НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій (назва факультету(ННІ))	Комп'ютерних систем, мереж та кібербезпеки (назва кафедри)
_____ Болбот І.М., д.т.н, проф. (підпис) (ПІБ, вчене звання і ступінь)	_____ Касаткін Д.Ю., к. пед.н., доц. (підпис) (ПІБ, вчене звання і ступінь)
«___»_____2025 р.	«___»_____2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження комп'ютерної системи виявлення конфіденційних і захищених даних в інформаційному середовищі»

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи захисту інформації
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми
д.т.н., професор _____ Мамченко С.М.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи
д.т.н., доцент _____ Шкарупило В.В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Прохоров М.О.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
к.пед.н., доц. _____ Касаткін Д.Ю.
(вчене звання і ступінь) (підпис) (ПІБ)
«__» _____ 20__ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Прохорову Михайлу Олексійовичу
(прізвище, ім'я, по батькові)

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи захисту інформації
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи: «Дослідження комп'ютерної системи виявлення конфіденційних і захищених даних в інформаційному середовищі»

затверджена наказом ректора НУБіП України від "29" жовтня 2024р. № 1941 «С»

Термін подання завершеної роботи на кафедру 14 листопада 2025 р.

Вихідні дані до магістерської кваліфікаційної роботи Методологія розробки автоматизованої системи ідентифікації конфіденційних даних на основі регулярних та S-виразів та її переваги над існуючими Data Leak Prevention системами

Перелік питань, що підлягають дослідженню:

1. Аналіз предметної області методів автоматизації ідентифікації конфіденційної інформації
2. Дослідження використання регулярних та S-виразів для формалізованого опису конфіденційних файлів
3. Програмування демонстраційних програмних засобів виявлення конфіденційних даних
4. Тестування ефективності запропонованого методу на практичних прикладах
5. Аналіз отриманих результатів з точки зору переваг над існуючими DLP-рішеннями

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання " 29 **"** жовтня 2024 р.

Керівник магістерської кваліфікаційної роботи _____ Шкарупило В.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Прохоров М.О.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Магістерська робота викладена на 63 сторінках (без урахування додатків), містить 3 розділи, 11 рисунків, 3 таблиці та 5 додатків. Використано 30 допоміжних джерел.

Об’єкт дослідження – інформаційне середовище у вигляді файлової системи, яке потенційно містить аналітично незахищені та криптографічно (або, взагалі кажучи, стеганографічно) захищені конфіденційні файли.

Предмет дослідження – методи автоматизації виявлення конфіденційної інформації шляхом аналізу вмісту та структури файлів.

Мета роботи – розробка формального опису вмісту та структури певних файлів на основі регулярних виразів (для аналітично незахищеної інформації) та S-виразів (для захищеної інформації) з подальшим накопиченням таких описів в якості початкових даних для відповідної автоматизованої системи, або навчання певної нейромережі для подальшого залучення засобів так званого штучного інтелекту.

У першому розділі розглянуто загальні принципи ідентифікації конфіденційних файлів за особливостями їх структури, наявністю грифів, міток та інших ознак.

У другому розділі досліджено можливості використання регулярних виразів для формалізованого опису аналітично незахищених конфіденційних файлів та розроблено відповідний демонстраційний програмний агент.

У третьому розділі продемонстровано метод ідентифікації шифрованих файлів на основі формалізованих описів їх структури у вигляді S-виразів. В якості демонстраційного прикладу наведена добре відома структура шифрованого ZIP-архіву.

Розроблено демонстраційні програмні засоби для:

– виявлення за регулярними виразами конфіденційної інформації в текстових файлах;

– порівняльного виявлення шифрованих ZIP-архівів з та без використання відповідного S-виразу;

– інтерактивної побудови S-виразів за результатами візуального аналізу стійкої структурності певної множини екземплярів однотипних криптофайлів.

Ключові слова: конфіденційна інформація, шифрування, регулярні вирази, S-вирази, структура файлу, pattern matching, автоматизація виявлення, ідентифікація файлів, криптографічний захист, стеганографія, файлові формати, структурний аналіз, машинне навчання, нейронні мережі, інформаційна безпека, гриф секретності, мітки доступу, моніторинг даних, аудит, програмні агенти, прикладні системи, захист даних, криптозахищені файли, формалізований опис, інтерактивна побудова, аналіз вмісту, інтелектуальні системи, візуальний аналіз, демонстраційні засоби, Sensitive information type, Data Leak Prevention.

ЗМІСТ

ВСТУП

Відомо, що конфіденційність, цілісність та доступність (тріада CIA) – це три наріжні камені надійної системи захисту інформації [18]. Термін «конфіденційна» походить від англійського confidence – довіра, запобігання втраті. У контексті інформаційної безпеки конфіденційність визначається як властивість, що запобігає несанкціонованому ознайомленню (розкриттю) інформації особами, процесами чи системами. Прикладами конфіденційної інформації є робочі документи компаній, результати досліджень, особисті дані співробітників та клієнтів, відомості про фінансові операції та звіти, інтелектуальна власність, комерційна таємниця й інші компоненти, що мають високу цінність і значущість.

Захищеність інформації вимагає ретельного контролю доступу, формалізованої політики безпеки та постійного моніторингу [30] стану інформаційної інфраструктури.

З початком використання комп'ютерних технологій і переходом документообігу у цифрове середовище виникла низка проблем, пов'язаних саме із захистом конфіденційності, оскільки більшість колишніх адміністративних заходів стали непридатними. Навіть сучасні системи запобігання вторгнень, антивіруси чи міжмережні екрани не завжди здатні нейтралізувати випадковий або умисний витік інформації [14].

Окремо слід зазначити двоїстий характер поняття «захисту інформації»:

– збереження даних, ідентифікованих як конфіденційні, від витоку чи несанкціонованого доступу (через політики доступу, шифрування тощо);

– виявлення, ідентифікація та аналіз даних у контексті проведення власних кібероперацій або оцінки загроз.

Таким чином, своєчасна ідентифікація конфіденційної інформації стає найактуальнішим питанням у сфері її захисту.

Зазвичай використовують суб'єктивний, нечіткий, неформальний метод опису ознак конфіденційних даних, який унеможливлює автоматизацію їх

розпізнавання та контролю. Тому постає потреба у створенні формального, алгоритмічного підходу до вирішення цієї задачи.

Мета роботи – залучення новітніх методів формализованого опису структурних особливостей конфіденційних документів для автоматизації їх ідентифікації в інформаційному середовищі.

Досягнення цієї мети передбачає:

- аналіз існуючих підходів до пошуку та виявлення конфіденційних файлів у комп'ютерних системах;
- дослідження доцільності застосування регулярних виразів для виявлення необхідних текстових документів;
- відображення структурних особливостей криптозахищених файлів за допомогою S-виразів (продемонстровано на прикладі шифрованого ZIP-архіву);
- розробку прототипу програмного засобу для демонстрації пошуку конфіденційних файлів за формалізованими описами їх структурних особливостей.

Об'єкт дослідження – автоматизація процесу ідентифікації конфіденційних та зашифрованих файлів у комп'ютерних системах.

Предмет дослідження – методи аналізу та формального опису структурних ознак файлів за допомогою регулярних виразів (для відкритих документів) і S-виразів (для структурованих і зашифрованих даних).

Методологічна основа роботи включає системний аналіз, формалізацію та опис структурних ознак даних, а також створення і тестування програмних прототипів пошуку конфіденційної інформації.

В якості засобів реалізації використані мови програмування Python та C++, стандартні бібліотеки для роботи з файлами, регулярними виразами, а також інструменти для візуалізації структурованих даних.

Практична цінність роботи полягає у створенні працюючих прототипів відповідного програмного забезпечення, які можуть бути використані при розробці власних корпоративних систем захисту інформації та рішень класу DLP (Data Leak Prevention) для внутрішнього аудиту і моніторингу без залучення сторонніх послуг.

Структура роботи включає вступ, три розділи, висновки, список використаних джерел та додатки. У першому розділі викладені загальні принципи ідентифікації конфіденційних файлів за структурними ознаками, у другому – розглянуто застосування регулярних виразів для пошуку відкритих документів і створення демонстраційного програмного агента, у третьому – продемонстровано застосування S-виразів для ідентифікації криптозахищених файлів (на прикладі шифрованих ZIP-архівів).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ МЕТОДІВ ІДЕНТИФІКАЦІЇ КОНФІДЕНЦІЙНОЇ ІНФОРМАЦІЇ

1.1 Конфіденційність, обумовлена специфікою форматів та структурними особливостями вмісту

У комп'ютерному середовищі дані – це інформація, що зберігається у формі, здатній для переміщення та обробки. Для зручності в якості даних будемо розглядати файлову систему, а для вихідних даних частини програмних рішень – текстові файли, оскільки зазвичай існують певні можливості конвертування в них інших форматів файлів.

Цікавими з точки зору конфіденційного вмісту вважаємо:

– «відкриті» файли із вмістом специфічної структури, яка явно позначає даний документ, як конфіденційний;

– файли – результати криптографічних перетворень, оскільки сама процедура шифрування подекуди використовується зловмисниками або шкідливим програмним забезпеченням, що опинилось у комп'ютерному середовищі, для викрадення конфіденційних файлів;

– файли – стеганографічні контейнери (в роботі не розглядаються);

– програмні вироби, що реалізують криптографічні та стеганографічні перетворення, оскільки сама їх наявність у конкретному комп'ютерному середовищі зазвичай потребує певних пояснень.

Менш розповсюджені інструменти для приховування конфіденційних даних, наприклад, перетворення PDF на QR-код, тощо, в роботі не розглядаються.

Моніторинг та аудит конфіденційної інформації – головні засоби підтримки безпеки даних та дотримання нормативних вимог [14]. Відмінність полягає в тому, що моніторинг надає безперервний контроль над даними, відстежуючи будь-які зміни чи аномалії, а аудит передбачає періодичний детальний розгляд дотримання системами встановлених стандартів та правил. Оптимізація цих засобів – важливе завдання, яке вирішується за рахунок більш прискіпливого використання

специфічних властивостей електронних документів. Втрату конфіденційної інформації практично важко виявити, бо вона може бути спричинена:

– ненавмисними помилками користувачів, операторів, системних адміністраторів та інших осіб, які обслуговують інформаційні системи (найчастіша і найбільша небезпека);

– крадіжками та підробкою;

– загрозами від стихійних ситуацій зовнішнього середовища та зараження вірусами.

Специфічні властивості структури вмісту конфіденційних файлів в першу чергу обумовлені системою грифування документів. Грифи конфіденційності та обмеження доступу являють собою реквізит (елемент, службову позначку, послід), що свідчить про конфіденційність відомостей, що містяться в документі, і проставляється у самому документі. Найбільш поширеними є грифи:

– «Конфіденційно», «Конфіденційна інформація»;

– «Строго конфіденційно», «Строго конфіденційна інформація», «Особливий контроль»;

– «Для службового користування» («ДСК»).

Гриф обмеження доступу на документі пишеться повністю, тобто не скорочується. Під позначенням грифа зазначаються номер екземпляра документа, термін дії грифа та інші умови його зняття. Якщо на паперових документах гриф розташовується на першому та титульному аркушах документа, а також на обкладинці справи у правому верхньому кутку, то на електроних документах гриф позначається на всіх аркушах. Нижче грифа, або нижче адресата можуть позначатися обмежувальні прикмети: «Особисто», «Тільки в руки», «Тільки адресату», «Особисто в руки» та інші. Документи, конфіденційні в цілому, у своїй масі (наприклад, документація служби персоналу, служби безпеки, документи, віднесені до професійної таємниці тощо) зазвичай не маркуються, тому що в повному обсязі мають суворе обмеження доступу до них персоналу. Але вони теж мають певні структурні особливості. На цінних, але не конфіденційних,

документах може проставлятися послід (позначка, напис, штамп), що передбачає особливу увагу до збереження таких документів: «Власна інформація фірми», «Інформація особливої уваги», «Копії не знімати», «Зберігати в сейфі», тощо.

1.2 Сучасний стан розвитку технологій аудиту та моніторингу конфіденційної інформації

В основі пошуку, визначення рівня конфіденційності даних і їх захисту лежить практика класифікації даних. Вона дозволяє:

– дізнатися місця розташування, обсяг, контекст, тощо, незалежно від того, де вони знаходяться в комп'ютерній системі;

– знайти дані, які зазвичай залишаються прихованими, наприклад, неструктуровані дані;

– скоротити обсяг конфіденційних даних доти, доки вони не стануть необхідні для роботи, щоб знизити ризик.

В існуючих автоматизованих системах ідентифікація конфіденційної інформації здійснюється за допомогою класифікаторів (SIT – Sensitive information type), які працюють на основі певних шаблонів та ключових слів [9]. Умовно такі шаблони можна розділити на:

– попередньо створені, які відповідають загальним типам даних, що зазвичай вважаються конфіденційними, наприклад, номер банківського рахунку, номер посвідчення водія, номер паспорта, фізичні адреси, номер податкового файлу, номер медичного рахунку, тощо;

– користувальницькі шаблони, які створюються на основі вимог певних організацій;

– іменовані ідентифікатори сутностей, що включають складні ідентифікатори на основі словника, такі як фізичні адреси певних країн, тощо;

– набори шаблонів точного зіставлення даних, які створюються з урахуванням фактичних конфіденційних даних;

– шаблони відбитків документів, що базуються на форматі документів, а не на їхньому вмісті, тому що часто використовувані форми можуть мати «візерунки» слів, які є унікальними для них;

– шаблони, що стосуються мережевої чи інформаційної безпеки.

Після ідентифікації конфіденційної інформації за допомогою шаблонів, отримані знання використовуються для реалізації основних принципів управління інформацією: забезпечення захисту даних, запобігання втратам і ефективного управління.

Існуючі автоматизовані системи захисту даних від втрати (DLP) – це, взагалі кажучи, засіб для зниження інформаційних ризиків, який допомагає запобігти витоку конфіденційної інформації. Політики захисту від втрати даних налаштовуються на основі «розташування», «умов» та «дій». DLP-системи забезпечують контроль за переміщенням даних, їх обробкою та зберіганням, їх основне завдання – запобігання втрати чи витоку даних. Головна частина будь-якої DLP системи – вбудований механізм визначення ступеня конфіденційності документів через:

– аналіз спеціальних маркерів документів;

– аналіз їх вмісту.

Перший спосіб дозволяє уникнути похибок другого роду, але вимагає попередньої класифікації документів. Похибки першого роду (пропуски конфіденційної інформації) при цьому цілком імовірні. Другий спосіб, природно, інколи спрацьовує помилково. DLP системи бувають активними, які блокують передачу даних, та пасивними, які лише фіксують дії користувачів. Активні системи більш ефективні у запобіганні випадковим витокам, але можуть негативно позначитися на бізнес-процесах, тоді як пасивні системи менш інвазивні, але можуть не справлятися з систематичними витоками. Також DLP-системи можуть бути поділені на шлюзові та хостові, залежно від того, де вони встановлюються та як здійснюють контроль за даними.

Розробка DLP-систем інтегруються в загальний контекст безпеки і грунтується на концепції «нульової довіри», тобто кожна спроба доступу супроводжується підтвердженням та автентифікацією незалежно від того, наскільки взаємодіючі з системою елементи довірені. Такий підхід забезпечує захист як від зовнішніх загроз, так і від внутрішніх зловмисників.

Загальний обсяг світового ринку DLP-рішень оцінюється в 400 мільйонів доларів [22]. Ключова тенденція їх розвитку – перехід від «платних» систем до єдиних інтегрованих програмних комплексів. Це дозволяє уникнути проблем сумісності, спростити керування налаштуваннями для великих масивів клієнтських робочих станцій та більш ефективно контролювати всі канали передачі даних.

Приклади сучасних реалізацій DLP-систем:

– Symantec Data Loss Prevention, одна з найпопулярніших систем, використовує комплексний підхід до захисту даних, включає механізми мінімізації ризиків витоків через різні канали;

– McAfee Total Protection for Data Loss Prevention, фокусується на захисті даних на рівні кінцевих точок та мереж, має просунуті можливості моніторингу та миттєвої реакції на інциденти;

– Digital Guardian, спеціалізується на захисті даних у реальному часі, використовує технології машинного навчання для виявлення аномалій в поведінці користувачів, підтримує захист даних на рівні програм та файлів;

– Forcepoint Data Loss Prevention здатна проводити поведінковий аналіз ризиків витоку даних, пропонує інтеграцію з хмарними сервісами та мобільними пристроями;

– Microsoft Information Protection, частина екосистеми Microsoft 365, пропонує інструменти захисту даних як у cloud, так і на локальних пристроях. Дозволяє класифікувати та захищати дані за допомогою міток та керувати доступом до них, легко інтегрується в існуючі IT-інфраструктури.

Недоліки DLP-систем:

– висока вартість впровадження та довгострокової підтримки, яка часто виправдовується використанням технологій машинного навчання та штучного інтелекту, що не завжди надає чітке розуміння застосовуваних алгоритмів;

– співпраця із зовнішніми постачальниками, залежність від їхньої надійності, ризик передачі конфіденційної інформації стороннім організаціям, яким довіряється забезпечення безпеки та моніторинг даних.

2 ІДЕНТИФІКАЦІЯ КОНФІДЕНЦІЙНОСТІ КРИПТОГРАФІЧНО НЕЗАХИЩЕНИХ ФАЙЛІВ

2.1. Регулярні вирази, як засіб опису структурних особливостей даних

Регулярний вираз – це стиснутий опис (шаблон, pattern), певної множини рядків текстового фрагменту [2]. Якщо абстрактному процесору регулярних виразів надати рядки текстового фрагменту та конкретний регулярний вираз, отримаємо підрядки, які задовольняють цьому виразу – шаблону. Взагалі кажучи, вихідний текстовий фрагмент будується з друкованих (видимих) символів, невидимих символів (наприклад, простий пробіл або символи, які з якихось причин не можуть бути виведені на пристрій) та керуючих послідовностей, які потрібні тільки пристрою відображення (символи табуляції, символи перенесення рядка, тощо). Хоча керуючі послідовності в більшості випадків не друкуються, процесор регулярних виразів не ігнорує їх запис у вихідному тексті і вважає їх частиною фрагмента, що входить. Вперше регулярні вирази були задіяні у Unix-редакторі ed [2]. Команда редактора, яка дозволяла шукати за регулярними виразами рядки з файлу, була настільки корисна, що була розроблена програма grep, яка наразі портована на безліч платформ. Початковий синтаксис grep постійно, але безсистемно доповнювався. Як тільки якісь програмні вироби набували популярності, ставали популярними і реалізовані в них діалекти. Одним із найперших стандартів стали Basic Regular Expressions (BRE), що використовувалися в Unix-системах [3]. Вони зберігають значення більшості символів як літералів, що забезпечує простоту запису. Попри те, що POSIX визначив цей синтаксис як застарілий, він і досі залишається важливим через зворотну сумісність [2, 6]. Наразі популярні наступні діалекти:

– POSIX діалекти BRE (програма Grep) та ERE (від Extended Regular Expressions, программа Egrep);

– Perl діалект, мова програмування Perl;

– PCRE діалект [7] (від Perl Compatible Regular Expressions), мова PHP, різноманітні графічні текстові редактори. PCRE пропонують багатіший синтаксис,

ніж ERE, підтримують додаткові метасимволи, просунуті групи та інші можливості для складніших шаблонів;

– ECMAScript діалект, мови специфікації ECMA (наприклад, JavaScript), включається за замовчуванням у стандартній бібліотеці STL, починаючи з стандарту C++11;

– Python діалект, мова Python.

