Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій

Кафедра комп'ютерних систем і мереж

Пиж Роман Юрійович

УДК 007

БАКАЛАВРСЬКА РОБОТА

Менеджмент криптографічних профілів користувачів розподілених комп'ютерних систем

Комп'ютерна інженерія
(назва освітньої програми)

123 – Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач освітнього ступеня _____ *Пиж Р.Ю.* .
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ *Воронич А.Р., к.т.н. доцент* _____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри
д.т.н., професор _____ /С. І. Мельничук/
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025 рік

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітньо-кваліфікаційний рівень *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

С.І. Мельничук

«05» травня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Пижу Роману Юрійовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи*) Менеджмент криптографічних профілів користувачів розподілених комп'ютерних систем*

керівник проекту (роботи) *Воронич Артур Романович, доцент* .

затверджені наказом вищого навчального закладу від *05.05.2025 № 275/7*

2. Строк подання студентом проекту (роботи) *12 червня 2025р.*

3. Вихідні дані до роботи *Методичні вказівки, технічна література* .

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1 Аналіз предметної області та специфікації вимог до додатку інформаційного супроводу видавництва. 2 Проєктування додатку інформаційного супроводу видавництва. 3 Реалізація програмного забезпечення додатку інформаційного супроводу видавництва 4 Перевірка працездатностірозробленого додатку*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) .

6. Консультанти розділів роботи

7. Дата видачі завдання *29 січня 2025 р.* .

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	*Постановка задачі та вибір технології для розробки.*	*Лютий, 2025*	
2	*Огляд наявних аналогів. Збір інформації про інформаційне та програмне забезпечення.*	*Березень, 2025*	
3	*Розробка програмного забезпечення*	*Квітень, 2025*	
4	*Тестування програмного забезпечення*	*Травень, 2025*	
5	*Оформлення пояснювальної записки*	*Червень, 2025*	

Студент _____ *Пиж Р.Ю.*

Керівник роботи _____ *Воронич А.Р.*

АНОТАЦІЯ

У нашому технологічно розвиненому світі кібербезпека стала критично важливою для захисту систем, мереж і програм від нових загроз. Криптографія, історичний наріжний камінь кібербезпеки, використовує математичні принципи для захисту інформації. Вибір мови програмування для реалізації алгоритмів шифрування збільшує складність у цій галузі. Оскільки програмування відіграє центральну роль у впровадженні шифрування, зростаюча різноманітність мов програмування створює проблему для вибору найбільш прийнятної мови для шифрування даних. Ця робота підкреслює актуальність мов Python і GO для підтримки трьох криптографічних алгоритмів, AES, RSA і 3DES. Робота присвячена продуктивності, простоті впровадження та підтримці криптографічних бібліотек. Розуміння сильних сторін цих мов для криптографічних завдань може допомогти розробникам програмного забезпечення приймати обґрунтовані рішення. Результати показують, що і Python, і GO мають переваги в окремих сферах. GO має хорошу продуктивність, тоді як Python має добре підтримувані криптографічні бібліотеки.

ABSTRACT

In our technologically advanced world, cybersecurity has become critical to protecting systems, networks, and applications from emerging threats. Cryptography, a historical cornerstone of cybersecurity, uses mathematical principles to protect information. Choosing a programming language to implement encryption algorithms adds complexity to the field. Since programming plays a central role in implementing encryption, the growing variety of programming languages creates a challenge for choosing the most appropriate language for encrypting data. This paper highlights the relevance of Python and GO for supporting three cryptographic algorithms, AES, RSA, and 3DES. The paper focuses on performance, ease of implementation, and support for cryptographic libraries. Understanding the strengths of these languages for cryptographic tasks can help software developers make informed decisions. The results show that both Python and GO have advantages in specific areas. GO has good performance, while Python has well-supported cryptographic libraries.

ЗМІСТ

ВСТУП

У сучасному цифровому світі, де розподілені комп'ютерні системи відіграють ключову роль у забезпеченні безпеки даних, захисті конфіденційності та підтримці довіри між користувачами, ефективне управління криптографічними профілями стає критично важливим. Криптографія є основою для захисту інформації в таких системах, як блокчейн, хмарні сервіси та децентралізовані мережі, але її неправильне використання може призвести до серйозних вразливостей. Розробка програмного забезпечення для управління криптографічними профілями, яке враховує сучасні алгоритми шифрування, безпечні бібліотеки та оптимальні мови програмування, відкриває нові можливості для підвищення безпеки, продуктивності та простоти впровадження в розподілених системах.

Актуальність теми обумовлена швидким розвитком розподілених комп'ютерних систем і зростанням кількості кібератак, які використовують слабкі місця в криптографічних реалізаціях. Існуючі рішення часто страждають від поганої криптографічної практики, застарілих бібліотек або недостатньої уваги до продуктивності, що підкреслює необхідність створення нових підходів до управління криптографічними профілями користувачів.

Метою роботи є розробка системи управління криптографічними профілями користувачів розподілених комп'ютерних систем, яка забезпечить безпечне, ефективне та просте у впровадженні рішення для захисту даних.

Завданнями дослідження є аналіз сучасних криптографічних рішень, вивчення останніх досліджень поганої криптографічної практики, розробка моделі порівняння алгоритмів шифрування, вибір безпечних бібліотек, тестування продуктивності та створення інструментарію для програмування криптографічного програмного забезпечення.

Об'єктом дослідження є процеси розробки програмного забезпечення для управління криптографічними профілями в розподілених комп'ютерних системах,

що охоплюють аналіз вимог, проектування, реалізацію, тестування та оцінку ефективності.

Предметом дослідження є методи, моделі, алгоритми та інструменти, що застосовуються під час розробки таких систем, зокрема алгоритми шифрування, криптографічні бібліотеки, мови програмування, а також методи оцінки продуктивності й безпеки.

Для досягнення мети використано методи аналізу літературних джерел, порівняльного аналізу алгоритмів, моделювання криптографічних систем, емпіричного програмування, тестування продуктивності та безпеки, а також оцінки простоти впровадження.

Розроблена система передбачає управління криптографічними профілями користувачів, використання сучасних алгоритмів шифрування, таких як AES і RSA, інтеграцію безпечних бібліотек, таких як OpenSSL або Libsodium, і забезпечення високої продуктивності в розподілених середовищах. Очікується, що впровадження системи сприятиме підвищенню безпеки даних, спрощенню управління криптографічними ресурсами та встановленню нових стандартів для розробки криптографічного програмного забезпечення в розподілених системах.

1 АНАЛІЗ РІШЕНЬ ДАНОГО КЛАСУ

1.1 Передумови та мотивація

Деякі широко використовувані алгоритми шифрування є Рівест-Шамір-Адлеман (ЮАР), Стандарт потрійного шифрування даних (3DES), Криптографія еліптичної кривої (ECC) і Розширений стандарт шифрування (AES). Спосіб реалізації цих алгоритмів буде відрізнятися залежно від того, яка мова програмування використовується, оскільки в криптографії немає мови програмування. Але одними з найпоширеніших мов для криптографії є Python, GO, Ruby, C++ і Java [9].

Ви не можете знати, чи мова, до якої ви найбільше звикли, є найкращою для криптографії. Зрештою, існує безліч змінних, які слід враховувати, щоб визначити, наскільки мова підходить для шифрування та дешифрування. Як-от час виконання, простота впровадження, інструменти, які допомагають у процесі написання коду, мовні особливості та підтримка криптографічних бібліотек тощо. Справедливу оцінку можна дати, лише якщо порівнювати кожну мову програмування в однакових категоріях.

У цьому тексті криптографічне програмне забезпечення – це програмне забезпечення, яке має своєю метою справжню потребу у захисті або збереженні деяких основних цілей інформаційної безпеки (а саме цілісності, автентичності, конфіденційності та неспростовності) за допомогою криптографічної технології. Для досягнення цих цілей програмне забезпечення може використовувати криптографію безпосередньо, за допомогою власних реалізацій або через багаторазові бібліотеки та фреймворки.

У будь-якому випадку, реалізації криптографічних алгоритмів мають бути ретельно створені, щоб у них не було проблем, які ставлять під загрозу безпеку програмного забезпечення. Крім того, ці безпечні реалізації мають безпечно використовуватися програмістами додатків, які сприймають як належне якість внутрішніх алгоритмів.

Незважаючи на чотири десятиліття після того, як Зальцер і Шредер опублікували золоті правила безпеки програмного забезпечення [1], розробка безпечного програмного забезпечення [2, 3], здається, не стосується безпосередньо питання криптографічної безпеки. Насправді протягом майже двадцяти років дослідження показали, що уразливості в криптографічному програмному забезпеченні були в основному спричинені дефектами програмного забезпечення та поганим керуванням криптографічними параметрами та іншим конфіденційним матеріалом [4, 5, 6, 7, 8].

Крім того, нещодавні дослідження [9, 10, 11, 12, 13, 14, 15] показали повторювану появу добре відомих поганих практик використання криптографії в різних програмних системах (зокрема в мобільних додатках). Фактично, починаючи з 2012 року, інтерес академічних кіл до «криптографії реального світу» [16] став явним і набирає обертів.

Важливо відрізняти безпечне (або захисне) програмування криптографічного програмного забезпечення від програмування безпечного криптографічного програмного забезпечення. Перше пов'язане з використанням загальних методів безпечного кодування під час програмування криптографічного програмного забезпечення. Останній починається в точці, де закінчується перший, і охоплює спеціальні безпечні методи кодування та програмування контрзаходів для кращого захисту криптографічного програмного забезпечення від конкретних зловживань, а також від поганої конструкції криптографічних методів. Сьогодні криптографічне програмне забезпечення визнаної якості зазвичай розглядається як «твір мистецтва», конструктивний процес якого навряд чи може відтворити пересічний програміст. Для уважного спостерігача існує велика кількість поганих реалізацій, а також брак хороших.

Це твердження підтверджується низкою сучасних прикладів:

1 Неадекватність поточних програмних засобів (наприклад, SDK, мов програмування та компіляторів тощо) для вирішення проблем безпеки в криптографічному програмуванні. Наприклад, перегляньте нещодавні випадки

відомих уразливостей у криптографічному програмному забезпеченні (наприклад, HeartBleed [17] і помилка Apple GoTo Fail [18]).

2 Низькорівневі криптографічні служби зловживали звичайними програмістами без належного інструментарію чи освіти. Основні криптографічні процеси не вивчаються ефективно програмістами [10, 11, 14, 15], які роблять ті самі помилки знову і знову.

3 Складні концепції безпеки не представлені в існуючих рамках. Наприклад, перевірка цифрових сертифікатів ускладнює програмістів непотрібну складність через неправильне розуміння того, як мають бути сформовані криптографічні структури [9, 12, 13].

Уразливості, зрештою пов’язані з архітектурними аспектами криптографічних служб і дизайном API, можуть виявити несподівані побічні канали та витік інформації. Наприклад, Padding Oracles [19] може виникнути через невідповідну обробку помилок на верхніх рівнях під час організації криптографічних служб і потенційний витік інформації [20, 21, 22].

