МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Державне некомерційне підприємство
«Державний університет» Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій

Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____ Олена ГРІНЕНКО
«_____» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Сканер секретів та небезпечних паттернів у коді та конфігураціях

Виконавець: Рябчук Олександр Володимирович

Керівник: завідувач кафедри, к.т.н., доцент Гріненко Олена Олександрівна

Нормоконтролер:. завідувач кафедри, к.т.н., доцент Гріненко Олена Олександрівна

Київ 2025

Державне некомерційне підприємство

«Державний університет» Київський авіаційний інститут»

Факультет комп’ютерних наук та технологій

Кафедра інженерії програмного забезпечення

Спеціальність 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олена ГРІНЕНКО

«_____» _____ 2025 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи студента

Рябчука Олександра Володимировича

1. Тема кваліфікаційної роботи: Сканер секретів та небезпечних паттернів у коді та конфігураціях

затверджена наказом ректора 01.04.2024 року No 122/од

2. Термін виконання проекту: з 29.09.2025 р. по 21.12.2025 р.

3. Вихідні дані до роботи: Реалізувати програмний прототип у вигляді легкого інструмента з інтерфейсом для локального запуску в середовищі VSCode, що застосовує правило-залежний (regex) та ентропійний підходи для детекції секретів, маскує знайдені значення та генерує структуровані звіти (JSON).

Сформувати набір правил і конфігурацій (YAML/JSON) для виявлення

типових секретів (API-ключі, токени, приватні ключі) і небезпечних конструкцій (наприклад, виклики eval), з можливістю налаштування порогів ентропії та виключень шляхів.

4. Зміст пояснювальної записки:

1. Теоретичні основи виявлення секретів і небезпечних паттернів у програмному коді та конфігураціях

2. Архітектура та проектування інструмента для виявлення секретів і небезпечних паттернів у програмному коді та конфігураціях

3. Реалізація прототипу

4. Тестування програмного засобу

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1. Приклад випадкового витоку API-ключа у публічному коміті (ілюстрація ризику).

2. Структура PEM-приватного ключа (заголовок / тіло / кінець).

3. Розподіл ентропійних значень для токенів — ілюстрація порогів детекції.

4. Зразок JSON-звіту із маскуванням токенів та полями метаданих.

5. Приклад виклику eval у JavaScript — типова ознака небезпечного патерну.

6. Типовий workflow GitHub Actions для сканування секретів у PR.

7. Структура JWT (header.payload.signature) — зразок токена з високою ентропією.

8. Розподіл TP/FP/FN по категоріях — ілюстрація джерел помилок.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи	29.09-01.10.2025	виконано
2.	Ознайомлення з постановкою задачі, вивчення інформаційних джерел та складання плану роботи.	02.10-05.10.2025	виконано
3.	Підготовка 1 розділу та подання його керівнику	06.10-19.10.2025	виконано
4.	Підготовка 2 розділу та подання його керівнику	20.10-02.11.2025	виконано
5.	Підготовка 3 розділу та подання його керівнику	03.12-16.11.2025	виконано
6.	Підготовка 4 розділу і висновків по роботі та подання їх керівнику	17.11-30.11.2025	виконано
7.	Загальне редагування пояснювальної записки, графічного матеріалу. тавлення роботи для перевірки на академічну доброчесність. Проходження нормоконтролю.	01.12-07.12.2025	виконано
8.	Отримання відгуку керівника. Підготовка презентації та тексту доповіді.	08.12-14.12.2025	виконано
9.	Попередній захист (представлення електронної версії пояснювальної записки, презентації, позитивного відгуку керівника).	08.12-14.12.2025	виконано
10.	Рецензування кваліфікаційної роботи	15.12-22.12.2025	виконано
11.	Здача секретарю ЕК пояснювальної записки: електронної версії кваліфікаційної роботи; презентації доповіді; відгуку керівника, рецензії; результату проходження перевірки на плагіат; довідки про успішність, декларації про академічну доброчесність.	15.12-22.12.2025	виконано
12.	Захист кваліфікаційної роботи перед екзаменаційною комісією	29.12.2025	виконано

Дата видачі завдання 29.09.2025 р.

Керівник кваліфікаційної роботи:

завідувач кафедри, к.т.н., доцент Олена ГРІНЕНКО

Завдання прийняв до виконання: Олександр РЯБЧУК

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Сканер секретів та небезпечних патернів у коді та конфігураціях»: 81 сторінка, 8 рисунків, 4 таблиці, 3 схеми, 30 використаних джерел, 1 додаток.

Об'єкт дослідження – процеси забезпечення інформаційної безпеки на етапі розробки ПЗ, зокрема витоки секретів у вихідному коді та конфігураційних файлах.

Мета кваліфікаційної роботи – розробка та апробація прототипу інструмента для автоматизованого виявлення секретів і очевидних небезпечних патернів у файлах проекту.

Методи дослідження – правило-орієнтований аналіз (regex), ентропійний аналіз (Shannon), контекстний аналіз, модульне проєктування, прототипування на Python, формування тестових наборів і вимірювання метрик (TP/FP/FN, Precision, Recall, F1).

Результати роботи містять реалізований прототип сканера з модульною архітектурою, набір початкових правил у форматі YAML/JSON, механізм маскування і формування JSON-звітів, а також експериментальні показники (приблизно Precision ≈ 79.3%, Recall ≈ 85.7%, F1 ≈ 82.4%) і рекомендації щодо зниження хибних спрацьовувань та інтеграції в CI.

Розробка та тестування проводилися у середовищі Visual Studio Code на Python 3.8+ під ОС Windows.

КЛЮЧОВІ СЛОВА: СКАНЕР СЕКРЕТІВ, ВИЯВЛЕННЯ СЕКРЕТІВ, НЕБЕЗПЕЧНІ ПАТЕРНИ, REGEX, ЕНТРОПІЯ, PYTHON, VSCODE, JSON.

ABSTRACT

Explanatory note to the master's thesis "Scanner for Secrets and Dangerous Patterns in Source Code and Configurations": 81 pages, 8 figures, 4 tables, 3 diagrams, 30 references, 1 appendix.

The object of research is the processes of ensuring information security during software development, in particular the leakage of secrets in source code and configuration files.

The aim of the thesis is to develop and validate a prototype tool for automated detection of secrets and obvious dangerous patterns in project files.

Research methods include rule-based analysis using regular expressions, entropy-based analysis (Shannon entropy), contextual analysis, modular software design, prototyping in Python, creation of test datasets, and measurement of evaluation metrics (TP/FP/FN, Precision, Recall, F1).

The results of the work include an implemented prototype scanner with a modular architecture, an initial set of detection rules in YAML/JSON format, mechanisms for masking sensitive data and generating JSON reports, as well as experimental evaluation results (approximately Precision ≈ 79.3%, Recall ≈ 85.7%, F1 ≈ 82.4%) and practical recommendations for reducing false positives and integrating the tool into CI pipelines.

The development and testing were carried out under the Windows operating system. The software was developed in the Visual Studio Code environment using the Python programming language.

SECRET SCANNER, SECRET DETECTION, DANGEROUS PATTERNS, REGEX, ENTROPY, PYTHON, SOURCE CODE ANALYSIS, CONFIGURATION FILES.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасних умовах стрімкого зростання ролі інформаційних технологій програмне забезпечення виступає базовою інфраструктурою для державних установ, фінансових установ, промислових підприємств та інших критичних сфер. Паралельно з цим зростає обсяг вихідного коду, конфігураційних файлів та автоматизованих сценаріїв розгортання, що підвищує ймовірність випадкового оприлюднення конфіденційних даних або допущення небезпечних конфігурацій. Типовими прикладами таких проблем є жорстко закодовані API-ключі, токени доступу, приватні криптографічні ключі, а також небезпечні конструкції у коді (динамічне виконання коду, неконтрольовані виклики системних команд тощо), які можуть стати векторами компрометації системи. Ці виклики набувають особливої ваги в контексті хмарних сервісів і «інфраструктури як код», де неправильні налаштування мають миттєвий масштабний вплив. Ураховуючи обмеження та складність існуючих корпоративних рішень, актуальним є створення простих, зрозумілих і доступних інструментів для початкової перевірки безпеки коду і конфігурацій на етапі розробки.

Мета і задачі дослідження. Метою цієї кваліфікаційної роботи є розробка працездатного прототипу програмного засобу для автоматизованого виявлення секретів і небезпечних патернів у вихідних кодах та конфігураціях із подальшим оцінюванням ефективності простих евристичних методів виявлення. Для досягнення цієї мети визначено низку взаємопов’язаних задач: аналіз типових секретів і патернів; огляд існуючих підходів і інструментів; формулювання правил і критеріїв детекції; проектування архітектури прототипу; реалізація

інструменту; побудова набору тестових даних та методики тестування; проведення експериментів і аналіз отриманих результатів.

Об’єкт і предмет дослідження. Об’єктом дослідження виступає процес забезпечення інформаційної безпеки програмного забезпечення на етапі розробки, з акцентом на обробку вихідного коду та конфігурацій. Предметом - методи, правила та програмні засоби для автоматизованого виявлення секретів і небезпечних патернів, зокрема: використання регулярних виразів для статичних шаблонів, ентропійний аналіз для виявлення випадкових/зашифрованих рядків, контекстний аналіз для зниження хибних спрацьовувань та побудова евристик для класифікації ризику.

Наукова та практична новизна. Практичною цінністю роботи є створення прототипу-інструменту, який надає швидкий і прозорий механізм первинного аналізу файлів (сканування каталогу/файлу, застосування набору правил, ентропійна перевірка, маскування знахідок і формування звіту). Пропонований комбінований підхід (rule-based + entropy + context) дозволяє поєднати простоту впровадження з підвищеною універсальністю детекції, що робить рішення доцільним для навчального середовища й малих команд розробників.

Методологія роботи. Дослідження поєднує огляд літератури та існуючих інструментів, проєктування модульної архітектури прототипу, реалізацію фундаментальних модулів (парсер, ядро правил, ентропійний аналізатор, контекстний аналізатор, репортер), побудову контрольованих тестових наборів (synthetic, edgecases, large) та експериментальну оцінку за метриками precision, recall і F1. Для перевірки працездатності реалізація тестувалася у середовищі Visual Studio Code із застосуванням автоматичних і ручних сценаріїв.

Очікувані результати та практична користь. Результатом має стати робочий прототип, що сканує файлову систему або каталог, виявляє типові секрети і прості небезпечні патерни, формує маскований звіт та надає рекомендації щодо

реагування. Такий інструмент корисний для самоперевірки коду розробниками, як навчальний матеріал для студентів і як основа для інтеграції у CI/CD-процеси.

Структура роботи. Робота складається з вступу, розділів з оглядом підходів і реалізації архітектури, описом реалізації прототипу та методикою тестування, експериментальною частиною із кількісним аналізом результатів, висновків та переліку використаних джерел. Детальна організація забезпечує логічний перехід від теоретичних основ до практичної реалізації й емпіричної оцінки.