Незважаючи на таку різноманітність, більшість базових речей залишаються у всіх діалектах незмінними. А саме, регулярний вираз завжди складається з літералів та метасимволів (англ. Wildcards, що буквально означає «шалена карта», під wildcard розуміється не сам метасимвол, а метод його інтерпретації процесором). Літерали відповідають конкретним символам результату, а метасимволи лише диктують вимоги для вмісту певних позицій результату. Будь-який символ у шаблоні сприймається як простий літерал, якщо він не є метасимволом [3]. Щоб метасимвол сприймався як простий літерал, його потрібно екранувати метасимволом. Наприклад, `\\` – це літерал `\`.

Будь-який простий літерал, на якому застосований метасимвол екранування, продовжує залишатися простим літералом (наприклад, `\k` і «k» еквівалентні). Екранування потрібне лише для метасимволів. Для звичайних літер воно нічого не змінює. Однак, слід бути обережним, тому що не всі реальні процесори регулярних виразів дотримуються цього твердження.

Для багатьох мов програмування регулярні вирази дозволяють витягати дані зі строк і обробляти їх як масиви або колекції [4]. Якщо групи іменовані, їх можна сприймати як ключ-значення у словниках, де імена груп виступають ключами. Це особливо корисно під час роботи з великими обсягами даних, коли потрібно максимально ефективно організовувати та отримувати доступ до досліджуваних елементів [4].

Водночас слід враховувати, що не всі конфіденційні документи містять явні патерни, також, частина даних може навмисно маскуватися або бути зашифрованою. Тому регулярні вирази повинні використовуватися із залученням

додаткових перевірок, наприклад, валідації структури знайденого значення [9] або контрольної суми для типових ідентифікаторів.

Усі символи послідовності простих літералів сприймаються як прості літерали, якщо вони є метасимволами.

Для більшої гнучкості, регулярні вирази підтримують модифікатори, які впливають на різні аспекти пошуку. Наприклад, чутливість до регістру (i), обробка багаторядкових даних (m і s), а також глобальний пошук (g). PCRE додають модифікатори, як A, D, X, і u, що розширюють можливості обробки даних, зокрема для UTF-8.

Реалізації NFA (недетерміновані скінченні автомати) використовують «жадібний» алгоритм з відкатом [2, 3], перевіряючи всі можливі розширення регулярного виразу і вибираючи перше відповідне значення. Вони здатні обробляти підвирази і зворотні посилання, але через відкати можуть перевіряти одні й ті самі ділянки тексту кілька разів, що може знижувати продуктивність. Оскільки NFA у своїй класичній формі приймає перший знайдений збіг, це може перешкодити виявленню найдовшого збігу. Однак існують модифікації NFA, такі як у GNU sed, які відповідають стандарту POSIX і продовжують пошук для виявлення найдовшого збігу.

DFA (детерміновані скінченні автомати) працюють більш ефективно, оскільки не вимагають відкатів і гарантовано знаходять найдовший рядок із можливих варіантів, не перевіряючи текст повторно [2, 3, 6, 26]. Однак вони не можуть обробляти зворотні посилання і не підтримують конструкції з явним розширенням, такі як підвирази. DFA широко використовуються в таких інструментах, як lex і egrep, де потрібне ефективне опрацювання регулярних виразів без необхідності складних розширень

У всіх діалектах регулярних виразів метасимволи можна умовно поділити на кілька категорій, кожна з яких виконує окрему функцію у формуванні шаблону.

Першу категорію становлять прості метасимволи, що зазвичай складаються з одного символу (`\`, `^`, `.`, `$`, `|`) і позначають характерну частину текстового фрагмента. Наприклад, крапка (`.`) у більшості діалектів відповідає будь-якому

символу, окрім символу нового рядка. Перерахування (або альтернатива), що позначається вертикальною рискою `|`, дозволяє обирати з кількох можливих варіантів. Наприклад, вираз «gray|grey» охоплює обидва написання слова, враховуючи різницю в орфографії. До простих метасимволів також належать пари з дужок: квадратні `[ ]`, що формують класи літералів, та круглі `( )`, які утворюють групи. Наприклад, «gr(a|e)y» і «gray|grey» обидва описують варіанти написання слів, зберігаючи при цьому зміст різноманіття вибору. У традиційних регулярних виразах символи між квадратними дужками ([…]) можна заперечувати за допомогою символу `^`. Це означає, що вираз [^abc] відповідає будь-якому символу, окрім «a», «b» або «c», а вираз [^a-z] – будь-якому символу, крім малих латинських літер. Крім того, метасимвол `^`, використаний на початку виразу, вказує на початок тексту або рядка в багаторядковому режимі, а `$` вказує на кінець тексту чи рядка.

Другу категорію становлять екрановані спеціальні символи. Метасимволи регулярних виразів, такі як точка (.) або квадратні дужки ([]), вимагають використання зворотного слеша для їх інтерпретації як буквальні символи. Наприклад, щоб використовувати крапку як літерал, потрібно вказати «\.». Аналогічно, для квадратних дужок та інших спеціальних символів необхідно використовувати зворотний слеш для їх екранування. Щоб уникнути цього, можна подвоїти зворотний слеш (\\), і він буде сприйнятий як звичайний символ. Це важливо для того, щоб уникнути непередбаченого трактування цих символів як частини регулярного виразу. Регулярні вирази використовують не тільки буквальні символи, але й спеціальні метасимволи для вираження більш складних шаблонів. Якорі (`^` і `$`) визначають позицію збігу в рядку [6, 7]. Приклади метасимволів та їхні значення:

– «^Привіт» відповідає рядку, що починається з «Привіт»;

– «бувай$» відповідає рядку, що закінчується на «бувай»;

– «^Привіт Бувай$» задає точну відповідність, коли рядок починається і закінчується фразою «Привіт бувай».

Екранування спеціальних символів – деякі символи (^, ., [, $, (,), |, *, +, ?, {, }) мають спеціальне значення і потребують екранування зворотним слешем:

– \$\d відповідає рядку, де після символу $ йде цифра;
– непридатні до друку символи позначаються так:
  - \t – табуляція;
  - \n – новий рядок;
  - \r – повернення каретки.

Шаблон в контексті регулярних виразів представляє собою рядок, який може включати як літерали, так і спеціальні символи. Наприклад, вираз $\d{4}$ вимагає, щоб вказана позиція містила рівно чотири десяткові цифри після символу $.

Третю категорію становлять квантифікатори, які використовуються для опису серій символів. Вони завжди прив’язані до іншого метасимволу або літералу шаблону. Конструкція \{x,y\} дозволяє задати кількість повторень, де x відповідає мінімальній кількості, а y – максимальній. Наприклад, a\{3,5\} відповідає рядкам ааа, аааа або ааааа. Це вимагає використання зворотного слеша для забезпечення коректної інтерпретації, що залежить від конкретної реалізації регулярних виразів. Наприклад, egrep та Perl можуть по-різному інтерпретувати зворотні слеші перед метасимволами, такими як дужки або вертикальні палички. Тож, важливо знати тонкощі синтаксичного розбору в конкретному контексті.

Також квантифікація в регулярних виразах дозволяє вказати, скільки разів символи можуть повторюватися. Наприклад, {n,m} позначає кількість повторень від n до m включно, {n,} – n і більше разів, {,m} – не більше m разів. Спеціальні квантифікатори включають ? для 0 або 1 разу, * для 0 і більше разів, та + для 1 і більше разів. Конкретна реалізація цих конструкцій може варіюватися в залежності від процесора регулярних виразів, який використовується для аналізу тексту.

Квантифікатори (`*`, `+`, `?`, `{}`) керують кількістю повторень символів:

– abc* відповідає рядку, де після «ab» іде 0 або більше символів «c»;
– abc+ вимагає хоча б один символ «c» після «ab»;

– азбука? відповідає рядку, в якому після «азбука» йде 0 або 1 символ «к»;

– abc{2} відповідає рядку з рівно двома символами «c» після «ab»;

– abc{2,} – від двох і більше;

– abc{2,5} – від двох до п’яти;

– a(bc)* і a(bc){2,5} аналогічним чином обробляють послідовності з групуванням.

За замовчуванням квантифікатори є «жадібними» – вони намагаються захопити максимально можливу кількість символів. Якщо потрібно зробити їх «лінивими», тобто обмежити до мінімально необхідного збігу, варто додати ? після квантифікатора. Наприклад, шаблон <.+?> у тексті HTML захопить лише окремі теги, а не весь вміст між ними. Щоб уникнути використання крапки (.), яка відповідає будь-якому символу, можна застосовувати точніші конструкції, наприклад <[a-zA-Z]+> для знаходження тегів.

Четверту категорію становлять флаги, які дозволяють змінювати поведінку регулярного виразу. Шаблон зазвичай записується у вигляді /abc/, де він оточений косими рисками /. Після них указуються флаги, що можуть комбінуватися для зміни стандартних налаштувань:

– g – глобальний пошук, знаходить усі збіги в тексті, а не тільки перший;

– m – багаторядковий режим, змінює поведінку ^ і $, дозволяючи їм визначати початок і кінець кожного рядка, а не всього тексту;

– i – нечутливість до регістру, /aBc/i відповідатиме «abc», «ABC», «aBC», «AbC» тощо.

П’яту категорію становлять класи літералів, які описують конкретну множину допустимих символів, наприклад:

– \t - клас, що складається з одного символу табуляції;

– \d (PCRE) або [:digit:] (BRE) – клас, що представляє десяткові цифри.

В регулярних виразах багато діапазонів символів залежать від налаштувань локалізації. Стандарт POSIX визначив спосіб оголошення деяких класів і категорій символів. Наприклад:

– [:upper:] відповідає [A-Z] для великих літер;
– [:lower:] – для малих ([a-z]);
– [:alpha:] включає як великі, так і малі літери [A-Za-z].

Для цілих чисел використовується клас [:digit:], що відповідає [0-9], або скорочується до \d. Шестнадцяткові цифри позначаються [:xdigit:], що охоплює [0-9A-Fa-f]. Символи пунктуації визначаються як [:punct:], а пробіли та табуляція належать до класу [:blank:] (що відповідає [\t]). Клас [:space:] включає всі види пропусків та нові рядки, тобто [\t\n\r\f\v] (також можна позначити як \s).

Шосту категорію становлять уявні метасимволи (якорі), які не розкриваються в якийсь видимий символ, а покликані позначити місце у фрагменті, наприклад:

– PCRE \b – межа слова;
– \A – початок тексту;
– \Z – кінець тексту.

Сьому категорію становлять символічні класи, які спрощують запис шаблонів і є скороченнями для поширених множин символів. Символічні класи (\d, \w, \s і .) спрощують запис шаблонів:

– \d відповідає цифрі;
– \w – будь-якому символу слова (букви, цифри, підкреслення);
– \s – пробільному символу (включаючи табуляцію та перенесення рядка).

Крапка (.) відповідає будь-якому символу, але її слід використовувати обережно – альтернативи через класи символів часто точніші та швидші. Заперечення символічних класів позначаються як \D, \W, \S – вони шукають символи, що не належать до відповідного класу [6]. Ці класи підвищують зручність і продуктивність регулярних виразів, особливо в комбінації з квантифікаторами.

Восьму категорію становлять розширені метасимволи, які значно розширюють можливості стандартних виразів за рахунок додаткових конструкцій для умовного збігу та посилань. Для збільшення можливостей стандартних виразів розширені регулярні вирази включають додаткові метасимволи. Наприклад, границі слів (\b та \B) дозволяють вказати початок або кінець слова. Вираз \babc\b відповідатиме «abc», якщо це слово є на межі іншого слова. Також існують зворотні посилання, що дозволяють посилатися на вже захоплені групи в шаблоні, наприклад, у виразі ([abc])\1 ми знаходимо два однакові символи поруч. Важливою особливістю є перевірки виду (?=) та (?<=), що дозволяють задавати умови для наявності або відсутності певного шаблону перед або після основного, при цьому не включаючи його в результат. Ці можливості значно підвищують гнучкість регулярних виразів при складній обробці текстів, дозволяючи створювати шаблони для валідації або парсингу без зайвого захоплення тексту.

Дев’яту категорію становлять просунуті групи, які дозволяють вводити умови пошуку у шаблон без включення їх у підсумковий збіг, наприклад, група позитивного прямого перегляду (Positive Lookahead) – (?=). Скобочні групи в регулярних виразах дозволяють групувати частини шаблона. Стандартна група «a(bc)» створює групу зі значенням «bc». Можна відключити захоплення, використовуючи оператор ?:, як у «a(?:bc)*», що корисно, коли потрібно лише логічне об’єднання без збереження результату. Також можна присвоювати імена групам, використовуючи синтаксис «(?P<name>bc)» [4, 7], що в подальшому спрощує вилучення потрібних даних. Іменовані групи підвищують читабельність коду та полегшують доступ до захопленої інформації. Просунуті групи дозволяють вводити умови пошуку без включення їх у підсумковий збіг. Наприклад, група позитивного прямого перегляду (Positive Lookahead) позначається як (?=...). Так, вираз foo(?=bar) знайде «foo», якщо після нього одразу йде «bar», але «bar» не увійде в збіг. Аналогічно, негативний прямий перегляд ((?!...)) працює навпаки – foo(?!bar) знайде «foo» лише якщо після нього не йде «bar».

Ці конструкції, разом з іншими розширеними (наприклад, lookbehind (?<=...) та (?<!...)), дозволяють створювати складніші шаблони без зайвого захоплення тексту.

Десяту категорію становлять довільні класи літералів, які дають змогу точно визначати множини символів через перелік. Довільний клас літералів використовує метасимвол `[]`, у якому перераховуються всі бажані літерали. Усередині цього метасимволу роздільників немає. Якщо ви знаєте про клас літерала на певній позиції фрагмента, то краще використовувати метасимволи типу клас літералів. Довільний клас літералів завжди розкривається в один із допустимих літералів, якщо не використовується будь-який квантифікатор. Всередині класу можуть створюватись діапазони з погляду таблиці кодування. Діапазон створюється за допомогою тире символу. Наприклад, щоб створити клас із цифр з 1 по 7, потрібно написати [1-7]. У цьому сенсі тире сприймається не як символ класу, а як діапазон. Щоб включити тире до класу треба написати його першим чи останнім, тобто [-1-7] або [1-7-]. Всередині класу також можна екранувати літерали, які можуть трактуватися подвійно: наприклад, щоб створити клас із символів -, ^ і \ можна написати [\-\^\\]. Спеціальні випадки стосуються дужок: щоб відповідати `[` або `]`, закриваюча дужка повинна бути першим символом після відкриваючої, наприклад, `[][ab]` відповідає `]`, `[`, `a` або `b`.

Одинадцяту категорію становлять альтернативи, які забезпечують вибір між варіантами в шаблоні. Альтернатива визначається через метасимвол `|`, тобто `a|b` означає «або a, або b». Альтернатив може бути більше двох (a|b|c|d), але розкривається при цьому для кожного зіставлення шаблону завжди лише один варіант. На вибір альтернатив може впливати жадібність квантифікатора, якщо він використовується. Наприклад, у виразі gray|grey обирається один з варіантів залежно від контексту тексту, що корисно для обробки орфографічних варіацій.

Дванадцяту категорію становлять групи, які дозволяють об’єднувати та повторювати частини шаблону з збереженням результатів. Група `()` дозволяє використовувати повторювані частини шуканої послідовності літералів багаторазово в одному шаблоні в межах фрагмента тексту, що обробляється. Група

представляє окремий шаблон, результат розкриття за яким зберігається в окремій області пам’яті, на яку можна послатися. Групи можуть вкладатися одна в одну. Група завжди розкривається в тій же позиції, в якій вона знаходиться у зовнішньому до неї шаблоні. Група завжди розкривається рівно один раз. На групу може бути застосований квантифікатор, проте він не діє на результат, що зберігається, тобто в пам’яті буде збережено лише перше входження після квантифікації. Посилання на групу оформляються як \n, де n – цифра, починаючи з одиниці. Від процесора регулярних виразів потрібна підтримка щонайменше 9 груп. Для вкладених груп окремий стек не створюється і нумерація продовжується за наскрізним принципом у глибину. Наприклад, якщо записати так (( )), то зовнішня група дужок утворює групу \1, а внутрішня – \2. Цей механізм дозволяє обробляти такі повторювані структури, як дати чи email-адреси.

2.2 Демоверсія програми-агента для виявлення за регулярними виразами конфіденційної інформації в текстових файлах

Як зазначалося вище, одним із недоліків DLP-систем є передача контролю за конфіденційною інформацією зовнішнім організаціям. Це може призвести до того, що ваші дані більше не належать виключно вам. Навіть якщо постачальник послуг використовує технології з відкритим вихідним кодом, по суті, інформація стає частково доступною цій компанії, що може викликати питання щодо збереження та конфіденційності. В умовах зростання кіберзагроз така ситуація навряд чи може вважатися безпечною, особливо якщо врахувати, що навіть найнадійніші компанії схильні до атак або випадкових витоків інформації. У зв’язку з цим, актуальним стає питання щодо можливості розробки власних систем контролю та моніторингу конфіденційних даних.

