту тестування, можливість повторного використання модулів та гнучкість під час розгортання.

Структура проєкту організована так, як вказано на рис. 3.1:

```
aggregator/
├── fetcher/      ← модуль для завантаження фідів
├── parser/       ← модуль для обробки (парсингу) даних
├── tester/       ← модуль для перевірки живості ресурсів
├── generator/    ← модуль для генерації вихідних форматів
├── config/       ← конфігураційні файли (YAML)
├── data/
│   ├── downloads/ ← збереження завантажених фідів
│   ├── parsed/   ← база даних SQLite (db.sqlite)
│   └── output/   ← сформовані результати генерації
```

Рисунок 3.1 – Структура проєкту «aggregator»

Усі утиліти є CLI-програмами (Command Line Interface), які можуть викликатися незалежно або по черзі відповідно до логіки роботи. Це дозволяє запускати кожен етап автоматично через планувальник задач (наприклад, cron), або вручну.

Основні утиліти:

- aggregator.fetcher – завантажує фіди згідно з конфігураційним YAML-файлом ;

- aggregator.parser – обробляє локальні файли, нормалізує дані та записує в базу даних;
- aggregator.tester – перевіряє актуальність кожного ресурсу, оновлюючи відповідні поля в БД;
- aggregator.generator – генерує списки блокування у форматах для робочих пристроїв, для роутерів, чи для досілдження.

Конфігураційний файл містить шляхи до всіх ключових директорій. Утиліти використовують ці шляхи для пошуку завантажених фідів, збереження вихідних списків та доступу до єдиної бази даних. Завдяки централізованій конфігурації, переміщення каталогу проєкту або переналаштування середовища не потребує змін у коді самих утиліт.

3.2 Структура бази даних

Для організації зберігання та обробки інформації про потенційно небезпечні інтернет-ресурси в системі використовується реляційна база даних SQLite. Такий вибір обумовлений її легкістю, простотою інтеграції в Python-проєкти, відсутністю необхідності запуску окремого серверу та повною автономністю. База даних є файлом (db.sqlite), який зберігається в директорії «data/parsed/».

SQLite відповідає реляційній моделі даних і підтримує основні елементи SQL: таблиці, індекси, унікальні обмеження, зовнішні ключі та прості транзакції. У контексті цього проєкту вона використовується як централізоване сховище між модулями parser, tester та generator.

У базі даних реалізовано три ключові таблиці:

- categories – містить список категорій шкідливих або небажаних ресурсів;
- types – визначає тип ресурсу: IP-адреса або доменне ім’я;
- threats – основна таблиця, що акумулює всі знайдені записи.

Кожна з таблиць створюється за допомогою SQL-інструкцій, об’єднаних у скрипт init_db.sql. Повний код скрипта наведений у додатку (Додаток Б), а нижче подано опис полів таблиць(таблицях 3.1-3.3)

Таблиця 3.1 – опис таблиці «categories»

Поле	**Тип**	**Призначення**
id	INTEGER PK	Унікальний ідентифікатор
name	TEXT UNIQUE NOT NULL	Назва категорії (наприклад, "malware")

Таблиця 3.2 – опис таблиці «types»

Поле	Тип	Призначення
id	INTEGER PK	Унікальний ідентифікатор
name	TEXT UNIQUE NOT NULL	Тип ресурсу ("domain", "ipv4")

Таблиця 3.3 – опис таблиці «threats»

Поле	**Тип**	**Призначення**
id	INTEGER PK	Унікальний запис
value	TEXT	Домен або IP-адреса
type_id	INTEGER FK	Зовнішній ключ до types(id)
category_id	INTEGER FK	Зовнішній ключ до categories(id)
source	TEXT	Джерело, з якого було завантажено
added_at	DATE	Дата та час додавання
is_alive	INTEGER	Статус (1 – доступний, 0 – недоступний, NULL – не перевірено)
last_date	DATE	Дата останньої перевірки
last_time	TIME	Час останньої перевірки

Отже, діаграма зв’язків таблиць БД буде виглядати як на рисунку 3.2:

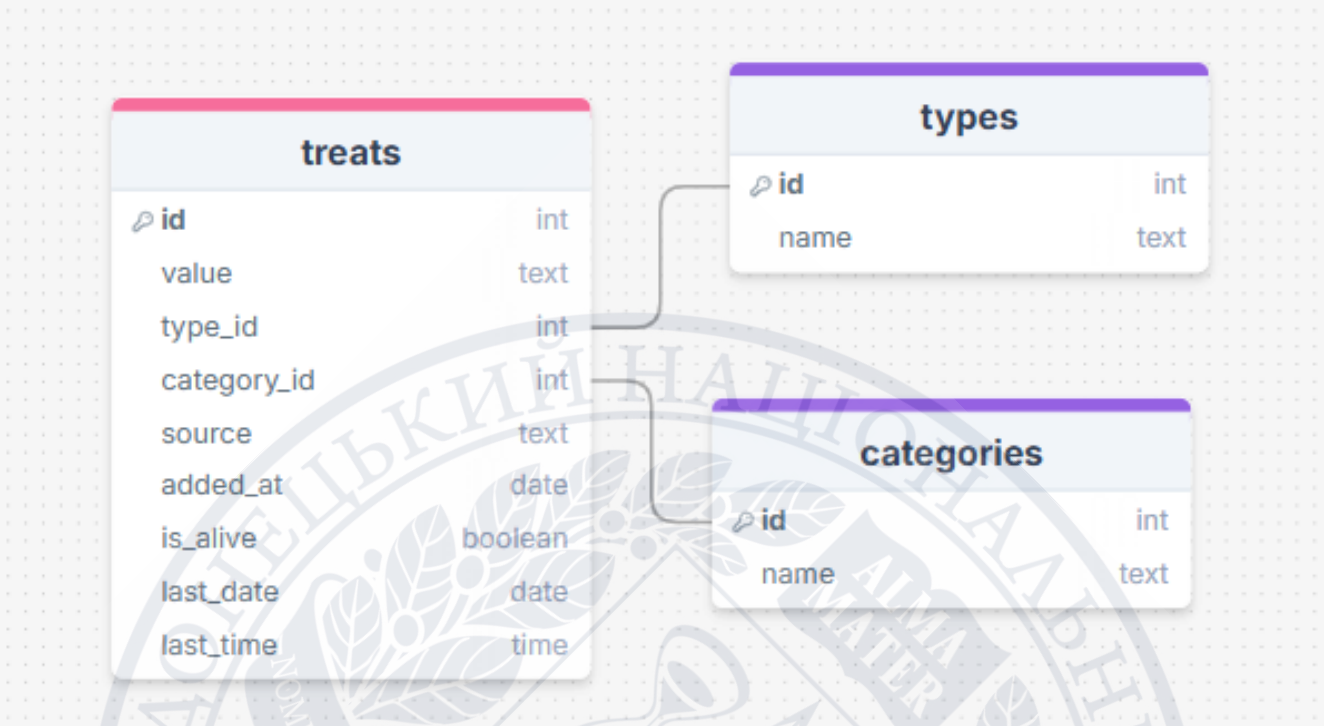

Рисунок 3.2 – Фрагмент діаграми зв’язків таблиць реалізованої БД

Поле value є унікальним у межах джерела (source). Це забезпечується обмеженням UNIQUE(value, source) у SQL-скрипті, що дозволяє уникати дублювання записів при повторному парсингу.