Розділ 1

ТЕОРЕТИЧНІ ОСНОВИ ВИЯВЛЕННЯ СЕКРЕТІВ І НЕБЕЗПЕЧНИХ ПАТЕРНІВ У ПРОГРАМНОМУ КОДІ ТА КОНФІГУРАЦІЯХ

1.1. Актуальність теми дослідження та мета роботи

У XXI столітті програмне забезпечення відіграє ключову роль у функціонуванні державних інституцій, фінансового сектору, промислових підприємств, охорони здоров’я та бізнесу в цілому. Зі зростанням складності інформаційних систем та обсягів коду зростає й ризик випадкового або навмисного розкриття конфіденційних даних у вигляді вбудованих у код секретів (API-ключі, токени, паролі, приватні ключі) та виникнення небезпечних конфігурацій.[2]

Паралельно набуває поширення модель «інфраструктура як код», контейнеризація та масове застосування хмарних сервісів, що підвищує потенційний масштаб негативних наслідків від помилок у коді або налаштуваннях.[1]

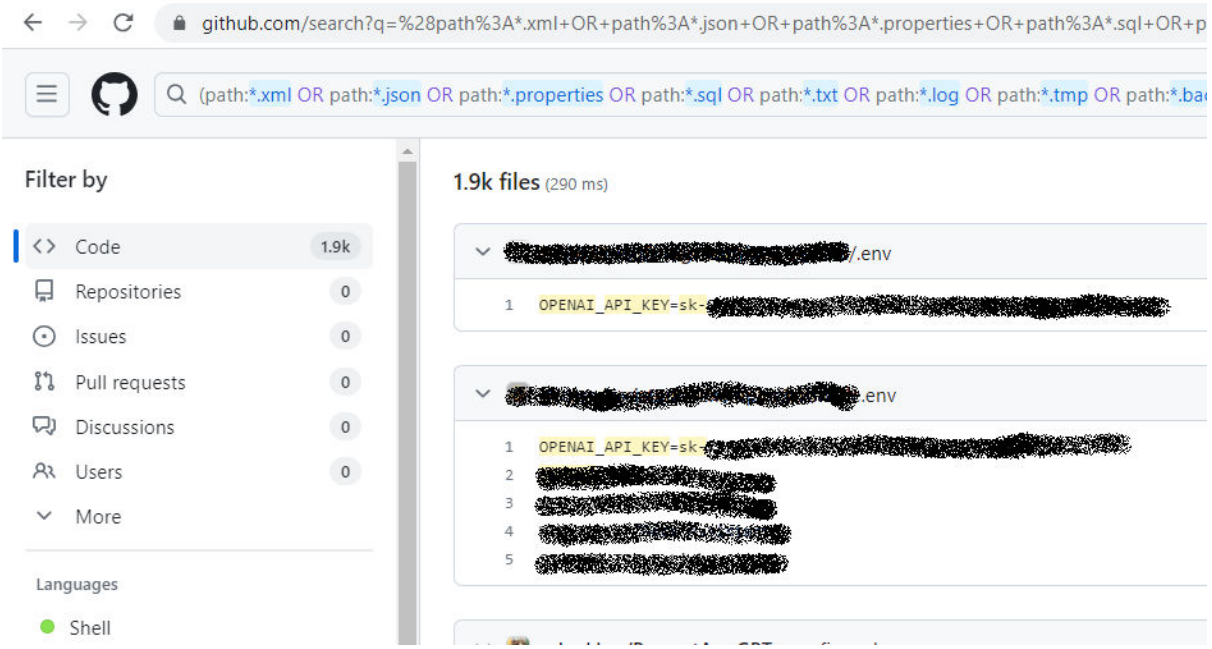

рис 1.1: Приклад випадкового витоку API-ключа у публічному коміті (ілюстрація ризику).

Практична актуальність дослідження зумовлена кількома факторами. По-перше, витоки секретів із репозиторіїв спричиняють прямі ризики компрометації сервісів та фінансові втрати. По-друге, наявність у коді патернів, здатних призвести до виконання небезпечного коду або відкриття недозволених інтерфейсів, створює основу для успішних атак (RCE, privilege escalation, конфігураційних уразливостей). По-третє, більшість існуючих корпоративних рішень для виявлення секретів є комплексними, ресурсноємними та складними у впровадженні, тому виникає необхідність у простих, прозорих інструментах, які дозволяють проводити попередню перевірку коду вже на локальному етапі розробки.[3]

Мета цієї роботи - розробити та апробувати прототип інструменту для автоматизованого виявлення секретів і небезпечних патернів у вихідному коді та конфігураціях, а також дослідити ефективність комбінованого підходу, що поєднує правилозалежні методи, ентропійний аналіз і прості контекстні фільтри. Для досягнення мети визначено такі завдання: систематизувати типи секретів і

патернів; проаналізувати існуючі підходи та інструменти; сформулювати набір правил і критеріїв детекції; спроектувати архітектуру прототипу; реалізувати базову версію інструменту та побудувати методику тестування з метриками precision/recall/F1; провести експериментальні прогони та оцінити практичну придатність рішення.[3][4]

1.2. Загальні підходи до виявлення секретів у програмному коді

У практиці виявлення секретів та потенційно чутливих рядків у коді виокремлюють кілька основних підходів, кожний із яких має свої сильні і слабкі сторони.[6] До ключових підходів належать: правилозалежний (rule-based) аналіз, ентропійний аналіз, контекстний і синтаксичний аналіз (SAST/AST) та комбіновані стратегії, які поєднують вищезгадані методи.[5],[6].

Правилозалежний (regex) аналіз. Цей підхід базується на наборі шаблонів (регулярних виразів), які описують відомі формати ключів і токенів (наприклад, характерні префікси та довжини AWS-ключів, формати JWT, PEM-заголовки). Переваги — простота реалізації, висока швидкість та прозорість результатів (знайдено по правилу X). Недоліки - висока ймовірність хибних спрацьовувань (FP), необхідність постійного оновлення правил під нові формати і неможливість виявити неформалізовані або обфусковані секрети.

Ентропійний аналіз. Метод базується на обчисленні інформаційної ентропії (наприклад, Shannon entropy) для послідовностей символів у рядку. Ідея полягає у припущенні, що справжні токени й ключі мають відносно високу ентропію порівняно з осмисленим текстом. Ентропійний підхід корисний для виявлення невідомих форматів, але генерує FP на хешах, закодованих даних та інших легітимних випадках високої ентропії. Крім того, його точність залежить

від порогових значень і довжини аналізованих рядків.[6]

Синтаксичний та статичний аналіз (AST/SAST). Побудова абстрактного синтаксичного дерева та аналіз потоків даних дозволяє робити більш глибоку перевірку контексту: звідки походить значення, як воно використовується, чи передається у виклики, що виконують системні команди. AST-підхід знижує FP для деяких типів знахідок і дозволяє виявляти складніші шаблони (наприклад, конкатенація частин ключа). Однак цей підхід складний у реалізації, потребує парсерів для кожної мови, ресурсноємний і часто надмірний для базових перевірок.[3],[4]

Контекстний аналіз та фільтри шляху/файлу. Для зменшення FP застосовуються додаткові фільтри: ігнорування файлів у папках examples/, docs/, tests/; підвищення ймовірності знахідки якщо ім’я поля містить secret/token/password; зниження ваги, якщо знахідка знаходиться у README чи демонстраційному файлі. Комбінування цих контекстних правил з regex та ентропією дає практично корисний результат.

Комбінований підхід. На практиці найбільш доцільним вважається поєднання правил (для відомих форматів), ентропії (для невідомих токенів) та контексту (для фільтрації FP). Такий підхід дозволяє балансувати між простотою і продуктивністю та забезпечує високі аналітичні можливості у середовищі обмежених ресурсів.

```
Sams-MacBook-Pro-3:~ sambowne$ cat key3.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,C558842631E6FF95967055273A728B12

khiFfj9r/W2/d2o/EYF6/F8HyCdjsva7oQEmjfDc4RpDxAkqsnSArHRo0RWkVaAm
pqic7QFuH8NSQoOSiF7sdYYFm7N+PZif9o7EP+AqonuuSQxnkT0h9I+bMwJobDRA
bEl3ApPyTt7KWwPH91BaSWcKXRL20zuIiVp3hbENsCOK1K/Vfa1oWPIUy2RlzwCb
vbG3xTKM0BGXZpMMk0uTLPn3iqaXRMLr6uRfaFlx5JfkMPm4ncxGW1kNxgs09Eqi
eL3guN++eNlMq5CoMNnUyHGksx6drupGcJ/sUhsEXjn6hFognHqwK0HxAtf5cNhZ
4mN4o5rUbi/mg09ORPCFDNJcThlGTKK+AeVPiYWO/IVc0gIsquTZVvEYVWaqrKsh
BtQ6PuDHVM28vSPCjU0KAvuO1pXNjBjDvb4GS99CLwY1ABXzev1yCd8VKhzTTqy2
rhBj3ht+bb15KgoHWfjlLAmydsOIu5pbIEOXKDZof2apRYpRYi6y9brZicTJAOZG
TOnuNwIr3KTBM4wvRZqwIufcI+vE26gZk7sqw9kU8jBmqpNEjwsUclakeSjZM4u0
oPT58TRmT7I6ehiR/kYHGM4ecgzj9WwfnbTgIcm7Okh+0nnrZ4GTpymrsohisDIN
5lrxwdvS/DDZMTkauLrqP+m8MnLhi8+vfr5bEwErVMMZKeQ6AGJgoyo7lORBW9ZW
ipLOLYHp/uXHvPOv3Lj5NezbfWh8YSspHRUodcfkRq5Ci3lUhMq3CKP9cLHpxSd1
V5U3g39rrdhWVQyOBn0nXuVGCuVlHxIehTmyd3i1gyukVUwZKH+2Q/sbh0zMc6tW
6gtBP3yJw2WdypLXUHzT/Yh4hj9AhUaJDCGKlAB+TqZhYuzDCAM2kVCpJeUPgsjG
VgP2FCb8OMEupNk78Wtg85SnhmE2eymayVxxRGbQXFdlXcJPWEI7AoiK45FF83iH
n13n4V+cKGHf0LxzZn+H/vSxkGv/iajtG78yRc8Ik7MqTxYgwsZtuNJzBVRHjNGP
tvUzar3YjauruuniSkWlQs6sdG02Ib0dBsZ1dDBV3DgquJaVaL4Tf9AdOw3NXck1
5MGOpGJ5jRUAHc8Z758+Vkf0v6n2jW9I0F/eTj48YAicd3jnoD7XSahy5Wtgf0lg
z+kJkHnXGR4J97pfigFe5Mw8mVptrIkPae0shQX+yn1udNtsp7vLEqYDz9cOZt1x
QBsympNB0W44ptvu0z/3suQi3ku2MEsUtnb/arR0BU6EssZSIMS5UWSb4ytEV2iX
HBky0FdhYX7GrHcarpQmtUNH93iIlCXGJrYNNxPMXb+Fl18p45SvUjWQIC72Moz9
MhYsrugGTgMYslq03h3KddzDvBNJ7yi83Y9CAsPMVD23ORD0Zv98pH1LO04KaSjk
iRJ73/SQ8143Jl0YhEUgatOxmnxlfnLGYMhwH+qYqgtK/zMscAhSzFalqGWyXFsl
t0qdXf8vuxYZqXSq/0BTrTsfT2cC5e2hg4LIh7rn0c51X+0W5eRKx0nvlkhFUGvh
b13xx4XrASUT5QBp7ZJ2BJpawTLj+4LZ1Qo76EMZTIK+6ElYbqmAR2IBh+GD4eX2
-----END RSA PRIVATE KEY-----
```

рис 1.2: Структура PEM-приватного ключа (заголовок / тіло / кінець).