1.2 Останні дослідження криптографічної поганої практики

У цьому розділі аналізуються нещодавні дослідження щодо неправильного використання криптографічного програмного забезпечення. Згідно з останніми дослідженнями Egele та ін. [11] і Shuai та ін. [14], найпоширенішим зловживанням є використання детермінованого шифрування, де симетричний шифр у режимі електронної кодової книги (ECB) з’являється в основному за двох обставин: AES/ECB та 3DES/ECB. Існують випадки криптографічних бібліотек, у яких режим ECB є параметром за замовчуванням, який автоматично вибирається, коли режим роботи явно не вказано програмістом. Можливо, гіршим варіантом цього неправильного використання є RSA в режимі ланцюжка блоків шифру (CBC) без рандомізації, який також доступний у сучасних бібліотеках, незважаючи на те, що він був ідентифікований більше 10 років тому Гутманом [5].

Іншим частим неправильним використанням є жорстко закодовані вектори ініціалізації (IV), навіть із фіксованими чи постійними значеннями [11]. Вектори ініціалізації, майже у всіх режимах роботи блокових шифрів, повинні бути як унікальними, так і непередбачуваними. Винятком є режим CTR, який вимагає унікальних IV (без повторення). Ця вимога поширюється на режим шифрування з автентифікацією GCM. Пов'язане неправильне використання - це використання звичайним програмістом жорстко закодованих початкових значень для PRNG [11]. Поширене непорозуміння щодо правильного використання IV виникає, коли (з будь-якої причини) програмістам потрібно змінити режими роботи блокових шифрів. Наприклад, Java Cryptographic API [23] дозволяє легко змінювати режими роботи без урахування вимог IV.

Перевірка цифрових сертифікатів у веб-браузерах є відносно добре побудованою та надійною, незважаючи на те, що користувачі зазвичай ігнорують попередження щодо недійсності сертифікати. У програмному забезпеченні, відмінному від веб-браузерів, особливо в програмах для мобільних пристроїв, існує широкий спектр бібліотек для обробки з'єднань SSL/TLS. Нещодавні дослідження Георгієва та ін. [12] і Фахла та ін. [13, 24] показують, що всі вони дозволяють програмісту ігнорувати деякі кроки під час перевірки сертифіката, щоб підвищити зручність використання або продуктивність, але створюють уразливості. Зокрема, збій під час перевірки підпису або перевірки доменного імені сприяє атаці Man-in-the-Middle (MITM).

Нарешті, нещодавнє дослідження Лазара та інших [15] показало, що на основі аналізу 269 уразливостей, пов'язаних із криптографією, лише 17% помилок знаходяться в криптографічних бібліотеках, а решта 83% – це неправильне використання криптографічних бібліотек програмами.

1.3 Проблема

Програмування відіграє ключову роль у реалізації алгоритмів шифрування. Оскільки існує дедалі більше різноманітних мов програмування, багато з яких

мають власні переваги для шифрування та дешифрування, може бути важко вирішити, яку мову використовувати для ши фрування даних. Це призводить до наступного дослідницького питання щодо **в**ідмінностей в підтримці шифрування різними мовами програмування.

Метою цієї дипломної роботи є висвітлення важливих властивостей різних мов програмування. Аналізуються:

1 Продуктивність: наскільки швидкі та ефективні операції шифрування/дешифрування.

2 Простота впровадження: якщо реалізувати алгоритми шифрування легко і просто.

3 Криптографічні бібліотеки: скільки і якої якості є доступні криптографічні бібліотеки?

Основна мета цього проекту — пояснити розробникам програмного забезпечення переваги різних мов програмування при роботі з шифруванням. Це робиться шляхом детального опису того, як працює кожна мова, які типи функцій вони мають, наскільки легко реалізувати різні алгоритми шифрування та до якого типу криптографічних бібліотек вони мають доступ.

1.4 Методологія дослідження

Ми провели дослідження літератури, щоб ознайомитися з різними мовами програмування та алгоритмами шифрування. Це буде ключем до розуміння того, як буде здійснюватися цей проект. Це включало, але не обмежувалося, читання документації мови програмування та аналіз підтримки криптографічної бібліотеки. У цій дипломній роботі порівнюються мови програмування щодо їхньої підтримки алгоритмів шифрування, тому критерії оцінки були обговорені та оцінені для створення відповідних категорій у порівнянні. Після вивчення літератури було проведено тематичне дослідження, щоб зібрати дані щодо виконання алгоритмів шифрування за допомогою вибраної мови програмування. Реалізовані алгоритми потім були проаналізовані за вибраними

критеріями.Нарешті, інформацію з літературного дослідження та прикладу було ретельно проаналізовано та сформульовано відповідь на дослідницьке запитання. Цільова аудиторія включає студентів, викладачів, дослідників і розробників, які цікавляться або зараз працюють з кібербезпекою. Розробники в організаціях можуть використовувати наше дослідження, щоб розрізнити різні потрібні аспекти мов програмування з метою безпеки, які можуть стосуватися продуктивності, простоти впровадження або підтримки криптографічних бібліотек.

Розмежування.

Необхідно визнати розмежування, оскільки ця широка дослідницька область містить багато мов програмування та алгоритмів шифрування. Щоб дипломна робота була продуктивною, було вирішено обмежити кількість мов програмування та алгоритмів. Це призведе до того, що згенерований результат буде менш точним, оскільки кількість ресурсів обмежена. Робота також включатиме спеціальний набір критеріїв, які використовуються для визначення мов програмування для порівняння, які не визнають енергоспоживання як критерій оцінки.

Метою є надання інформації про аспекти мов програмування під час роботи з різними алгоритмами шифрування. Тому всі алгоритми не будуть реалізовані з нуля, оскільки це не є метою дипломної роботи.

Переваги, етика та сталість.

Дослідження алгоритмів шифрування разом із набором конкретних мов забезпечує глибоке розуміння переваги однієї мови над іншими через потребу в певному аспекті, який позитивно допомагає певним алгоритмам. Для прикладу, скажімо, мова програмування «A» з підтримкою бібліотеки для криптографічних алгоритмів має в середньому на 20 відсотків нижчий час виконання для певного алгоритму порівняно з мовою «B». Якщо швидкість є важливим фактором, то «A» може бути можливим кандидатом для розгляду, а не мова програмування «B».

1.5 Теоретичні основи

Цей розділ починається з короткого вступу до історії шифрування, після чого йде вступ до криптографії та двох основних типів криптографічних систем: асиметричної та симетричної. Для кожної системи надаються відповідні базові пояснення пов’язаних алгоритмів, оскільки це необхідно для цієї галузі дослідження. Після цього йде короткий вступ до різних мов програмування, які будуть використані для реалізації пізніше в дослідженні. Нарешті, представлені деякі роботи, які проводили подібні дослідження, як ця дисертація.

Коротка історія шифрування.

Шифрування існувало протягом більшої частини людської цивілізації, причому шифри заміни та шифри транспозиції були найпоширенішою формою стародавнього шифрування. Шифри заміни працюють шляхом заміни літер або груп літер іншими літерами чи групами літер, наприклад, замінюючи кожну літеру в алфавіті літерою після неї; A стає B, B стає C і так далі. Тоді як шифри транспозиції працюють, змінюючи порядок літер у слові, тобто слово «будинок» буде написано як «оухес».

Перші письмові свідчення про шифрування можна простежити до стародавнього Єгипту. У гробниці з 1900 року до нашої ери були знайдені незвичайні ієрогліфи в сценарії, що записує вчинки окремої гробниці померлого. Ці ієрогліфи використовувалися як спроба приховати оригінальний текст з незрозумілих причин.

Перемотайте 1400 років вперед, і єврейські писарі винайшли шифр підстанції, відомий як Атбаш. Цей шифр кодує повідомлення шляхом зміни алфавіту: A стає Z, B стає Y, C стає X і так далі.У ту ж епоху спартанці використовували тип шифру транспозиції для таємної передачі повідомлень під час війни. Це було зроблено за допомогою типу стрижня, який називається скітал. Тонкий шматок пергаменту був намотаний навколо стрижня, уздовж якого був написаний текст. Після розмотування пергаменту текст ставав нечитабельним, доки його не прикріпив до ідентичного циліндра одержувач. Це був перший

випадок, коли концепція відкритого ключа, яку можна побачити навіть сьогодні в асиметричному шифруванні, використовувалася як для шифрування, так і для дешифрування.

Лише в 1800-х роках з'явилися більш складні та потужні методи шифрування даних. Починаючи з Едгара Аллана По, який написав есе про численні різні криптографічні методи, які пізніше використовували британці для розшифровки німецьких кодів і шифрів під час Першої світової війни. До Другої світової війни почали використовувати механічні та електромеханічні шифрувальні машини. Прикладом тому є загадкова машина, яка за допомогою електромеханічного роторного механізму шифрувала літери алфавіту [11]. Розкриття загадкового коду виявилося важливим для військових зусиль союзників, що призвело до створення криптографії як наукової галузі та надихнуло Алана Тюрінга на розробку та використання першої машини, здатної використовувати обчислювальну потужність для зламу шифрування. До кінця Другої світової війни завдання підготовки та розшифровки шифрів перейшли від машин до комп'ютерів, що привело людей у еру сучасної криптографії. Завдяки комп'ютерам і математичним функціям стало можливим безпечніше представляти дані. Існували різні підходи до шифрування даних. Їх можна класифікувати як алгоритми з асиметричним ключем і алгоритми з відкритим ключем [11].

1.6 Алгоритми шифрування

Шифрування передбачає захист зв'язку та даних шляхом перетворення інформації у форму, зрозумілу лише тим, хто має відповідний ключ або знання. Відправник перетворює простий текст ($П$) у нерозбірливий зашифрований текст ($С$) за деяким алгоритмом шифрування за допомогою ключа шифрування ($К_1$). Алгоритм визначає процес перетворення, тоді як ключ впливає на конкретне перетворення, застосоване до звичайного тексту. Так само, щоб перетворити зашифрований текст назад у звичайний текст, одержувач використовує алгоритм

дешифрування з ключем дешифрування ($К_2$). Це можна інтерпретувати як те, що відправник передає повідомлення $П$ шляхом передачі зашифрованого тексту

$$С : С = І(К_1, П)$$

Одержувач розшифровує зашифрований текст, щоб отримати повідомлення $П$ використовуючи $К_2$, оскільки $П = Д(К_2, С)$, як показано на рисунку 1.1.

Рисунок 1.1 – Процеси шифрування та дешифрування [12]

Алгоритми шифрування, які зазвичай називають «шифрами», поділяються на дві категорії: асиметричні та симетричні.

Асиметричні алгоритми.

Асиметричні алгоритми (також відомі як алгоритми з відкритим ключем) покладаються на пару ключів для виконання як шифрування, так і дешифрування для того самого потоку даних, один відкритий ключ для шифрування та відповідний закритий ключ для дешифрування. Відкритий ключ, як випливає з назви, можна вільно ділитися, тоді як закритий ключ слід зберігати в секреті. Будь-хто, хто має відкритий ключ, може зашифрувати повідомлення, але лише ті, хто знає відповідний закритий ключ, можуть розшифрувати зашифрований текст, щоб отримати вихідне повідомлення [13].

Симетричні алгоритми.