Загалом, база даних створюється за допомогою SQL-скрипту init_d.sql, який виконується окремо на етапі підготовки проєкту. Скрипт включає:

- видалення таблиць, якщо існували (DROP TABLE IF EXISTS);
- створення нових з усіма ключами та обмеженнями;
- автоматичне індексування первинних ключів.

Приклад команди виконання скрипта:

«sqlite3 data/parsed/db.sqlite < init_d.sql».

Усі утиліти звертаються до db.sqlite через модуль sqlite3. Основні операції:

- вставка даних (INSERT OR IGNORE);
- оновлення статусу (UPDATE threats SET is_alive = ...);
- фільтрація та агрегація (через SELECT).

У лістингу 3.3 представлено частину коду, яка вписує результат перевірки в таблицю «threats»:

```
def _check_row(self, row):
    if not self._should_check(row):
        return

    is_alive = False
    if row["type"] == "domain":
        is_alive = self._is_domain_alive(row["value"])
    elif row["type"] == "ipv4":
        is_alive = self._is_ip_alive(row["value"])

    now = datetime.now()
    last_date = now.strftime("%Y-%m-%d")
    last_time = now.strftime("%H:%M:%S")

    try:
        self.conn.execute(
            "UPDATE threats SET is_alive = ?, last_date = ?, \
            last_time = ? WHERE id = ?",
            (1 if is_alive else 0, last_date, last_time,
row["id"])
        )
        self.conn.commit()
        self.updated += 1
    except Exception as e:
        print(f"[X] Помилка при оновленні запису {row['id']}:
{e}")
```

Лістинг 3.1 – фрагмент коду програми «checker.py» утиліти terster

З опису таблиць можна побачити, що побудова структури відповідає вимогам 2-ї нормальної форми (2NF), оскільки всі залежності між атрибутами зведено до зовнішніх ключів, що виключає надлишковість. Розділення таблиць «types» і «categories» дозволяє централізовано оновлювати або змінювати типи без дублювання по всій базі.

3.3 Реалізація утиліти агрегації

Утиліта fetcher відповідає за початкову фазу життєвого циклу системи – автоматичне завантаження списків потенційно шкідливих ресурсів (фідів) з інтернету. Вона реалізує функцію агрегації даних з розподілених джерел та зберігає отримані файли в локальному каталозі data/downloads/ для подальшої обробки.

Програма реалізована як модуль із CLI-інтерфейсом (файл __main__.py), що викликає клас FeedFetcher (файл downloader.py). Основні етапи її роботи:

1. Зчитування конфігураційного YAML-файлу settings.yaml, де вказано:
   - список URL-адрес джерел;
   - шлях до папки для збереження (paths.download_dir);
2. Послідовне завантаження кожного фіду через HTTP/HTTPS;
3. Збереження кожного отриманого файлу з унікальним ідентифікатором у вигляді timestamp+назва;
4. Виведення логів про результати (успішно / помилка).

Розглянемо для ночності фрагмент коду, що відповідає за завантаження фідів (див. лістинг 3.2).

```
def download_file(self, url: str, category: str) -> bool:
    filename = self.generate_filename(category, url)
    filepath = os.path.join(self.download_dir, filename)
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        with open(filepath, "wb") as f:
            f.write(response.content)
        logger.info(f"[✔] Завантажено: {url} → {filepath}")
        return True
    except requests.RequestException as e:
        logger.error(f"[X] Помилка при завантаженні {url}: {e}")
        return False

def fetch(self, selected_categories: Optional[List[str]] = None):
    categories = self.config.get("categories", {})
    for category, info in categories.items():
        if selected_categories and category not in
selected_categories:
            continue
        urls = info.get("urls", [])
        if not urls:
            logger.warning(f"[!] Категорія '{category}' не містить
URL-адрес.")
            continue
        for url in urls:
            self.download_file(url, category)
```

Лістинг 3.2 – Два основні методи з downloader.py

Розроблена програма передбачає обробку таких винятків:

- неправильний формат URL;
- таймаут або недоступність сервера;
- невдале збереження файлу.

У разі виникнення помилки програма фіксує її в консолі, але не припиняє виконання – це дозволяє продовжити завантаження інших джерел.

Крім того, проводиться базова валідація: перевірка, чи містить посилання схему (http або https), та коректність отриманого вмісту.

Налаштування винесено в окремий YAML-файл settings.yaml (див лістинг 3.3), структура якого дозволяє адаптувати поведінку утиліти без зміни коду. Це відповідає принципам зменшення зв’язності та підвищення масштабованості програмних систем.[31]

```
gambling: #Сайти, пов'язані з терористичною діяльністю
  urls:
    - "https://blocklistproject.github.io/Lists/gambling.txt"
    -
"https://raw.githubusercontent.com/StevenBlack/hosts/refs/heads/ma
ster/alternates/gambling/hosts"

drugs: #Ресурси з інформацією про наркотики
  urls:
    - "https://blocklistproject.github.io/Lists/drugs.txt"

malware: #Ресурси зі шкідливим ПЗ
  urls:
    - "https://urlhaus.abuse.ch/downloads/text/"
```

Лістинг 3.3 – Фрагмент конфігураційного файлу settings.yaml

3.4 Реалізація утиліти парсингу

Після етапу завантаження файлів фідів з мережі, дані переходять до наступної фази обробки – парсингу. Утиліта parser виконує розбір збережених текстових файлів та перетворення вмісту у внутрішній уніфікований формат, що зберігається в базі даних SQLite.

Утиліта реалізована як окремий Python-модуль із CLI-запуском. Головний клас Parser розташований у файлі parser.py і викликається через файл __main__.py.

Парсер опрацьовує кожен файл у папці data/downloads/, ітеративно зчитуючи його рядки. На цьому етапі:

1. Перевіряє, чи містить файл не порожні дані.
2. Зчитує всі рядки, очищаючи їх від пробілів, коментарів (#) і службових символів.
3. Кожен рядок проходить базову перевірку регулярним виразом на відповідність:
    - доменному імені;
    - IP-адресі (формат IPv4).
4. Для кожного допустимого рядка:
    - визначається тип ресурсу (domain або ipv4);
    - додається джерело (source) – шлях до файлу або URL;
    - встановлюється час додавання (added_at) у форматі ISO 8601;
    - визначається категорія (наприклад, malware або proxy).
5. Дані зберігаються у таблиці threats через запити INSERT OR IGNORE (рис 3.3), та з обмеженням UNIQUE(value, source) щоб уникнути дублювання.

```
try:
    category = filename.split("_")[0].lower()
    source = "_".join(filename.split("_")[2:]) or "unknown"

    category_id = self.get_or_create("categories", category)

    with open(filepath, "r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            if line.startswith("#") or line.strip() == "":
                continue  # ігноруємо коментарі й порожні рядки
            value, value_type = self.extract_value(line)
            if not value or not value_type:
                continue

            type_id = self.get_or_create("types", value_type)
            added_at = datetime.now().strftime("%Y-%m-%d")

            self.conn.execute("""
                INSERT OR IGNORE INTO threats
                (value, type_id, category_id, source, added_at)
                VALUES (?, ?, ?, ?, ?)
            """, (value, type_id, category_id, source, added_at))

    self.conn.commit()

except Exception as e:
    print(f"[X] Помилка обробки файлу {filename}: {e}")
```