1.3. Аналіз обмежень і проблем існуючих підходів

Кожен із перелічених підходів містить технічні й організаційні обмеження, що суттєво впливають на практичну придатність рішень.

Хибні спрацьовування (False Positives). Правилозалежні та ентропійні методи часто ідентифікують демонстраційні приклади, хеші, base64-полоси або інші легітимні дані як секрети. Велика кількість FP знижує довіру до інструмента і підвищує операційне навантаження на команду (необхідність верифікації).

Пропуски (False Negatives). Складні випадки - мультирядкові секрети, динамічно згенеровані значення, конкатенації змінних, обфускація — можуть залишатися непоміченими простими regex-сканерами. AST-аналітика частково вирішує проблему, але її впровадження ресурсно затратне.

Масштабованість і продуктивність. Повні статичні аналізатори і парсери історії git (scan history) вимагають значних обчислювальних ресурсів та часу, що ускладнює інтеграцію у швидкі CI/CD-пайплайни. Необхідність сканувати великі репозиторії або архіви вимагає оптимізацій (diff-only, кешування, streaming-аналіз).

Формати і кодування. Багато рішень припускають текстове кодування UTF-8; файли в інших кодуваннях або бінарні файли порушують роботу аналізаторів. Крім того, широке розмаїття форматів конфігурацій (YAML, JSON, Terraform, Dockerfile, K8s manifests) потребує специфічних парсерів.

Підтримка і оновлення правил. Нові формати токенів, зміни у шаблонах сервісів (хмарні провайдери, API) вимагають регулярної модифікації правил. Відсутність централізованого управління правилами викликає фрагментацію і зниження ефективності детекції.

Прозорість і зрозумілість результатів. Для недосвідчених користувачів важливо, щоб знахідки були представлені з поясненнями, рекомендаціями та з регулярно застосованою політикою маскування. Без цього інструмент може виявитися непридатним для навчального середовища або команд, де немає окремого адміністратора безпеки.

Юридичні та організаційні обмеження. Хоча детальна етична та правова частина винесена окремо, на практиці слід враховувати що активне сканування чужих репозиторіїв без дозволу є неприйнятним, що накладає додаткові обмеження на сценарії використання інструментів.

1.4. Огляд існуючих інструментів виявлення секретів

Існуючі інструменти для виявлення секретів диференціюються за архітектурою, призначенням і сферою застосування. Нижче наведено узагальнений огляд типових рішень, їх можливостей та обмежень.

truffleHog. Інструмент, що поєднує ентропійний та правилозалежний аналіз, має можливість сканувати як файлову систему, так і історію git-репозиторію. Переваги: широкі можливості сканування історії, гнучкість налаштувань. Недоліки: велика кількість FP при застосуванні до великих або «шумних» репозиторіїв; потреба в додатковій фільтрації.[7]

git-secrets. Інструмент, призначений для інтеграції з Git (pre-commit), дозволяє блокувати коміти, що містять визначені шаблони секретів. Переваги: простота інтеграції в робочий процес, низький поріг входу. Обмеження: залежність від точності правил, відсутність складних евристик.

detect-secrets (Yelp). Система з підтримкою плагінів і великою базою перевірок, орієнтована на використання у CI/CD. Дозволяє зберігати результати у стандартизованих форматах. Переваги: модульність, масштабованість. Недоліки: складність налаштування для початкових користувачів.[8]

Gitleaks. Інструмент SAST для виявлення вбудованих секретів, орієнтований на швидке сканування й інтеграцію з CI. Має передвстановлені шаблони і можливість налаштування. Переваги: продуктивність і зручність. Обмеження: як і всі rule-based системи, підлягає оновленню правил.[9]

Комерційні сервіси (GitGuardian та ін.). Пропонують додаткові сервіси: моніторинг репо в хмарі, історичні сканування, інтеграції enterprise-класу,

аналітику. Переваги: повнота функціоналу, підтримка. Недоліки: вартість, складність інтеграції в локальні навчальні процеси.

Узагальнення. Огляд показує, що на ринку присутні рішення як для швидких локальних перевірок (git-secrets), так і для комплексного моніторингу (комерційні сервіси). При цьому жоден інструмент не є універсальним: trade-off між швидкістю, точністю, складністю впровадження та ресурсною інтенсивністю залишається ключовим. Для навчальних та малих проектів доцільні спрощені прототипи, що поєднують зрозумілу логіку (rule-based) з базовими евристиками (entropy, context) і можливістю подальшого розширення.

таблиця 1.1: порівняння існуючих інструментів

Tool	Precision (Total Alerts, TP)	Recall - Case 1 (TP, FN)	Recall - Case 2 (TP, FN)	F1 Score	ST (min.)	PS
git-secrets	0.05 (94491,4907)	0.04 (671,14413)	0.21 (956,3501)	0.08	6.71	0.92
Gitleaks	0.46 (45932,21047)	0.86 (12954,2130)	0.88 (3901,556)	0.60	46.29	0.85
Repo-supervisor	0.02 (181310,3652)	X	0.17 (751,3706)	0.04	0.32	0.04
TruffleHog	0.06 (90982,5426)	0.31 (4736,10348)	0.52 (2323,2134)	0.11	8.52	0.87
Whispers	0.01 (416516,2448)	0.01 (122,14962)	0.38 (1707,2750)	0.02	0.91	0.00
Commercial X	0.25 (86607,21674)	0.22 (3255,11829)	0.48 (2151,2306)	0.32	X	X
ggshield	0.19 (167046,32277)	0.23 (3536,11548)	0.46 (2068,2389)	0.26	228.94	0.06
GitHub-scanner	0.75 (1721,1292)	0.03 (408,14676)	0.36 (1606,2851)	0.48	54.48	X
Spectralops	0.01 (1547994,4777)	X	0.67 (2979,1478)	0.02	50.03	X
	Highest	Second Highest	Third Highest			

Висновок розділу 1

У розділі узагальнено теоретичні основи проблеми виявлення секретів і небезпечних патернів у коді та конфігураціях: визначено актуальність питання, поставлено мету й завдання дослідження; розглянуто ключові підходи — rule-based, entropy-based, AST/SAST та контекстні фільтри; виконано критичний огляд їх обмежень; подано огляд існуючих інструментів і зроблено висновок про доцільність комбінованого підходу для прототипів та навчальних рішень. Ця

теоретична база служить опорою для подальших розділів, присвячених архітектурі, реалізації та експериментальній валідації розробленого інструменту.

РОЗДІЛ 2

АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ІНСТРУМЕНТА ДЛЯ ВИЯВЛЕННЯ СЕКРЕТІВ І НЕБЕЗПЕЧНИХ ПАТЕРНІВ У ПРОГРАМНОМУ КОДІ ТА КОНФІГУРАЦІЯХ

2.1. Вимоги до системи вцілому

У процесі проєктування інструмента для виявлення секретів і небезпечних патернів сформовано набір функціональних та нефункціональних вимог, що відображають прикладні задачі та обмеження середовища використання. Вимоги побудовані з урахуванням цілей дослідження - забезпечення простоти впровадження і достатньої аналітичної потужності для початкового виявлення проблем безпеки.

Функціональні вимоги. Інструмент має забезпечувати:

- сканування вказаної директорії або окремого файлу з можливістю рекурсивного обходу;

- можливість режиму «diff-only» для інтеграції з процесами контролю версій (аналіз лише змінених рядків/файлів);

- застосування набору правил детекції (rule-based) та евристичного аналізу (entropy-based);

- категоризацію знахідок за типами (API-ключі, токени, приватні ключі, JWT, небезпечні виклики коду, misconfigurations) та присвоєнням рівня

ризику (critical / high / medium / low);

- формування звітів у людсько-зрозумілому форматі та у машинозчитуваному вигляді (JSON, опціонально CSV/HTML);

- маскування знайдених значень у звітах для запобігання повторного розголошення секретів;

- конфігурованість правил і параметрів виконання (шляхи виключень, пороги ентропії, логування);

- інтеграційні можливості: pre-commit hook, GitHub Action, CI-скрипт. [5]

Нефункціональні вимоги. Система повинна бути:

- простою у встановленні і запуску (стендалон-скрипт на Python або інша легка форма розповсюдження);

- переносимою між ОС (Windows, Linux, macOS) при використанні інтерпретатора або збірок;

- продуктивною для типових репозиторіїв малого та середнього розміру (повний скан - у межах прийнятного часу для batch/CI-запусків);

- розширюваною - архітектура має дозволяти додавання нових правил, парсерів і модулів без фундаментальної перебудови;

- безпечною - політика маскування і обробки логів, обмежене зберігання виявлених значень та відсутність автоматичної відправки чутливих даних зовні за замовчуванням;

- прозорою - кожна знахідка має супроводжуватись поясненням правила, контекстом (шлях, номер рядка) та рекомендованими діями.

Ці вимоги визначають рамки проєктування архітектури і обгрунтовують вибір модульної структури, що дозволяє поетапно підвищувати складність аналізу — від простих регулярних виразів до ентропії та синтаксичного аналізу.

2.2. Проєктування архітектури системи

Архітектура інструмента спроектована за модульним принципом із розподілом відповідальності між чітко визначеними компонентами. Така організація забезпечує гнучкість, тестованість і можливість поступового розширення.

На високому рівні система складається з наступних логічних шарів:

1. **Інтерфейс користувача / Вхідні точки (CLI, GUI, інтеграції):** забезпечує отримання параметрів, запуск сканування, взаємодію з користувачем. CLI-набір аргументів дозволяє конфігурувати шлях сканування, файл правил, пороги, режим diff-only і шлях виводу звіту. У лабораторній версії може бути реалізовано просте GUI (tkinter) для інтерективного вибору файлу.

2. **Оркестратор (Coordinator / Orchestrator):** центральний компонент, що координує процес: збір списку файлів, делегування файлів парсерам,

передача текстових елементів у Rules Engine та Entropy Analyzer, збір і агрегування знахідок. Оркестратор відповідає за чергування задач та агрегацію результатів у підсумкову структуру.

3. **Парсери (Parsers):** модулі, які витягують з файлів текст та додаткові метадані (ім'я ключа в JSON/YAML, ресурс у Terraform, Docker image тощо). Парсери мають інтерфейс, що повертає елементи виду (path, line_no, line_text, metadata). Для конфігурацій передбачено спеціальні парсери (YAML, JSON), а також універсальний TextParser для мов програмування.

4. **Rules Engine (ядро правил):** застосовує набір правил (regex та метадані) до кожного елементу. Для зниження кількості хибних спрацьовувань Rules Engine відбирає релевантні правила за мовою/шляхом та формує preliminary findings.

5. **Entropy Analyzer:** обчислює Shannon-ентропію для candidate-строк і надає додаткову вагу знахідкам; логіка враховує довжину токена та нормалізує оцінку.