Симетричні алгоритми використовують один ключ як для шифрування, так і для дешифрування для ретрансляції одного повідомлення. Таким чином, ключ повинен бути секретним, відомим лише відправнику та одержувачу. У симетричних алгоритмах існує два підтипи: потокові шифри та блочні шифри.

Основна відмінність полягає в тому, що в системах на основі потокового шифру дані обробляються або шифруються на основі потоку бітів, тоді як у системах на основі блокового шифру дані обробляються на основі групи бітів фіксованої довжини, яка називається блоком [14]. Рисунок 1.2 показує класифікації алгоритмів шифрування разом із прикладами кожної класифікації та роком їх створення.

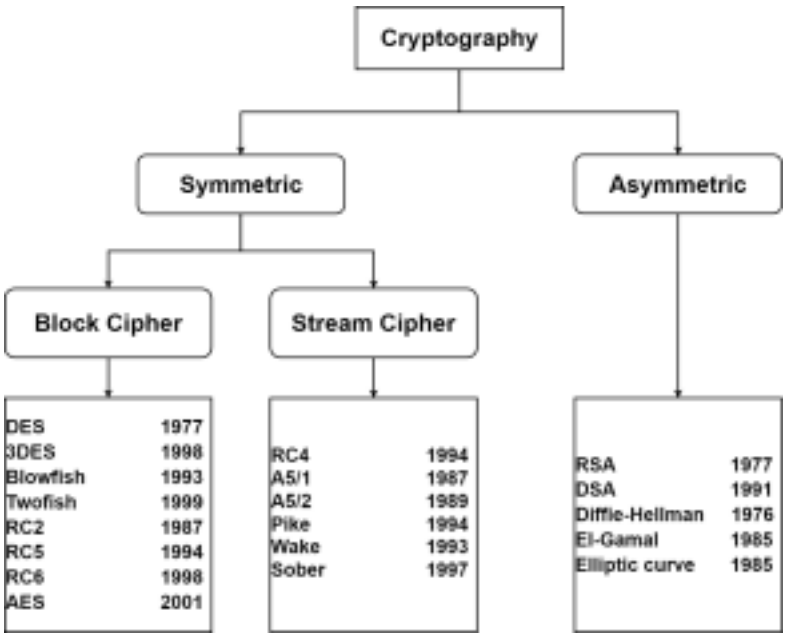

Рисунок 1.2 – Класифікація алгоритмів шифрування [12]

Вивчені алгоритми.

Численні криптографічні алгоритми широко використовуються в різних сферах. У цьому розділі пропонується короткий вступ до алгоритмів, які пройшли ретельний аналіз і слугували еталоном для тестування та дослідження різних мов програмування.

AES.

Advanced Encryption Standard (AES) — це алгоритм шифрування з симетричним блочним шифруванням, який спочатку був запропонований як заміна DES, оскільки він містив уразливості. AES був просуваний NIST (Національний інститут стандартів і технологій) у 2000 році, оскільки він був оцінений серед п’яти інших претендентів на алгоритм шифрування з

використанням різних критеріїв і важливих параметрів безпеки [15]. Для довідки NIST – це організація США, яка вимірює стандарти, технології та інновації [16].

AES використовує фіксований розмір блоку 128 біт (блочний чипер) і унікальний ключ як для шифрування, так і для дешифрування зі змінним розміром ключа 128, 196 або 256 біт [17], що позначатиметься назвою AES-128, AES-196 і AES-256 відповідно. Алгоритм розроблено з мережею заміни та перестановки, яка використовує підстановку та перестановку в алгоритмі замість мережі Фістеля, яка використовується в DES. Відзначається, що використання мережі заміни-перестановки швидко обробляється апаратним і програмним забезпеченням, що робить AES гнучким і швидким [18].

RSA.

Rivest-Shamir-Adleman (RSA) працює як загальнодоступна криптосистема, яка є асиметричним блочним шифром, який широко використовується в рішеннях, пов'язаних з Інтернетом, як-от перевірка автентичності електронної пошти, веб-сервери та браузери для безпеки веб-трафіку, ймовірно, через те, що RSA надає цифрові підписи. Цей алгоритм також працює як ядро в системах електронних карткових платежів. Він був розроблений і названий на честь Рона **Р**івест, Аді **С**Хамір і Леонард **А**dleman у 1978 році в статті, опублікованій у науковому журналі «Scientific American» [19, 20, 21,22].

RSA — це повільніший алгоритм, заснований на теорії чисел, і його безпека значною мірою залежить від надійності ефективного розкладання простих чисел. На цій ноті RSA був би небезпечним, якби був розроблений ефективний алгоритм факторингу, але багато спроб розробити цей тип концепції атак виявилися недостатньо ефективними. Оглядаючись назад, більшість спроб показують неправильну реалізацію алгоритму [21, 20].

3DES.

Стандарт потрійного шифрування даних (3DES) — це симетричний блоковий шифр, який був попереднім стандартом безпеки AES (Rijindael). 3DES був першою відповіддю на недоліки безпеки свого попередника, DES. 3DES — це версія DES, яка шифрує/дешифрує три рази, а також використовує набір із трьох

ключів для шифрування та дешифрування, використовуючи кожен ключ для кожного етапу шифрування та дешифрування. 3DES і DES базуються на мережі Fiestel, завдяки чому стадії шифрування та дешифрування майже ідентичні, що спрощує реалізацію всього алгоритму. Зверніть увагу, що ключі можуть використовуватися в різному порядку, але вибрано порядок шифрування, дешифрування-шифрування (EDE) із 3 варіантами [19, 18, 23].

Зважаючи на недоліки безпеки, DES було відкликано в 2005 році, і 3DES став варіантом, який було досить легко реалізувати, щоб багато програм могли прийняти його, забезпечуючи більшу безпеку. Проте ці програмиперейшли від 3DES до кращих варіантів, таких як AES, оскільки 3DES є повільнішим блоковим шифром, а також оскільки 3DES було заборонено 31 грудня 2023 року, але продовження використання дозволено під час роботи з уже захищеними даними або для застарілих програм [23, 24].

1.7 Мови програмування

Існує величезна кількість мов програмування, одні з них загального призначення, інші – більш предметно-спеціальні. У цьому розділі подано коротку презентацію мов, які було вибрано для аналізу та використано для реалізації обраних алгоритмів шифрування.

Python — це популярна інтерпретована мова високого рівня з функціями, які забезпечують швидку розробку додатків, а також її можна використовувати як мову сценаріїв або мову «склею», що з'єднує різні компоненти [25, 26]. Ця мова була вперше випущена в першому кварталі 1991 року голландським розробником Гвідо ван Россумом, який мав на меті створити просту та зрозумілу англійською мовою програмування [27]. Поточний випуск Python поставляється в комплекті з багатьма функціями зі стандартної бібліотеки [26], а також розширюється до C і C++ для інших функцій або типів даних.

Багато аспектів цієї мови спрощують створення сценаріїв автоматизації, роботу з веб-розробкою, аналіз даних і машинне навчання [28]. Завдяки системі

динамічного набору тексту Python [29], у цих сферах простіше працювати навіть новачкам.

Мова програмування GO є статично типізованою, процедурною мовою і, перш за все, з відкритим кодом [30]. Він був розроблений командою Google у 2007 році [31], який хотів створити просту, легку для вивчення та ефективну мову програмування. Перевагами Go є насамперед його простота та читабельність, які можна розглядати через його фокусування на простій інтеграції важливих функцій, таких як паралелізм, збір сміття та низький час компіляції [30, 32].

Go став дуже необхідною мовою серед співробітників Google через те що дедалі складніша кодова база Google потребувала обслуговування. Таким чином, Go добре обізнаний у сприянні покращенню інфраструктури та мережі, оскільки ці аспекти були однією з основних цілей Go [31].

Цілісність програмного забезпечення є невід'ємною частиною будь-якої розробки програмного забезпечення, оскільки вона має вирішальне значення для тривалої інтеграції. Темою Go є розробка програмного забезпечення для систем високої цілісності [33], що робить його ідеальним для різних цілей програмування. Go використовувався в багатьох сферах, серед яких хмарні служби або серверні програми, а також розробка інструментів командного рядка [31].

Існують численні порівняльні дослідження криптографічних алгоритмів та їх продуктивності. Ці обговорення зазвичай зосереджуються на функціях і безпеці алгоритмів, часто включаючи міркування щодо мов програмування, які використовуються для реалізації. Однак оцінки та порівняння різних мов програмування для реалізації криптографічних алгоритмів рідко знаходяться в центрі уваги.

Аналіз та використання методів криптографії в мові програмування C#.

Бафті *та ін.* представляє дослідження, що аналізує криптографічні шифри та їхні криптографічні особливості, а також програму на C#, яка шифрувала та

дешифрувала різні тексти. Вони показали кожен шифр на прикладі та показали, як цей шифр працює, порівнюючи між собою. Дослідження в основному представляє порівняння як частину свого висновку. Порівняння можна використати для диференціації інформації про різні шифри та їхні унікальні властивості, а також для отримання та розуміння того, як це реалізувати в C# [34].

Впровадження криптографічних алгоритмів у програмне забезпечення: аналіз ефективності.

Оцінюється безпечність реалізації та швидкість виконання для кількох різних класів криптографічних алгоритмів, наприклад, у межахшифрування, односторонні хеш-функції, коди автентифікації повідомлень і цифровий підпис. Вони зробили власну реалізацію, використовуючи програмні бібліотеки та фреймворки для Java, C# та C++, щоб мати можливість оцінити ефективність різних криптографічних алгоритмів.

Їх реалізації дуже актуальні для наших досліджень, зокрема їхні реалізації алгоритмів шифрування, оскільки вони показують нам швидкість виконання для різних мов програмування [35].

Опитування щодо алгоритмів криптографії.

Abood *та ін.* проводить порівняльне дослідження, щоб визначити найважливіший алгоритм з точки зору безпеки даних. Вони обговорюють і досліджують кілька популярних алгоритмів шифрування, таких як AES, DES, 3DES, RSA, ECC, Алгоритм цифрового підпису (DSA), і Rivest Cipher 4 (RC4). Вони порівнюють їх за розміром ключа, розміром блоку, круглістю, структурою, гнучкістю та функціями. Вони знайшли переваги для різних алгоритмів, але віддали перевагу AES як найнадійнішому алгоритму з точки зору швидкості шифрування та декодування, складності, довжини ключа, структури та гнучкості [36].

2 МЕТОДИ, МОДЕЛІ ТА АЛГОРИТМИ

Методологія дослідження.

У цьому розділі представлено методологію дослідження, використану в дисертації. Ось детальна інформація про етапи дослідження, підхід до дослідження та інструменти дослідження. Крім того, обговорюється валідність обраної методології.

2.1 Етапи дослідження

Цей розділ охоплює різні етапи дослідження, які використовуються для виконання цієї дипломної роботи. Кожна фаза відповідає діяльності, яка допомагає спрямувати наше дослідження до остаточного висновку, її можна розглядати як процес з однією або кількома запланованими цілями. Ці фази: (1) Попереднє дослідження, (2) Створення критеріїв оцінювання мов програмування, (3) Реалізація алгоритмів шифруванняі (4) Порівняння можливостей шифрування мов програмування рисунок 2.1 показує ці фази.