В даній роботи пропонується приклад агентського рішення, коли на кінцевий пристрій користувачів встановлюється спеціальна програма-монітор (агент), яка здійснює рекурсивний огляд файлової системи та контентно аналізує наявність файлів з певним вмістом. Програма працює виключно з текстовими файлами, проте, її функціонал може бути розширено за рахунок додавання

додаткових класів та бібліотек, щоб включати підтримку PDF, Word та інших типів файлів. Існують API та бібліотеки для C++, такі як Aspose, які дозволяють відкривати, читати та редагувати документи формату PDF та Word, що уможливлює впровадження аналогічного функціоналу у програму. Використовуючи такі SDK, можна легко інтегрувати роботу з різними форматами документів, досягнувши більш широкого спектру застосування програми. Шаблони пошуку є вихідними даними і мають бути підготовлені за певними правилами. При цьому, звичайний набір конкретних слів труднощів не викликає, що не можна сказати про «неявні» набори лексем у документі, наприклад, щоб відрізнити гриф від фрази у тексті. Для вирішення таких питань програма детектує конкретні об'єкти в комп'ютерному середовищі за допомогою регулярних виразів. Для цього використовуються бібліотеки C++ <regex> та <filesystem>.

Початковими даними для програми є регулярні вирази, які описують конфіденційні текстові файли. В якості результату переліковуються відповідні конфіденційні файли, які містятся у заданому каталозі, чи у його підкаталогах.

Клас RegexSearcher використовується для пошуку рядків у текстових файлах з використанням шаблонів регулярних виразів. Він володіє різними полями та методами, які забезпечують створення та використання регулярних виразів. Усередині класу визначається статичний масив рядків s_regWords, що містить шаблони, які є основою для побудови регулярних виразів. Ці шаблони допомагають ідентифікувати текстові фрагменти, що становлять інтерес, наприклад, тему, дату, номер документа, тощо. Клас також містить покажчики p_begin, p_end і p_filler, які є компонентами шаблону регулярного виразу – початок, кінець і роздільник відповідно. Ці покажчики дозволяють динамічно формувати вираз з огляду на потреби конкретного пошуку. Підсумковий регулярний вираз зберігається в змінній p_regExpression, а його розмір – в змінній p_exprSize. Ці параметри дозволяють забезпечувати контроль над обробкою та зберіганням регулярних виразів, що також полегшує їх подальшу обробку.

Методи, реалізовані у класі, забезпечують його основну функціональність. Метод createRegex створює регулярний вираз, використовуючи список ключових

слів. Алгоритм методу передбачає динамічне виділення пам’яті для зберігання підсумкового виразу, а також використовує внутрішній допоміжний метод copyString, який відповідає за коректне копіювання текстових даних та їх приєднання до виразу, що формується. Процес створення шаблону може бути розпочато з будь-якого обраного початку, при цьому метод повертає структуру, що містить підсумковий вираз та його розмір.

Метод linesContaining використовується для пошуку рядків, які відповідають заданому регулярному виразу у вказаному текстовому файлі. Цей метод читає файл рядок за рядком, випробовуючи регулярний вираз для всіх рядків файлу. Якщо рядок відповідає виразу, він зберігається у векторі, що є результатом виконання пошуку. Знайдені рядки можуть бути оброблені окремо та виведені на екран.

Метод findFiles виконує пошук файлів у зазначеному каталозі, використовуючи заданий регулярний вираз для фільтрації. Він ініціалізує рекурсивний ітератор для обходу файлової системи та повертає список файлів, які відповідають заздалегідь встановленому патерну. Завдяки цьому методу можна динамічно знаходити та обробляти лише ті файли, які відповідають встановленим вимогам, що економить ресурси та час не працюючи з непотрібними файлами.

Метод init запускає загальний процес пошуку, комбінуючи функціонал усіх попередніх методів. Він створює регулярний вираз на основі списку ключових слів і виконує пошук текстових файлів у поточній робочій директорії. Потім він обробляє кожен знайдений файл, виводячи на консоль інформацію про знайдені рядки, які відповідають виразу. Якщо хоча б один вираз ідентифіковано успішно, метод завершує роботу поверненням цього результату, інакше він повертає неуспішний результат. Таким чином, клас RegexSearcher інкапсулює всю логіку програми та дозволяє ефективно працювати з регулярними виразами для пошуку інформації у текстових файлах.

2.3 Гіпотетичний приклад конфіденційного файлу та відповідного регулярного виразу для його ідентифікації

Розглянемо приклад конфіденційного текстового файлу, на наявність якого у комп'ютерному середовищі має реагувати наведена вище демонстраційна версія програми-агента. Узагальнену структуру такого документа наведено нижче:

ГРИФ_ТАЄМНОСТІ

екз.000

НАЗВА_УСТАНОВИ

ПІДПРИЄМСТВО_АБО_УСТАНОВА

Назва_Документу

м. Місто

00 00 0000 р. № 00000_____

Про Тема_Мета_Документа

Текст текст

ПОСАДА ІМ'Я_ПРІЗВИЩЕ

Рисунок 2.1 – Узагальнена структура гіпотетичного конфіденційного документа з позначеними літералами (жовтим) та метасимволами (зеленим).

Цей документ містить виділені жовтим кольором металітерали, які замінюють конкретні символи або рядки, що мають точно збігатися з шуканими

даними, та зеленим кольором метасимволи, які замінюють символи поділу між словами (пробіли), кінця рядків (абзаци), а також позначають місця, де можуть бути змінні значення, такі як дати, номери документів, текст або ПІБ.

Ідентифікація наведеного документу здійснюється за допомогою регулярних виразів, які поєднують у собі сталі текстові фрагменти та узагальнені шаблони для змінних значень. Для наочності, наведемо у скороченому вигляді використані метасимволи:

Таблиця 2.1 – Пояснення метасимволів регулярних виразів, використаних для ідентифікації документа.

Метасимвол	Опис
^	Початок рядка
$	Кінець рядка
\.	Екранований символ точки, який відповідає символу
.*	Відповідає будь-якій кількості будь-яких символів або їх відсутності
\d	Будь-яка цифра (0–9)
{n}	Вказує, що попередній символ має повторюватися рівно n разів
{m,n}	Повторення від m до n разів
\s	Відповідає будь-якому символу редагування (пробіл, табуляція тощо)
\w	Відповідає будь-якому алфавітно-цифровому символу (літери та цифри)
|	Логічний оператор «або», розділяє альтернативні варіанти

Наведений нижче приклад регулярного виразу для ідентифікації документа ґрунтується на принципі заміщення умовних даних шаблонами. Це забезпечує гнучкість ідентифікації при збереженні ключових структурних елементів:

– умовний заповнювач «ГРИФ_Таємності» замінений на логічний оператор альтернативи (|), що охоплює найбільш імовірні грифи (наприклад, Таємно, Цілком таємно, ДСК);

– замість «м. Місто» прописано набір конкретних альтернативних назв міст (наприклад, Київ, Одеса, Харків або Львів);

– об’ємний вміст документа («Текст текст текст...») та пробіли між елементами замінені на узагальнений шаблон, що відповідає будь-якій

послідовності символів або їх відсутності (.*). Це дозволяє ігнорувати варіативний текст між основними структурними елементами;

– значення дати, номера документа та інші числові поля замінені на відповідні символьні класи (\d, {n}), тоді як сталі структурні елементи (НАЗВА_УСТАНОВИ, Назва_Документу, тощо) включені як літерали для забезпечення точності ідентифікації.

Приклад регулярного виразу для ідентифікації документа: .*(Таємно|Цілком_таємно|ДСК).*екз\.\d+.*НАЗВА_УСТАНОВИ.*ПІДПРИЄМСТВО_або_УСТАНОВА.*Назва_Документу.*м\. (Київ|Одеса|Харків|Львів).* \d{2} \d{2} \d{4} р\..*№ \d{1,5}.*Про .*Документа.*текст.*[А-ЯІЇЄ][а-яіїє’]+([А-ЯІЇЄ][а-яіїє’]+){0,2} [А-ЯІЇЄ]\. [А-ЯІЇЄ]\.

Наступна таблиця описує структуру наведеного регулярного виразу, пояснюючи функціональне призначення та логіку застосування кожного його компонента відповідно до структурних елементів гіпотетичного конфіденційного файлу.

Таблиця 2.2 – Декомпозиція регулярного виразу для ідентифікації структурних елементів конфіденційного документа.

№	Компонент регулярного виразу	Опис
1	.*(Таємно|Цілком_таємно|ДСК).*	Пошук грифа секретності. Підтримуються варіанти: «Таємно», «Цілком_таємно», «ДСК». .* дозволяє наявність будь-яких символів до і після.
2	екз\.\d+	Пошук позначення кількості екземплярів документа: «екз.» + число.
3	.*НАЗВА_УСТАНОВИ.*ПІДПРИЄМСТВО_або_УСТАНОВА.*	Пошук назви установи та організаційної форми. .* допускає будь-який текст між цими фразами.
4	.*Назва_Документу.*	Обов’язковий заголовок документа.
5	.*м\. (Київ|Одеса|Харків|Львів).*	Перед назвою має бути обов’язкова частина «м.». У прикладі використано кілька найбільш поширених назв.
6	\d{2} \d{2} \d{4}.*р\.	Пошук дати у форматі «день місяць рік», наприклад: 02 05 2025, з подальшим «р.».
7	.*№ \d{1,5}.*	Пошук реєстраційного номера документа від 1 до 5 цифр.
8	.*Про_ТЕМА_МЕТА_ДОКУМЕНТА.*	Пошук теми або мети документа (заголовок або вступ).

9	.*текст.*	Основна частина документа (тіло документа). умовно позначено словом «текст»)
10	[А-ЯІЇЄ][а-яіїє’]+([А-ЯІЇЄ][а-яіїє’]+){0,2} [А-ЯІЇЄ]\. [А-ЯІЇЄ]\.	ПІБ підписанта: посада (одне або кілька слів) + прізвище та ініціали (наприклад: Директор Сидоренко І. П.).

Шаблон регулярного виразу розроблено з урахуванням можливості вибору різних файлів і параметрів, таких як гриф таємності та дати, що дозволяє йому працювати з максимально різноманітними форматами документів та їх можливими варіаціями.

Регулярний вираз охоплює як типові сталі елементи документа (гриф секретності, службові слова, позначення «екз.», «№», «р.»), так і змінні дані (дата, номер, місто, прізвище підписанта). Це дозволяє програмі-агенту виявляти документи навіть у випадках, коли вони відрізняються за конкретними значеннями, але зберігають структуру. Код демонстраційної програми-монітора наведено в Додатку А.

2.4 Результати тестування демонстраційної програми-агента

Для перевірки працездатності розробленої програми-агента було створено набір тестових текстових файлів, які імітують різні типи документів. Тестування проводилося на десяти файлах (test0.txt, test1.txt, test2.txt, test3.txt, test4.txt, test5.txt, test6.txt, test7.txt, test8.txt, test9.txt), які мали різний текстовий вміст. Частина з них відповідала шаблону регулярного виразу, описаному в попередньому підрозділі, частина – ні, що дозволило продемонструвати селективність алгоритму пошуку.

Вихідні дані програми включають:

– каталог з тестовими текстовими файлами;

– регулярний вираз, закодований у статичному масиві s_regWords[] всередині класу RegexSearcher;

– виконуваний файл програми ConsoleApplication1.exe.

Структуру каталогів та файлів програми наведено на рисунку 2.2.

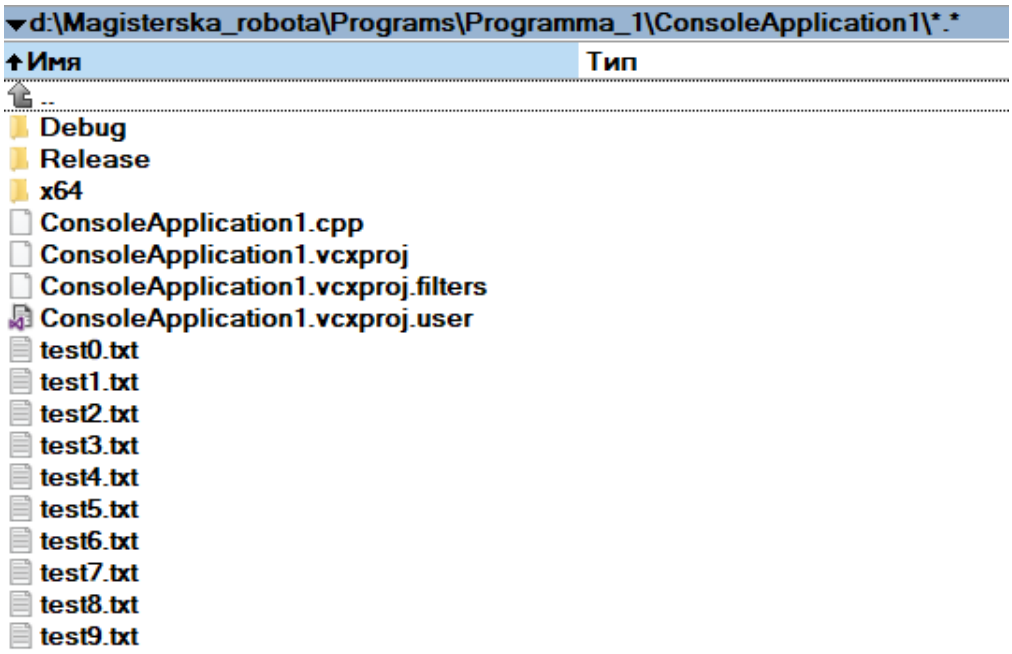


Рисунок 2.2 – Дерево каталогів та файлів програми-агента для ідентифікації конфіденційних документів.

Програма виконує рекурсивний пошук усіх текстових файлів у поточному каталозі та його підкаталогах, застосовуючи до кожного з них регулярний вираз. У разі збігу структури файлу з шаблоном програма виводить повідомлення «ЗНАЙДЕНО!», в іншому випадку – «ні». Підсумковий результат виконання програми відображається у консольному вікні (рисунок 2.3).

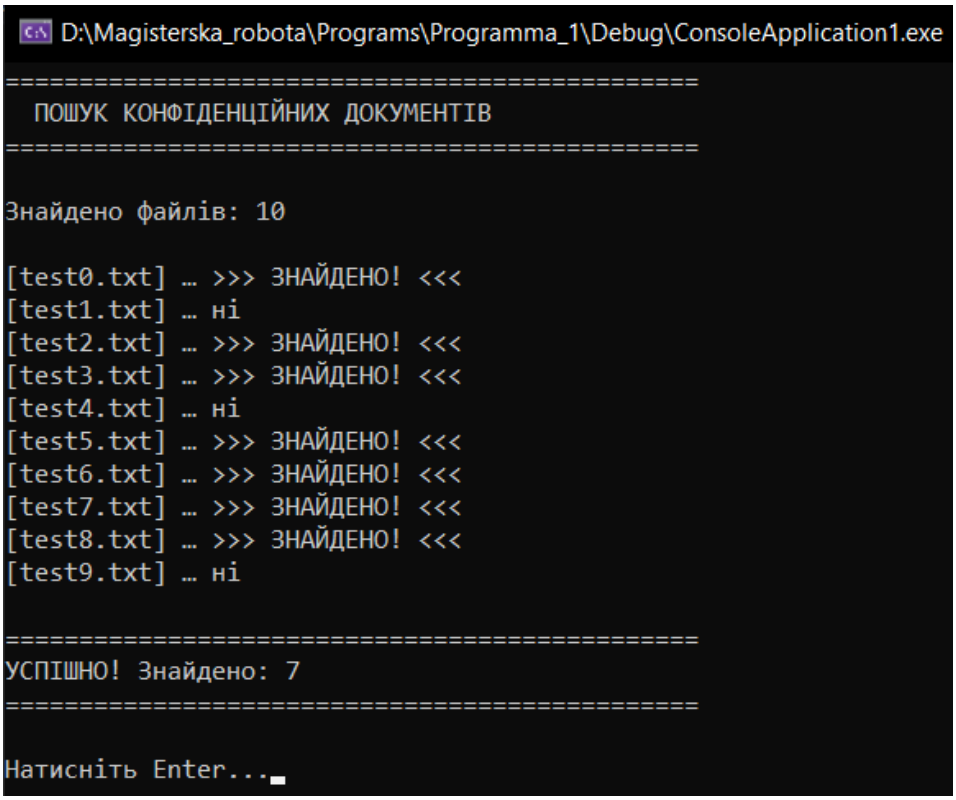


Рисунок 2.3 – Результат виконання програми-агента з відображенням знайдених конфіденційних файлів.

Програма успішно ідентифікувала файли, що відповідають заданому шаблону, та проігнорувала ті, які не містять структурних ознак конфіденційних документів. Це підтверджує ефективність застосування регулярних виразів для автоматизованого виявлення конфіденційної інформації в текстових файлах.

Для демонстрації різноманітності структур, які можуть зустрічатися на практиці, вміст текстових файлів, використаних для тестування програми, наведено на рисунку 2.4.

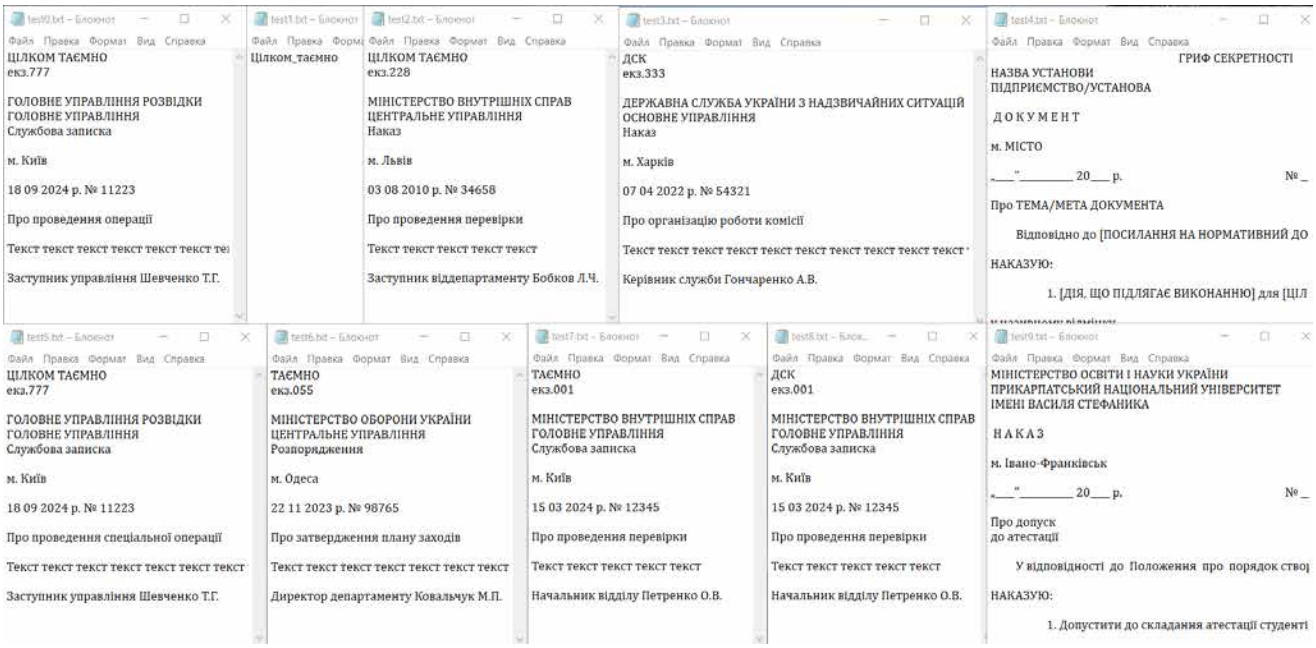


Рисунок 2.4 – Приклади вмісту тестових файлів.