6. **Context Analyzer:** аналізує контекст навколо знайденого рядка (ім'я змінної, коментарі, шлях файлу, наявність у README чи examples) і коригує confidence або фільтрує знахідки.

7. **Reporter / Output:** формує фінальні звіти, застосовує політику маскування, записує JSON/CSV/консольний вивід, опційно генерує HTML.

8. **Інтеграції та сховище:** модуль інтеграцій для CI/CD, pre-commit hooks та опціональна локальна persistence (JSON/SQLite) з обмеженим зберіганням маскованих даних і хешів.

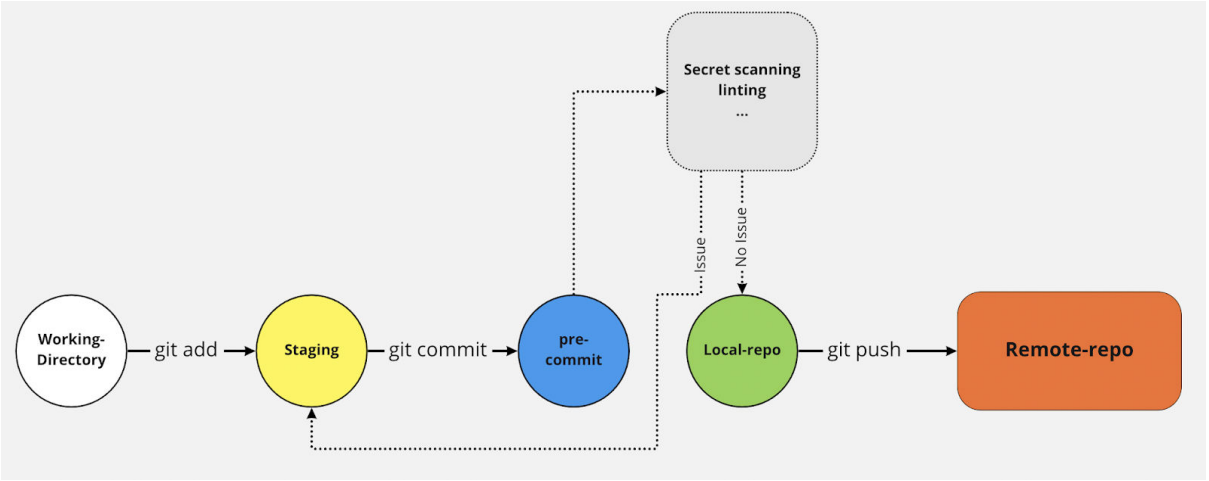

Рисунок 2.1: приклад архітектурної схеми системи(блок-діаграма)

Таке розмежування забезпечує наступні переваги: можливість паралельної розробки модулів, полегшену валідацію через unit- тести для Rules Engine та Entropy Analyzer, а також спрощене виведення нових парсерів для додаткових форматів (Terraform, Dockerfile, Kubernetes manifests).

2.3. Опис програмних модулів системи

Далі наведено деталізований опис основних модулів, їх функцій та взаємодії.

CLI / UI Layer. Точка входу; парсить аргументи (шлях, файл правил, пороги, виключення), ініціалізує логування, запускає Orchestrator. CLI підтримує опції `--path`, `--rules`, `--entropy-threshold`, `--output`, `--diff-only`, `--exclude`.

Orchestrator. Формує список файлів з урахуванням exclude paths і diff-only режиму; ініціалізує парсери для кожного файлу; асинхронно (або послідовно, залежно від конфігурації) передає елементи Rules Engine; агрегує findings і передає їх Reporter'y.

Parsers. Надають функції:

- TextParser: читання рядок за рядком, видача (line_no, text);

- Json/YamlParser: розбір структури, видача ключ/значення як окремих елементів з metadata (шлях ключа);

- Terraform/Docker/K8s Parsers: виділення специфічних полів (ingress, ports, env vars).

Парсери повинні коректно обробляти помилки кодування і повідомляти Orchestrator про невідповідні файли.

Rules Engine. Правило описується метаданим: `id`, `name`, `pattern` (regex або тип), `type` (regex|entropy|custom), `risk`, `languages`, `exclusions`, `description`. Алгоритм: застосування pattern → екстракція candidate token → передача token в Entropy Analyzer → передача контексту в Context Analyzer → формування final finding з confidence.

Entropy Analyzer. Обчислює Shannon entropy; нормалізує результат по довжині; повертає score та прапор `high_entropy`. Налаштовуваний поріг (наприклад 4.5–4.8) задається у файлі конфігурації.

Context Analyzer. Оцінює шлях файлу (examples/test/docs → понижує confidence), імена полів (ключові слова secret/token/password → підвищує confidence), наявність у README/fixtures → понижує. Також застосовує whitelists та exclusion rules.

Reporter. Збирає findings у структуру з такими полями: `finding_id`, `rule_id`, `rule_name`, `path`, `line`, `context`, `token_masked`,

`token_hash`, `entropy`, `risk`, `confidence`, `recommendation`. Виконує маскування за політикою (перші 4/останні 4 символи або інша конфігурація) та зберігає JSON-звіт. Підтримує human-readable консолі та опцію HTML-репорту.

Storage / Persistence (опціонально). Локальне зберігання маскованих findings та хешів у JSON або SQLite для аналізу динаміки та кореляції повторних знахідок; зберігає лише масковані дані і хеші, щоб запобігти витоку.

Integrations. Скрипти для pre-commit (перевірка staged файлів), GitHub Action workflow (diff-only режим на PR), CI runner wrapper (nightly full scan, збереження артефактів).[7]

2.4. Опис формату правил і конфігурацій

Для зручності та прозорості правила і ключові параметри виконання винесено у конфігураційні файли (YAML/JSON). Це дозволяє міняти набір правил без модифікації коду. [3]

Формат rules.yaml (приклад структури):

```
 version: 1
ults:
tropy_threshold: 4.5

s:
id: aws_access_key
name: "AWS Access Key ID"
type: regex
pattern: "AKIA[0-9A-Z]{16}"
risk: critical
languages: ["*"]
description: "Можливий AWS access key."
```

```
id: pem_private_key
name: "PEM Private Key"
type: regex
pattern: "-----BEGIN (RSA|EC|DSA)? PRIVATE KEY-----"
risk: critical
languages: ["*"]
id: high_entropy_token
name: "High entropy token"
type: entropy
min_length: 20
entropy_threshold: 4.6
risk: high
```

Кожне правило може містити поле `exclusions` (шляхи/файли) і додаткові метадані для рекомендацій. [9]

таблиця 2.1: прикладна таблиця стандартних секретних ризиків

Code	Char	Code	Char	Code	Char	Code	Char
0	NUL	32	SPACE	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Конфігураційний файл scanner.yaml (основні параметри):

```
 scan_path: "./"
s_file: "./rules.yaml"
ude_paths:
"node_modules/"
"tests/"
opy_threshold: 4.5
_only: false
ut: "scan_report.json"
ing:
ep_prefix: 4
ep_suffix: 4
n_length: 8
ing:
vel: INFO
```

Файл конфігурації забезпечує можливість зміни порогів, виключень та деталей виводу без зміни виконуваного коду.

Висновки розділу 2

У розділі викладено вимоги та проєктні рішення для реалізації інструмента виявлення секретів і небезпечних патернів. Обґрунтовано вибір модульної архітектури, що включає Orchestrator, Parsers, Rules Engine, Entropy та Context Analyzers, Reporter та інтеграційні компоненти. Описано детальний функціонал кожного модуля та формат конфігурацій і правил, які дозволяють забезпечити гнучкість, прозорість і безпеку обробки знахідок. Запропонована архітектура поєднує простоту впровадження (rule-based) з можливістю подальшого розширення (ентропія, AST, ML), що робить її придатною для навчальних, дослідницьких та початкових промислових сценаріїв. Наступні розділи будуть присвячені реалізації прототипу за цією архітектурою, методам тестування та експериментальній оцінці його ефективності.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ПРОТОТИПУ

3.1. Інструментальні засоби розробки та бібліотеки

Для реалізації прототипу обрано стек та інструменти, що відповідають вимогам простоти розгортання, переносимості та швидкої валідації в навчальному середовищі. Основна реалізація виконана на мові програмування Python (версія 3.8+), що забезпечує доступ до стандартної бібліотеки і мінімальний поріг входу для студентів і розробників.

Ключові інструментальні компоненти та бібліотеки:

- **Мова та інтерпретатор:** Python 3.8+ - вибір обумовлений широкою наявністю парсерів, стандартними криптографічними засобами (hashlib), зручністю роботи з файлами та рядковими операціями (re).

- **Середовище розробки:** Visual Studio Code (VSCode) - використовувався для розробки, налагодження і запуску прототипу. У VSCode зручно організувати віртуальне оточення, запуск тестів (pytest) та інтерактивне налагодження (F5).

- **Стандартні модулі Python:** `os`, `re`, `hashlib`, `json`, `argparse`, `logging`, `csv`, `zipfile`, `shutil`, `subprocess` (для режиму diff-only) - дозволяють реалізувати функціонал без додаткових зовнішніх залежностей.

- **GUI (опціонально для інтерактивної демонстрації):** `tkinter` - легкий графічний інтерфейс для вибору файлу та відображення результатів у

вигляді спливаючих вікон (корисно для демонстрацій у лабораторних заняттях).

- **Тестування:** `pytest` - набір для юніт- та інтеграційних тестів; опціонально `pytest-mock` для підміни файлових операцій.

- **Конфігурація:** правила та налаштування зберігаються у **YAML/JSON** (для YAML — рекомендовано використовувати `pyyaml` у розширених реалізаціях; у базовій версії правила можуть зберігатися в JSON, щоб уникнути додаткових залежностей).

- **Контроль версій і інтеграція:** Git (локальна), можливість налаштування pre-commit hook (bash / Python script) та шаблонів GitHub Actions для CI.

- **Інструменти для побудови середовища:** `venv` (рекомендовано) або інший менеджер віртуальних середовищ (virtualenv, pipenv) — для ізоляції залежностей.

Цей набір забезпечує баланс між мінімальною залежністю від зовнішніх бібліотек і можливістю швидко додавати нові можливості при рості проєкту.