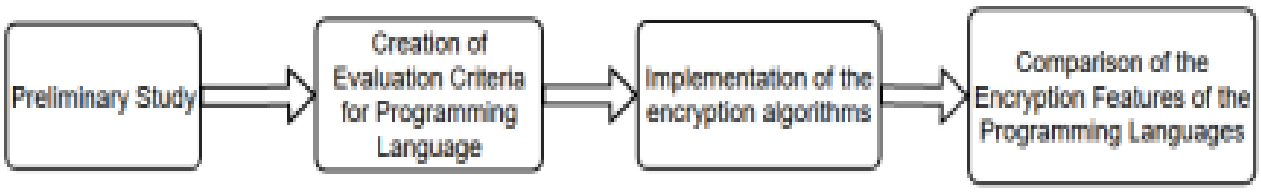

Рисунок 2.1 – Етапи дослідження

Це дослідження літератури було теоретично важким, де була зібрана інформація як про мови програмування, так і про алгоритми шифрування. Значна частина інформації була зібрана за допомогою онлайн-пошукових систем, а також великих наукових пошукових систем, таких як Google Scholar і DiVa Portal. Проте Google Scholar і DiVa Portal використовувалися ширше, оскільки вони були найзручнішими у використанні. На цьому етапі були обрані та ретельно вивчені

мови програмування. Таким чином було закладено основу для реалізації різноманітних алгоритмів шифрування.

Створення критеріїв оцінки мови програмування.

Метою цього етапу дослідження було визначення порівняльної моделі, яка використовується для порівняння різних мов програмування при використанні для шифрування. Ця оцінка мов мала бути важливою для розробників, які прагнуть реалізувати алгоритми шифрування.

Модель порівняння складалася з трьох різних критеріїв: Performance, Простота реалізації, і Криптографічна бібліотека. Метрики Криптографічна бібліотека використовувався для порівняння різних мов програмування, зібраних на етапі попереднього дослідження, тоді як Продуктивність і Простота реалізації були використані для порівняння реалізацій алгоритмів шифрування в цих мовах.

Реалізація алгоритмів шифрування.

Реалізація алгоритмів шифрування була практичною частиною з основною метою, як частину прикладу, впровадження алгоритмів у вибраних мовах програмування для вимірювання Продуктивність і Простота реалізації. Ми впроваджували алгоритми, одну мову за раз, а потім аналізували наші результати, перш ніж переходити до наступної мови. Кінцевий продукт цієї фази та результати приблизно Криптографічні бібліотеки з вивчення літератури були необхідні дані для наступного етапу Порівняння можливостей шифрування мов програмування.

Порівняння можливостей шифрування мов програмування.

Під час останнього етапу дані з вивчення літератури та етапу впровадження були проаналізовані разом із запропонованою моделлю порівняння. Порівняння були складені у форматі діаграми за критеріями. Хоча це називається «порівнянням», ми не обмежилися лише можливістю порівняти дві мови програмування. Кожну мову програмування можна порівняти з будь-якою іншою мовою програмування, оскільки необхідні матеріали були надані для кожної мови та критеріїв порівняння.

Прикладне дослідження відноситься до інтенсивного характеру охоплення більш детальної інформації з багатством і різноманіттям. Тематичні дослідження, як правило, проводяться різними способами, наприклад, кількісними, якісними чи іншими способами. Це можна пояснити, оскільки тематичні дослідження сягають всієї історії і завжди забезпечували ефективний спосіб оцінки теоретичного застосування в реальному світі, враховуючи, що більшість емпіричних даних походить від використання тематичних досліджень [37]. Кейсове дослідження як частина цієї дисертації було використано для отримання сучасних даних щодо сучасної доступності криптографічних бібліотек і демонстрації практичних переваг, які включають продуктивність виконання та загальне задоволення від написання коду вибраною мовою.

Якісні та кількісні.

Якісне дослідження стосується збору даних, які можуть бути використані для демонстрації відчуття глибшого розуміння проблеми, що розглядається. Завдяки отриманню розуміння, що виходить за межі одних лише числових даних, методології якісного дослідження пропонують можливість виявити проблеми, що лежать в основі проблеми. Це пов’язано з тим, що якісні дані часто походять із лінгвістичних або візуальних джерел, що забезпечує більш глибоке розуміння предмета.

Якісні методи дають більше свободи для інтерпретації та можуть абстрагуватися, узагальнюючи тему, щоб показати ширшу картину, але дані якісних методів важко отримати та узагальнити. Відтворити ці дослідження також важко, оскільки цей тип методології покладається на багато даних, які можуть сильно відрізнятися. Для вираження людських почуттів також необхідні якісні дані. Прикладами якісних методів дослідження є інтерв’ю, тематичні дослідження, фокус-групи та аналіз документів [38]. Кількісні методи дослідження також пропонують хороші альтернативи для збору інформації. Кількісні методи дослідження збирають більшу кількість даних порівняно з якісними. Це забезпечує об’єктивне уявлення про предмет із використанням числових і статистичних аспектів у методах, таких як рівень випадків, щоб

відповісти на такі запитання, як «скільки?» і «як часто?». Це полегшує необхідність представити, узагальнити та порівняти результати в цілому [39].

У цій дипломній роботі застосовувалася якісна методологія з деякими кількісними даними. Якісна частина дипломної роботи стосується аналізу документації та нашого досвіду впровадження алгоритмів шифрування. Кількісна частина виникла в результаті тестування наших реалізацій. Тому після кожного впровадження проводилися тести, щоб отримати необхідну інформацію. Наша ідея тестів полягала в тому, щоб накопичити достатньо даних, щоб представити середнє значення та використати його як наш результат. Це полягало в ітераціях фіксованої кількості разів, одночасно вимірюючи різницю в часі між початком і кінцем кожної ітерації алгоритму. Цей підхід було обрано, оскільки загальною потребою було зібрати сучасні дані. Це означає, що проведене тематичне дослідження було дійсним протягом періоду його проведення, але може відрізнятися в майбутньому, якщо буде проведено подібне дослідження.

Порівняльні дослідження.

Порівняльне дослідження — це метод пошуку спільного між спорідненими питаннями та визначення відповідних атрибутів, що забезпечує більший вплив як на дисертацію, так і на контекст предметної галузі. Ця форма дослідження була важливим вибором для цієї дисертації, враховуючи, що мета полягає в тому, щоб підкреслити відмінності, щоб довести, що певні мови програмування вигідні для використання шифрування. Ці спільні риси були визначені та виявлені на етапах, які використовували якісні та кількісні методології.

Щоб отримати результати, які могли б сприяти глибокому розумінню, накопичене дослідження було порівняно з трьома критеріями оцінки. Кожна мова відповідала кожному критерію по-своєму. Це означає, що мови не безпосередньо порівнювали одна з одною, а вимірювали окремо, щоб об’єктивно та ефективно продемонструвати відмінності. Це вигідно цільовій аудиторії, щоб дати їй неупереджену думку та, зокрема, полегшити пошук інформації в цій галузі, оскільки це є метою цієї роботи.

Дослідницькі інструменти.

Основними дослідницькими інструментами, використаними в роботі, були Порівняльна модель і Реалізації. Крім того, існували також інструменти, які використовувалися для допомоги інструментам, наприклад Google Scholar і Код Visual Studio. Що це були за інструменти та чому вони використовувалися, пояснюється в наступних пунктах.

Вивчення літератури: Було прочитано документацію для мов програмування, щоб зібрати дані про те, які типи бібліотек і функції мають мови програмування. Крім того, оскільки існували подібні дослідження, ми зібрали дані та вимірювання з існуючих робіт, щоб мати можливість використовувати їх у нашій роботі.

Модель порівняння: Порівняльну модель використовували під час порівняння різних мов програмування між собою. Це було важливо для того, щоб проект міг показати, яка мова програмування є вигідною для різних ситуацій.

Реалізації: Реалізації використовувалися для отримання вимірювань у практичних дослідженнях. Для кожного було три види реалізаціїмова програмування, яка аналізувалася: впровадження та тестування шифру AES, шифру RSA та шифру 3DES. Це передбачало шифрування текстових файлів різного розміру до 200 МБ і обчислення його пропускної здатності. Для створення шифрів використовувалася або стандартна бібліотека, або дуже поширена зовнішня бібліотека.

Код Visual Studio: Це комп’ютерна програма, яка дозволяє редагувати та виконувати програми. Код Visual Studio використовувався для створення реалізацій алгоритмів шифрування для обраних мов програмування.

Google Scholar: Google Scholar — це пошукова система, яка дозволяє користувачеві ефективно знаходити наукову літературу в межах теми. Його використовували для пошуку статей, книг та інших тез у галузі криптографії.

Комп'ютер: Вживаний комп’ютер мав наступні характеристики:

– процесор Intel 13-го покоління з 10 ядрами на базовій робочій частоті 1700 МГц;

– 16 ГБ фізичної пам'яті (RAM);4

– Операційна система Windows 11.

Портал DiVa. За допомогою цього інструменту ми могли знаходити публікації, які були зареєстровані в DiVa.

Термін дії. Критерії валідності використовуються для перевірки міцності та надійності методу дослідження [40]. Дослідження в дипломній роботі має переважно якісний характер, а тому необхідно підтвердити його відповідними критеріями. Якісне дослідження має враховувати наступні загрози дійсності: Достовірність, Переказність, Надійність і Підтверджуваність [41]. Усі критерії додатково пояснюються в наступних пунктах і як вони розглядалися:

Достовірність: Ця загроза стосується достовірності звіту. Це стосується того, як робляться посилання на твердження та цитування джерел. Важливою частиною цієї загрози є те, що вона змусила звіт зберегти претензії дійсними на момент їх подання, навіть якщо в майбутньому буде доведено, що вони недійсні.Твердження, зроблені в цьому звіті, були обґрунтовані джерелами та/або висновками.

Можливість передачі. Для того, щоб теза була застосована в інших ситуаціях, висновки тези необхідно було узагальнити, щоб зробити їх застосовними в цих інших ситуаціях. Наприклад, якщо результати стверджують, що метод дослідження є перевагою, це має бути відображено в інших дослідженнях з точки зору ефективності та загальних переваг, отриманих від методу дослідження. У цій дисертації це буде відображено у виборі мови програмування для реалізації шифрування розробниками або науковцями, яким потрібні дані щодо різних переваг мов програмування для кожного алгоритму шифрування.

Надійність. Надійність відноситься до узгодженості та повторюваності вимірювань, що гарантує, що висновки будуть послідовно повторюватися, якщо дослідження повторюватимуться з тими ж або схожими респондентами. Оскільки частина цього дослідження базувалася на нашому власному досвіді, може бути важко відтворити всі результати, оскільки вони можуть бути дуже ситуативними.

Можливість підтвердження: Можливість підтвердження гарантує, що упередження дослідника не вплинуть на результат дослідження. Наприклад, якщо ми особисто віддаємо перевагу Python над GO, існує ризик спотворення висновків дослідження на користь Python як найкращої мови програмування для шифрування.

Етичні вимоги. Етичні міркування в якісних дослідженнях зазвичай зосереджуються навколо поводження з учасниками, наголошуючи на таких принципах, як конфіденційність, пропаганда, голос учасників, особиста етика та запобігання втручанню [42].