Незважаючи на демонстраційний характер, наведена програма має певний потенціал для подальшого розвитку. Зокрема, можливе розширення функціоналу шляхом додавання підтримки інших форматів файлів (PDF, Word, Excel) за допомогою спеціалізованих бібліотек, таких як Aspose або Apache POI. Крім того, модульна архітектура програми дозволяє легко змінювати або доповнювати регулярні вирази для адаптації до нових типів конфіденційних документів без необхідності перебудови всього коду. Це робить програму гнучким інструментом для організацій, які прагнуть контролювати обіг чутливої інформації в межах власної інфраструктури.

3 ІДЕНТИФІКАЦІЯ КРИПТОГРАФІЧНО ЗАХИЩЕНИХ ФАЙЛІВ, ЯКІ ПОТЕНЦІЙНО МОЖУТЬ МІСТИТИ КОНФІДЕНЦІЙНІ ДАНІ

3.1 Труднощі розпізнавання результатів криптографічних перетворень

В попередній частині роботи запропоновано підхід, який, на відміну від існуючих «промислових» DLP, дозволяє контролювати конфіденційну інформацію виключно її власниками. Разом з тим, відомі DLP мають ще один значний недолік – вони не ідентифікують криптографічно, (взагалі кажучи, і стеганографічно) захищену інформацію. Вочевидь, неконфіденційну інформацію зазвичай не шифрують, тобто, якщо вона зашифрована, то вона конфіденційна, як то кажуть, «за визначенням» [14, 30]. Треба відмітити, що певні формати документів вже мають власний вбудований криптографічний захист. Наприклад, складові документи Word та Excel версій більш ніж Microsoft Office 97 містять потоки «\x01CompObj», «\x05DocumentSummaryInformation», «Workbook», «\x05SummaryInformation», «WordDocument», «1Table», «Data». При парольному захисті документів до них додається поток «encryption», а ознаки використання парольного захисту містяться в потоках «WordDocument» (для Word) та «Workbook» (для Excel). Аналогічно, для PDF-файлів ознакою використання парольного захисту є наявність послідовності байт вигляду «/Encrypt "деяке десяткове число"». Отже, щоб ідентифікувати зашифровані конфіденційні документи, доводиться аналізувати не стільки їх зміст, скільки структуру їх оформлення. З огляду на те, що раніше була порівняно невелика кількість стандартів оформлення шифрованих файлів, а мови опису форматів даних були недостатньо розвинуті, формалізований опис характерних структурованих особливостей аналітично захищених файлів не вважався доцільним. Нині ситуація змінилася. На ринку доступна велика кількість програмних засобів криптографічного і стеганографічного захисту, що реалізують широкий спектр сучасних алгоритмів шифрування. Крім створення спеціалізованих програмних виробів, активно поширюються «універсальні криптографічні інтерфейси» (наприклад, інтерфейси GSS-API, IDUP-GSS-API, GCS-API, CryptoAPI,

CRYPTOKI), за допомогою яких користувачі розробляють власні засоби шифрування.

3.2 Аналіз засобів криптографічного захисту конфіденційної інформації з метою отримання ознак результатів їх роботи

Для ототожнення певних шифрованих файлів з породжуючими їх засобами криптозахисту при написанні даної роботи було протестовано декілька популярних програмних засобів шифрування [22] файлів на предмет встановлення характерних ознак їх вихідних даних. Для цього була написана технологічна програма, що здійснює пошук структурних повторень. Результати її роботи продемонструємо на прикладі:

2_L=188, K=42, N1=0, N2=0, F1=1.txt, F2=2.txt

0D@ 0D@ 0A@ 3C< 3C< 53S 54T 41A 52R 54T 5F_ 50P 43C 5F_ 45E 6En 63c 72r 79y 70p 74t 5F_ 44D 41A 54T 41A 3E> 3E> 0D@ 0D@ 0A@ [2] 41A 41A 41A [3] 41A 41A [66] 0D@ 0D@ 0A@ [76] 0D@ 0D@ 0A@

Це означає, що між файлами 1.txt і 2.txt з 0-го місця виявлено ділянку завдовжки 188 символів, з яких повторилося на відповідних позиціях 42 символи (її наведено в явному вигляді, де зліва від кожного повторюваного символу розміщене його шістнадцятиричне представлення, а в квадратних дужках зазначено довжини послідовностей, які не збіглися).

У Додатку Б наведені отримані таким чином відповідні відомості щодо протестованих засобів програмного закриття інформації, які свідчать про наявність певної структурованості.

Зрозуміло, що виявити факт використання криптозасобів, безсистемно аналізуючи результати їх роботи, швидше за все неможливо. Але перспективи системного підходу грунтуються на тому, що:

– факти використання певних засобів шифрування, відповідно до принципу Кіркхофа (правило, згідно з яким стійкість криптографічного алгоритму не має залежати від архітектури алгоритму, а має залежати тільки від ключів, тобто,

вважається, що про систему шифрування відомо все, окрім ключів), не зберігатимуться в секреті особливо ретельно [13, 14];

– засоби шифрування для координації та передачі даних дотримуються певних протокольних угод, а ці умовності утворюють моделі, які можна розпізнати та проаналізувати.

Головна причина існування таких моделей – різноманітність телекомунікаційних протоколів та програмних засобів шифрування, які використовують криптографічні інтерфейси, що призводить до уніфікації процедур маніпулювання структурними даними у гетерогенних комп'ютерних мережах.

3.3 Використання технології S-виразів для опису результатів криптографічних перетворень

Існуючі методи опису криптофайлів базуються на емпіричних спостереженнях, які характеризуються високим рівнем варіативності та інтерпретаційної невизначеності. Такі підходи не забезпечують систематизації, не володіють достатньою повнотою даних і, як наслідок, не можуть служити основою для розроблення автоматизованих алгоритмів розпізнавання та класифікації захищеної інформації. Проте, методи формального опису структурних особливостей шифрованих файлів можуть забезпечити не тільки ідентифікацію конкретного засобу закриття інформації [25], але й стимулювати розвиток формалізованого підходу до розпізнавання конкретної математичної моделі, що лежить в основі відповідного криптоперетворення. Це передбачає певну методику тестування криптографічних засобів та необхідну дисципліну роботи з результатами їх виконання.

В даній роботі розглянуті:

– мови опису моделей та структура відповідного сховища таких описів;

– технологія інтерактивного виготовлення описів;

– рекомендації щодо розробки програмних засобів для ідентифікації криптофайлів за допомогою інтерпретації описів результатів їх роботи.

3.4 Мова опису моделей та структура відповідного сховища описів

Сучасні мови опису форматів даних орієнтовані на спеціальні формати зовнішнього подання даних (ASN.1/BER/DER, HDF, NDF) та передбачають наявність спеціальних компіляторів для створення за описами даних статичних процедур перетворення на зовнішнє/внутрішнє оформлення (XDR/RPCGEN, IDL/MIDL). Вони мають певні недоліки, коли йдеться про опис і динамічний розбір довільних структурованих даних. Існує також підхід, коли опис даних вже сам по собі є своєрідною програмою для інтерпретації форматів цих даних [28]. Важливим елементом синтаксису такої мови є можливість зручного відображення ієрархічності (структурності) даних. Отже, на даний момент існує можливість практичного впровадження технології уніфікації описів форматів шифрованих файлів [29] і автоматизації процесів ідентифікації відомих засобів шифрування. Задля цього в роботі пропонується використовувати так звані S-вирази [1,12]. S-expressions – структури, що описують складні дані, подібно S-виразам мови програмування LISP [1, 12, 20], яка була запропонована Джоном Маккарті. Деталі синтаксису або підтримуваних типів даних можуть відрізнятися для різних представників сімейства Lisp-мов, однак спільною рисою є представлення S-виразів у дужковій префіксній формі запису. Неформально кажучи, S-вираз - це або атомарне значення (число, рядок, символ тощо), або список S-виразів [1, 12, 20]. S-вирази в сімействі Lisp-мов можуть представляти не тільки дані, а й записаний в префіксній нотації відповідний код. У контексті XML, за допомогою S-виразів можна також описувати і XML-документи, перетворюючи їх у деревоподібну структуру [12], де кожен вузол дерева відповідає S-виразу. Навіть існує так званий SXML – спосіб представлення XML-документів у вигляді S-виразів. У плані можливостей щодо мови опису XML та S-вирази рівносильні і підтримують кодування base-64, що дозволяє працювати з двійковими даними, але є і відмінності:

Таблиця 3.1 – Порівняльна характеристика S-виразів та XML, як мов опису структурованих даних.

	Переваги	Недоліки
S-вирази	- наочність; - проста реалізація; - компактність; - швидкодія.	- Невелика програмна підтримка.
XML	- є де-факто стандартом; - промисловий масштаб підтримки.	- етап становлення ще продовжується.

Обидва варіанти підтримують:

– наявність базових (байт, слово, рядок, число…) та контейнерних типів (вектор, список);

– можливість конструювання похідних типів;

– дані з динамічним розміром або типом (тобто коли розмір або тип даних, що розбираються, визначаються або обчислюються за вмістом інших полів даних;

– логічні умови на дані (рівність значенню, шаблону…);

– регулярні вирази.

У термінах S-виразів існує багато різних способів представлення 8-бітових даних, наприклад: abc (лексема), «abc» (рядок) [13], #616263# (шістнадцяткове представлення), 3:abc (представлення з покажчиком довжини), {MzphYmM=} (у кодуванні base-64 з покажчиком довжини), |YWJjj| (у кодуванні base-64). Усі ці приклади описують одні й ті самі дані. Окрім бінарного, усі представлення використовують символи US-ASCII: літери (A-Z, a-z), числа (0-9), розділювачі (пробіл, горизонтальний табулятор, вертикальний табулятор, переведення форми, повернення каретки, переведення рядка), а також псевдографіку (- hyphen or minus, . period, / slash, _ underscore, : colon, * asterisk, + plus, = equal) та символи редагування ((left parenthesis,) right parenthesis, [left bracket,] right bracket, { left brace, } right brace, | vertical bar, # number sign, " double quote, & ampersand, \ backslash). Крім бінарного, у всіх представленнях заборонені такі символи: !

(exclamation point), % (percent), ^ (circumflex), ~ (tilde), ; (semicolon), ‘ (apostrophe), , (comma), < (less than), > (greater than), ? (question mark).

Бінарне представлення байтових рядків складається з таких компонентів: довжина рядка (кількість байтів у десятковій системі без лідируючих нулів), роздільник (двокрапка) та дані (безпосереднє значення рядка). Приклади:

– 3:abc;
– 7:subject;
– 4::::: ;
– 12:hello world!;
– 10:abcdefghij;
– 0:.

Quoted-string представлення містить такі складові: довжина рядка у десятковій системі, відкриваюча лапка («), рядок із урахуванням спеціальних символів редагування та закривна лапка (»).

Символи редагування включають:

– \b – backspace;
– \t – horizontal tab;
– \v – vertical tab;
– \n – new-line;
– \f – form-feed;
– \r – carriage return;
– \" – double quote;
– \' – single quote;
– \\ – backslash;
– \ooo – восьмиричний запис (усі три цифри мають бути присутніми);
– \xhh – шістнадцятковий запис (обидві цифри мають бути присутніми) ;
– \<carriage-return> – ігнорує carriage return;
– \<line-feed> – ігнорує line feed;
– \<carriage-return><line-feed> – ігнорує CRLF;

– \<line-feed><carriage-return> – ігнорує LFCR.

Приклади:

– «subject»;

– «hi there»;

– 7«subject»;

– 3«\n\n\n»;

– «This has\n two lines.»;

– ««».

Літеральне представлення рядків дотримується таких правил: рядок не може починатися з цифри; допустимі символи – літери, цифри, а також - . / _ : * + =. Приклади:

– subject;

– not-before;

– class-of-1997;

– //microsoft.com/names/smith;

– *.

Шістнадцяткове представлення рядків містить такі компоненти: довжина рядка у десятковій системі (необов'язкова), символ # на початку, шістнадцяткове представлення кожного байта рядка та символ # у кінці. Приклади:

– #616263# – представляє рядок «abc»;

– 3#616263# – також представляє «abc».

Base-64 представлення (використовує 4 символи замість кожних 3) містить: довжину рядка у десятковій системі (необов'язкова), символ `|` на початку, кодування рядка відповідно до RFC 1521 та символ `|` у кінці. Приклади:

– |YWJjj| – представляє «abc»;

– 3|YWJJ| – також представляє «abc»;

– |YWJjjZA==| – представляє «abcd»;

– |YWJjjZA| – також представляє «abcd».

Для зручності візуалізації формат даних може бути зазначений у квадратних дужках, наприклад:

– [image/gif];
– [URI];
– [charset=unicode-1-1];
– [text/richtext];
– [application/postscript];
– [audio/basic];
– [«http://abc.com/display-types/funky.html»].

За замовчуванням використовується MIME-формат: text/plain; charset=iso-8859-1.

S-вирази, як концепція алгоритмічної мови LISP, можуть бути елементами списку. Список має бути укладений у дужки, а його елементи розділені відповідними символами. Приклади S-виразів:

– (a b c);
– (a (b c) (((d e) (e f))));
– (11:certificate(6:issuer3:bob)(7:subject5:alice));
– ({3Rt=} «1997» murphy 3:{XC++});
– (abc (de #6667#) «ghi jkl»).

Варіанти кодування S-виразів класифікуються за категоріями:

- transport (для передачі S-виразів між комп’ютерами);
- canonical (для стандартного запису S-виразів);
- advanced (для покращеної читабельності);
- in-memory (для зберігання у пам’яті комп’ютера).

Канонічна форма використовується для представлення цифрових та інших сигнатур [1]. Вона однозначно визначає кожний S-вираз і представляє рядок у бінарному вигляді, а кожен список – без надлишкових пробілів та інших роздільників. Приклади:

– (6:issuer3:bob);

– (4:icon[12:image/bitmap]9:xxxxxxxxxxx);

– (7:subject(3:ref5:alice6:mother)).

Базова транспортна форма використовується як універсальний механізм передачі S-виразів між комп’ютерами. Приклади:

– (1:a1:b1:c);

– {KDE6YTE6YjE6YykA} (те саме у форматі Base-64).

Синтаксис канонічної та просунутої транспортних форм запишемо за допомогою бекусових нормальних форм (BNF), використовуючи такі позначення:

– * – 0 або більше повторень;

– + – 1 або більше повторень;

– ? – 0 або 1 повторення.

Канонічна транспортна форма:

```
<sexpr> ::= <string> | <list>
<string> ::= <display>? <simple-string> ;
<simple-string> ::= <raw> ;
<display> ::= "[" <simple-string> <simple-string> "]" ;
<raw> ::= <decimal> ":" <bytes> ;
<decimal> ::= <decimal-digit>+ ;
  -- десяткові числа не повинні мати зайвих початкових нулів
<bytes> ::= будь-який рядок байтів вказаної довжини
<list> ::= "(" <sexpr>* ")" ;
<decimal-digit> ::= "0" | ... | "9" ;
```

Просунута транспортна форма:

```
<sexpr> ::= <string> | <list>
<string> ::= <display>? <simple-string> ;
<simple-string> ::= <raw> | <token> | <base-64> | <hexadecimal> |
<quoted-string> ;
<display> ::= “[“ <simple-string> “]” ;
<raw> ::= <decimal> “:” <bytes> ;
<decimal> ::= <decimal-digit>+ ;
  -- десяткові числа не повинні мати зайвих початкових нулів
<bytes> ::= довільний рядок байтів вказаної довжини
```

```
<token> ::= <token-char>+ ;
<base-64> ::= <decimal>? "|" ( <base-64-char> | <whitespace> )* "|" ;
<hexadecimal> ::= "#" ( <hex-digit> | <whitespace> )* "#" ;
<quoted-string> ::= <decimal>? <quoted-string-body> ;
<quoted-string-body> ::= "\"" <bytes> "\"" ;
<list> ::= " (" ( <sexpr> | <whitespace> )* ")" ;
<whitespace> ::= <whitespace-char>* ;
<token-char> ::= <alpha> | <decimal-digit> | <simple-punctuation> ;
<alpha> ::= <uppercase> | <lowercase> | <decimal-digit> ;
<lowercase> ::= "a" | ... | "z" ;
<uppercase> ::= "A" | ... | "Z" ;
<decimal-digit> ::= "0" | ... | "9" ;
<hex-digit> ::= <decimal-digit> | "A" | ... | "F" | "a" | ... | "f" ;
<simple-punctuation> ::= "-" | "." | "/" | "_" | ":" | "*" | "+" |
"=";
<whitespace-char> ::= " " | "\t" | "\r" | "\n" ;
<base-64-char> ::= <alpha> | <decimal-digit> | "+" | "/" | "=" ;
<null> ::= "".
```

Описи результатів роботи засобів криптозахисту містять метадані, тобто дані про дані, не включаючи самі назви файлів та їх вміст. Кожен описувач окрім відповідних S-виразів містить також певну інформацію про конкретний засіб шифрування. Взагалі кажучи, сховище описів може бути окремим файлом, базою даних, тощо. В даній роботі в демонстраційних цілях сховище організовано у вигляді списку.

3.5 Технологія інтерактивного виготовлення описів, демонстраційна версія програми побудови S-виразів за результатами інтерактивного аналізу шифрованих файлів

Аналітичним шляхом створити S-вираз, відповідний нетривіальним структурованим даним, досить складно. Набагато простіше це робити в інтерактивному режимі, з наочними ілюстраціями та автоматичною перевіркою не

лише синтаксису, а й достовірності описів. З цією метою розроблена демонстраційна версія програми, яка:

– аналізує позначені користувачем ділянки даних;

– будує за їхньою сукупністю об’єкт деревоподібного відображення (MFC-клас CTreeCtrl);

– для всіх «гілок» запам’ятовує необхідну інформацію (ім’я, тощо), а для «листя», крім того, дає можливість користувачу задати відповідні регулярні вирази;

– автоматично генерує відповідний S-вираз та перевіряє його достовірність на інших даних.