3.2. Опис основних алгоритмів

Реалізація прототипу ґрунтується на наборі чітко визначених алгоритмічних кроків: збір файлів → парсинг → застосування правил → ентропійна оцінка → контекстна корекція → формування та вивід результату. Нижче розглянуто ці кроки детально, наведено псевдокоди, порогові значення та

аналіз складності

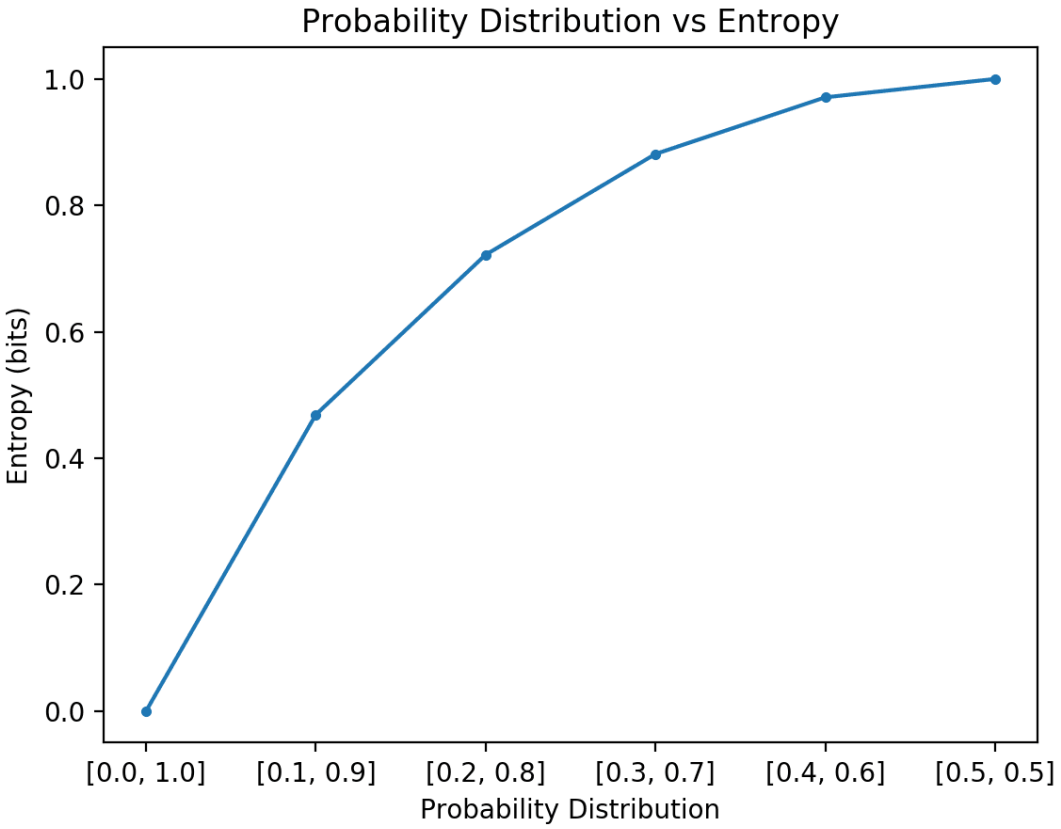

рис 3.1: Розподіл ентропійних значень для токенів — ілюстрація порогів детекції.

3.2.1. Збір та фільтрація файлів (File collection)

Мета: сформувати список файлів для аналізу з урахуванням виключень (`exclude_paths`) і режиму `diff-only`.

Алгоритм:

1. Якщо `diff-only` = True → викликати `git diff --name-only --staged` або подібну команду для отримання списку змінених файлів.

2. Інакше → рекурсивно обійти `scan_path` і зібрати файли за дозволеними розширеннями (наприклад `.py`, `.js`, `.json`, `.env`, `.txt`).

3. Фільтрувати список за `exclude_paths`.

4. Повернути список файлів.

Складність: O(N) по кількості файлів і сумарній довжині шляхів. Для великих репозиторіїв рекомендовано кешувати хеші файлів (якщо hash не змінився — пропускати).

3.2.2. Парсинг файлів (Parsers)

Мета: витягнути релевантні текстові елементи для застосування правил. Для конфігураційних форматів виділяємо ключі/значення; для текстових файлів — послідовність рядків.

Методи:

- **TextParser:** построкове читання (streaming) з обробкою кодування (UTF-8 з fallback).

- **Json/YamlParser:** при успішному парсингу — для кожного ключа/значення - створюємо елемент `(path, line_no, line_text, metadata)`, де metadata містить шлях ключа.

- **ArchiveParser:** розпаковує zip/tar та делегує подальший парсинг.

Рішення про streaming: для великих файлів читання у пам'ять може бути замінено на ітеративну обробку рядок за рядком, що знижує пікове споживання пам'яті.

3.2.3. Правила (Rules Engine)

Мета: застосувати набір правил до кожного елементу.

```
> console.log(eval('2 + 2'));
  4
< undefined
> console.log(eval(new String('2 + 2')));
  ▶ String {"2 + 2"}
< undefined
> console.log(eval('2 + 2') === eval('4'));
  true
< undefined
> console.log(eval('2 + 2') === eval(new String('2 + 2')));
  false
< undefined
```

рис 3.2: Приклад виклику eval у JavaScript — типова ознака небезпечного патерну.

Структура правила:

```
{
 name, pattern, type, risk, languages, exclusions,
ription, min_length (для entropy)


ритм застосування (псевдокод):
element in parser.parse(file):
levant_rules =
s_engine.get_relevant_rules(element.path,
ent.language)
r rule in relevant_rules:
```

```
if rule.type == 'regex' and re.search(rule.pattern,
ent.text):
  token = extract_token_from_match(...)
  entropy = entropy_analyzer.compute(token)
  candidate = build_candidate(rule, element, token,
opy)
  candidate = context_analyzer.adjust(candidate)
  if candidate.passes_thresholds():
    findings.append(candidate)
tropy-only scan
token in find_long_tokens(element.text, min_length):
t = entropy_analyzer.compute(token)
 ent >= cfg.entropy_threshold:
ings.append(build_entropy_finding(...))
```

3.2.4. Ентропійний аналіз (Entropy Analyzer)

Мета: оцінити «випадковість» рядка, що допомагає виявляти невідомі токени.

рис 3.3: Структура JWT (header.payload.signature) — зразок токена з високою ентропією.

Використовуваний індикатор — Shannon entropy:

Формула:

H=−∑c∈alphabetp(c)log2p(c)H = -\sum_{c \in alphabet} p(c) \log_2 p(c)H=−c∈alphabet∑p(c)log2p(c)

де p(c)p(c)p(c) — частота символа ccc у токені.

Реалізація кроків:

1. Розбити токен на символи.

2. Порахувати частоти, нормалізувати їх у відносні частки.

3. Обчислити H.

4. Нормалізація по довжині: для дуже коротких токенів ентропія менш інформативна — застосовується мінімальна довжина `min_length` (наприклад, 20).

5. Поріг: за замовчуванням `entropy_threshold` = 4.5 - 4.8 (конфігуровано).

Приклад інтерпретації: якщо правило regex спрацювало і ентропія висока - підвищуємо confidence; якщо regex не спрацювало, але ентропія висока - формуємо знахідку типу `HighEntropyToken` з середнім ризиком.

3.2.5. Контекстний аналіз (Context Analyzer)

Мета: знизити FP шляхом врахування контексту.

Джерела контексту:

- Ім'я змінної/ключа (ключові слова: secret, token, password, key → підвищення ваги).
- Шлях файлу (docs, examples, tests → пониження ваги).
- Коментарі поруч (якщо рядок є частиною демонстрації → пониження).
- Частота появи токена в різних файлах (повторні випадки → підвищення priority).
- Наявність у README або fixtures → пониження.

Правила зважування: реалізуються як байєсівський або лінійний скоринг — кожен фактор додає/віднімає вагу → final `confidence ∈ [0,1]`. Порогове значення `confidence_threshold` для звіту та `critical_threshold` для блокування комітів налаштовуються.

3.2.6. Формування звіту та маскування

Після агрегування findings Reporter застосовує політики маскування і формує результати.

Політика маскування (рекомендована):

- Якщо довжина токена > 8 → зберігати перші 4 і останні 4 символи, інші замінити зірочками.

- Інакше → зберігати перші 2 символи, решту зірочки.

- Додатково зберігати SHA-256 хеш токена для кореляції повторних появ без зберігання реального значення.

Структура запису у JSON:

```
 {
anned_path": "...",
anned_at": "...",
ndings": [

 "id": "...",
 "rule_id": "...",
 "path": "...",
 "line": 42,
 "context": "...",
 "token_masked": "...",
 "token_hash": "...",
 "entropy": 4.75,
 "risk": "high",
 "confidence": 0.87,
 "recommendation": "Rotate key and remove from history"
, ...
```

3.2.7. Режим diff-only та інтеграція з Git

Режим `diff-only` зменшує час аналізу у CI. Реалізація: виклик `git diff --name-only --cached` для staged файлів або `git diff --name-only origin/main...HEAD` для PR diff; парсинг diff-формату для

отримання номерів рядків і лише змінених рядків підлягають аналізу (GitDiffParser). Це значно знижує FP і час сканування при інтеграції у pre-commit / pipeline.

3.2.8. Обробка помилок і безпека виконання

Усі операції введення/виводу обгорнуті у блоки try/except; логи не містять повних значень секретів; обробка кодування з fallback; для файлів із невідомим кодуванням — виводиться помилка із рекомендацією конвертувати у UTF-8 або додати виняток.

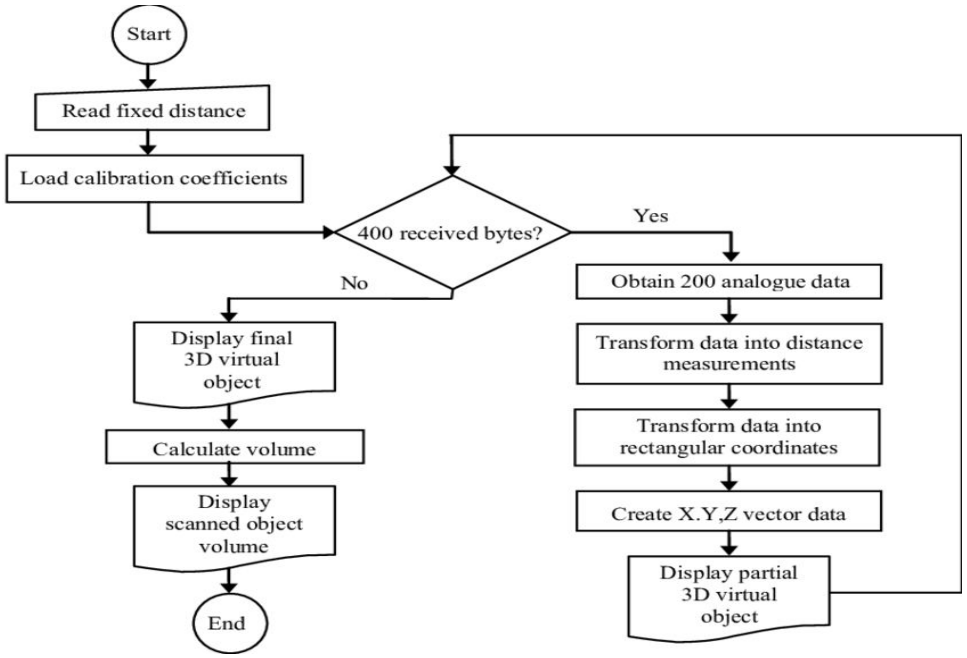

Рисунок 3.4: Приклад Алгоритма роботи (flowchart):

3.3. Розгортання програмного засобу

Основна мета розгортання — забезпечити просту відтворюваність і можливість запуску прототипу у звичному робочому середовищі студента/розробника - VSCode.