Рисунок 3.3 – Основний фрагмент коду, що парсить зібрані фіди

Для зручності користування передбачено вказання шляху до бази даних та шляху до папки з файлами-фідами. Аргументи, що це описують є частиною CLI модуля __main__.py (див рис 3.4)

```
def main():
    parser = argparse.ArgumentParser(description="Утиліта\
                                     для парсингу фідів та заповнення БД.")
    parser.add_argument(
        "--db",
        type=str,
        default="data/parsed/db.sqlite",
        help="Шлях до SQLite бази даних"
    )
    parser.add_argument(
        "--input",
        type=str,
        default="data/downloads",
        help="Шлях до папки з файлами для парсингу"
```

Рисунок 3.4 – Реалізація CLI інтерфейсу для утиліти parser

Розглянемо на прикладі. Нехай потрібно обробити файл з наступнм вмістом:

«# this is a comment

example.com

192.168.1.1

ftp.badactor.biz»

У такому випадку утиліта визначить:

- example.com та ftp.badactor.biz як домени;
- 192.168.1.1 як IP;
- # this is a comment буде проігноровано.

Ці значення буде збережено в базі даних, а їх тип (type_id) і категорія (category_id) визначатимуться через зовнішні таблиці types та categories.

Особливості реалізації

- Вставка запису відбувається тільки якщо такого value і source раніше не існувало.
- Значення added_at встановлюється як ISO-формат поточного часу.
- Значення is_alive, last_date і last_time залишаються порожніми до виконання перевірки.

Парсинг реалізовано з урахуванням масштабування: кожен формат фіду (txt, csv тощо) можна підтримати окремим парсером

Поділ процесу на окремі етапи агрегації, парсингу та зберігання базується на принципах ETL (Extract–Transform–Load), що широко застосовуються у системах обробки великих даних

3.5 Реалізація утиліти перевірки доступності

Перевірка життєздатності (активності) інтернет-ресурсів є критично важливою частиною системи. Сам факт того, що певний домен або IP-адреса на момент виявлення був у списку потенційно небезпечних, ще не означає, що він залишається таким на момент застосування блокування. Ресурси можуть бути деактивовані, переміщені, заблоковані або навіть змінити призначення. Саме

тому необхідно регулярно оновлювати статуси в базі даних, аби в актуальних списках блокування не опинялися застарілі чи неактивні записи.

З цією метою реалізовано автономну CLI-утиліту tester, яка призначена для автоматичної перевірки активності ресурсів, збережених у базі даних. Її основне завдання – встановити, чи відповідає вказаний домен або IP-адреса на запит у межах певного часу.

З точки зору архітектури модуль tester реалізовано у вигляді двох основних файлів:

- checker.py – містить клас ThreatChecker, що реалізує логіку перевірки;
- __main__.py – забезпечує зручний CLI-запуск з параметрами (--range, --id тощо).

Завдяки модульній структурі та підтримці параметрів запуску, tester легко інтегрується у будь-який планувальник задач або CI/CD-конвеєр. Наприклад, можна щогодини запускати утиліту на останні 50 записів, або щодоби – на всі нові за останній день.

Перевірка здійснюється за допомогою модуля socket, вбудованого у Python. Суть методу полягає у спробі встановлення з’єднання з ресурсом через TCP-протокол на стандартних портах 80 (HTTP) або 443 (HTTPS). Це дозволяє не тільки визначити, чи існує домен, але й отримати інформацію про його реальну активність у мережі.

Процес включає:

- створення TCP-сокета;
- застосування таймауту (2–5 секунд для кожної перевірки);
- фіксацію помилок (socket.timeout, socket.gaierror) як індикаторів недоступності;
- запис результату в БД через UPDATE threats SET is_alive = ...

Цей підхід є значно надійнішим за просте DNS-розпізнавання (nslookup), оскільки деякі небезпечні ресурси можуть все ще відповідати на DNS-запити, але не бути активними з точки зору мережевих з’єднань.

Щоб уникнути повторних звернень до тих самих ресурсів, у БД реалізовано поля last_date і last_time. Утиліта зчитує ці значення, і якщо від моменту останньої перевірки минуло менше ніж 5 хвилин, запис пропускається. Це дозволяє уникнути надмірного навантаження на мережу, зменшити ризики фільтрації з боку провайдерів або хостингів, а також підвищити етичність перевірок.

Ключовим елементом утиліти tester є модуль socket, який входить до стандартної бібліотеки Python і забезпечує інтерфейс для роботи з мережею на низькому рівні. Модуль базується на системних викликах POSIX-сумісних операційних систем (BSD sockets API) та реалізує кросплатформену взаємодію на рівні транспортного рівня (L4) моделі OSI – передусім TCP і UDP[33].

Основні принципи роботи бібліотеки socket:

1. Ініціалізація з’єднання:
    a. створюється об’єкт сокета (socket.socket()), який за замовчуванням використовує тип AF_INET (IPv4) і SOCK_STREAM (TCP);
    b. встановлюється таймаут для уникнення зависань (socket.setdefaulttimeout() або sock.settimeout()).
2. Розв’язання доменного імені:
    a. використовується функція socket.gethostbyname() або socket.getaddrinfo() для розв’язання імені у IP;
    b. важливо: ця операція залежить від налаштувань системи DNS (тобто клієнтська ОС).
3. Встановлення з’єднання:
    a. метод ((host, port)) пробує ініціювати TCP-з’єднання;
    b. якщо підключення успішне – ресурс вважається доступним.
4. Обробка винятків:
    a. socket.timeout: сервер не відповів;
    b. socket.gaierror: помилка розв’язання імені (ім’я неіснуюче або DNS не відповів);

c. ConnectionRefusedError: порт закритий або сервер недоступний.

Такий підхід дозволяє отримати результат без активного запиту HTTP/HTTPS, не інтерпретуючи жодних даних, не відкриваючи жодної сторінки та не виконуючи JavaScript, що повністю відповідає вимогам пасивної безпеки.

Порівняємо з іншими засобами (табл. 3.4):

Таблиця 3.4 – Порівняння інструментів доступності інтернет-ресурсів

Підхід	Бібліотека	Рівень доступу	Активність	Призначення
DNS запит	dns.resolver[34]	Службовий (L7)	Пасивний	Тільки перевірка запису DNS
HTTP-запит	Requests[35]	Високий (L7)	Активний	Повна взаємодія з веб-сервером
ICMP Ping	ping3 / OS[36]	Мережевий (L3)	Активний	Вимагає прав суперкористувача, обхід фаєрволів
socket	Socket[37]	Низький (L4)	Пасивний	Мінімальна перевірка доступності

На відміну від бібліотек вищого рівня (наприклад, requests, urllib, http.client), які реалізують повноцінний HTTP-протокол, бібліотека socket не обробляє відповіді сервера, не інтерпретує заголовки, не використовує cookies або сесії.

Такий спосіб:

- не викликає підозри в систем виявлення атак;
- не активує шкідливі скрипти;
- не порушує політики CORS/CSRF;
- не залишає помітного сліду у веб-логах цільового ресурсу.