Наведений далі аналіз чотирьох ZIP-архівів з різним вмістом та параметрами стиснення/шифрування (aaa.zip, bbb.zip, ccc.zip та ddd.zip) демонструє можливості програми щодо ідентифікації ключових характеристик файлу на основі його бітового представлення.

aaa.zip та bbb.zip містять ідентичний текстовий файл. Проте, aaa.zip є зашифрованим і захищеним паролем, тоді як bbb.zip – незашифрований.

ccc.zip не зашифрований, але містить інший вміст.

ddd.zip має унікальний вміст і збережений з максимальним рівнем стиснення, що дозволило значно зменшити його розмір.

Програма візуалізує файлову структуру як дерево, відповідно до стандартного формату ZIP-архівів. Ця структура є ієрархічною і може бути представлена таким чином:

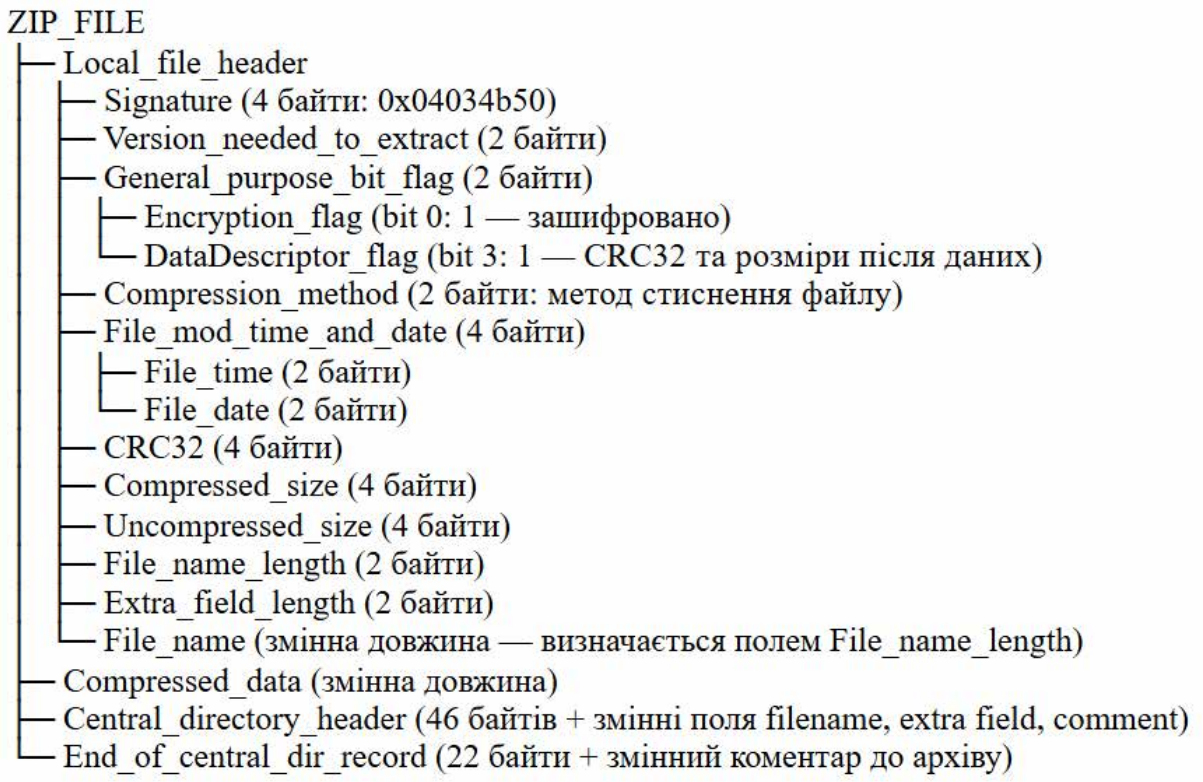


Рисунок 3.1 – Ієрархічна структура ZIP-архіву згідно специфікації PKZIP (байтове представлення для інтерактивного аналізу).

Програма аналізує ключові структурні блоки, що визначають метадані файлів, методи стиснення та шифрування. Ці блоки відповідають специфікації формату ZIP (PKZIP Application Note 6.3.8), де кожен елемент представлено у вигляді послідовності байтів для забезпечення цілісності та доступності даних. Нижче наведено опис основних блоків, що аналізуються, з акцентом на їх роль у виявленні захищеної інформації:

– Local_file_header – заголовок локального файлу, який містить основні метадані про вміст ZIP-архіву та передує стисненим даним. Цей блок включає сигнатуру (стандартну для ZIP-формату: 0x04034b50, або #50 4B 03 04# у шістнадцятковому поданні), версію, необхідну для вилучення (мінімальна версія програми для розпакування), бітовий прапор загального призначення (де біт 0 вказує на шифрування, а біт 3 – на наявність дескриптора даних), метод стиснення (наприклад, 0 для без стиснення, 8 для Deflate), час і дату останньої модифікації файлу, а також ім’я файлу (змінної довжини). Аналіз цього блоку дозволяє виявити

ознаки конфіденційності, такі як флаг шифрування, що блокує прямий доступ до вмісту;

– Data_descriptor – додатковий дескриптор даних, що розміщується після стиснених даних файлу (якщо встановлено відповідний біт у прапорі). Він містить контрольну суму CRC-32 для перевірки цілісності даних, стиснутий розмір файлу та нестиснутий розмір, що допомагає у валідації архіву без розпакування. У контексті безпеки цей блок критичний для ідентифікації пошкоджень або маніпуляцій з конфіденційними файлами, оскільки CRC-32 дозволяє виявити несанкціоновані зміни;

– Central_directory_header (або File_header) – заголовок центрального каталогу, що агрегує інформацію про всі файли в архіві та розміщується ближче до кінця файлу. Цей блок дублює ключові метадані з локальних заголовків (сигнатура 0x02014b50, версія, метод стиснення, CRC-32, розміри файлів, ім’я та додаткові поля), а також включає відносне зміщення до локального заголовка для швидкого доступу. У програмі аналіз центрального каталогу забезпечує повний огляд архіву, дозволяючи виявляти приховані конфіденційні файли без повного сканування;

– End_of_central_dir_record – кінцевий запис центрального каталогу, що завершує структуру архіву та містить загальну інформацію: сигнатуру (0x06054b50), кількість файлів, розмір каталогу, зміщення до його початку та можливий коментар (до 65535 байтів). Цей блок слугує «індексом» для читання архіву, і його аналіз у програмі підтверджує валідність ZIP-структури, виявляючи аномалії, такі як невідповідність розмірів для захищених архівів.

На рисунку 3.2 наведено скріншот з виділенням окремих іменованих байтів з метою детального аналізу кожного з цих блоків.

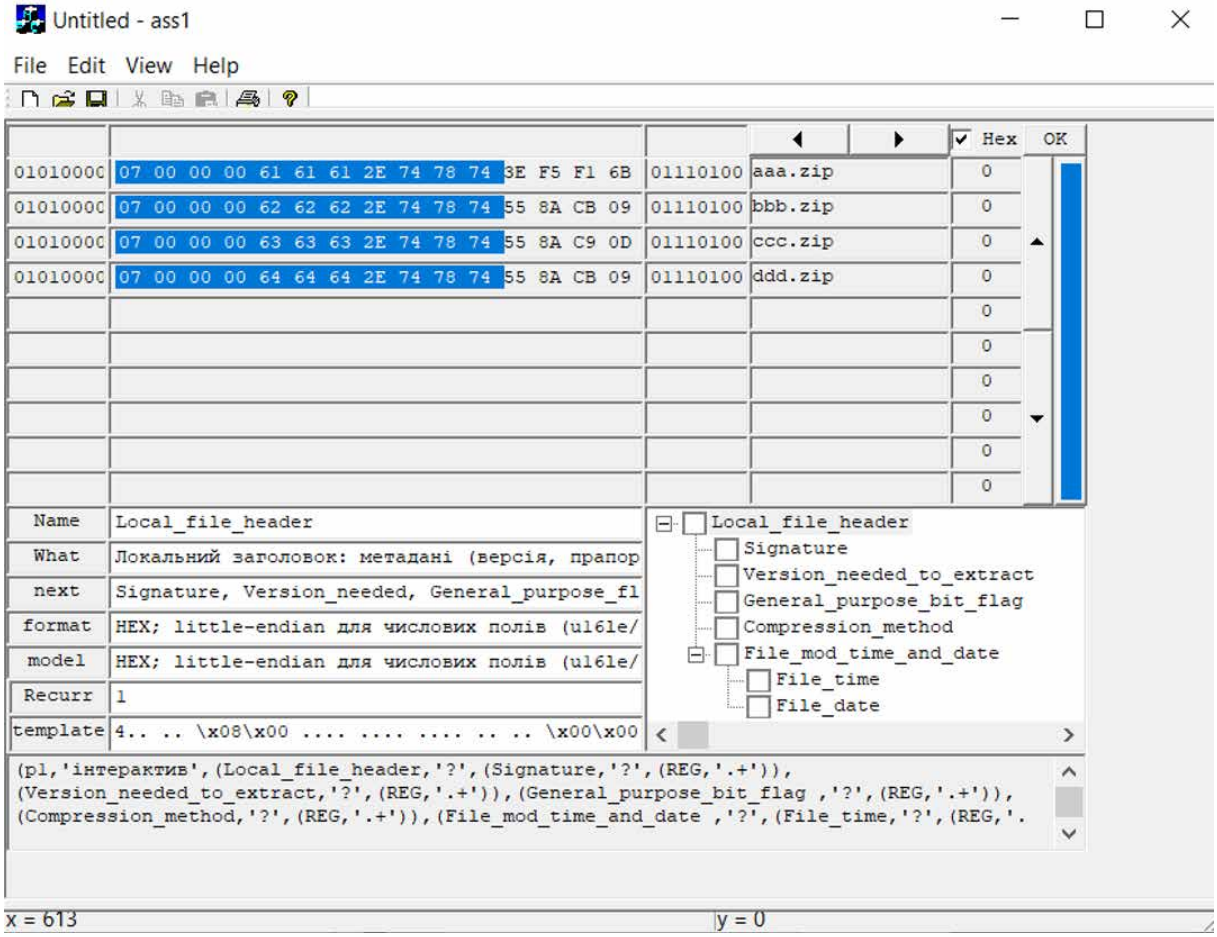


Рисунок 3.2 – Візуалізація структури локального заголовка ZIP-архіву та метаданих у програмі інтерактивного аналізу.

Особливий інтерес становить поле General_purpose_bit_flag, яке займає два байти та аналізується побітово. Зокрема, біт 0 відповідає за шифрування, а біт 3 вказує на використання додаткового дескриптора даних.

Для незашифрованих файлів (наприклад, bbb.zip) це поле має значення 02 00 у шістнадцятковій системі (little-endian: молодший байт 0x02, старший 0x00), що у двійковому вигляді є 0000 0000 0000 0010; тут біт 0 = 0 (немає шифрування), а біт 1 = 1 (зазвичай позначає інші атрибути, як наявність дескриптора).

Для зашифрованого файлу (aaa.zip), значення становить 03 00, що у двійковій формі є 0000 0011. Тут біт 0 має значення 1 (увімкнене шифрування).

Програма дозволяє виділити це поле та створити для нього детальний опис, використовуючи спеціалізовані службові позначки.

General_purpose_bit_flag (2 байти):

– name: General_purpose_bit_flag;

– what: Бітові прапори: біт 0 = шифрування, біт 3 = Data Descriptor;
– інші – сумісність/опції;
– next: Encryption_flag (bit0), DataDescriptor_flag (bit3);
– format: u16le (bitfield);
– model: flags{enc=0x0001, dataDesc=0x0008};
– occurrence: 1;
– template: \x03\x00 (для зашифрованого) \x02\x00 (для незашифрованого).

Після інтерактивного аналізу та опису всіх необхідних полів програма автоматично генерує спеціальний рядок, показаний на рисунку 3.3, у форматі, подібному до S-виразів.

Демонстраційний S-вираз, сформований програмою, в даному випадку виглядає так: (pl,'інтерактив', (Local_file_header,'?',(Signature,'?',(REG,'.+'))), (Version_needed_to_extract,'?',(REG,'.+')), (General_purpose_bit_flag,'?',(REG,'.+')))

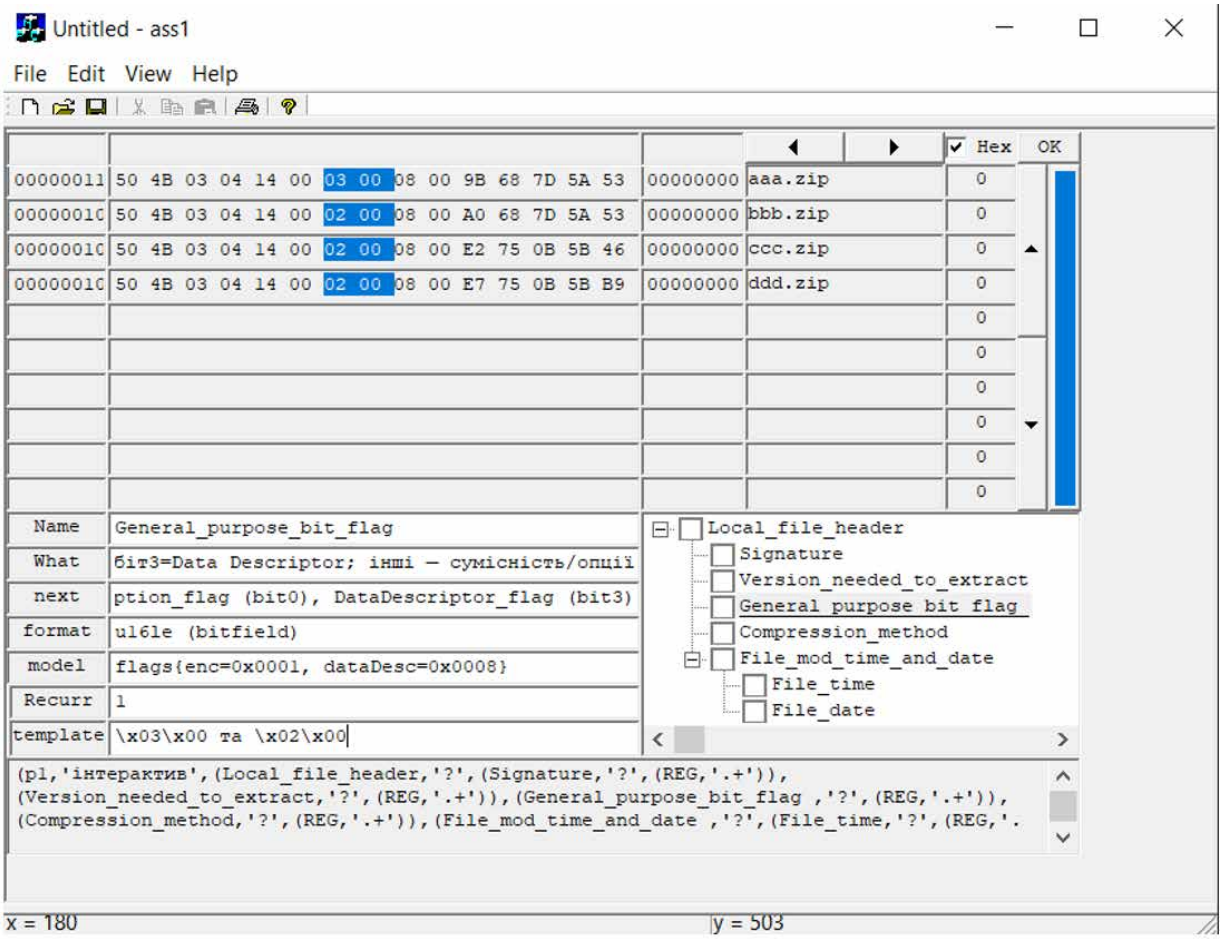


Рисунок 3.3 – Автоматично згенерований S-вираз для структури ZIP-архіву після інтерактивного аналізу.

Це не є класичним «чистим» S-виразом, оскільки він містить додаткові службові позначки ('?', 'REG', 'інтерактив'), що використовуються програмою для внутрішніх потреб: позначення невідомих значень, регулярних виразів та режиму формування структури. Однак логіка вкладеної структури в дужках збережена, що дозволяє стверджувати про створення розширеного варіанта S-виразу, адаптованого для опису бінарних форматів файлів.

У такий спосіб за допомогою інтерактивного аналізу програма не тільки візуалізує структуру криптофайлу, але й уніфікує її опис у форматі S-виразу. Це є важливим кроком для подальшої автоматизованої обробки та аналізу файлових структур, включаючи ідентифікацію шифрованих даних.

3.6 Рекомендації щодо розробки програмних засобів для ідентифікації криптофайлів за допомогою інтерпретації описів результатів їх роботи

Як наведено вище, S-вирази дозволяють моделювати довільні ієрархічні структури, включно з деревами, графами, списками та іншими рекурсивними об’єктами [24]. Вони ідеально підходять для опису внутрішньої структури файлів, що містять вкладені заголовки, службові блоки та секції, як це було продемонстровано на прикладі ZIP-формату. Працювати з ними можна як за допомогою базових LISP-функцій (розчленування S-виразів – селектори, складання S-виразів – конструктори та аналіз S-виразів – предикати), так і спеціалізованими засобами синтаксичного аналізу.

Для навігації та обробки S-виразів існує декілька методів, які різняться за рівнем абстракції та обраною парадигмою програмування.

3.6.1. Базова обробка списків (car, cdr, cons)

У класичному стилі Lisp або Scheme робота з S-виразами зводиться до використання базових функцій, таких як car (голова списку), cdr (хвіст списку) і cons (побудова нового списку). Цей підхід дозволяє будувати рекурсивні алгоритми для обходу, порівняння та трансформації деревоподібних структур [1, 12, 20, 27], а також реалізовувати прості засоби pattern matching «вручну».

Хоча такий метод є найпростішим, він потребує великої кількості коду і не завжди зручний для читання. Він підходить для низькорівневої обробки або вбудованих інтерпретаторів, де важлива максимальна гнучкість програмування.

3.6.2. Відображення S-виразів у модель XML / XPath

Більш високорівневою альтернативою є відображення (mapping) S-виразів у XPath-модель даних [12], що дозволяє використовувати потужні механізми навігації, фільтрації та вибірки частин дерева. Ідея полягає в перетворенні кожного вузла S-виразу на відповідний XML-елемент або атрибут, після чого можна застосовувати XPath-запити для пошуку певних патернів, таких як, наприклад, «знайти всі вузли, де флаг шифрування дорівнює 1».