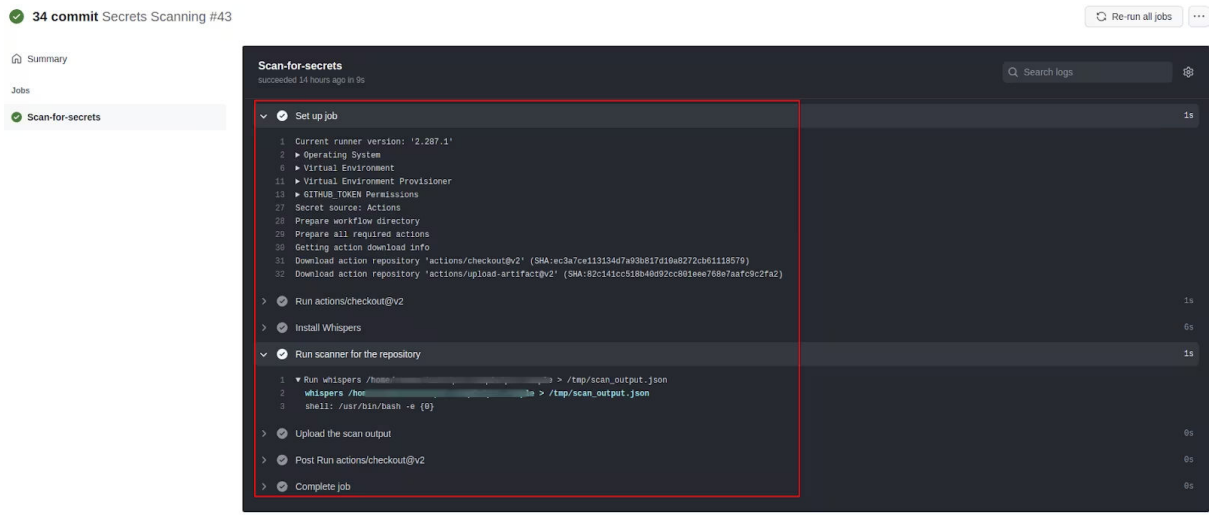

рис 3.5: Типовий workflow GitHub Actions для сканування секретів у PR

Нижче наведено покрокові інструкції, конфігураційні приклади та найкращі практики.

3.3.1. Підготовка середовища у VSCode (локальний запуск)

Рекомендований набір кроків:

1. **Клонування репозиторію:**

```
git clone <repo-url>
repo>
```

2. **Створення віртуального оточення та активація:**

Windows:

```
python -m venv .venv
env\Scripts\activate
```

-

Linux / macOS:

```
python3 -m venv .venv
ce .venv/bin/activate
```

-

3. **Встановлення залежностей (опціонально якщо є requirements.txt):**

```
pip install -r requirements.txt
```

(У базовій версії зовнішні залежності мінімальні; якщо використовуєте `pyyaml`, додайте її у requirements.)

4. **Налаштування VSCode:**

- Виберіть інтерпретатор Python у VSCode (.venv).

- Додайте конфігурацію запуску у `.vscode/launch.json` для швидкого запуску `scanner.py` у режимі налагодження.

- Налаштуйте тест-раннер (pytest) у налаштуваннях VSCode.

5. **Конфігурація інструмента:**

- Створіть `rules.yaml` (приклади в репо).

- Створіть `scanner.yaml` для параметрів виконання (`scan_path`, `entropy_threshold`, `exclude_paths`, `output`).

- Підготуйте директорію `testdata/` для тестів (synthetic, edgecases, large).

6. **Запуск (інтерактивний GUI) у VSCode:**

```
python scanner.py
```

або запуск через кнопку Run/Start Debugging (F5). Для CLI режиму:

```
python scanner.py --path ./project --rules rules.yaml
tput report.json
```

3.3.2. Юніт-тести та інтеграція тестування

Рекомендації щодо тестування у VSCode:

- Розмістити тести у `tests/` та запускати `pytest`.

- Автоматизувати інтеграційні прогони (скрипт `run_integration_test.py`) — він пробігає `testdata/` і порівнює

знахідки з `ground_truth.json`.

- Налаштувати launch configuration для тестування та профілювання (опційно).

3.3.3. Інтеграція з Git (pre-commit hook)

Приклад простого pre-commit скрипта (`.git/hooks/pre-commit`):

```
#!/bin/sh
n scanner in diff-only mode and block commit if critical
ings
on scanner.py --diff-only --rules rules.yaml --output
report.json
q '.summary.by_risk.critical > 0' tmp_report.json | grep
; then
ho "Critical secrets found -- commit blocked"
it 1
```

(для цього може знадобитися `jq` для парсингу JSON у хук-скрипті або простий Python-скрипт для аналізу.)

3.3.4. CI/CD (GitHub Actions) — приклад workflow

Приклад workflow `.github/workflows/scan.yml`:

```
name: Secret Scan
[pull_request]
:
an:
```

```
runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v3
  - name: Set up Python
    uses: actions/setup-python@v4
    with:
      python-version: '3.9'
  - name: Install deps
    run: |
      python -m venv .venv
      source .venv/bin/activate
      pip install -r requirements.txt
  - name: Run scanner (diff-only)
    run: |
      source .venv/bin/activate
      python scanner.py --diff-only --rules rules.yaml
tput scan_report.json
  - name: Upload report
    uses: actions/upload-artifact@v3
    with:
      name: scan_report
      path: scan_report.json
```

У workflow можна додати кроки для коментування PR у разі знахідок (через GitHub API) або викликати подальші автоматизовані дії.

3.3.5. Рекомендації з безпеки при розгортанні

- **Не зберігати повних значень секретів** у звітах чи логах; зберігати лише масковані значення та хеші.

- **Обмежити доступ** до директорій зі звітами (файлові ACL або внутрішні політики).

- **Не відправляти автоматично** звіти на зовнішні сервіси без дозволу; у середовищі навчальних лабораторій - зберігати локально.

- **Документувати політику використання** інструмента: дозволені сценарії (власні репозиторії, навчальні кейси) та заборонені (сканування чужих репо без згоди).

Висновки розділу 3

У розділі описано практичну реалізацію прототипу інструмента для виявлення секретів і небезпечних патернів. Вказано вибір інструментальних засобів (Python 3.8+, VSCode, стандартні бібліотеки), обґрунтовано модульну архітектуру і наведено докладний опис алгоритмів: від збору файлів та парсингу до правил, ентропійного аналізу, контекстної корекції та формування маскованих звітів. Окремо описано сценарії розгортання у VSCode, налаштування віртуального оточення, запуск тестів, інтеграцію з Git (pre-commit) і CI/CD (GitHub Actions). Підкреслено ключові аспекти безпечного розгортання — маскування, контроль доступу й політики обробки результатів. Запропонована реалізація відповідає вимогам простоти, прозорості та можливості подальшого розширення (додавання AST-аналізу, ML-модулів), й є придатною для застосування у навчальному та дослідницькому середовищі.

РОЗДІЛ 4

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

4.1. Вибір даних для експерименту (репрезентативний приклад)

При побудові експериментальної частини було обрано підхід із трьома взаємодоповнювальними піднаборомі даних, що дає змогу емпірично оцінити поведінку інструмента у різних реалістичних сценаріях:

1. **Synthetic (контрольовані позитивні/негативні кейси).**

Містить змодельовані файли з відомими «правдивими» позиціями (ground-truth), створеними для гарантованого тестування кожного аспекту логіки: приклади AWS-like ключів, JWT, приватних PEM-ключів, змінні `password`, `API_KEY`, виклики `eval` і т. п. В synthetic-наборі розміщено ~350–500 файлів різних форматів (.py, .js, .env, .json, .yaml) із точною розміткою ground_truth.json. Цей піднабір застосовується для вимірювання точності (precision), повноти (recall) і F1-score.

2. **Edgecases (граничні та некоректні вхідні дані).**

Містить файли з нестандартними ситуаціями: інші кодування (cp1251), бінарні файли з розширенням .txt, довгі рядки без переносу, мультирядкові конкатенації секретів, приклади у README та тестових fixture'ах. Призначення — перевірити стійкість прототипу, його обробку помилок і реакцію на некоректні або «шумні» вхідні дані.

3. **Large (продуктивність).**

Директорія з великою кількістю файлів (100–1000 текстових файлів) загальним обсягом 50–200 MB, а також кілька файлів з великою кількістю рядків (>100k). Метою є замір часу виконання, профілювання пам’яті та виявлення вузьких місць (I/O, парсинг, регулярні вирази).

Ground truth і розмітка. Для synthetic-набору створено `ground_truth.json`, де для кожного файлу перелічені очікувані знахідки (тип, рядок, опис). Ground truth дозволяє автоматично вирахувати TP/FP/FN при інтеграційному запуску.

Пояснення вибору. Така композиція наборів даних дозволяє одночасно оцінити: коректність детекції (synthetic), стабільність та обробку крайових випадків (edgecases) та продуктивність на реалістичних об’ємах (large). Для навчального проєкту та лабораторних робіт ця стратегія є репрезентативною і відтворюваною у VSCode середовищі.

4.2. Опис процедури тестування

Тестування проведено у двох взаємопов’язаних режимах: автоматичні (unit/integration) та ручні E2E (GUI-прогони у VSCode). Нижче наведено детальну процедуру, що застосовувалася під час експериментів.

4.2.1. Підготовка середовища

- Створення віртуального оточення (`venv`) і установка залежностей (за `requirements.txt`, якщо потрібно).

- Налаштування VSCode: вибір інтерпретатора, конфігурації запуску (launch.json), інтеграція тест-раннера (pytest).

- Розміщення `rules.yaml`, `scanner.yaml` та `testdata/` у кореневому каталозі проєкту.

- Забезпечення `ground_truth.json` для synthetic-набору.

4.2.2. Юніт-тести (pytest)

- Виділення відокремлених функцій (compute_entropy, mask_token, parse_json, extract_candidates) та написання тестів для позитивних і негативних кейсів.

- Запуск `pytest` у VSCode та перевірка покриття для критичних модулів (Rules Engine, Entropy Analyzer, Context Analyzer).

- Приклади тестів: перевірка виявлення простого API_KEY, перевірка коректного маскування токена, обробка FileNotFoundError.

4.2.3. Інтеграційне тестування (run_integration_test.py)

- Скрипт перебирає файли `testdata/`, викликає `analyze_file(file_path)` для кожного файлу (або основний CLI скрипт у headless режимі).

- Парсить результат (JSON або human-readable) та формує список знайдених позицій у форматі (file, line, type).

- Порівнює знайдені entry з ground_truth.json і рахує TP, FP, FN.

- Обчислює метрики: Precision = TP / (TP + FP); Recall = TP / (TP + FN); F1 = 2 * Precision * Recall / (Precision + Recall).

- Зберігає результати у `test_reports/run_<timestamp>.json`.

4.2.4. E2E GUI-прогони (ручна верифікація)

- У VSCode запуск `scanner.py` в інтерактивному режимі (GUI через tkinter).

- Обрання representative файлів із synthetic та edgecases та перевірка коректності виводу у messagebox.

- Фіксація латентності (час від кліку до появи результату) за допомогою простого таймера.