2.2 Модель порівняння

Критерії. Модель порівняння складалася з трьох критеріїв: Продуктивність, Простота реалізації, і Криптографічні бібліотеки. Критерії використовувалися для оцінки та порівняння різних мов програмування одна з одною при використанні для шифрування. Таким чином, критерії повинні бути легкими для вимірювання та простими у використанні для порівняння між мовами програмування.

Дані про Криптографічні бібліотеки були зібрані з вивчення літератури. Дані щодо Продуктивність і Простота реалізації було зібрано шляхом реалізації алгоритмів шифрування в кількох мовах програмування, використовуючи бібліотеки шифрування мов програмування.

Продуктивність, основним критерієм, вказана швидкість шифрування. Він оцінив час, необхідний алгоритму шифрування для шифрування повідомлення. Потім виміряний час шифрування повідомлень використовувався для розрахунку пропускної здатності алгоритму. Пропускна здатність алгоритму шифрування обчислювалася як загальна кількість зашифрованого відкритого тексту, поділена на час шифрування. Це було зроблено для всіх файлів різного розміру. Пропускна здатність цього алгоритму шифрування оцінювалася за середньою пропускною здатністю цих файлів.

Причина включення Продуктивність полягала в тому, що він виступав індикатором ефективності шифрування в різних мовах. Таким чином, для цього критерію стверджувалося, що вищі швидкості означають більш ефективне шифрування.

Простота впровадження.

Простота реалізації був другим критерієм моделі порівняння. Ця категорія охоплює наші спостереження щодо простоти або складності реалізації алгоритмів шифрування в кожній мові програмування. Ми проаналізували фактори, що сприяють цій простоті чи труднощам, у тому числі наявність мовних функцій. Задокументувавши конкретні випадки простоти чи складності, разом із поясненнями додаткових елементів, ми прагнули надати вичерпний огляд нашого досвіду впровадження.

Простота реалізації спрямований на те, щоб висвітлити сильні та слабкі сторони використання різних мов програмування, як їх сприймають розробники. Ця інформація призначена для того, щоб скерувати людей, які прагнуть вивчити та використовувати ці мови для завдань шифрування, пропонуючи зазирнути в потенційний досвід, з яким вони можуть зіткнутися. Для цього критерії включають три шкали вимірювання: «Простота у використанні», «Незалежна доступність» і «Час розробки». Обидві ці шкали вимірюються шляхом позначення хорошого, поганого або середнього.

«Простота у використанні» підкреслює, наскільки легко було читати та писати код поточною мовою програмування, оскільки синтаксис не був надто складним.

«Незалежне зручність використання» стосується того, до якої міри можливо використовувати лише одну бібліотеку на мову програмування для всіх галузевих цілей, це означає, що якщо метою є шифрування, то бібліотека повинна доповнювати всі необхідніоперації з цією метою. «Час розробки» стосується часу від початку розробки до готового продукту. Зауважте, що цей критерій вимірювався з поточного досвіду авторів, суб'єктивність була неминуча.

Крипотографічні бібліотеки.

Остаточним критерієм було Криптографічні бібліотеки. Це був єдиний критерій, за яким дані були зібрані з літературних досліджень, а не з наших реалізацій. Критерій аналізував якість цих бібліотек. Основна увага була зосереджена на підтримці бібліотеками різних симетричних і асиметричних алгоритмів шифрування. Ми б перевірили як офіційну, так і зовнішню бібліотечну підтримку. Оцінюючи якість, ми досліджували вміст бібліотеки, зокрема, скільки алгоритмів шифрування та режимів блокового шифрування вона підтримує. Зокрема, режими блокового шифру допомагають змінити використання алгоритмів шифрування блокового шифру, що сприймалося як важливе, оскільки деякі режими блокового шифру сприймалися як небезпечні [43]. Крім того, ми розглянули можливість використовувати бібліотеку незалежно, не покладаючись на інші бібліотеки, наприклад, якщо бібліотека має власні методи доповнення або інші функції, специфічні для бібліотеки. Нарешті, ми перевірили, наскільки подобається бібліотека, перевіривши, скільки людей уже нею користується, наскільки доглянутою є бібліотека, перевіривши, як часто вона оновлюється, і, нарешті, якість їхньої документації бібліотек, поставивши хороший, середній або поганий бал, на основі поточного вибору дизайну для макета документації. Враховуючи, що час був обмежений, і ймовірність того, що стандартна бібліотека для мови програмування може не мати функціональної бібліотеки шифрування, це може означати, що існує зовнішня бібліотека шифрування, яка краще підходить для цієї мети. Тому ми вирішили детальніше дослідити 3 зовнішні бібліотеки на випадок, якщо стандартна бібліотека не підтримує алгоритми шифрування, а також оцінити аспекти якості, згадані раніше.

Включення підтримки криптографічної бібліотеки як критерію відображає універсальність мов програмування в шифруванні. Він сигналізує розробникам, що мова пропонує вбудовану підтримку кількох алгоритмів шифрування, забезпечуючи гнучкість для різних потреб шифрування. Можна стверджувати, що чим більше підтримують різні алгоритми шифрування, тим більше вони оптимізовані для використанняце шифрування.

2.3 Вибір бібліотек для впровадження

Вибираючи криптографічну бібліотеку для наших реалізацій, ми вирішили використовувати стандартну бібліотеку, якщо це можливо. Якщо це було неможливо, ми хотіли, щоб зовнішня бібліотека була добре відомою та добре задокументованою. Популярність та/або те, що це стандартна бібліотека, свідчить про високу ймовірність того, що люди, які обирають ту саму мову програмування, виберуть цю бібліотеку, що відображає довіру до її надійності та ефективності для шифрування. Крім того, вичерпна документація забезпечує легкість використання та обслуговування, ще більше підтверджуючи придатність бібліотеки для наших цілей.

За Python, ми використали зовнішню бібліотеку під назвою *Криптографія* який використали понад 635 000 осіб [44]. Він підтримує безліч криптографічних примітивів, включаючи симетричне та асиметричне шифрування. Він навіть може похвалитися можливістю реалізації ключових функцій деривації, цифрових підписів і безпечних кодів автентифікації повідомлень. Він містить обширну документацію з внутрішніми посиланнями, щоб полегшити навігацію різними класами шифрування та їхніми параметрами.

В GO, ми користувалися бібліотекою *Крипто* який є частиною стандартного дистрибутива Go [45]. Ця бібліотека, розроблена та підтримувана командою Go, пропонує ряд криптографічних функцій. Від симетричних і асиметричних алгоритмів до хеш-функцій, цифрових підписів і безпечних генераторів випадкових чисел. Crypto надає розробникам необхідні інструменти для створення безпечних програм. Крім того, він підтримує такі компоненти, як автентифікація повідомлень на основі хеш-кодування, отримання ключів і захист транспортного рівня.

2.4 Тестування алгоритмів

Ми реалізували тести для кожного вибраного алгоритму, створюючи все більші файли для шифрування, повторюючи 500 разів для кожного розміру файлу.

Це дало необхідну інформацію, яка стосувалася в першу чергу Продуктивність критерій, а також опосередковано Простота реалізації критерій, оскільки це означало більше роботи з мовою.

Для тестування AES і 3DES ми використовували файли розміром від 200 КБ до 200 МБ. Длятестуючи RSA, ми використовували файли розміром від 100 байт до 400 байт. Це пов'язано з тим, що RSA може шифрувати лише файли, розмір яких менший або дорівнює розміру ключа, тому найбільший можливий розмір файлу буде 512 байт, оскільки ми використовували ключ такого розміру.

2.5 Інструменти та методи для безпечного криптографічного програмного забезпечення

У цьому розділі розглядаються інструменти та методи допоміжного програмування та перевірки криптографічного програмного забезпечення. Набір інструментів і методів було розділено на дві категорії: безпечне криптографічне програмування та перевірка безпеки криптографічного програмного забезпечення.

Безпечне криптографічне програмування. Інструменти безпечного програмування криптографічного програмного забезпечення складаються з певних мов програмування, інструментів для автоматичного створення коду, криптографічних API та фреймворків.

Криптографічні мови програмування. Використання певних мов програмування не є стандартною практикою розробки безпечного програмного забезпечення. З іншого боку, експерти, такі як криптологи, зазвичай віддають перевагу своїм знанням, вираженим у власному синтаксисі предметно-орієнтованими мовами [25, 26, 27, 28].

cPLC [25] — це мова криптографічного програмування та компілятор для генерації реалізацій Java двосторонніх криптографічних протоколів, таких як Diffie Hellman. Мова введення cPCL сильно натхненна стандартною нотацією для визначення протоколів і, нібито, є першим інструментом, який можуть

використовувати криптографічно непідготовлені інженери програмного забезпечення для отримання надійних реалізацій довільних двосторонніх протоколів, а також криптографи, які хочуть ефективно реалізувати свої протоколи, розроблені на папері.

Барбоза, Мосс і Пейдж [26] працювали з мовою програмування CAO, щоб створити предметно-спеціальну мову з підтримкою криптографії та відповідний компілятор, призначений для роботи як механізм для передачі та автоматизації експертних знань криптографів у формі, доступній для будь-кого, хто пише засоби безпеки.

Свідоме програмне забезпечення. CAO дозволив описувати програмне забезпечення для криптографії з еліптичною кривою (ECC) у спосіб, близький до оригінальної математики, а його компілятор дозволяв автоматичне створення виконуваного коду, що конкурує з реалізаціями, оптимізованими вручну. Нещодавно система типізації CAO (набір правил для призначення типів змінним, виразам, функціям та іншим конструкціям на мові програмування) була офіційно специфікована, підтверджена та реалізована таким чином, щоб підтримувати реалізацію зовнішніх інтерфейсів для компіляції CAO та інструментів формальної перевірки [29]. Нарешті був випущений компілятор для CAO [30]. Інструмент використовує високорівневі специфікації криптографічного алгоритму та перетворює їх на реалізацію C за допомогою низки перетворень та оптимізацій з урахуванням безпеки.

Cryptol [27] є функціональною предметно-орієнтованою мовою для визначення криптографічних алгоритмів. Мова працює таким чином, що реалізація алгоритму нагадує його математичну специфікацію. Cryptol може виробляти код C, але його основна мета полягає в підтримці виробництва формально перевірених апаратних реалізацій. Cryptol підтримується набором інструментів для формальної специфікації, впровадження та перевірки криптографічних алгоритмів [31].

Останньою роботою, яку варто згадати, є доменно-орієнтована мова для обчислень із зашифрованими даними [28], яку можна назвати вбудованою

доменно-специфічною мовою для безпечних хмарних обчислень (EDSLSCC). Мова була розроблена для безпечних хмарних обчислень, і передбачається, що вона дозволить програмістам розробляти код, який працює на будь-якій безпечній платформі виконання, що підтримує операції, що використовуються у вихідному коді.

Автоматизована генерація коду.

Автоматизоване створення коду не є звичайною практикою безпечного кодування. Незважаючи на це, його успішно використовували для створення криптографічного вихідного коду. Недавня робота Алмейди та інших [32] розширює сертифікований компілятор CompCert механізмом міркувань про програми, що покладаються на надійні бібліотеки, а також перевірку перекладу на основі механізму анотації CompCert. Цих механізмів, разом із надійною бібліотекою для арифметичних операцій і екземплярів ідеалізованих операцій, виявилося достатньо, щоб зберегти як коректність, так і властивості безпеки вихідного коду на C під час перекладу до його скомпільованого виконуваного файлу збірки.