Цей підхід широко застосовується в системах, де важлива виразність запитів і доступність інструментів валідації структури (наприклад, XSLT, XPath engines). Зокрема, він може бути реалізований у мові Scheme або навіть у Python за допомогою відповідних XML-бібліотек.

Так, у роботі [11] запропоновано формалізований механізм виразів регулярних шаблонів над XML-структурами, що може бути адаптовано і для дерев S-виразів.

3.6.3. Регулярні шаблони для дерев (Tree Pattern Matching)

У мові Scheme також існують експериментальні бібліотеки для деревного pattern matching, які дозволяють задавати шаблони (tree patterns), що описують конкретну форму чи зміст дерева. Прикладом може бути мова trx, наведена в роботі [11], яка дозволяє описувати патерни для рекурсивної перевірки відповідності деревоподібних виразів.

Така система дає змогу, наприклад, задати правило: «знайди дерево, де перший підвузол має значення pkzip і містить підвузол flags, де біт 0 встановлений».

Попри свою потужність, ці підходи поки що не мають широкого поширення, а реалізації trx наразі недоступні або не підтримуються.

3.6.4. Pattern Matching у функціональних мовах (ML, Haskell, Prolog)

Функціональні мови (Haskell, OCaml, ML) та логічні (Prolog) традиційно мають вбудовані засоби pattern matching, які природним чином підходять для роботи з S-виразами [11]. Наприклад, у Haskell можна описати тип SExpr і реалізувати патерн «Cons (Atom «flags») (Cons (BitField 1) Nil)» – тобто шаблон, що відповідає структурі заголовка певного криптофайла.

Однак застосування цих мов потребує додаткового компіляційного середовища та не завжди є зручним у контексті Python-проектів.

3.6.5. Pattern Matching у Scheme (Chicken, Guile, Gauche)

У межах інтерпретаторів мови Scheme також існують реалізації pattern matching, зокрема від Andrew Wright. Вони дозволяють будувати шаблони без явної рекурсії, однак їх застосування обмежене:

- не всі реалізації Scheme підтримують їх однаково;
- вони більше підходять для списків, ніж для повноцінних дерев;
- відсутня стандартизація API.

Тим не менш, цей підхід може бути ефективним у випадку статичного аналізу описів файлів, які мають фіксовану або частково регулярну структуру.

3.7 Приклад типової програмної реалізації виявлення шифрування у ZIP-архівах без використання S-виразів

У додатку В наведено код програми, яка реалізує пошук конкретно заданого типу криптофайлів (для простоти викладу йдеться знов про ZIP-архіви) без використання технології S-виразів. Зверніть увагу, що кожен засіб криптозахисту потребує свій унікальний програмний код такого типу. Програма зчитує заголовки усіх файлів архіву та перевіряє значення поля general purpose bit flag у структурі Local_file_header або File_header, яке складається з 16 бітів, кожен з яких може вказувати на певну властивість, наприклад, увімкнене шифрування, стиснення, тощо [16].

Алгоритм роботи програми включає кроки [16,17]:

1. імпорт стандартної бібліотеки os та zipfile;
2. складання списку усіх ZIP-архівів у вказаній папці (наприклад, d:\Magisterska_robota\Programs\Method_Traditional_Zipfile\test_archives) (рисунок 3.1);
3. Для кожного архіву:

- відкриває його за допомогою zipfile.ZipFile;
- зчитує заголовки кожного внутрішнього файлу;
- перевіряє значення прапорця general purpose bit flag;

– якщо активовано біт 0x1 – файл вважається зашифрованим.

4. Усі зашифровані архіви фіксуються у текстовому файлі d:\Magisterska_robota\Programs\Method_Traditional_Zipfile\results\encrypted_zips.txt . (рисунок 3.5)

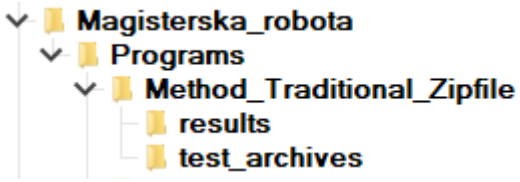


Рисунок 3.4 – Дерево каталогів та файлів методу типової ідентифікації ZIP-архівів.

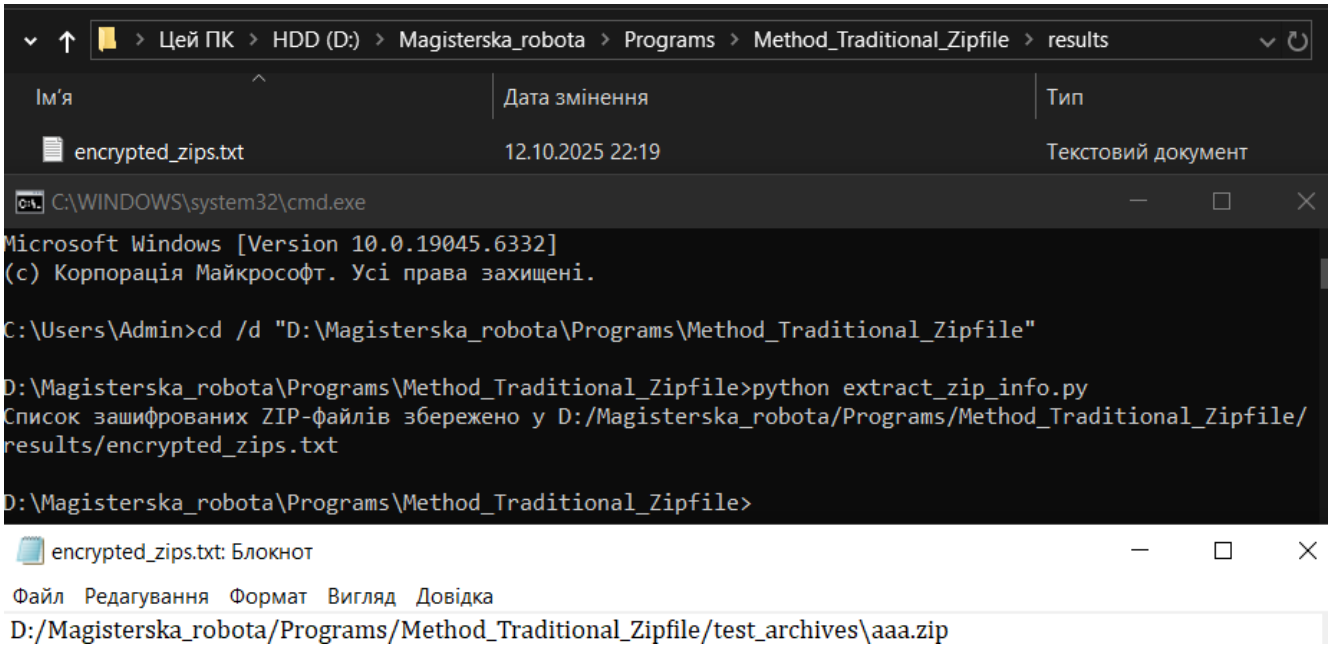


Рисунок 3.5 – Результат виконання програми, а саме, відображення списку зашифрованих ZIP-архівів у файлі encrypted_zips.txt.

В процесі роботи програма орієнтується на наступні значення поля general purpose bit flag у ZIP-архівах з різним рівнем захисту:

– незашифрований файл (наприклад, bbb.zip): поле general purpose bit flag має значення #00 00#, що означає упаковку без шифрування;

– зашифрований файл (наприклад, aaa.zip): поле general purpose bit flag має значення #09 00#, де біт 0x1 вказує на активоване шифрування (PKZIP-Encryption).

Інші можливі значення general purpose bit flag, які вказують на різні варіанти захисту або їх відсутність:

– #01 00# – файл не зашифрований, лише стиснення без додаткових опцій;

– #03 00# – стиснення + шифрування (PKZIP-Encryption);

– #07 00# – стиснення + шифрування + увімкнено додаткові службові біти (наприклад, дата/час або спеціальні розширення);

– #09 08# – альтернативна комбінація, яка може вказувати на використання нестандартного шифрування або додаткового дескриптора даних.

Вихідні дані програми:

– папка d:\Magisterska_robota\Programs\Method_Traditional_Zipfile\test_archives з файлами aaa.zip, bbb.zip…;

– файл d:\...\Method_Traditional_Zipfile\results\encrypted_zips.txt;

– скрипт extract_zip_info.py у папці Method_Traditional_Zipfile.

Також у цій папці наведено скрипт extract_zip_info.py з кодом програми, який запускається за допомогою команди:

```
python extract_zip_info.py
```

Результати роботи заносяться в текстовий файл encrypted_zips.txt, де перелічуються лише зашифровані архіви. У нашому випадку це файл aaa.zip.

3.8 Приклад універсальної програмної ідентифікації та візуалізації шифрованих файлів за множиною відповідних S-виразів

У додатку Г наведено код програми, яка реалізує пошук різних типів криптофайлів (знов таки, на прикладі ZIP-архівів) із візуалізацією їх структури за допомогою наданих S-виразів. На відміну від традиційного підходу, описаного в розділі 3.7, в даному випадку кожен засіб криптозахисту потребує лише відповідного S-виразу, що описує його структурні особливості, без необхідності написання окремого програмного коду. Такі S-вирази також можуть бути корисними і для криптоаналітичних досліджень.

Вихідні дані програми включають:

– каталог криптофайлів (d:\...\Method_Universal_S_Expr\test_archives);

– каталог з текстовими файлами, що містять S-вирази (d:\Magisterska_robota\Programs\Method_Universal_S_Expr\s_expressions).

Структуру каталогів та файлів методу наведено на рисунку 3.6

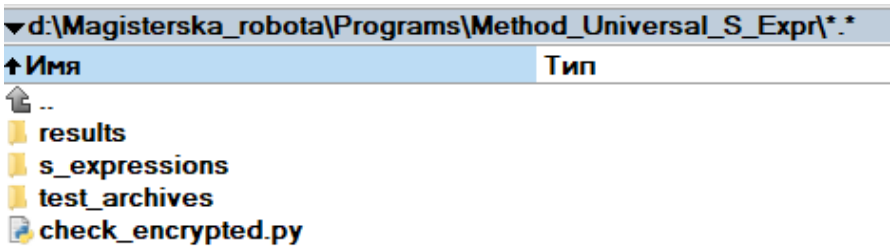


Рисунок 3.6 – Дерево каталогів та файлів методу універсальної програмної ідентифікації з використанням S-виразів.

Програма реалізує декартовий добуток множин криптофайлів та S-виразів, тобто перевіряє кожен криптофайл на відповідність кожному наявному S-виразу. Якщо структура криптофайлу збігається з якимось S-виразом, робиться відповідна відмітка в текстовому файлі encrypted_found.txt із зазначенням імені файлу та виявленого засобу шифрування (рисунок 3.7).

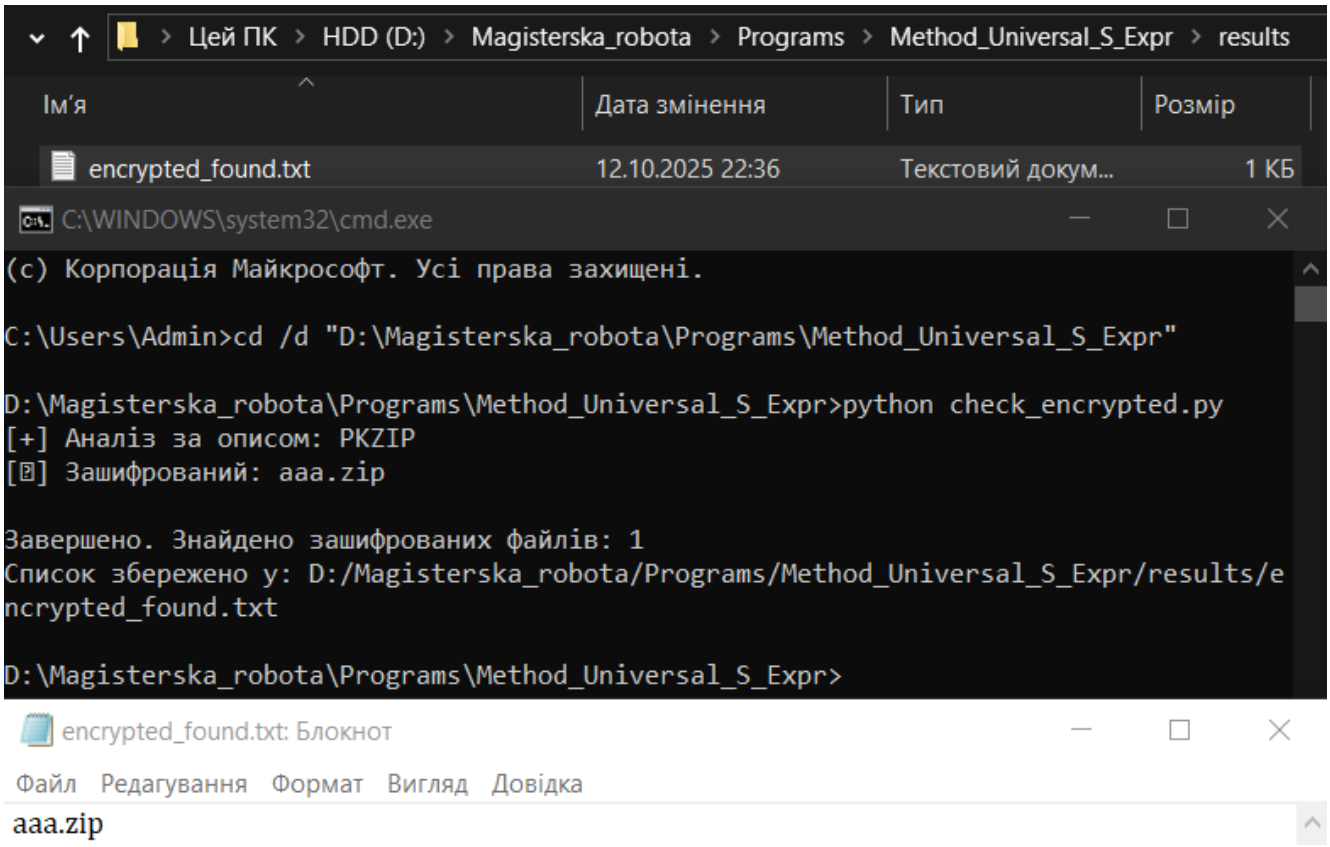


Рисунок 3.7 – Результат процесу ідентифікації – перелік зашифрованих файлів у encrypted_found.txt після аналізу S-виразів.

У додатку Ґ наведений приклад S-виразу, який використовується для виявлення наявності шифрування у ZIP-архівах, створених програмою-архіватором PKZIP. Цей S-вираз описує структуру поля General_purpose_bit_flag та критерії ідентифікації активованого біту шифрування (біт 0x1), що дозволяє автоматично розпізнавати зашифровані архіви без ручного аналізу їх байтової структури.

3.8.1 Алгоритмічна реалізація та переваги підходу

У запропонованій програмі перебираються всі файли із структурними описами у вигляді S-виразів, які є шаблонами для різних типів криптофайлів. Далі аналізується кожен файл у цільовій папці, на предмет збігу з кожним із шаблонів.

Такий підхід дає змогу розпізнавати шифрування навіть у нестандартних чи пошкоджених архівах, оскільки не залежить ні від імені файлу, ні від метаданих.

Програма здійснює аналіз на рівні байтів файлової структури, без використання спеціалізованих бібліотек для обробки конкретних форматів архівів.

Основні переваги запропонованого методу:

– додавання підтримки нових форматів або шифрувальників відбувається шляхом створення нового S-виразу – без змін у коді;

– автоматичний розбір структури через рекурсивний обхід, який нагадує роботу з функціональними структурами даних;

– можливість застосування в якості базового інструменту для подальшої автоматизації й інтеграції в системи моніторингу.

Для використання програми потрібно:

– розмістити криптофайли у папці для аналізу;

– помістити всі S-вирази у окрему папку;

– виконати скрипт check_encrypted.py;

– переглянути результати у файлі encrypted_found.txt, де буде список виявлених шифрфайлів.

ВИСНОВКИ

Магістерська робота присвячена проблемам автоматизації процесів пошуку та ідентифікації конфіденційної інформації в інформаційному середовищі. В умовах зростання інтенсивності кіберзагроз і ускладнення технологій цифрової трансформації суспільства вирішення цього питання стає необхідною умовою безпеки інформаційних структур.

У першому розділі роботи проведено узагальнений аналіз сучасних методів ідентифікації конфіденційних файлів у комп'ютерних системах. Розглянуто специфіку форматів даних, особливості маркування документів, системи грифів та інші явні ознаки наявності конфіденційної інформації. Також наведені відомості про існуючі технологічні засоби контролю чутливих даних, зокрема DLP-систем, та окреслені напрямки їх вдосконалення.

Другий розділ присвячений ідентифікації аналітично незахищених файлів із конфіденційним вмістом. Особлива увага приділялася методам зіставлення зі зразком (pattern matching) за допомогою регулярних виразів. Детально розглянуто принципи побудови шаблонів, модифікаторів, груп символів і квантифікаторів, а також особливості повного переліку діалектів регулярних виразів. У результаті роботи розроблено програмний агент, який здійснює пошук певних файлів у файловій системі, використовуючи гнучку платформу для автоматизованого аналізу вмісту документів.

У третьому розділі розглянуто ідентифікацію криптографічно захищених файлів, які потенційно можуть містити конфіденційну інформацію. Запропоновано застосування методу pattern matching на основі S-виразів. Це дало змогу формально описати внутрішню структуру файлів, будувати шаблони для виявлення певних властивостей і автоматизувати процес пошуку та класифікації зашифрованих даних. Розроблена демонстраційна версія програми дозволяє додавати нові формати без змін у коді, що підтверджує гнучкість та масштабованість підходу. В плані практичної доцільності на прикладі ZIP-архівів продемонстрована також різниця між традиційним трудомістким бітовим аналізом та більш універсальним методом на основі S-виразів.

Отримані результати можуть стати підґрунтям для створення єдиної концепції автоматизованої системи дослідження, пошуку та аналізу конфіденційної інформації у комп’ютерному середовищі. Запропоновані методи дозволяють підвищити якість моніторингу інформаційних потоків, забезпечити гнучку адаптацію під нові типи загроз, формати файлів і механізми захисту. Поєднання регулярних виразів (для роботи з відкритими текстовими структурами) та S-виразів (для структурованого бінарного аналізу) створює передумови для проектування більш універсальних і стійких індустріальних систем безпеки [14, 23, 30].