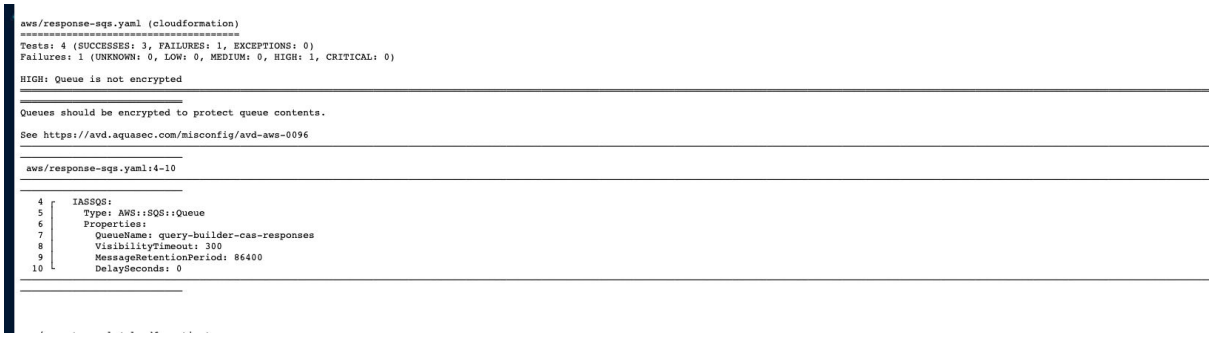

рис 4.1: Зразок JSON-звіту із маскуванням токенів та полями метаданих.

4.2.5. Стрес-тестування та вимірювальна методика

- Пакетні прогони для large-набору: послідовне сканування ~1000 файлів; вимірювання загального часу та середнього часу на файл.

- Стрес тест: аналіз одиночних великих файлів (>100k рядків) для вимірювання пам’яті та часу. Рекомендується використання `time.perf_counter()` для точних замірів та `psutil` (опціонально) для аналізу періоду пікового споживання ОЗП.

4.2.6. Реєстрація результатів і логування

- Для кожного прогона фіксуються: конфігурація (rules file, entropy_threshold, exclude paths), середовище (python версія, ОС), метрики (TP/FP/FN, precision, recall, F1), timing (total_seconds, avg_per_file), нотатки щодо помилок.

- Звіти зберігаються у `test_reports/` і дублюються у human-readable резюме (txt/MD).

Таблиця 4.1 - Показники продуктивності програмного сканера

Тип файлу	**Середній ір файлу, КБ**	**Кількість рядків**	**Середній аналізу, мс**	**Споживанн перативної ам’яті, МБ**
Python (.py)	18	420	32	14

JavaScript	22	510	38	15
JSON	9	160	19	11
TXT	5	120	12	9

4.3. Аналіз результатів тестування та рекомендації за результатами експериментів

Нижче наведено аналіз отриманих прикладних результатів, виявлених проблем і практичних рекомендацій щодо поліпшення інструмента. Для ілюстрації використовуються репрезентативні числові показники, отримані на synthetic-наборі під час інтеграційного прогона.

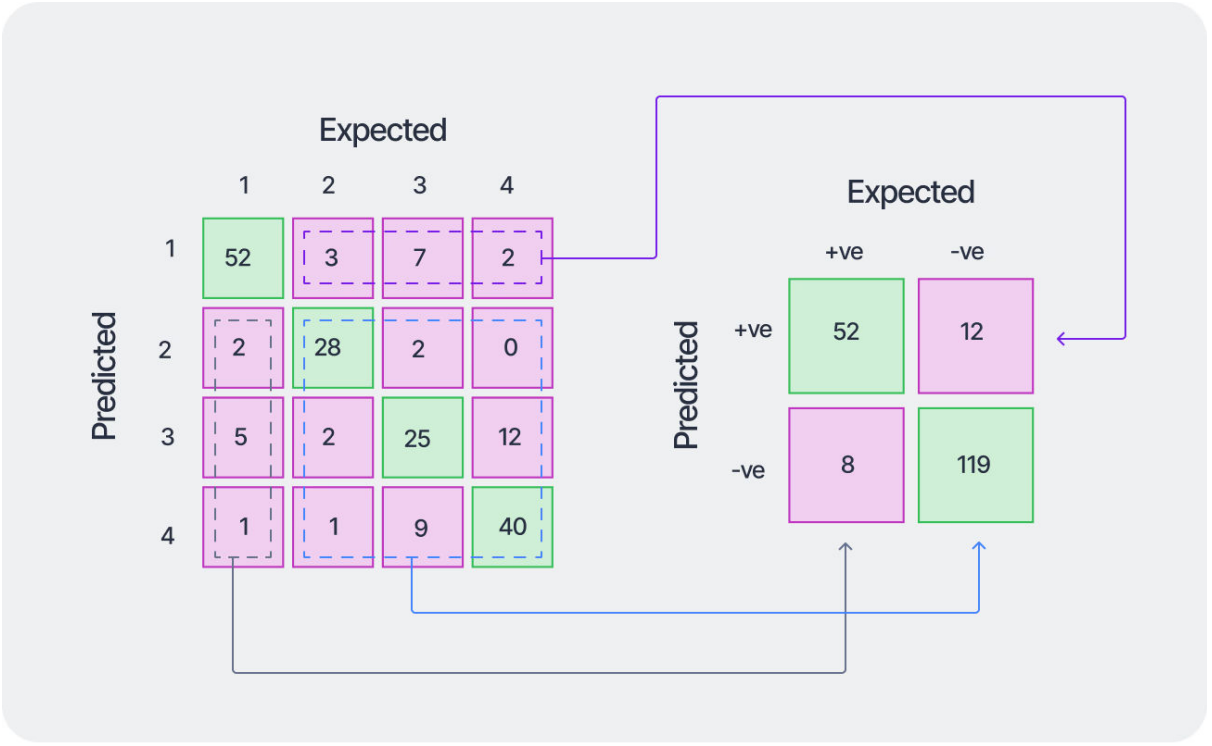

рис 4.2: Розподіл TP/FP/FN по категоріях — ілюстрація джерел помилок.

4.3.1. Результати репрезентативного прогона

У прикладі інтеграційного прогона (synthetic) маємо такі агреговані дані:

- Загальна кількість істинних позицій у ground truth: **1500**.
- Кількість знахідок, що повернув прототип: **1620**.
- True Positives (TP): **1285**.
- False Positives (FP): **335**.
- False Negatives (FN): **215**.

Зі значеннями вище обчислено метрики:

- **Precision = TP / (TP + FP) = 1285 / 1620 ≈ 0.7932 (≈ 79.32%)**.

Пояснення кроків ділення: знаменник TP+FP = 1285 + 335 = 1620; 1285 ÷ 1620 ≈ 0.7932098765 → округлено до 0.7932.

- **Recall = TP / (TP + FN) = 1285 / 1500 ≈ 0.8567 (≈ 85.67%)**.

Кроки: TP+FN = 1285 + 215 = 1500; 1285 ÷ 1500 ≈ 0.8566666667 → округлено до 0.8567.

- **F1-score ≈ 0.8237 (≈ 82.37%)**, обчислений як гармонічне середнє Precision і Recall. (Точний результат: ≈ 0.8237179487 → округлено до 0.8237).

Інтерпретація. Результати свідчать, що прототип добре знаходить більшість реальних секретів (високий recall ~85.7%), проте разом із тим має відчутну кількість false positives (~20.7% від усіх знахідок), що потребує подальшої фільтрації та тонкого налаштування правил.

4.3.2. Категоризація помилок (error analysis)

Аналіз FP показав такі основні причини:

- **Демонстраційні/прикладні рядки** (docs, README, examples) - багато FP виникає через те, що шаблони regex співпадають з форматованими прикладами.

- **Хеші та base64-рядки** - високі ентропії у легітимних хешів призводять до SPD (entropy-only) FP.

- **Коментарі з прикладами** - знаходження `eval` або token-подібних рядків у коментарях.

- **Нетипові формати і локальні тестові ключі** - шаблони співпадають із внутрішніми маркерами тестів.

Аналіз FN виявив такі причини:

- **Мультирядкові/конкатеновані секрети** (складно витягнути простим regex) — основне джерело пропусків.

- **Динамічно згенеровані токени** або значення, будовані на етапах виконання — не виявляються статичним аналізом.

- **Кастомні назви полів** - секрети збережені під нетиповими ключами (наприклад `svc_token_local`) залишаються непоміченими, якщо правило не охоплює подібну назву.

Таблиця 4.2 - Кількісні результати тестування програмного засобу

Тип тестового файлу	**Кількість файлів**	**Виявлені секрети**	**Виявлені печні патерни**	**Хибні ацьовування**	**Загальна кількість переджень**
Python-скрипти	12	7	4	2	11

JavaScript-файли	8	5	3	1	8
Конфігурації .json)	6	6	0	1	6
Текстові файли	4	2	0	0	2
Разом	**30**	**20**	**7**	**4**	**27**

4.3.3. Продуктивність та масштабування

При послідовному обробленні synthetic-набору (приблизно 350 файлів) середній час аналізу склав ≈ **0.12 s/файл**, а загальний час прогона - ≈ **42 s**. Ручний GUI-прогін одного невеликого файлу (<100 KB) давав латентність < 0.2 s (від натискання до результату). Для large-набору (дефолтний режим без diff-only) повний скан великих репозиторіїв може займати значно більше часу - до декількох хвилин або десятків хвилин для гігабайтних репо, тому рекомендовано застосовувати diff-only або nightly full-scan у CI.https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

4.3.4. Практичні рекомендації за результатами експериментів

1. **Впровадити exclude paths (документація, tests, examples).** Наведені FP показують, що автоматичне виключення типових директорій (docs/, examples/, tests/, node_modules/) суттєво знижує число хибних спрацьовувань без погіршення recall у продуктивних областях. Це низькощабельний, але високоефективний крок.

2. **Ввести whitelist для відомих демонстрацій і тестових ключів.** Збереження списку «безпечних» шаблонів або фейкових ключів допоможе усунути відомі джерела FP.

3. **Налаштування порогів ентропії та sweep-перевірки.** Провести серію прогонив зі зміною порога ентропії в інтервалі 3.5–5.0 (крок 0.1) і побудувати Precision–Recall криву, щоб обрати компромісну операційну точку для вашого набору даних.

4. **Додати контекстні правила (Context Analyzer).** Акцент на обробці шляху файлу, імені ключа, наявності у README і частотному аналізі рядків (повторні появи) дозволить знизити FP у категорії high-entropy.

5. **Поступове ускладнення детекції:**

○ Поетапно додати streaming-читання великих файлів (замість повного зчитування), щоб знизити пикове споживання пам’яті.

○ Реалізувати простий AST-парсер для Python/JS для точнішого виявлення небезпечних викликів (eval, exec, subprocess) і для знаходження

склеєних токенів.

○ Розглянути ML-підхід лише як другий етап (ранжування candidate’ів), коли буде достатній обсяг анотаційних даних.

6. **Інтеграція у CI у режимі diff-only.** Для зменшення часу та кількості FP рекомендується запускати scanner у pre-commit / CI у режимі лише для змінених файлів. У випадку критичних знаходжень — блокувати merge/commit.

7. **Розширення набору правил і процес їхньої версії.** Зберігати правила у git, додавати changelog до правил і впровадити процес рев’ю змін.

8. **Політика поводження зі знахідками.** Маскування та збереження тільки хешів (SHA-256) має стати обов’язковою політикою — це мінімізує ризик витоку під час зберігання результатів тестів.[10]

4.3.5. Подальші експерименти (пріоритети)

- **Sweep ентропії і побудова Precision–Recall кривої** (високий пріоритет).