Отже, до реалізованого через socket модуля checker.py можна звернутись наступним чином:

- «python -m aggregator.tester --id 123» – перевірити конкретні записи за ID;

- «python -m aggregator.tester --range 100-200» – перевіряти діапазон записів, наприклад з ID від 100 до 200;
- за замовчуванням – перевіряти останні 20 доданих записів

Приклад використання зображено на рисунках 3.5 та 3.6:

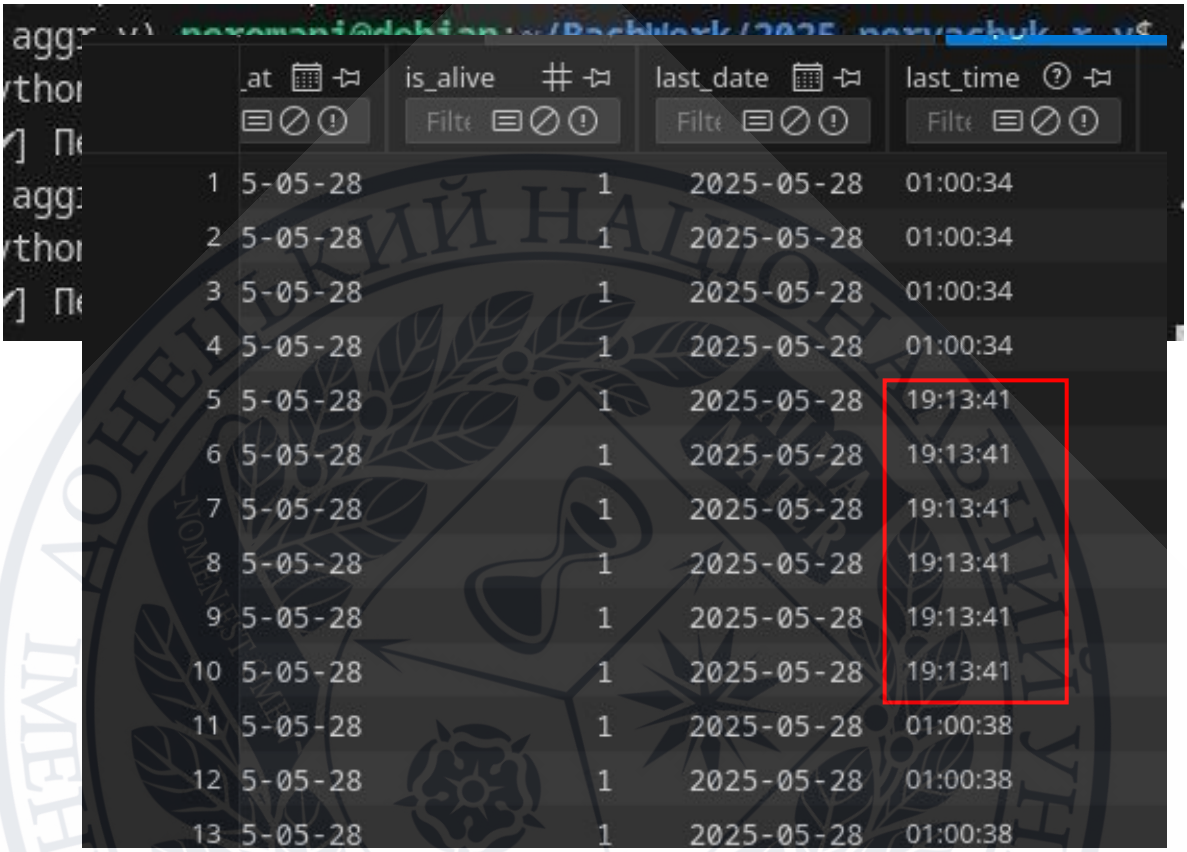

Рисунок 3.6 – Перевірена доступність ресурсів з id від 5 до 10

З прикладу видно, що програма не перевіряє щойно перевірені рядки та коректно виконує вкладені в неї функції.

3.6 Реалізація утиліти генерації

Завершальним етапом функціонального ланцюга системи є формування списків блокування на основі даних, зібраних, нормалізованих та збережених у внутрішній базі. Ці списки мають бути придатними для використання в конкретному середовищі – на рівні операційної системи, в налаштуваннях маршрутизатора або в інструментах адміністрування мережевого трафіку. Завдання цього етапу полягає у перетворенні внутрішньої структурованої моделі даних у текстовий або табличний формат відповідно до вимог кінцевої системи.

Утиліта generator створена для автоматичного формування вихідних списків блокування з бази даних SQLite на основі заданих критеріїв. Це дозволяє

адаптувати список до різних середовищ фільтрації – наприклад, hosts для локального комп’ютера, dnsmasq.conf для маршрутизатора або adblock для браузера.

Побудова списків починається із формування запиту до бази даних. Залежно від параметрів, заданих у конфігураційному файлі YAML, обираються лише ті записи, які відповідають зазначеним умовам: типу ресурсу, категорії загрози, джерелу та актуальності. Наприклад, може бути використано запит, який повертає всі живі домени категорії malware, додані не раніше ніж 30 днів тому. Це дозволяє адаптувати фільтрацію до актуального контексту загроз і потреб користувача.

У реалізації генерації використано клас BlocklistGenerator у файлі generator.py, який:

- зчитує конфігураційні параметри та аргументи запуску;
- формує SQL-запит з умовами фільтрації;
- обробляє результати через форматувальні шаблони;
- записує у відповідний файл у директорії data/output.

Процес генерації включає кілька алгоритмічних кроків:

1. Формування SQL-запиту з урахуванням фільтрів (category_id, type_id, is_alive, added_at).
2. Отримання результатів (value, category, type) із JOIN-запитом.
3. Форматування кожного запису відповідно до обраного формату.
4. Запис усіх рядків у вихідний файл (hosts, dnsmasq, adblock, csv).

Алгоритм має бути максимально відмовостійким, повторюваним і незалежним від формату: логіка вибірки та фільтрації залишається єдиною, змінюється лише кінцева форма подання даних. Завдяки цьому система залишається гнучкою та придатною до розширення – додавання нових форматів або сценаріїв використання не потребує зміни базової логіки.

Підтримується декілька типів форматів, що охоплюють основні сценарії використання в локальних і мережевих системах.

1. hosts – найпростіший і найпоширеніший формат. Кожен домен блокується перенаправленням на 0.0.0.0, чи іншу IP-адресу- заглушку(рис. 3.7). Універсальний, але має обмеження: не охоплює піддомени, не дозволяє гнучкого управління правилами.

```
# example.biz
216.194.171.230  www.example.biz
216.194.171.230  mail.example.biz
216.194.171.230  cpanel.example.biz
216.194.171.230  webmail.example.biz
216.194.171.230  example.biz
```

Рисунок 3.7 – Приклад hosts файлу [38]

2. dnsmasq – для мереж із власним DNS-резолвером [39]. Підтримується OpenWRT, MikroTik, DD-WRT. Забезпечує централізоване блокування на рівні локальної мережі. Приклад вигляду запису в цьому форматі зображено на рисунку 3.8:

```
address=/router/192.168.1.1
```

Рисунок 3.8 – Приклад формату запису для dnsmasq.conf [40]

3. CSV – табличне представлення для аналізу та дослідження (рис. 3.9). Зручно для перегляду в Excel або імпорту в аналітичні системи. Не придатне для прямого блокування.

example.csv > data

```
1  value,category,type
2  badexample.com,malware,domain
3
```

Рисунок 3.9 – Приклад CSV файлу

Отже, враховуючи вищеописане, доречно сформувати порівняльну таблицю для наочності (див табл. 3.5).