Перспективи подальшого розвитку проведених досліджень полягають у напрямках:

– впровадження нейромережевих алгоритмів для більш складних методів зіставлення зі зразком;

– інтеграції запропонованих підходів у корпоративні DLP-системи, платформи SIEM та хмарні служби;

– формування та підтримки актуальної універсальної бази регулярних та S-виразів за аналогією з Microsoft Purview;

– розширений аналіз стеганографічних контейнерів;

– організації масштабованих агентських програм для моніторингу великих обсягів інформації у розподілених середовищах.

Це дозволяє розглядати результати роботи не лише як академічний експеримент, а як практичний крок до створення багатофункціональних систем ідентифікації конфіденційних і захищених даних, що відповідають сучасним технологіям і реальним потребам кіберзахисту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rivest R. S-expressions (SEXP). [Електронний ресурс] – Режим доступу: https://datatracker.ietf.org/doc/draft-rivest-sexp/00/ (дата звернення: 14.10.2025).

2. Сміт Б. Методи та алгоритми обчислень у рядках (регулярні вирази). – К.: Вільямс, 2006. – 496 с. – ISBN 0-201-39839-7.

3. Friedl, J.E.F. Mastering Regular Expressions. 3rd ed. O'Reilly Media, 2006. 542 p.ISBN 978-0-596-52812-6.

4. Форта Б. Оволодій самостійно регулярними виразами. PHP, Perl, JavaScript, Java, C#, Visual Basic, ASP.NET, JSP, MySQL, Unix, Linux. – К.: Вільямс, 2004. – 192 с. – ISBN 0-672-32566-7.

5. Гойвертс Я., Левітан С. Регулярні вирази. Збірник рецептів. – Х.: Символ-Плюс, 2010. – 352 с. – ISBN 978-5-93286-181-3.

6. Regular-Expressions.info – Tutorial, Examples and Reference. [Електронний ресурс] – Режим доступу: https://www.regular-expressions.info/ (дата звернення: 14.10.2025).

7. Онлайн тестер регулярних виразів (JavaScript, Python, PHP). [Електронний ресурс] – Режим доступу: https://regex101.com/ (дата звернення: 14.10.2025).

8. Microsoft: Ідентифікація конфіденційної інформації (PSPF). [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/uk-ua/compliance/anz/pspf-identifying-info (дата звернення: 14.10.2025).

9. Microsoft PSPF – Sensitive Information Types. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/uk-ua/compliance/anz/pspf-identifying-info#sensitive-information-types (дата звернення: 14.10.2025).

10. Bagrak, I. and Shivers, O. TRX: Regular-tree expressions, now in Scheme. Proceedings of the Workshop on Scheme and Functional Programming. 2004. P. 21. URL: http://shivers.com/~shivers/scheme04/tmp/scheme04/proceedings.pdf (дата звернення: 02.11.2025).

11. Hosoya, H. and Pierce, B.C. Regular expression pattern matching for XML. Journal of Functional Programming. 2003. 13(6). P. 961–1004. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/regular-expression-pattern-matching-for-xml/C845B41E6B150FBD7731EE396FBCB911 (дата звернення: 02.11.2025).

12. Rivest, R.L. S-expressions: The syntax for S-expressions. MIT Computer Science and Artificial Intelligence Laboratory. URL: http://theory.lcs.mit.edu/~rivest/sexp.html (дата звернення: 02.11.2025).

13. Stallings W. Cryptography and Network Security: Principles and Practice. – 8th ed. – Pearson, 2023. – 752 с.

14. Schneier, B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 20th Anniversary Edition. Wiley, 2015. 784 p..

15. Tanenbaum, A.S., Feamster, N., and Wetherall, D.J. Computer Networks. 6th ed. Pearson, 2021. 960 p..

16. PKWARE Inc. .ZIP File Format Specification – APPNOTE.TXT.Version 6.3.10. 2023. URL: https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.10.TXT.

17. Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and Thayer, R.OpenPGP Message Format. RFC 4880. IETF, November 2007.URL: https://www.rfc-editor.org/rfc/rfc4880.

18. ISO/IEC 27001:2023. Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою. – К.: ДП «УкрНДНЦ», 2023.

19. ISO/IEC 14764:2006. Software Engineering – Software Life Cycle Processes – Maintenance.Geneva: ISO, 2006 (confirmed 2021).

20. Norvig P. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. – Morgan Kaufmann, 2021. – 944 с.

21. Symantec Corporation. Data Loss Prevention Best Practices Guide. – 2023.

22. Microsoft Corporation. Sensitive Information Types and DLP Policies in Microsoft 365. – Microsoft Docs, 2024.

23. Viega J., Messier M. Secure Programming Cookbook for C and C++. – O'Reilly, 2020. – 820 c.

24. Symantec Corporation. Data Loss Prevention Best Practices Guide. Broadcom, 2024.

25. Viega, J., and Messier, M. Secure Programming Cookbook for C and C++. O'Reilly Media, 2020. 820 p.

26. Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D. Compilers: Principles, Techniques, and Tools. 2nd ed. Pearson, 2006. 1009 p. ISBN 978-0-321-48681-3.

27. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. Introduction to Algorithms. 4th ed. MIT Press, 2022. 1312 p. ISBN 978-0-262-04630-5.

28. ISO/IEC 19770-2:2015. Information technology — Software asset management — Part 2: Software identification tag. Geneva: ISO, 2015.

29. NIST Special Publication 800-57 Part 1 Revision 5. Recommendation for Key Management: Part 1 – General. NIST, May 2020. URL: https://doi.org/10.6028/NIST.SP.800-57pt1r5

30. Anderson, R. Security Engineering: A Guide to Building Dependable Distributed Systems. 3rd ed. Wiley, 2020. 1232 p. ISBN 978-1-119-64278-7.

ДОДАТОК А

Код демонстраційної програми-монітора

```
#define _CRT_SECURE_NO_WARNINGS // Дозволяє використовувати функції
C без попереджень MSVC щодо безпеки
#include <fstream> // Підключаємо роботу з файлами
#include <iostream> // Стандартний ввід/вивід у консоль
#include <windows.h> // Функції WinAPI (для встановлення кодування
консолі)
#include <regex> // Підтримка регулярних виразів
#include <filesystem> // Обхід файлової системи (C++17)
#include <vector> // Динамічні масиви
#include <algorithm> // Алгоритми стандартної бібліотеки

class RegexSearcher // Оголошення класу пошуку за регулярними
виразами
{ // Початок тіла класу
private: // Приватна секція (лише для внутрішнього використання
класом)
  static inline const char* s_regWords[] = // Масив рядків з
регулярними виразами (шаблонами пошуку)
  { // Початок ініціалізації масиву

u8".*?(ТАЄМНО|ЦІЛКОМ[\\s_]ТАЄМНО|ДСК)[\\s\\S]{0,200}?екз\\.?[\\s]*\\
d+[\\s\\S]{0,800}?[\\s\\S]{0,1000}?(?:ГОЛОВНЕ УПРАВЛІННЯ|ЦЕНТРАЛЬНЕ
УПРАВЛІННЯ|ОСНОВНЕ УПРАВЛІННЯ)[\\s\\S]{0,200}?(?:Службова
записка|Наказ|Розпорядження)[\\s\\S]{0,400}?м\\.?[\\s]*(?:Київ|Одеса
|Харків|Львів)[\\s\\S]{0,300}?\\d{1,2}[\\s./-]+\\d{1,2}[\\s./-
]+\\d{4}[\\s]*р?\\.?[\\s]*(?:№|N|#)[\\s]*\\d+[\\s\\S]{0,600}?Про[\\s
]+[^\\r\\n]*[\\s\\S]{0,700}?(?:Начальник|Заступник|Директор|Керівник
)[\\s\\S]{0,200}?(?:відділу|управління|департаменту|служби)[\\s\\S]{
0,200}?[А-ЯІЇЄ][а-яіїє]+[\\s]+[А-ЯІЇЄ]\\.?[\\s]*[А-ЯІЇЄ]\\." //
Основний регулярний вираз: знаходить службові документи з
грифом/містом/датою/№/посадою/ПІБ
  }; // Кінець масиву регулярних виразів

public: // Публічна секція (доступна ззовні)
  static inline const char* _filePattern = R"(\w+\.txt)"; // Шаблон
імен файлів: усі *.txt у поточному дереві каталогів

  bool linesContaining(const std::string& a_filename, const char*
_expression) // Перевіряє, чи містить файл збіг з регулярним виразом
  { // Початок тіла функції
    std::ifstream file(a_filename, std::ios::binary); // Відкриваємо
файл у двійковому режимі для коректного читання байтів
    if (!file) return false; // Якщо файл не відкрився – одразу
повертаємо false

    std::string content((std::istreambuf_iterator<char>(file)),
std::istreambuf_iterator<char>()); // Зчитуємо весь файл у рядок
```

```
    // Удаляем BOM UTF-8, если есть // Коментар оригіналу російською
залишено; нижче – пояснення українською
    if (content.size() >= 3 && // Перевіряємо, що довжина дозволяє
доступ до перших трьох байтів
      (unsigned char)content[0] == 0xEF && // Перший байт BOM UTF-8
      (unsigned char)content[1] == 0xBB && // Другий байт BOM UTF-8
      (unsigned char)content[2] == 0xBF) // Третій байт BOM UTF-8
    { // Початок блоку видалення BOM
      content.erase(0, 3); // Видаляємо три байти BOM з початку
рядка
    } // Кінець блоку видалення BOM

    try { // Блок перехоплення можливих виключень під час роботи з
regex
      std::regex _re(_expression, std::regex_constants::ECMAScript);
// Компілюємо регулярний вираз у діалекті ECMAScript
      return std::regex_search(content, _re); // Повертаємо true,
якщо знайдено будь-який збіг у вмісті
    } // Кінець блоку try
    catch (...) { // На будь-яку помилку компіляції/пошуку
      return false; // Повертаємо false, щоб не зривати виконання
    } // Кінець блоку catch
  } // Кінець функції linesContaining

  std::vector<std::filesystem::directory_entry> findFiles(const
std::filesystem::path& path, const char* _expression) // Повертає
список файлів, що відповідають шаблону імені
  { // Початок функції findFiles
    std::vector<std::filesystem::directory_entry> _res; // Контейнер
для результатів
    std::regex r(_expression, std::regex_constants::ECMAScript); //
Компілюємо регулярний вираз імені файла

    for (auto& entry :
std::filesystem::recursive_directory_iterator(path)) // Рекурсивно
обходимо каталог і підкаталоги
    { // Початок циклу по елементах файлової системи
      try { // Захищаємо доступ до атрибутів файлів від винятків
        if (entry.is_regular_file()) // Перевіряємо, що елемент –
звичайний файл
        { // Початок блоку обробки звичайного файла
          auto fname = entry.path().filename().string(); //
Отримуємо лише ім'я файла (без шляху)
          if (std::regex_match(fname, r)) // Перевіряємо, чи ім'я
відповідає шаблону _filePattern
            _res.push_back(entry); // Якщо так – додаємо файл у
результати
        } // Кінець блоку обробки звичайного файла
      } // Кінець блоку try
      catch (...) {} // Ігноруємо можливі помилки доступу/дозволів
    } // Кінець циклу обходу файлової системи
```

```
    return _res; // Повертаємо зібраний список файлів
  } // Кінець функції findFiles

  void init() // Основна процедура: пошук та звітування у консоль
  { // Початок функції init
    const char* _regExpression = s_regWords[0]; // Обираємо перший
(і єдиний) регулярний вираз для пошуку

    std::cout <<
"================================================\n"; // Рамка зверху
    std::cout << " ПОШУК КОНФІДЕНЦІЙНИХ ДОКУМЕНТІВ\n"; // Заголовок
програми
    std::cout <<
"================================================\n\n"; // Рамка з
відступом

    const auto _fileEntries =
findFiles(std::filesystem::current_path(), _filePattern); //
Знаходимо всі .txt у поточному каталозі (рекурсивно)

    std::cout << "Знайдено файлів: " << _fileEntries.size() <<
"\n\n"; // Виводимо кількість знайдених файлів

    int count = 0; // Лічильник файлів, у яких виявлено збіг

    for (const auto& entry : _fileEntries) // Перебираємо кожен
знайдений файл
    { // Початок циклу по знайдених файлах
      std::string filename = entry.path().filename().string(); //
Беремо ім'я файла для виводу
      std::cout << "[" << filename << "] … "; // Друкуємо мітку
файла з еліпсисом, поки триває перевірка

      if (linesContaining(entry.path().string(), _regExpression)) //
Перевіряємо вміст файла на збіг регулярному виразу
      { // Початок гілки, коли збіг знайдено
        std::cout << ">>> ЗНАЙДЕНО! <<<\n"; // Повідомляємо про
успішний збіг
        count++; // Збільшуємо лічильник знайдених збігів
      } // Кінець гілки знайдено
      else // Якщо збіг не знайдено
      { // Початок гілки, коли збіг відсутній
        std::cout << "ні\n"; // Повідомляємо, що збігів немає
      } // Кінець гілки, коли збіг відсутній
    } // Кінець циклу по файлах

    std::cout <<
"\n================================================\n"; // Рамка
перед підсумком
    if (count > 0) // Якщо знайдено хоча б один збіг
      std::cout << "УСПІШНО! Знайдено: " << count << "\n"; //
Позитивний підсумок із кількістю
```

```
    else // Якщо збігів не знайдено
      std::cout << "НЕ ЗНАЙДЕНО: " << count << "\n"; // Негативний
підсумок (0)
    std::cout <<
"=============================================\n"; // Нижня рамка
  } // Кінець функції init
}; // Кінець оголошення класу RegexSearcher

int main() // Точка входу програми
{ // Початок main
  // Устанавливаем кодовую страницу UTF-8 для консоли //
Оригінальний коментар; нижче – фактичні виклики для UTF-8
  //SetConsoleOutputCP(65001); // Альтернативний варіант: ручне
встановлення CP для виводу (закоментовано навмисно)
  //SetConsoleCP(65001); // Альтернативний варіант: ручне
встановлення CP для вводу (закоментовано навмисно)

  //system("chcp 866 > nul"); // Приклад перемикання OEM-сторінки
(DOS-866); не потрібен з CP_UTF8

  SetConsoleOutputCP(CP_UTF8); // Встановлюємо кодування UTF-8 для
виводу у консоль Windows
  SetConsoleCP(CP_UTF8); // Встановлюємо кодування UTF-8 для вводу у
консоль Windows

  RegexSearcher().init(); // Створюємо тимчасовий об'єкт і
запускаємо основну процедуру пошуку

  std::cout << "\nНатисни Enter..."; // Просимо користувача
натиснути Enter перед виходом
  std::cin.get(); // Чекаємо на натискання Enter

  return 0; // Повертаємо код успішного завершення
} // Кінець main
```

ДОДАТОК Б

Коротка характеристика криптографічних програм

Програма Crypt-O-Text

Crypt-O-Text – програма для шифрування текстових повідомлень, що формує компактні зашифровані блоки у спеціальному форматі з характерною сигнатурою. Вміст шифрованого файлу розташовується між спеціальними маркерами START та END, що спрощує його автоматичне виявлення. Нижче наведено приклад вмісту такого файлу:

********** START Crypt-O-Text **********

```
CoT/0124/03/8/436

CTltw1F3eEHT3eGkQnYuXyU3eWB8Aol4q0NyXsVc4gRJKt-79r0SL+QvChL1MI-FC

CI2KMkLSLi2ZtSITFaBGJivJcqWm2rBx1M5Vkb0YxjyXxL3PVb8CCKBRoUt55y3tt

...

Crz8e+pyXc2klct16EEJP

CoT/16637
```

********** END Crypt-O-Text **********

Особливістю програми є відсутність загальновідомого алгоритму шифрування, але також – надзвичайно слабкий захист пароля. Завдяки аналізу за допомогою дизасемблера SoftIce було встановлено:

- структура CoT/0124/03/8/436 означає:
- 8 – довжина пароля;
- 436 – кількість зашифрованих символів;
- версія програми 1.24 підтримує паролі довжиною від 1 до 45 символів;
- пароль відновлюється за менше ніж 1 секунду незалежно від його довжини.

Програма EasyCrypt v.1.0c

Програма EasyCrypt v.1.0c реалізує два алгоритми шифрування: DES (56 біт) і CAS. Під час шифрування вона додає до файлу сигнатуру, яка дозволяє розпізнати використовуваний алгоритм:

- Приклад заголовку для CAS:

```
EZC/CAS F9A129E00D35409DCF4B5BB9@@EG(.w $КОФяс+■5 8
```

- Приклад заголовку для DES:

```
EZC/DES 2177B0E2E6A83D32C80D1158@@FC▌┌2 >j :J1ъ╬К 4в
```

Таким чином, розпізнавання алгоритму можливо здійснити шляхом аналізу першого рядка файлу.

Програма WinCrypt

WinCrypt – шифрувальний інструмент з графічним інтерфейсом користувача. Підтримує ключі довжиною 128, 192 та 256 біт. Нижче наведено приклад зашифрованого вмісту:

```
LOKI
      ч                ╬_eOPKAK5цsобцN╝fm&K}:_}■ь=ъ√aж╤|·ьш_Дzyh″═_J_
```

Заголовок LOKI свідчить про використання внутрішнього формату програми. Типові особливості:

- обов’язковий префікс LOKI;
- складний набір байтів з непридатним до друку виводом;
- фіксовані довжини ключа (AES-подібна реалізація).

Программа Figa 2.01e

Figa 2.01e – утиліта для симетричного шифрування будь-яких файлів за допомогою довільного пароля.

Ключові особливості:

- Підтримка довільної довжини пароля (у розумних межах);
- Рекомендоване використання латинських символів;
- Розширення зашифрованого файлу – .fga;
- Шифрування блочне, з перемішуванням байтів пароля із зашифрованими даними;
- Існує можливість не записувати контрольну суму для ускладнення зловмисного підбору пароля.

Приклад зашифрованого файлу:

```
Figa2.01 _Жa$.........
```

Дані:

- Password – строка з паролем;
- PassLen – довжина пароля;
- DataBuf – дані для обробки;
- DataLen – кількість байтів для обробки;
- OutBuf – буфер для оброблених даних.