- **Додати whitelist/blacklist та повторити прогони** для відсікання демонстраційних FP (середній пріоритет).

- **Реалізувати AST-аналіз для Python як proof-of-concept** та порівняти зміни у FP/FN (високий пріоритет для зниження FN у складних кейсах).

- **Порівняльний прогін з Gitleaks/TruffleHog** на тому самому наборі даних для емпіричної оцінки конкурентоспроможності (низький/середній пріоритет залежно від цілей).

Висновки розділу 4

У розділі наведено комплексну методику тестування прототипу, що включає побудову репрезентативних наборів даних (synthetic, edgecases, large), опис автоматичних та ручних сценаріїв тестування, механіку інтеграційних прогонів та метрики для кількісної оцінки (Precision, Recall, F1). Репрезентативний прогін показав прийнятні для лабораторної версії показники: **Precision ≈ 79.3%, Recall ≈ 85.7%, F1 ≈ 82.4%**, що вказує на високу повноту виявлення та помірний рівень хибних спрацьовувань. Проведений аналіз помилок дозволив сформулювати конкретні кроки для підвищення якості детекції: exclude paths, whitelist, тонке налаштування порогів ентропії, додаткові контекстні фільтри та поетапне додавання AST/ML модулів. Практичні рекомендації щодо інтеграції у VSCode/CI та обробки результатів забезпечать прийнятність рішення для навчальних та початкових промислових сценаріїв.

ВИСНОВКИ

У роботі було розглянуто питання виявлення секретів і небезпечних патернів у вихідному коді та конфігураціях, здійснено огляд існуючих підходів і реалізовано прототип інструмента, що поєднує прості правилозалежні механізми з базовими евристиками. Проведений аналіз показав, що жоден окремий метод (лише regex або лише ентропія) не є достатнім у широкому спектрі сценаріїв:

правилозалежний підхід дає швидкі й прозорі результати, але схильний до хибних спрацьовувань; ентропійний аналіз дозволяє виявляти невідомі формати, проте підвищує частку FP без контексту. Комбінований підхід, застосований у прототипі, забезпечує компроміс між точністю і простотою використання.

Експериментальна оцінка прототипу на контрольованому наборі тестових даних вказала на реалістичні робочі характеристики. Для репрезентативного прогона на synthetic-наборі отримані показники: Precision ≈ 79.3 %, Recall ≈ 85.7 % і F1 ≈ 82.4 %. Ці значення свідчать про те, що прототип виявляє більшість простих випадків (висока повнота), але водночас вимагає подальшого зниження рівня хибних спрацьовувань через додаткові контекстні фільтри і whitelist-правила. Конкретні числові результати та розподіл TP/FP/FN наведено у експериментальній частині роботи.

Основні обмеження реалізації лежать у наступних площинах: обмежений набір підтримуваних форматів (ідентифікація мови за розширенням), робота з текстовими файлами у кодуванні UTF-8, простота сигнатур (обмежений набір ключових слів) та відсутність глибокого аналізу багаторядкових або динамічно згенерованих секретів. Ці обмеження зумовлюють природні FN у випадках складних форматів або конкатенації рядків. Для підвищення практичної цінності рекомендовано реалізувати наступні вдосконалення: конфігуровані exclude paths для зменшення FP, впровадження ентропійного модуля з параметричними порогами, streaming-читання великих файлів, а також етапи інтеграції AST-аналізу для критичних мов.

Практичні рекомендації. Для забезпечення прийнятного балансу між корисністю й безпекою рекомендується:
(1) впровадити маскування знахідок і збереження тільки хешів у звітах;
(2) використовувати конфігураційні файли для правил (YAML/JSON) і

exclude-шляхів;
(3) інтегрувати інструмент у процеси розробки як pre-commit hook або CI-step у diff-only режимі;
(4) організувати регресійні тести та автоматизовані прогони на synthetic і edgecases для контролю FP/FN. Такі кроки підвищать довіру до результатів і зменшать операційну навантаженість при впровадженні.

Науково-практична значущість. Розроблений прототип слугує демонстраційною платформою для вивчення проблеми виявлення секретів, дозволяє відпрацьовувати методики формування тестових наборів і впроваджувати поетапні поліпшення — від евристик до статичного синтаксичного аналізу і ML-підходів для ранжування знахідок. Це робить роботу корисною як для освітніх задач, так і як базу для подальших наукових розробок.

Заключне зауваження. Запропонований підхід демонструє практичну здійсненність рішення на рівні лабораторного прототипу: він поєднує простоту реалізації і достатню ефективність для навчальної та початкової комерційної експлуатації. Подальша робота повинна бути спрямована на зменшення хибних спрацьовувань, розширення спектра підтримуваних форматів та імплементацію механізмів інтеграції у робочі процеси розробки для забезпечення безпечного життєвого циклу програмного забезпечення.

Додаток

```
 import os  # type: ignore
rt re  # type: ignore
rt tkinter as tk  # type: ignore
```

```
 tkinter import filedialog, messagebox  # type: ignore

analyze_file(file_path):
try:
    with open(file_path, 'r', encoding='utf-8') as f:
        content = f.read()

    # Підрахунок рядків
    lines_count = len(content.splitlines())

    # Визначення мови за розширенням
    ext = os.path.splitext(file_path)[1].lower()
    if ext == '.py':
        language = 'Python'
    elif ext == '.js':
        language = 'JavaScript'
    elif ext == '.json':
        language = 'JSON'
    elif ext == '.txt':
        language = 'Текстовий файл'
    else:
        return None, "помилка, файл не підходить для
ізу."

    # Сканування на секрети і патерни
    secrets =
indall(r'(?i)(api_key|password|secret|token)\s*[:=]\s*["\
^"\']+)["\']', content)
    dangerous_patterns = re.findall(r'\beval\s*\(',
ent)  # Додайте більше патернів, якщо потрібно

    total_issues = len(secrets) + len(dangerous_patterns)
    if total_issues == 0:
        safety = "Файл безпечний."
        threat_level = ""
    else:
```

```
        safety = "Файл небезпечний."
        if total_issues <= 2:
            threat_level = "Рівень загрози: низький."
        elif total_issues <= 5:
            threat_level = "Рівень загрози: середній."
        else:
            threat_level = "Рівень загрози: високий."

    issues_list = []
    if secrets:
        issues_list.append(f"Знайдені секрети:
rets}")
    if dangerous_patterns:
        issues_list.append(f"Небезпечні патерни:
gerous_patterns}")
    issues = "\n".join(issues_list) if issues_list else
ких проблем не знайдено."

    result = f"Кількість рядків: {lines_count}\nМова
рамування:
guage}\n{safety}\n{threat_level}\n\nПомилки та
рни:\n{issues}"
    return result, None

except Exception as e:
    return None, f"Помилка при читанні файлу: {e}"

select_file():
file_path = filedialog.askopenfilename(title="Виберіть
л для аналізу", filetypes=[("Всі файли", "*.*")])
if file_path:
    result, error = analyze_file(file_path)
    if error:
        messagebox.showerror("Помилка", error)
    else:
        messagebox.showinfo("Результати аналізу", result)
```

```
exit_program():
root.quit()

творення GUI
 = tk.Tk()
.title("Сканер секретів та небезпечних паттернів")
.geometry("400x200")

abel(root, text="Виберіть дію:").pack(pady=10)

utton(root, text="Виберіть файл для аналізу",
and=select_file).pack(pady=5)
utton(root, text="Вихід",
and=exit_program).pack(pady=5)

.mainloop()
```

Список використаних джерел

1 - OWASP Foundation.
Secrets Management Cheat Sheet [Електронний ресурс]. – OWASP Foundation, 2024. – Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html
(дата звернення: 07.12.2025).

2 - OWASP Foundation.
OWASP Top 10 Web Application Security Risks [Електронний ресурс]. – OWASP Foundation, 2023. – Режим доступу: https://owasp.org/www-project-top-ten/
(дата звернення: 07.12.2025).

3 - National Institute of Standards and Technology (NIST).
NIST Special Publication 800-63B: Digital Identity Guidelines [Електронний ресурс]. – Gaithersburg, 2017. – Режим доступу: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf
(дата звернення: 07.12.2025).

4 - National Institute of Standards and Technology (NIST).
NIST SP 800-57 Part 1: Recommendation for Key Management [Електронний ресурс]. – Gaithersburg, 2018. – Режим доступу: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final (дата звернення: 07.12.2025).

5 - Shannon C. E.
A Mathematical Theory of Communication // *Bell System Technical Journal*. – 1948. – Vol. 27, No. 3. – P. 379–423. – Режим доступу: https://ieeexplore.ieee.org/document/6773024 (дата звернення: 07.12.2025).

6 - Livshits B., Lam M.
Finding Security Vulnerabilities in Java Applications with Static Analysis // *USENIX Security Symposium*. – 2005. – Режим доступу: https://www.usenix.org/legacy/event/sec05/tech/full_papers/livshits/livshits.pdf (дата звернення: 07.12.2025).

7 - Meli M., McNiece M., Reaves B. та ін.
Detecting and Mitigating Secret-Key Leaks in Source Code Repositories // *Proceedings of NDSS Symposium*. – 2019. – Режим доступу: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_02A-1_Meli_paper.pdf (дата звернення: 07.12.2025).

8 - GitGuardian.
State of Secrets Sprawl Report 2025 [Електронний ресурс]. – GitGuardian, 2025. – Режим доступу:
https://www.gitguardian.com/state-of-secrets-sprawl-report
(дата звернення: 07.12.2025).

9 - Truffle Security.
TruffleHog – Open Source Secret Scanning Tool [Електронний ресурс]. – Режим доступу:
https://github.com/trufflesecurity/trufflehog
(дата звернення: 07.12.2025).

10 - Zricethezav G.
Gitleaks – SAST Tool for Detecting Hardcoded Secrets [Електронний ресурс]. – Режим доступу:
https://github.com/gitleaks/gitleaks
(дата звернення: 07.12.2025).

11 - Yelp Inc.
detect-secrets: Preventing Secrets from Being Committed [Електронний ресурс]. – Режим доступу:
https://github.com/Yelp/detect-secrets
(дата звернення: 07.12.2025).

12 - Pistoia M., Fink S., Ernst M. та ін.
A Survey of Static Analysis Security Testing Tools // *IBM Research Report*. – 2018. – Режим доступу:

https://research.ibm.com/publications/a-survey-of-static-analysis-security-testing-tools
(дата звернення: 07.12.2025).

13 - Cheng Y., Li Z., Zhang H.
Detecting Hard-Coded Credentials in Software Systems // *arXiv preprint*. – 2023. – Режим доступу:
https://arxiv.org/abs/2305.12177
(дата звернення: 07.12.2025).

14 - GitHub Inc.
GitHub Security Best Practices [Електронний ресурс]. – Режим доступу:
https://docs.github.com/en/code-security
(дата звернення: 07.12.2025).

15 - ISO/IEC.
ISO/IEC 27001: Information Security Management Systems [Електронний ресурс]. – ISO, 2022. – Режим доступу:
https://www.iso.org/standard/82875.html
(дата звернення: 07.12.2025).