Інша категорія інструментів перетворює код, додаючи безпечні елементи керування. Дві недавні роботи Мосса та інших [33, 34] описують автоматичне введення контрзаходів диференціального аналізу потужності (DPA) на основі методів маскування. Інша робота Agosta та ін. [35] виконує орієнтований на безпеку аналіз потоку даних і містить інструмент на основі компілятора для автоматичного створення необхідного набору контрзаходів маскування.

Розширені криптографічні API.

Інтерфейс прикладного програмування (API) — це набір сигнатур, які експортуються та доступні користувачам бібліотеки чи фреймворку [36]. У цьому розділі показано криптографічні бібліотеки, які виходять за межі звичайного криптографічного API, або представляючи диференційовану архітектуру програмного забезпечення, або пропонуючи послуги різними способами, нагадуючи фреймворки програмного забезпечення.

Два недавніх дослідження Браги та Насіменто [37] та Гонсалеса та інших [38] показали, що, незважаючи на різноманітність криптографічних бібліотек, що спостерігається в академічній літературі, ці бібліотеки не обов'язково є загальнодоступними або готовими для інтеграції з програмним забезпеченням сторонніх розробників. Незважаючи на багато претензій щодо загальності, майже всі вони були побудовані з урахуванням вузького охоплення та віддавали пріоритет науковим інтересам, а не іншим.

Стандартна криптографія. Крім того, портативність на сучасні платформи, такі як Android, зазвичай була проблемою, якою зазвичай нехтували в криптографічних бібліотеках [37]. Настільки, що сучасні платформи за замовчуванням пропонують звичайному програмісту лише кілька варіантів звичайних криптографічних сервісів [38].

Cryptlib Гутмана [39] — це і криптографічна бібліотека, і API, які підкреслюють дизайн внутрішньої архітектури безпеки для криптографічних служб, які обертаються навколо об'єктно-орієнтованого API, забезпечуючи багатошаровий дизайн із повною ізоляцією внутрішніх елементів архітектури від зовнішнього коду. Тим не менш, Cryptlib все ще виглядає як криптографічний API загального призначення (хоча і об'єктно-орієнтований), представляючи набір сервісів, інкапсульованих об'єктами. Крім того, це не усуває потребу в криптографічній експертизі, особливо під час створення служб. Проблеми, з якими стикаються під час використання Cryptlib у реальних програмах, уже задокументовані [5].

NaCl, запропонований Бернштейном, Ланге та Швабе [40], розшифровується як «бібліотека мереж і криптографії». NaCl пропонує, як окремі операції, композиції криптографічних послуг, які раніше виконувалися в кілька кроків. Наприклад, з NaCl автентифіковане шифрування є однією операцією.

Функція

$$c = \text{crypto\_box}(м, н, пк, ск),$$

де *ск* є закритим ключем відправника, *шт* є відкритим ключем одержувача, *м* це повідомлення і *n* є *nonce*, інкапсулює весь сценарій шифрування з

автентифікацією відкритого ключа з точки зору відправника. Вихід автентифікується та шифрується за допомогою ключів від відправника та одержувача.

Функція *crypto_box* має свої переваги. З більшістю криптографічних бібліотек писати незахищені програми легше, ніж писати програми, які включають належну автентифікацію, оскільки додавання підписів автентифікації до зашифрованих даних неможливо виконати без додаткової роботи з програмування. На жаль, NaCl не пропонує рішення для одного з найпоширеніших джерел помилок у використанні криптографії звичайними програмістами, а саме генерації та керування nonces. NaCl залишає генерування та керування nonce виклику функції, під аргументом, що nonce інтегровані в протоколи високого рівня різними способами [40]. Цю проблему частково вирішив libadacrypt [41], стійкий до неправильного використання криптографічний API для мови Ada.

Криптографічні рамки.

Відповідно до Фаяда та Шмідта [42], фреймворк програми — це багаторазово використовуваний, "напівзавершений" програмний додаток, який можна спеціалізувати для створення користувальницьких програм. Джонсон [43] стверджує, що, на відміну від бібліотек класів, фреймворки націлені на конкретні домени додатків, оскільки фреймворк є багаторазово використовуваною конструкцією системи або скелетом, який може бути налаштований розробником програми, забезпечуючи варіанти повторного використання. Фреймворки додатків для криптографії мають потенціал для зменшення зусиль кодування та помилок для складних випадків використання, таких як перевірка сертифікації, керування IV та автентифіковане шифрування.

Зараз існує багато бібліотек SSL, які спрямовані на полегшення інтеграції SSL у програми. Однак, як показали недавні дослідження Георгієва та ін. [12] і Фахла та ін. [13], багато з цих бібліотек або зламані, або схильні до помилок, тому неправильна перевірка SSL вважається широко поширеною проблемою, і просте спрощення бібліотек SSL або навчання розробників безпеки SSL не вважаються

вирішенням проблеми [24]. Натомість ідеальним рішенням було б дозволити розробникам правильно використовувати SSL без зусиль з кодування, таким чином запобігаючи порушенню перевірки SSL через незахищені налаштування.

Нещодавня робота Fahl та інших [24] пропонує зміну парадигми у використанні SSL: замість того, щоб дозволяти розробникам впроваджувати власний код зв'язування SSL, основні шаблони використання SSL мають надаватися службами операційних систем, які можна додавати до програм через конфігурацію, а не впровадження. Дотримуючись мети цієї нової парадигми, запропонована структура SSL автоматизує етапи перевірки сертифікатів, а також вносить зміни в те, як SSL зараз використовується мобільними програмами [24]. За допомогою параметрів конфігурації він замінює необхідність написання коду SSL майже в усіх випадках використання, перешкоджаючи небезпечним налаштуванням. Крім того, під час розробки програмного забезпечення він робить різницю між пристроями розробників і пристроями кінцевих користувачів, дозволяючи самопідписані сертифікати. Нарешті, проблеми з SSL, які можуть призвести до атак MITM, надійно інформуються через неминучі попередження ОС.

Ідея надання функцій, пов'язаних з криптографією, як послуг операційної системи, не є новою і вже була запропонована з міркувань продуктивності [44, 45]. З точки зору програміста, нововведення в структурі SSL полягає в аспекті зручності використання, що полягає у наявності високорівневої функціональності SSL замість примітивних криптографічних функцій у якості конфігурованих служб ОС.

Криптографічна перевірка безпеки.

Засоби перевірки безпеки включають засоби статичного та динамічного аналізу (тестування). Інструменти статичного аналізу виконують синтаксичний і семантичний аналіз вихідного коду без його виконання. З іншого боку, інструменти тестування (динамічного аналізу) виконують динамічну перевірку очікуваної поведінки програми на кінцевому наборі тестових випадків, відповідним чином вибраних із звичайно нескінченної області виконання.

Інструменти статичного аналізу.

Використання автоматичних інструментів для статичного аналізу коду є досить поширеним у безпечному програмуванні та може вважатися стандартною найкращою практикою. Здається цілком природним, що практики безпечного кодування також були перенесені на криптографічне програмування. Насправді більшість стандартів кодування [46, 47] містять прості правила щодо використання криптографії, які можна легко автоматизувати звичайними інструментами статичного аналізу. Наприклад, такі інструменти можуть попереджати про використання застарілих криптографічних функцій: використання MD5, SHA-1 і DES може бути автоматично виявлено як неправильне кодування, а також використання коротких ключів (наприклад, 128-бітний ключ для AES або 512-бітний ключ для RSA).

На жаль, існують криптографічні проблеми, які неможливо виявити за допомогою звичайних інструментів і простих методів [48]. Ці проблеми були вирішені передовими інструментами в академічних дослідженнях [11, 14, 49, 50, 51, 52]. Інструмент CryptoLint [11] бере необроблений двійковий файл Android, розбирає його та перевіряє типові криптографічні зловживання. Cryptography Misuse Analyzer (CMA) [14] — це інструмент аналізу для програм Android, який визначає попередньо визначений набір уразливостей криптографічного зловживання через виклики API.

Side Channel Finder (SCF) [49] — це інструмент статичного аналізу для виявлення каналів синхронізації в реалізаціях Java криптографічних алгоритмів. Ці бічні канали часто викликані розгалуженням потоку керування з умовами розгалуження залежно від атакованих секретів. Інструмент Sleuth [50] — це інструмент автоматичної перевірки, підключений до компілятора LLVM, який може автоматично виявляти кілька прикладів класичні підводні камені в реалізації контрзаходів DPA. CacheAudit [51] — це інструмент аналізу для автоматичного виявлення каналів на стороні кешу. Він приймає двійковий код і конфігурацію кешу та отримує гарантії безпеки на основі спостереження за станом кешу, слідами звернень і промахів і часом виконання. CAOVerif [52] — це

інструмент статичного аналізу для CAO [30], який використовувався для перевірки коду NaCl [40].

Важливо диференціювати сферу застосування цих засобів. У той час як CAOVerif, SCF, Sleuth і CacheAudit працюють усередині криптографічної реалізації, шукаючи певні типи побічних каналів та інші проблеми, CryptoLint і CMA працюють поза межами криптографічних API і виявляють відомі зловживання криптографічними бібліотеками.

Тести функціональної безпеки.

У криптографічній валідації [53] тестові вектори є тестовими випадками для криптографічних функцій безпеки і роками використовувалися для перевірки криптографічних реалізацій, переважно для постконструювання сертифікації продукту. Відповідно до Браги та Шваба [54], тестові вектори також можна використовувати для перевірки під час розробки програмного забезпечення та виконувати автоматизованими приймальними тестами. Приймальні тести перевіряють, чи система задовольняє критерії прийнятності, перевіряючи її поведінку на відповідність вимогам [36].

Тестові вектори є хорошими тестами на прийняття, оскільки вони стоять посередині між криптологами та розробниками [54]. Тестові вектори — це тестові випадки, надані криптологами, які замінюють специфікації вимог, а також незалежні перевірки того, що криптографічне програмне забезпечення правильно реалізувало вимоги. На основі тестових векторів розробники можуть писати модульні тести до написання криптографічного коду для тестування. Потім тестові вектори можна використовувати для оцінки правильності реалізацій, а не їх безпеки. Функціональна правильність є умовою безпеки, оскільки неправильні реалізації є ненадійними та небезпечними.

Існують загальнодоступні тестові вектори (наприклад, [53]), які побудовані за допомогою статистичної вибірки. Успішна перевірка за допомогою статистичної вибірки передбачає лише вагомі докази, але не абсолютну впевненість у правильності. Щоб бути статистично доречними, навіть невеликі набори даних містять тисячі зразків, тому потрібна автоматизація. Після того, як

автоматизовані тести стануть доступними для криптографічних реалізацій, можна буде внести подальші вдосконалення коду, щоб вирішити такі галузеві проблеми, як оптимізація продуктивності, енергоспоживання та захист від атак по бічних каналах та інших уразливостей. Навіть після всіх цих перетворень приймальні тести зберігають довіру, надаючи вагомі, хоча й неформальні, докази правильності [54].