Функція:

```
ChangePassword() – перетворює пароль наступним чином:
{
Sum – глобальна змінна
i = 1
ПОКИ (i <= PassLen) ВИКОНУВАТИ
```

```
 {
 Sum = Sum + Password[i]
 i = i + 1
 }
i = 1
ПОКИ (i <= PassLen) ВИКОНУВАТИ
 {
 Password[i] = Password[i] + Sum
 i = i + 1
 }
}
```

Алгоритм шифрування:

```
Sum = 0 – глобальна змінна
i = 1
ПОКИ (i <= DataLen / PassLen) ВИКОНУВАТИ
 {
 k = 1
 ChangePassword()
 ПОКИ (k <= PassLen) ВИКОНУВАТИ
  {
  OutBuf[i + k] = DataBuf[i + k] + Password[k]
  Password[k] = Password[k] + DataBuf[i + k]
  k = k + 1
  }
 i = i + 1
 }
```

Алгоритм розшифрування:

```
Sum = 0 – глобальна змінна
i = 1
ПОКИ (i <= DataLen / PassLen) ВИКОНУВАТИ
 {
 k = 1
 ChangePassword()
 ПОКИ (k <= PassLen) ВИКОНУВАТИ
  {
  OutBuf[i + k] = DataBuf[i + k] - Password[k]
  Password[k] = Password[k] + OutBuf[i + k]
  k = k + 1
  }
 i = i + 1
 }
```

Формат зашифрованого файлу

Перші 16 байт:

- ProgName – назва програми (4 байти);
- ProgVer – версія програми (4 байти);

- CheckSum – контрольна сума заголовку (4 байти);
- HeaderLen – довжина заголовку (4 байти).

Зашифрований заголовок:

- Level – рівень випадковості (1 байт);
- RandomBytes – 4 × (PassLen + Level);
- NextBlockLen – розмір наступного блоку (4 байти);
- FileName – ім’я початкового файлу.

Блоки даних:

- RandomBytes;
- NextBlockLen;
- ata – зашифровані байти

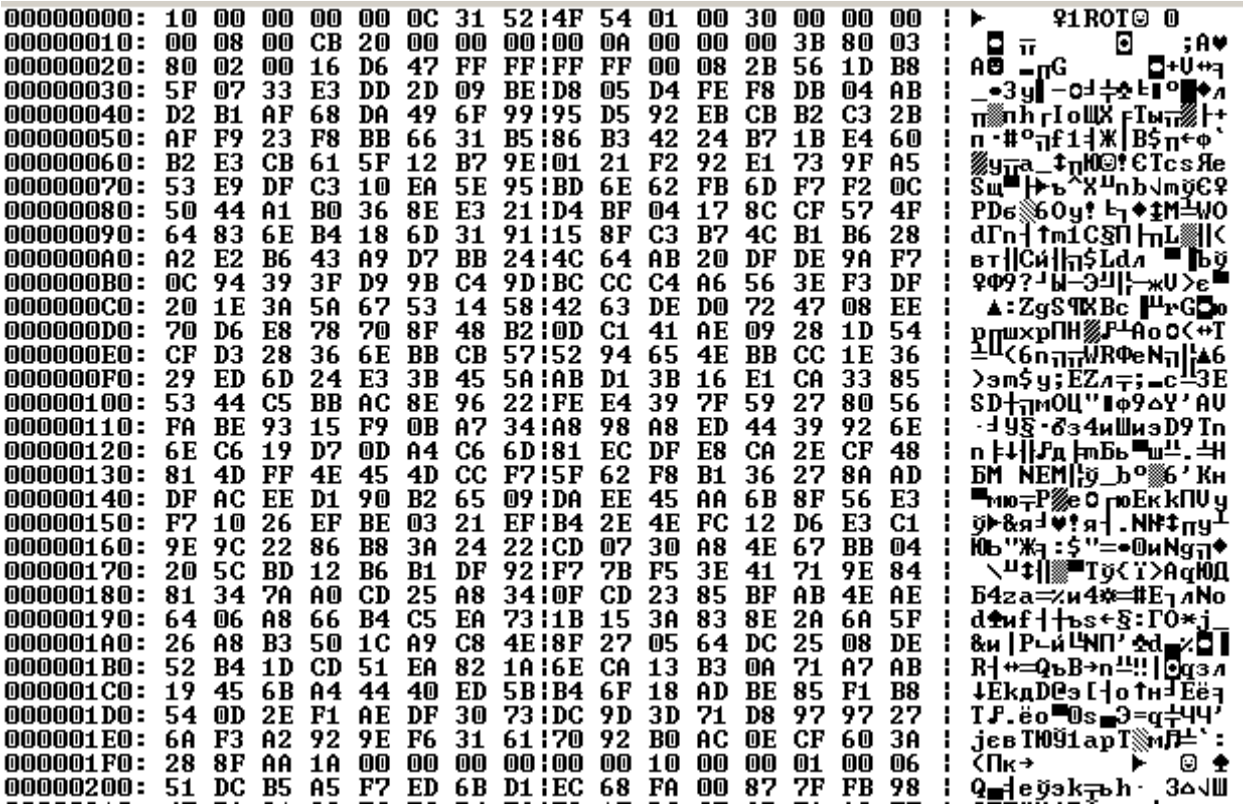
Розмір зашифрованого файлу:

```
21 + NameLen + 4 × (PassLen + Level + 1) × (BlocksNumber + 1)
```

Для файлів < 20 КБайт, BlocksNumber = 1. Наприклад, якщо test.txt шифрується паролем «pAroL_20*?» на рівні 20, то розмір зросте на 277 байт, а на рівні 100 – на 917 байт.

Приклади оформлення результатів роботи програм та аналізу їх структурних повторень.

Програма 4Safe StrongDisk 2.8.0.2.



```
2_L=97, K=41, N1=0, N2=0, F1=Image1.grd,  F2=Image1a.grd
 10►  00   00   00   00   0C♀  311  52R  4FO  54T  01☺  00   300  00   00   00
00   08◘  00   [1] 20   00   00   00   00   0A♪  00   00   17‡  FF   80A  03♥
80A [1] 00   16▬ [2] FF   FF   FF   FF   00   08◘ [53] 91C

2_L=43, K=42, N1=0, N2=0, F1=Image1.grd,  F2=Image2.grd
 10►  00   00   00   00   0C♀  311  52R  4FO  54T  01☺  00   300  00   00   00
00   08◘  00   CB╦  20   00   00   00   00   0A♪  00   00   17‡  FF   80A  03♥
80A  02☻  00   16▬ [2] FF   FF   FF   FF   00   08◘

2_L=43, K=40, N1=0, N2=0, F1=Image1.grd,  F2=Image3.grd
 10►  00   00   00   00   0C♀  311  52R  4FO  54T  01☺  00   300  00   00   00
00   08◘  00   CB╦  20   00   00   00   00   0A♪  00   00  [2] 80A  03♥  80A
02☻  00   16▬ [2] FF   FF   FF   FF   00   08◘

2_L=43, K=40, N1=0, N2=0, F1=Image1a.grd,  F2=Image2.grd
 10►  00   00   00   00   0C♀  311  52R  4FO  54T  01☺  00   300  00   00   00
00   08◘  00  [1] 20   00   00   00   00   0A♪  00   00   17‡  FF   80A  03♥
80A [1] 00   16▬ [2] FF   FF   FF   FF   00   08◘

2_L=43, K=40, N1=0, N2=0, F1=Image2.grd,  F2=Image3.grd
 10►  00   00   00   00   0C♀  311  52R  4FO  54T  01☺  00   300  00   00   00
00   08◘  00   CB╦  20   00   00   00   00   0A♪  00   00  [2] 80A  03♥  80A
02☻  00   16▬ [2] FF   FF   FF   FF   00   08◘
```

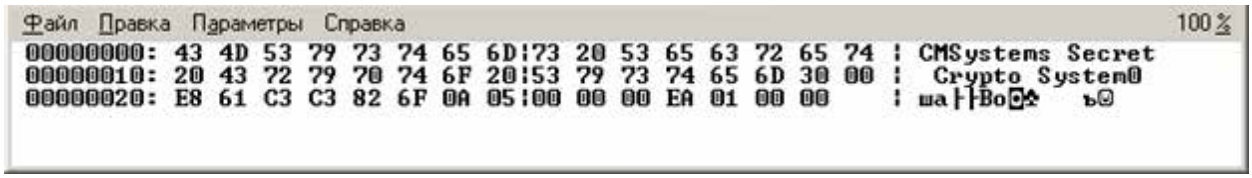
Програма: A-Lock 5.1. (початок файлу, упаковано MIME64).

```
<<START_PC_Encrypt_DATA>>
CgAAAIcBAAD/9cFk6do3mwc39NmwEStHZE2OeFKaF3v6/KrMJqg/44QhKPhC6zFT9LiLEyfA8uqU
1OnXHS5IUyNluOgBWIPJ8cRA11976vhOvhSPeztg+MuFp2y1WytPsrYbi20wiZYIx4/ILUHbnFa7
NLXYXlaahw3hIu0wv7zxRzjUAAAAAKB8GZDMLImMvp4hUnKhNSA5QmP81DAz4xJUxQKcnIhaDgb0
cbx4hf6fug7TgryzsX7bu0wD47NDrHe7w3NLANMz2UNwZ1lwdT2EZcFYkbHIdM5qqGrQ6amgMrcS
nNfq1E7yGD0OAzXyMC7hexqpS5S1KPNu1IOggJ7ra6WxTSWO5zuUvZ6oT/TJY+biS9sgqcS6oLKn
lf5ci2Oudlwo4CjlSBGpyPxi6BZEKyfzpQRdWy6rXmwK5gtp3WNKGUWdnLg/ebztF1Q/px+ehU4w
X3g+wjWTah9b7w8brd1S8FIJpbMKV8WjpkJalEHbN6/TwqxyXOayMw9uA4qlkHzaknmZcrZPgoza
m4emd/H6iL0DS4ihuNJWZzgmDC/AHjnBvmgMV1Qp66/5JdZTxHV01lSf/rVcXohLoQGUwu8BrfMi
aueivUCU5YHH2KTPtwlb2P1wMrQIwFkgO742ewjUtDKNIL4mXBWkY7tuphIRUVU0bnnBLCW/PGqP
jRNCOIz143aWKZAyySq4PL9Fz57bqyE23ot4DRL9o1DdJRt7ZpYW0PKvLeFPm4MSdlhO7PUf5+fn
MnvFuU2hviHv4vrmRsFG8K5/vC7tbO6HDB3Lo/XzvGm6mC66Mg0AnYkzEwxYWPZ1GrG64edmMQ0S
xmMA/fWF6RrYsPKwsxtZlekO/qWuQ+avTWkpiKBqmHumR8420g/+DgybBKZJHpq7hwx+CwkznXEb
kA5UmkfXGICoGl23DhS+E46y+EW418XxiySUMnDNvajh393j31v4RspB3ovacmuOC0s/1UWqozwb
Opk95tyd+7nCk7Qj3pI901LUeardJzAk+UpIN779Uq7clmbe+QKvTCg4XOjfJRrSARacORWmPo1Q
m7BZqYJCmm+9HK6f1cXFAdy8RH0Bt7HjHHr7J1i9VyP6XzWW739yPUQXh7lQ7XROvi6Eh0yDNGx0
..
```

```
2_L=18, K=8, N1=2, N2=2, F1=aaa1111.txt, F2=qqq!!!.txt
 41A 41A 41A [1] 63c [1] 41A 41A [1] 2F/ [8] 6Dm

2_L=29, K=7, N1=153, N2=153, F1=aaa1111.txt, F2=qqq!!!.txt
 20 [5] 6Cl [18] 41A 41A 41A 41A 41A
```

```
2_L=25, K=9, N1=1, N2=1, F1=aaa1111.txt, F2=qqqxxx.txt
 67g 41A 41A 41A [2] 42B 41A 41A [1] 2F/ [14] 6Dm

3_L=15, K=7, N1=175, N2=174, F1=aaa1111.txt, F2=qqqxxx.txt
 7Az 6Aj [2] 41A 41A 41A 41A [7] 4CL

1_L=29, K=7, N1=2, N2=180, F1=aaa2222.txt, F2=aaa2222.txt
 41A 41A 41A 4EN [13] 366 [7] 73s [3] 47G

1_L=29, K=7, N1=180, N2=2, F1=aaa2222.txt, F2=aaa2222.txt
 41A 41A 41A 4EN [13] 366 [7] 73s [3] 47G

2_L=29, K=7, N1=2, N2=2, F1=aaa2222.txt, F2=qqq!!!.txt
 41A 41A 41A [2] 44D 41A 41A [21] 47G

2_L=19, K=7, N1=178, N2=178, F1=aaa2222.txt, F2=qqq!!!.txt
 41A 41A 41A 41A 41A [12] 311 [1] 366
```

Пакет CMSystems Crypto



ДОДАТОК В

Код програми для виявлення шифрування у ZIP-архівах через бібліотеку zipfile

```
import os # робота з файлами
import zipfile # робота з ZIP-архівами

zip_folder =
"D:/Magisterska_robota/Programs/Method_Traditional_Zipfile/test_arch
ives" # папка з архівами
output_file =
"D:/Magisterska_robota/Programs/Method_Traditional_Zipfile/results/e
ncrypted_zips.txt" # вихідний файл

def check_zip_encryption(zip_path): # перевірка шифрування ZIP
  try:
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
      for file_info in zip_ref.infolist():
        if file_info.flag_bits & 0x1: # якщо файл зашифрований
          return True
  except zipfile.BadZipFile: # не ZIP або пошкоджений
    return False
  return False

encrypted_files = [] # список зашифрованих
for filename in os.listdir(zip_folder): # обхід файлів
  if filename.endswith(".zip"): # тільки .zip
    zip_path = os.path.join(zip_folder, filename)
    if check_zip_encryption(zip_path): # перевіряємо
      encrypted_files.append(zip_path)

with open(output_file, "w") as f: # запис результатів
  for file in encrypted_files:
    f.write(file + "\n")

print(f"Список зашифрованих ZIP-файлів збережено у {output_file}") #
повідомлення
```

ДОДАТОК Г

Код програми для виявлення зашифрованих ZIP-архівів на основі S-виразів

```
import os # робота з файлами
import struct # обробка байтів

ZIP_DIR =
"D:/Magisterska_robota/Programs/Method_Universal_S_Expr/test_archive
s" # тека ZIP
S_EXPR_DIR =
"D:/Magisterska_robota/Programs/Method_Universal_S_Expr/s_expression
s" # тека S-виразів
OUTPUT_FILE =
"D:/Magisterska_robota/Programs/Method_Universal_S_Expr/results/encr
ypted_found.txt" # результат

def parse_s_expr(file_path): # зчитування S-виразу
  with open(file_path, encoding="utf-8") as f:
    content = f.read().lower()
    if "pkzip" in content and "general purpose bit flag" in content:
# формат ZIP
      return {
        "name": "PKZIP",
        "type": "zip",
        "signature": b'\x50\x4B\x03\x04',
        "flag_offset": 6,
        "flag_bit": 0
      }
  return None # якщо не впізнано

def is_encrypted_zip(file_path, format_info): # перевірка ZIP на
шифрування
  try:
    with open(file_path, "rb") as f:
      data = f.read(100)
      sig_index = data.find(format_info["signature"]) # пошук
сигнатури
      if sig_index == -1:
        return False
      flag_start = sig_index + format_info["flag_offset"]
      flag_bytes = data[flag_start:flag_start+2]
      if len(flag_bytes) < 2:
        return False
      flag = struct.unpack("<H", flag_bytes)[0] # читання прапорців
      return bool(flag & (1 << format_info["flag_bit"])) # біт
шифрування
  except Exception as e:
    print(f"[Помилка] {file_path}: {e}")
    return False
```

```
def main(): # головна функція
  encrypted_files = []
  s_expr_files = [os.path.join(S_EXPR_DIR, f) for f in
os.listdir(S_EXPR_DIR) if f.endswith(".txt")]
  zip_files = [os.path.join(ZIP_DIR, f) for f in os.listdir(ZIP_DIR)
if f.lower().endswith(".zip")]

  for s_expr_path in s_expr_files: # аналіз кожного S-виразу
    format_info = parse_s_expr(s_expr_path)
    if not format_info:
      print(f"[!] Невідомий формат: {s_expr_path}")
      continue
    print(f"[+] Формат: {format_info['name']}")

    for zip_path in zip_files: # перевірка кожного ZIP
      if is_encrypted_zip(zip_path, format_info):
        print(f"[✓] Зашифрований: {os.path.basename(zip_path)}")
        encrypted_files.append(os.path.basename(zip_path))

  with open(OUTPUT_FILE, "w", encoding="utf-8") as f: # запис
результатів
    for name in encrypted_files:
      f.write(name + "\n")

  print(f"Знайдено: {len(encrypted_files)} | Результат:
{OUTPUT_FILE}") # підсумок

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

ДОДАТОК Д

Приклад опису ZIP-архіву за допомогою S-виразів

```
      (pkzip_v2.04
       (Local_file_header
        (local file header signature #50 4B 03 04#)
        (version needed to extract #14 00#)
        (general purpose bit flag #09 00#)
        (compression method #08 00#)
        (last mod file time #43 89#)
        (last mod file date #67 2E#)
        (crc-32 #CC 80 08 5B#)
        (compressed size #F8 00 00 00#)
        (uncompressed size #93 01 00 00#)
        (filename length #01 00#)
        (extra field length #00 00#)
        (filename #6D#)
        (compress data #31 CE 19 C8 05 3C 65 43 6C DE FC E6 90 E8 21
7C 77#))
       (Data_descriptor
       (data descriptor header signature #50 4B 07 08#)
       (crc-32 #CC 80 08 5B#)
       (compressed size #F8 00 00 00#)
       (uncompressed size #93 01 00 00#))
       (File_header
       (central file header signature #50 4B 01 02#)
       (version made by #14 00#)
       (version needed to extract #14 00#)
       (general purpose bit flag #09 00#)
       (compression method #08 00#)
       (last mod file time #43 89#)
       (last mod file date #67 2E#)
       (crc-32 #CC 80 08 5B#)
       (compressed size #F8 00 00 00#)
       (uncompressed size #93 01 00 00#)
       (filename length #01 00#)
       (extra field length #00 00#)
       (file comment length #00 00#)
       (disk number start #00 00#)
       (internal file attributes #01 00#)
       (external file attributes #20 00 00 00#)
       (relative offset of local header #00 00 00 00#)
       (filename #6D#))
       (End_of_central_dir_record
       (end of central dir signature #50 4B 05 06#)
       (number of this disk #00 00#)
       (start of the central directory #00 00#)
       (total number of entries in the central dir on this disk #01
00#)
       (total number of entries in the central dir #01 00#)
```

```
 (size of the central directory #2F 00 00 00#)
 (starting_disk_number #27 01 00 00#)
 (zipfile comment length #00 00#)
 (zipfile comment #00 00 00 00 00 00#))
)
```