Таблиця 3.5 – Порівняльна таблиця форматів для збереження списків блокування

Формат	**Призначення**	**Підтримка систем**	**Гнучкість**	**Підходить для блокування**
hosts	ОС, ручне блокування	Windows, Linux, macOS	низька	+
dnsmasq	DNS-сервери, мережеве обланання	OpenWRT, MikroTik	середня	+
csv	Аналітика, дослідження	Excel, BigQuery, pandas	висока	-

Так, для формування відповідних форматів у програмі generator.py прописано відповідно змінну-шаблон (див лістинг 3.4):

```
FORMATTERS = {
    "hosts": lambda d: f"0.0.0.0 {d}",
    "dnsmasq": lambda d: f"address=/{d}/0.0.0.0",
    "adblock": lambda d: f"||{d}^"
}
```

Лістинг 3.4 – Шаблон для форматів block-list-файлів

За замовчуванням «генератор» збирає лише ресурси, що попередня утиліта додала до активних. Проте, у CLI є опція включити усі (див. лістинг. 3.5):

```
def main():
    parser = argparse.ArgumentParser(description="Утиліта
генерації списків блокування.")
    parser.add_argument("--db", type=str,
default="data/parsed/db.sqlite", help="Шлях до бази даних SQLite")
    parser.add_argument("--format", type=str, choices=["hosts",
"dnsmasq", "adblock", "csv"], required=True, help="Формат
вихідного файлу")
    parser.add_argument("--output", type=str, required=True,
help="Шлях до файлу, у який буде записано список")
    parser.add_argument("--category", type=str, help="Фільтрація
за категорією")
```

```
    parser.add_argument("--include-dead", action="store_true",
help="Включити ресурси, які недоступні")
```

Лістинг 3.5 – Перелік опцій, доступних в інтерфейсі для утиліти aggregator.generator

Серед інших опцій також можна побачити:

- Можливість прописати інший шлях до бд
- Вибір формату збереженого файлу
- Вибір місця збереження файлу
- Обрання категорії загрози, за якою скаладатиметься список

Запуск утиліти, як і до цього, виконується через командну стрічку (рис 3.10) (рис 3.11) з введенням потрібних параметрів:

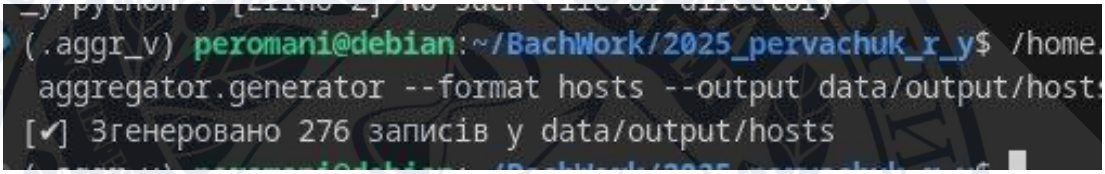

Рисунок 3.10 – Приклад введеня запиту для hosts

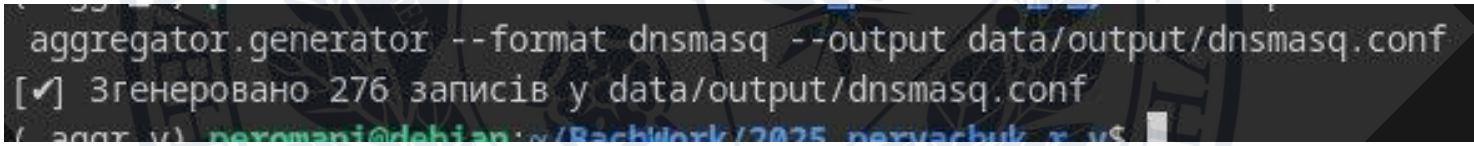

Рисунок 3.11 – Приклад введеня запиту для dnsmasq

Після виконання команд можна побачити відповідні сформовані файли (рис 3.12, рис 3.13)

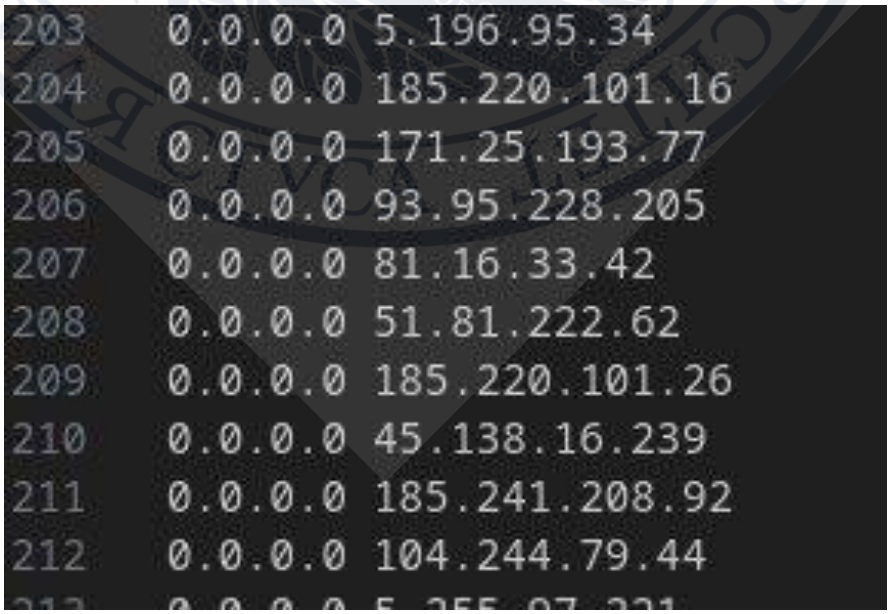

Рисунок 3.12 – Частина сформованого hosts-файлу

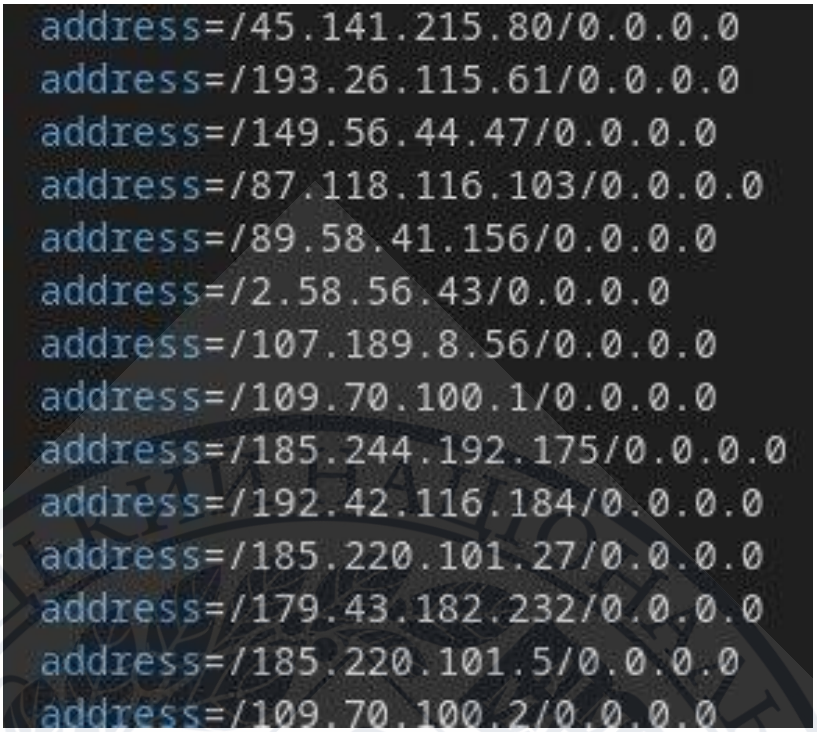

Рисунок 3.13 – Частина сформованого файлу dnsmasq

Можна побачити, що все спрацювало коректно.