Інструменти тестування безпеки.

Тестування безпеки фокусується на перевірці того, що програмне забезпечення захищено від зовнішніх атак [36]. Зазвичай тестування безпеки включає перевірку проти неправильного використання та зловживання програмним забезпеченням або системою [36]. Тести безпеки такі ж різноманітні, як і кількість вразливостей, які можна використовувати, і їх можна вважати загальноприйнятою практикою розробки безпечного програмного забезпечення. Варто згадати поточну галузеву практику тестування криптографії для веб-додатків [55, 56, 57] і криптографічних модулів [58]. У цьому розділі представлено три типи тестів і відповідні інструменти, які, як вважають, добре відображають поточні тенденції щодо тестів безпеки для криптографічного програмного забезпечення.

У безпеці веб-додатків для виявлення неправильної конфігурації HTTPS використовувалися автоматизовані тести для з'єднань SSL [56]. Нещодавно цей тип тестування перемістився в мобільні програми за допомогою MalloDroid [13], інструменту для виявлення потенційних уразливостей проти атак MITM реалізацій SSL у програмах Android. MalloDroid виконує статичний аналіз коду скомпільованих програм для Android, щоб досягти трьох цілей безпеки: отримати дійсні URL-адреси HTTP(S) із декомпільованих програм, аналізуючи виклики мережевого API; перевірити дійсність сертифікатів SSL усіх витягнутих хостів HTTPS; і ідентифікувати програми, які використовують ненормальне використання SSL (наприклад, містять нестандартні менеджери довіри, фабрики сокетів SSL або дозвільні верифікатори імен хостів).

Padding Oracle Exploitation Tool [20] (POET) — це інструмент, який автоматично знаходить і використовує оракули заповнення. Тести проти атак padding oracle зазвичай важко виконувати вручну через велику кількість ітерацій (від багатьох сотень до кількох тисяч), що виконуються для розшифровки одного блоку зашифрованого тексту [19]. Як тільки оракул заповнення буде виявлено, його експлуатація може бути автоматизована документовані алгоритми [22]. POET успішно використовувався для використання оракулів заповнення у веб-технологіях [20] (наприклад, шифрування XML [21] і ASP.NET [22]).

Нарешті, ін’єкція помилок – це різновид фазз-тестування, яке можна використовувати для тестування безпеки криптографічних пристроїв [59]. Фуз-тестування [36] — це спеціальна форма випадкового тестування (де тестові випадки генеруються випадковим чином), спрямованого на зламати програмне забезпечення. Атака з ін’єкцією помилок, автоматизована за допомогою інструменту Fault Injection Attack Tool (FIAT), — це тип атаки побічного каналу, що реалізується шляхом ін’єкції навмисних (зловмисних) помилок у криптографічний пристрій і спостереження за відповідними помилковими виходами. Було показано [59], що атаки з ін’єкцією несправності потребують недорогого обладнання та короткого проміжку часу порівняно з тестуванням на побічні канали споживання електроенергії або електромагнітних випромінювань, які зазвичай потребують дорогого налаштування.

3 ОБГРУНТУВАННЯ ВЛАСНОГО РІШЕННЯ

3.1 Складання інструментарію для програмування криптографічного програмного забезпечення

Згідно з Макгроу [3], не існує абсолютно безпечного програмного забезпечення в тому сенсі, що довіра до так званого безпечного програмного забезпечення завжди пов’язана з методами гарантії, які використовуються, і має бути виправдана структурованим використанням відповідних інструментів і методів.

Таким чином інструменти та методи, які обговорювалися досі, можна об’єднати в просту послідовність кроків для програмування безпечного криптографічного програмного забезпечення. Така послідовність кроків проілюстрована на рисунку 3.1 і, якщо її застосувати на практиці, зможе підвищити довіру до кінцевого криптографічного програмного забезпечення, надаючи вагомі докази безпеки. Цікаво спостерігати, що жоден із інструментів і технік не може поодинці задовольнити весь процес, який можна охопити лише спільним використанням кількох інструментів і технік.

Послідовність складається з трьох основних кроків, які можна виконувати ітеративно: програмування та перевірка бібліотеки, програмування та перевірка криптографічного програмного забезпечення та тестування криптографічного програмного забезпечення. На кожному кроці можна розмістити відповідні інструменти та методи для досягнення необхідної впевненості. Незважаючи на те, що багато інструментів і методів можна використовувати в кілька етапів, головну вигоду можна отримати, коли вони використовуються в конкретних (бажаних) місцях. Наприклад, безпечні мови, безпечна компіляція та безпечна генерація коду підходять для програмування криптографічних бібліотек. З іншого боку, інструменти статичного аналізу, (автоматичні) функціональні тести та фреймворки підходять для програмування криптографічного програмного забезпечення. Нарешті, динамічний аналіз, ін’єкція помилок, тести на

проникнення (SSL), тести оракулів заповнення та моніторинг краще підходять для перевірки.

Традиційно розробники програмного забезпечення вважають криптографію одним із найважчих для розуміння засобів контролю безпеки. Програмісти звикли покладатися на прості API, щоб забезпечити ефективність криптографії над функціями, пов'язаними з безпекою, тоді як правильність її внутрішніх функцій завжди сприймалася як належне. Однак зростаюча складність програмного забезпечення негативно вплинула на криптографію, що призвело до її неправильного використання програмістами та, зрештою, призвело до небезпечного програмного забезпечення.

Настав час придумати нові способи побудови сучасного криптографічного програмного забезпечення, шукаючи переваги останніх досягнень у підтримці інструментів для безпеки програмного забезпечення. Головний висновок полягає в тому, що не існує остаточного інструменту для програмування безпечного криптографічного програмного забезпечення. Натомість лише добре продуманий набір інструментів здається, здатний охопити весь ландшафт програмування криптографічного програмного забезпечення.

Програмування захищеного криптографічного програмного забезпечення, як це запропоновано в цьому тексті, є новою дисципліною в практиці програмування криптографічного програмного забезпечення. На момент написання статті звичайному програмісту було доступно лише кілька інструментів і технік. Більшість інструментів є прототипами, що представляють науковий інтерес, і не можуть конкурувати з комерційними, готовими інструментами безпеки. З іншого боку, індустрія програмного забезпечення має успішну історію інновацій у наданні нових технологій забезпечення якості пересічному програмісту. Тож є надія, що в у найближчому майбутньому нове покоління інструментів для безпеки програмного забезпечення може принести в щоденну практику всі академічні досягнення, згадані досі.

Поточне дослідження вказує на кілька можливостей для майбутньої роботи. Моделювання процесів розробки безпечного криптографічного програмного

забезпечення може допомогти знайти кращі способи використання всіх цих інструментів. Крім того, проведення експериментів із спеціальними інструментами безпеки під час розробки криптографічного програмного забезпечення може надати засіб для оцінки їх ефективності щодо зменшення криптографічної вразливості. Розробка кращих абстракцій програмного забезпечення для полегшення використання просунутих криптографічних концепцій звичайними програмістами є ще одним альтернативним напрямком дослідження. Нарешті, збір і аналіз даних щодо реальних звичок програмування розробників, відповідальних за кодування криптографічних примітивів, може спонукати до подальших досліджень, щоб краще зрозуміти, як програмісти зловживають криптографією.

3.2 Продуктивність

У цьому підрозділі представлено результати наших реалізацій. Кожен розділ охоплює різні алгоритми шифрування, показуючи витрати часу на всі наші тести та загальну пропускну здатність для кожної мови. Спочатку наводяться результати для AES, потім для RSA і, нарешті, для 3DES.

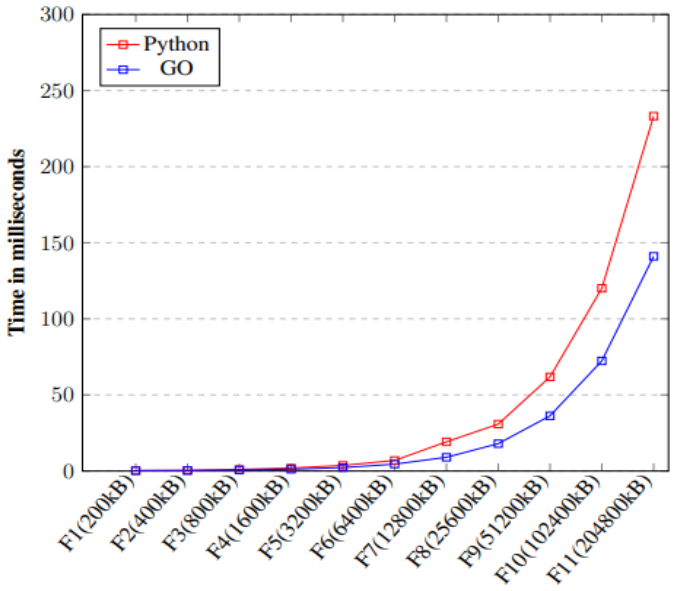

Рисунок 3.1 – Витрата часу на шифрування AES

AES. Реалізацію шифру AES перевіряли на одинадцяти різних відкритих текстових файлах розміром від 200 КБ до 204800 КБ. Кожен файл було зашифровано 500 разів за допомогою того самого 128-бітного ключа, щоб отримати середній час шифрування для кожного файлу.

Результати на рисунку 3.1 показати, як збільшується час шифрування зі збільшенням розміру файлу. Для кожного розміру файлу реалізація AES із використанням бібліотеки Python повільніша, ніж реалізація, яка використовує бібліотеку GO.

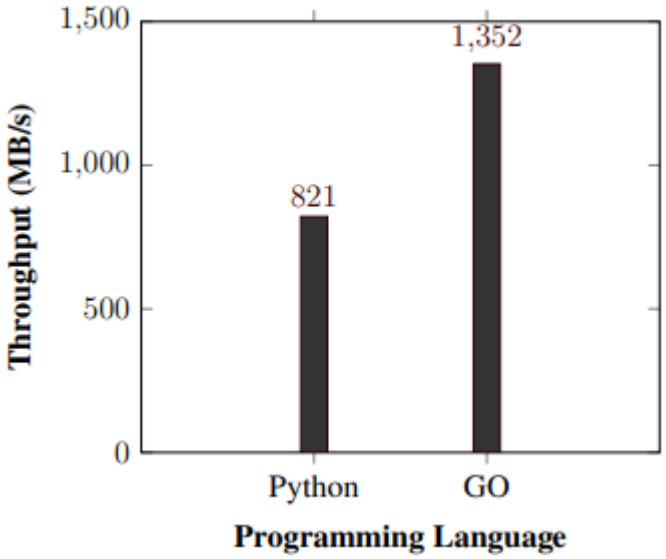

Рисунок 3.2 – Середня пропускна здатність шифрування AES

Результати на рисунку 3.2 показують, що пропускна здатність реалізації GO набагато вища, ніж пропускна здатність реалізації Python з точки зору часу обробки. Середній час обробки для всіх реалізацій GO становить 1352 МБ/с, тоді як середня пропускна здатність Python становить 821 МБ/с.

RSA. Реалізацію шифру RSA було перевірено на 4 різних файлах відкритого тексту розміром від 100 до 400 байт. Тестовий файл у цьому випадку має бути меншим за закритий ключ. Кожен файл було зашифровано 500 разів за допомогою 4096-бітного ключа, щоб отримати середній час шифрування для кожного файлу.