Висновки розділу 3

У третьому розділі було реалізовано прикладну частину розробленої системи для агрегації, обробки та фільтрації небезпечних інтернет-ресурсів. Основну увагу зосереджено на побудові автономних утиліт, що виконують окремі етапи роботи системи згідно з принципами Unix-філософії: одна утиліта — одна задача.

Було створено та протестовано чотири основні модулі:

- fetcher — для завантаження списків фідів;
- parser — для розбору та нормалізації даних;
- tester — для перевірки активності ресурсів;
- generator — для генерації списків блокування у прикладних форматах.

У межах реалізації було визначено й створено базу даних SQLite зі структурою, що підтримує нормалізоване зберігання ресурсів із категоріями, типами, часом додавання та перевірки. Було розроблено SQL-скрипти для створення схеми таблиць, а також реалізовано обробку дублювань, фільтрацію за ключовими ознаками (категорія, статус, час).

Кожна утиліта забезпечує власну CLI-інтерфейсну оболонку, що дозволяє гнучко налаштовувати її поведінку. Зокрема, підтримується вхідна конфігурація у форматі YAML, яка дозволяє централізовано зберігати налаштування шляху до даних, джерел фідів та параметрів виводу.

Особливу увагу приділено безпеці: перевірка доступності доменів реалізована через низькорівневі мережеві з'єднання (socket), що виключає небажану взаємодію з потенційно шкідливими серверами. Процеси генерації результатів охоплюють кілька форматів, придатних до практичного використання в локальних ОС, браузерах, маршрутизаторах тощо.

На всіх етапах проєктування та реалізації дотримано принципів модульності, повторного використання коду, масштабованості та відповідності архітектурним підходам ETL-процесів.

Таким чином, результати розділу демонструють повноцінну реалізацію системи з мінімальною залежністю між компонентами, що дозволяє легко адаптувати її до нових сценаріїв використання, форматів даних або змін у джерелах. Отриманий результат є готовою основою для подальшого тестування, розгортання та оптимізації.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було комплексно досліджено проблему блокування небезпечних інтернет-ресурсів та реалізовано автономну програмну систему для їх агрегації, аналізу та фільтрації. Отримані результати дозволяють зробити наступні узагальнення та висновки:

1. Систематизовано поняття інтернет-загроз та їх класифікацію: виокремлено ключові типи шкідливих ресурсів (шкідливе ПЗ, фішинг, порнографія, анонімайзери тощо) та охарактеризовано їхні властивості, методи поширення й небезпеки, які вони створюють. Проведено аналіз актуальних загроз на основі сучасних аналітичних звітів та фідів.
2. Уточнено архітектурну модель системи фільтрації: запропоновано модульну багатофазну структуру на основі принципу «одна утиліта — одна функція», що відповідає підходу Unix Philosophy. Система побудована як ланцюг взаємопов’язаних автономних програм з чітко визначеними зонами відповідальності.
3. Встановлено доцільність застосування ETL-моделі (Extract – Transform – Load) для обробки загрозливих ресурсів з відкритих джерел. Реалізація цієї моделі дозволила забезпечити логічну структуру даних, масштабованість системи та можливість подальшої адаптації під інші джерела або нові формати.
4. Розроблено й реалізовано набір CLI-утиліт, кожна з яких виконує окрему частину завдання:
    - завантаження списків (агрегація);
    - парсинг та нормалізація;
    - перевірка активності;
    - генерація списків блокування.

    Усі утиліти підтримують конфігурацію через YAML та інтегруються у загальну систему.
5. Підготовлено та реалізовано структуру бази даних SQLite, що включає нормалізовану схему з таблицями threats, categories, types. Забезпечено

унікальність записів, простоту запитів, підтримку оновлень, фільтрації та аналітики.

6. Описано та реалізовано алгоритми формування вихідних даних, придатних до використання в локальних системах, DNS-рішальниках, браузерах або аналітичних інструментах. Сформовано формати hosts, dnsmasq, adblock, csv, що охоплюють основні сценарії застосування.
7. Обґрунтовано вибір інструментів та технологій, таких як Python, SQLite, YAML, socket API. Розглянуто альтернативи та показано доцільність обраних рішень для задачі автономної обробки та формування списків фільтрації.
8. Надано рекомендації щодо подальшого розвитку системи: розширення форматів, інтеграція з мережею моніторингу, застосування хмарного зберігання, побудова веб-інтерфейсу або інтеграція з firewall-системами.

Загалом, поставлену мету — розробити автономну систему агрегації та обробки інтернет-ресурсів для формування списків блокування — досягнуто. Отримані результати є теоретично обґрунтованими та практично реалізованими, що підтверджує доцільність обраного підходу та перспективність розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ali Rafiei Taghanaki. TMP Universal Journal of Research and Review Archives. CYBER THREATS ANALYSIS IN THE INTERNET OF THINGS. 2025, 254-256 с.
2. What is ransomware? URL: https://www.ibm.com/think/topics/ransomware (дата звернення : 02.04.2025)
3. Info stealer. URL: https://www.trendmicro.com/vinfo/us/security/definition/info-stealer (дата звернення: 24.04.2025)
4. What is Qakbot? URL: https://www.darktrace.com/cyber-ai-glossary/qakbot (дата звернення: 30.04.2025)
5. Remote Access Trojan (RAT) URL: https://www.fortinet.com/uk/resources/cyberglossary/remote-access-trojan (дата звернення: 27.04.2025)
6. Advanced Persistent Threats (APT) Explained URL: https://www.crowdstrike.com/en-us/cybersecurity-101/threat-intelligence/advanced-persistent-threat-apt/ (дата звернення: 25.04.2025)
7. Backdoor Attacks. URL: https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/backdoor-attack/ (дата звернення: 05.05.2025)
8. What is a Botnet? URL: https://www.crowdstrike.com/en-us/cybersecurity-101/malware/botnets/ (дата звернення: 06.05.2025)
9. Cisco, Cyber Threat Trends Report: From Trojan Takeovers to Ransomware Roulette. 2024
10. Mohsen Bahrami, A Perspective Towards Using Aggregated Data in Research. 2023, 2 с.
11. What is DNS? How DNS works. URL: https://www.cloudflare.com/learning/dns/what-is-dns/ (дата звернення: 05.05.2025)
12. Документація Pi-Hole. URL: https://github.com/pi-hole/docs (дата звернення: 10.05.2025)

13. AdGuard. URL: https://adguard.com/uk/welcome.html (дата звернення: 12.05.2025)
14. Introduction to DNSFilter. URL: https://help.dnsfilter.com/hc/en-us/articles/1500008104522-Introduction-to-DNSFilter (дата звернення: 11.05.2025)
15. Basics of the Unix Philosophy. URL: http://www.catb.org/esr/writings/taoup/html/ch01s06.html (дата звернення: 15.05.2025)
16. Koffka Khan. Lecture Notes on Unix Programming. 2024, 8-10 с.
17. What is ETL? URL: https://www.ibm.com/think/topics/etl (дата звернення: 15.05.2025)
18. Babatunde Akande. DATA MONITORING FOR COMPLIANCE IN ETL PROCESSES. 2025, 2 с.
19. Антонов Ю.С, Гончар В.М. Розробка клієнт серверних систем Методичні рекомендації до виконання лабораторних завдань, Вінниця, 2023, 19 с.
20. SQL vs. NoSQL: The Differences Explained + When to Use Each? URL: https://www.coursera.org/articles/nosql-vs-sql (дата звернення: 20.05.2025)
21. Harold Castro. Comparative Analysis of NoSQL vs. SQL Databases for Big Data Analytic. 2024, 2 с.
22. PostgreSQL Global Development Group. Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 20.05.2025)
23. Hipp, D. Richard. "SQLite Home Page." https://www.sqlite.org (дата звернення: 20.05.2025)
24. MySQL 8.4 Reference Manual. URL: https://dev.mysql.com/doc/refman/8.4/en/ (дата звернення: 20.05.2025)
25. MongoDB Inc. *NoSQL Databases Explained*. https://www.mongodb.com/nosql-explained (дата звернення: 20.05.2025)
26. Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON) URL: https://datatracker.ietf.org/doc/html/rfc4627.html (дата звернення: 20.05.2025)

27. XML introduction. URL: https://developer.mozilla.org/en-US/docs/Web/XML/Guides/XML_introduction звернення: 20.05.2025)
28. What is an INI file? URL: https://docs.fileformat.com/system/ini/#keysproperties (дата звернення: 20.05.2025)
29. YAML Language. *YAML* Official Website. UML: https://yaml.org/ (дата звернення: 20.05.2025)
30. Python Software Foundation. re – Regular expression operations. URL: https://docs.python.org/3/library/re.html (дата звернення: 20.05.2025)
31. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 191 с.
32. Frost R. Parser Combinators for Ambiguous Left-Recursive Grammars. San Francisco, 2018, 167-181 с.
33. Cisco Networking Academy. Communication Principles. URL: https://www.ciscopress.com/articles/article.asp?p=3192417&seqNum=6 (дата звернення: 20.05.2025)
34. Dnspython Contributors. The dns.resolver.Resolver and dns.resolver.Answer Classes. URL: https://dnspython.readthedocs.io/en/latest/resolver-class.html (дата звернення: 20.05.2025)
35. Requests: HTTP for Humans. URL: https://requests.readthedocs.io/en/latest/ (дата звернення: 20.05.2025)
36. Опис проєкту Ping3. URL: https://pypi.org/project/ping3/ (дата звернення: 20.05.2025)
37. Socket – Low-level networking interface. URL: https://docs.python.org/3/library/socket.html (дата звернення: 20.05.2025)
38. How to Modify Your hosts File Using Windows URL: https://www.inmotionhosting.com/support/website/modifying-your-hosts-file/ (дата звернення: 15.05.2025)
39. Dnsmasq documentation. URL: https://thekelleys.org.uk/dnsmasq/doc.html (дата звернення: 21.05.2025)

40. Dnsmasq. URL: https://wiki.archlinux.org/title/Dnsmasq (дата звернення: 15.05.2025)

ДОДАТКИ

ДОДАТОК А

Лістинг основних програм

Лістинг А.1 – програма «Downloader.py»

```
import os
import requests
from pathlib import Path
from datetime import datetime
from urllib.parse import urlparse
import yaml
from logging import getLogger
from typing import Optional, List

logger = getLogger("fetcher")

class FeedFetcher:
    def __init__(self, config_path: str = "config/settings.yaml"):
        self.config_path = config_path
        self.config = self.load_config()
        self.download_dir = self.config.get("paths",
{}).get("download_dir", "downloads")

        os.makedirs(self.download_dir, exist_ok=True)

    def load_config(self) -> dict:
        with open(self.config_path, "r") as f:
            return yaml.safe_load(f)

    def load_config(self) -> dict:
        with open(self.config_path, "r", encoding="utf-8") as f:
            return yaml.safe_load(f)

    def get_filename_from_url(self, url: str) -> str:
        parsed = urlparse(url)
        return os.path.basename(parsed.path) or "unknown_file.txt"

    def generate_filename(self, category: str, url: str) -> str:
        date = datetime.now().strftime("%Y%m%d")
        original_filename = self.get_filename_from_url(url)
        return f"{category}_{date}_{original_filename}"

    def download_file(self, url: str, category: str) -> bool:
        filename = self.generate_filename(category, url)
        filepath = os.path.join(self.download_dir, filename)

        try:
            response = requests.get(url, timeout=10)
            response.raise_for_status()
```

```
            with open(filepath, "wb") as f:
                f.write(response.content)

            logger.info(f"[✔] Завантажено: {url} → {filepath}")
            return True

        except requests.RequestException as e:
            logger.error(f"[X] Помилка при завантаженні {url}:
{e}")
            return False

    def fetch(self, selected_categories: Optional[List[str]] =
None):
        categories = self.config.get("categories", {})

        for category, info in categories.items():
            if selected_categories and category not in
selected_categories:
                continue

            urls = info.get("urls", [])
            if not urls:
                logger.warning(f"[!] Категорія '{category}' не
містить URL-адрес.")
                continue

            for url in urls:
                self.download_file(url, category)
```

Лістинг А.2 – програма «parser.py»

```
import os
import re
import sqlite3
from datetime import datetime
from ipaddress import ip_address

class Parser:
    # Ініціалізація парсера:
    # - db_path: шлях до SQLite бази
    # - input_dir: директорія з фідами
    def __init__(self, db_path: str, input_dir: str):
        self.db_path = db_path
        self.input_dir = input_dir

        if not os.path.exists(self.db_path):
            raise FileNotFoundError(f"Базу даних не знайдено:
{self.db_path}")
        if not os.path.exists(self.input_dir):
```

```
            raise FileNotFoundError(f"Вхідна директорія не
знайдена: {self.input_dir}")

        self.conn = sqlite3.connect(self.db_path)

    #Повертає ID з таблиці або створює новий запис.
    def get_or_create(self, table: str, value: str) -> int:
        cursor = self.conn.cursor()
        cursor.execute(f"SELECT id FROM {table} WHERE name = ?",
(value,))
        row = cursor.fetchone()
        if row:
            return row[0]
        cursor.execute(f"INSERT INTO {table}(name) VALUES (?)",
(value,))
        self.conn.commit()
        return cursor.lastrowid

    #Вилучає домен або IP з рядка та визначає тип (domain, ipv4,
ipv6).
    def extract_value(self, line: str):
        line = line.strip().lower()
        line = line.replace("[.]", ".").replace("hxxp", "http")

        match = re.search(r"(?:https?://)?([^/\s]+)", line)
        if not match:
            return None, None
        value = match.group(1)

        try:
            ip_address(value)
            return value, "ipv4" if '.' in value else "ipv6"
        except ValueError:
            if "." in value:
                return value, "domain"
        return None, None

    # Основний метод парсингу – проходить усі файли, витягує й
зберігає значення.
    def parse_all(self):
        for filename in os.listdir(self.input_dir):
            filepath = os.path.join(self.input_dir, filename)
            if not os.path.isfile(filepath):
                continue

            print(f"Обробка: {filename}")
            try:
                category = filename.split("_")[0].lower()
                source = "_".join(filename.split("_")[2:]) or
"unknown"
```

```
                category_id = self.get_or_create("categories",
category)

                with open(filepath, "r", encoding="utf-8",
errors="ignore") as f:
                    for line in f:
                        if line.startswith("#") or line.strip() ==
"":
                            continue  # ігноруємо коментарі й
порожні рядки
                        value, value_type =
self.extract_value(line)
                        if not value or not value_type:
                            continue

                        type_id = self.get_or_create("types",
value_type)
                        added_at = datetime.now().strftime("%Y-%m-
%d")

                        self.conn.execute("""
                            INSERT OR IGNORE INTO threats
                            (value, type_id, category_id, source,
added_at)
                            VALUES (?, ?, ?, ?, ?)
                        """, (value, type_id, category_id, source,
added_at))

                self.conn.commit()

            except Exception as e:
                print(f"[X] Помилка обробки файлу {filename}:
{e}")

        self.conn.close()
        print("[✔] Парсинг завершено.")
```

Лістинг А.3 – програма «checker.py»

```
import sqlite3
import os
import platform
import subprocess
import socket
from datetime import datetime, timedelta

class ThreatChecker:
    def __init__(self, db_path: str):
        self.db_path = db_path
        if not os.path.exists(db_path):
```

```
            raise FileNotFoundError(f"Базу даних не знайдено:
{db_path}")
        self.conn = sqlite3.connect(db_path)
        self.conn.row_factory = sqlite3.Row
        self.total = 0
        self.updated = 0

    def _is_domain_alive(self, domain: str) -> bool:
        try:
            socket.gethostbyname(domain)
            return True
        except:
            return False

    def _is_ip_alive(self, ip: str) -> bool:
        param = "-n" if platform.system().lower() == "windows"
else "-c"
        cmd = ["ping", param, "1", ip]
        try:
            subprocess.run(cmd, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL, timeout=3)
            return True
        except:
            return False

    def _should_check(self, row):
        if not row["last_date"] or not row["last_time"]:
            return True
        try:
            last_str = f"{row['last_date']} {row['last_time']}"
            last_check = datetime.strptime(last_str, "%Y-%m-%d
%H:%M:%S")
            return datetime.now() - last_check >
timedelta(minutes=5)
        except:
            return True

    def _check_row(self, row):
        if not self._should_check(row):
            return

        is_alive = False
        if row["type"] == "domain":
            is_alive = self._is_domain_alive(row["value"])
        elif row["type"] == "ipv4":
            is_alive = self._is_ip_alive(row["value"])

        now = datetime.now()
        last_date = now.strftime("%Y-%m-%d")
        last_time = now.strftime("%H:%M:%S")
```

```
        try:
            self.conn.execute(
                "UPDATE threats SET is_alive = ?, last_date = ?,
last_time = ? WHERE id = ?",
                (1 if is_alive else 0, last_date, last_time,
row["id"])
            )
            self.conn.commit()
            self.updated += 1
        except Exception as e:
            print(f"[X] Помилка при оновленні запису {row['id']}:
{e}")

    def _fetch_and_check(self, query, params=()):
        cursor = self.conn.cursor()
        cursor.execute(query, params)
        rows = cursor.fetchall()

        for row in rows:
            self.total += 1
            self._check_row(row)

        print(f"[✔] Перевірено: {self.total}, оновлено:
{self.updated}")

    def check_recent(self, limit=20):
        self._fetch_and_check("""
            SELECT t.id, t.value, t.last_date, t.last_time,
ty.name AS type
            FROM threats t
            JOIN types ty ON t.type_id = ty.id
            WHERE is_alive IS NULL
            ORDER BY id DESC
            LIMIT ?
        """, (limit,))

    def check_single(self, id_):
        self._fetch_and_check("""
            SELECT t.id, t.value, t.last_date, t.last_time,
ty.name AS type
            FROM threats t
            JOIN types ty ON t.type_id = ty.id
            WHERE t.id = ?
        """, (id_,))

    def check_range(self, from_id, to_id):
        self._fetch_and_check("""
            SELECT t.id, t.value, t.last_date, t.last_time,
ty.name AS type
            FROM threats t
```

```
            JOIN types ty ON t.type_id = ty.id
            WHERE t.id BETWEEN ? AND ?
        """, (from_id, to_id))
```

Лістиг А.4 – програма «Generator.py»

```
import sqlite3
import os
import csv

FORMATTERS = {
    "hosts": lambda d: f"0.0.0.0 {d}",
    "dnsmasq": lambda d: f"address=/{d}/0.0.0.0",
    "adblock": lambda d: f"||{d}^"
}

class BlocklistGenerator:
    def __init__(self, db_path):
        self.db_path = db_path
        if not os.path.exists(db_path):
            raise FileNotFoundError(f"Базу даних не знайдено:
{db_path}")
        self.conn = sqlite3.connect(db_path)
        self.conn.row_factory = sqlite3.Row

    def fetch_threats(self, category=None, only_alive=True):
        cursor = self.conn.cursor()
        query = """
            SELECT t.value, c.name AS category, ty.name AS type
            FROM threats t
            JOIN categories c ON t.category_id = c.id
            JOIN types ty ON t.type_id = ty.id
            WHERE 1=1
        """
        params = []

        if only_alive:
            query += " AND t.is_alive = 1"
        if category:
            query += " AND c.name = ?"
            params.append(category)

        cursor.execute(query, params)
        return cursor.fetchall()

    def generate(self, format_name, output_path, category=None):
        if format_name not in FORMATTERS and format_name != "csv":
            raise ValueError(f"Невідомий формат: {format_name}")

        threats = self.fetch_threats(category)
        os.makedirs(os.path.dirname(output_path), exist_ok=True)
```

```
        if format_name == "csv":
            with open(output_path, "w", newline="") as f:
                writer = csv.writer(f)
                writer.writerow(["value", "category", "type"])
                for row in threats:
                    writer.writerow([row["value"],
row["category"], row["type"]])
        else:
            formatter = FORMATTERS[format_name]
            with open(output_path, "w", encoding="utf-8",
newline="\n") as f:
                for row in threats:
                    f.write(formatter(row["value"]) + "\n")

        print(f"[✔] Згенеровано {len(threats)} записів у
{output_path}")
```

Лістинг А.5 – SQL-скрипт Init_db.sql для бази даних

```
CREATE TABLE IF NOT EXISTS categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS types (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS threats (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    value TEXT UNIQUE NOT NULL,
    type_id INTEGER NOT NULL,
    category_id INTEGER NOT NULL,
    source TEXT,
    added_at DATE,
    is_alive INTEGER,
    last_date DATE,
    last_time TIME,
    UNIQUE(value, source),
    FOREIGN KEY(type_id) REFERENCES types(id),
    FOREIGN KEY(category_id) REFERENCES categories(id)
);
```

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

_____

_____

_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____

_____

(дата) (підпис)