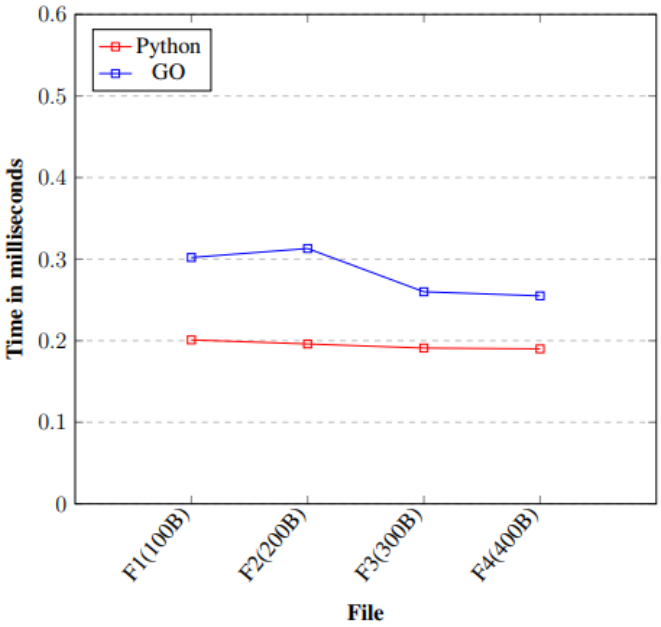

Рисунок 3.3 – Витрата часу на шифрування RSA

Результати рисунку 3.3 показати час, необхідний для шифрування за допомогою реалізацій RSA, як для бібліотеки Python, так і для бібліотеки GO, не сильно змінюється навіть із збільшенням розміру файлу. Також зрозуміло, що для кожного розміру файлу реалізація Python швидше, ніж реалізація GO.

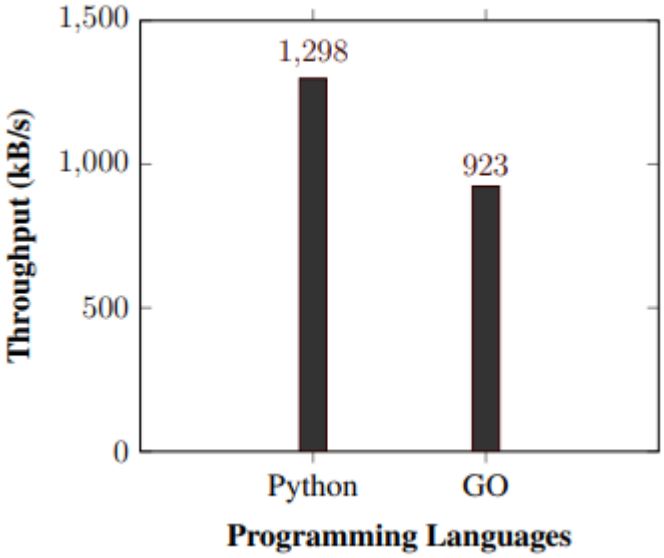

Рисунок 3.4 – Середня пропускна здатність шифрування RSA

Результати на рисунку 3.4 показують, що пропускна здатність реалізації Python вища, ніж пропускна здатність реалізації GO з точки зору часу обробки.

Середній час обробки для всіх реалізацій GO становить 923 кБ/с, тоді як середня пропускна здатність Python становить 1298 кБ/с.

3DES. Реалізацію шифру 3DES було перевірено на 11 різних файлах відкритого тексту розміром від 200 КБ до 204800 КБ. Кожен файл було зашифровано 500 разів за допомогою 128-бітного ключа, щоб отримати середній час шифрування.

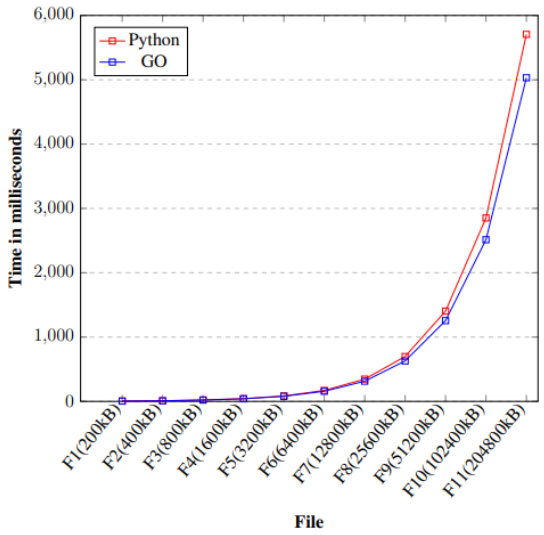

Рисунок 3.5 – Витрата часу на шифрування 3DES

Результати на рисунку 3.5 показують, що реалізації Python і Go мають однакову швидкість шифрування для 3DES. Тим не менш, впровадження в GO трохи швидше.

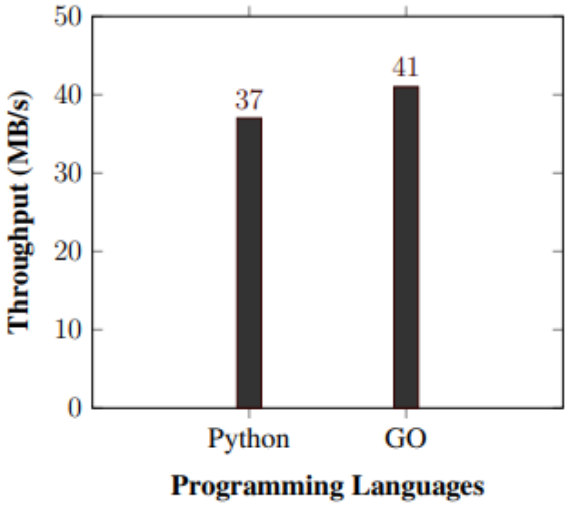

Рисунок 3.6 – Середня пропускна здатність шифрування 3DES

Результати на рисунку 3.6 показують, що пропускна здатність реалізацій GO трохи вища, ніж пропускна здатність реалізацій Python, з точки зору часу обробки. Середній час обробки для всіх реалізацій GO становить 41 МБ/с, тоді як середня пропускна здатність Python становить 37 МБ/с.

3.3 Простота впровадження

Цей підрозділ охоплює наш досвід роботи з алгоритмами шифрування та впровадження їх на різних мовах програмування. Кожен підрозділ охоплює іншу мову та пояснює, що нам здається важким, що зрозумілим і які елементи сприяють цим твердженням.

P y t h o n

Python була простою мовою для роботи. Його синтаксис був простим, що полегшувало читання та запис, оскільки він нагадував англійський текст. Використання відступів для визначення обсягу зробило код більш читабельним, оскільки використовувалося менше спеціальних символів. Виведення типу полегшило потребу в оголошенні змінних. Тим не менш, було важко оцінити типи змінних під час використання кількох викликів різних функцій, де повернуті значення зберігалися в змінних, виведених за типом. Однак визначення типу виявилося більш корисним, ніж перешкодою. Загалом, робота та читання Python були хорошими д о б р е .

Працювати з бібліотеками Python було просто. Процес потребує лише доступу до терміналу для завантаження будь-яких бібліотек (також званих «пакетом») за допомогою менеджера пакетів Python. Після завантаження потрібної бібліотеки можна було відкрити новий файл Python і імпортувати будь-що з цих бібліотек. Наявність цього простого способу використання нестандартних бібліотек допомогла прогресу в розробці.

Також можна було використовувати ту саму бібліотеку для всіх завдань шифрування, оскільки в цю бібліотеку входили всі необхідні функції. Це полегшило створення наших реалізацій, оскільки ми не були змушені вчитися та

покладатися на інші бібліотеки. Підсумовуючи, незалежне використання бібліотеки криптографії було д о б р е .

Під час впровадження AES це було просто з бібліотекою криптографії, оскільки враховувалося, що вона дуже добре задокументована. Ми згенерували ключ, вектор ініціалізації для режиму роботи та створили екземпляр шифру. Після цього ми використали вбудовану функцію для доповнення відкритого тексту таким чином, щоб його можна було зашифрувати. Оскільки ця процедура була стандартною для блочних симетричних шифрів, реалізація 3DES стала тривіальною, оскільки для її роботи потрібні були лише деякі зміни змінних. Здавалося, що обрана бібліотека має хороший рівень абстракції, де блочний шифр можна змінити так, ніби змінюється лише одна змінна, дотримуючись простоти використання цієї бібліотеки.

Реалізація RSA зайняла більше часу, оскільки згенерований приватний ключ і відкритий ключ складалися не з байтів, а з власних типів класів, які мали кілька функцій. Документація типів класів здавалася розпорошеною, що ускладнювало з'ясування того, як витягти приватний і відкритий ключі, а потім використовувати їх для шифрування та дешифрування. Тим не менш, загальний час розробки для трьох реалізацій був швидким, кожна з них займала менше чотирьох годин, що дало їй оцінку д о б р е .

Таблиця 3.1 – Результати Python для простоти впровадження

Criteria	Criteria Value
Ease of Implementation	*Good, Medium* or *Bad*
Ease-of-Use: ➔ How easy it was to read and write the code.	**Good**
Independent Usability: ➔ The library could independently fulfill the encryption process without dependencies on other libraries for implementation.	**Good**
Development Time: ➔ Evaluate whether the implementations were completed quickly.	**Good**

GO був простим і швидким для вивчення. Подібно до Python, GO був дуже

читабельним і мав такі функції, як визначення типу, але GO також нагадував мову програмування C із циклами for, синтаксисом оголошення списків і обробкою помилок. Наприклад, у C ви повинні явно обробляти помилки, виловлюючи помилку та перевіряючи, чи помилка порожня чи ні. Цикли for мають той самий синтаксис, за винятком того, що цикл GO розширено для більшої функціональності та замінює цикл while на C. Загалом GO більше зосереджувався на змінному вмісті, що означає суворішу декларацію. Наскільки це доречно, залежить від прихильності людини до мови програмування C.

Оголошення та ініціалізація змінних у GO спочатку виглядало інакше, оскільки специфікатор типу було розміщено після назви змінної. Однак у GO є метод скороченого оголошення, який зменшує кількість символів, що використовуються в рядку коду. Скорочена декларація в поєднанні з використанням виведення типу ще більше покращила читабельність і полегшила написання коду, якщо звертати увагу на те, який тип буде виведено для змінних.

Зважаючи на все, GO має такі функції, які покращують читабельність, як-от визначення типу та коротке оголошення. Однак обробка помилок має тенденцію перевантажувати код, тим самим ускладнюючи читабельність. Це зробило обробку помилок більш стомлюючою для написання коду, оскільки вимагало ретельної перевірки помилок для кожного виклику функції. Тому робота з ГО і читання були Середній.

Під час створення реалізацій ми виявили, що криптобібліотека забезпечує майже всі необхідні функції та властивості. Однак однією нестачею була можливість серіалізації та десеріалізації закритого ключа у формат, який підходить для зберігання та пошуку. Тому для виконання цього завдання потрібна була ще одна бібліотека. Крім того, Crypto не містив методів заповнення, що означало, що потрібно було або включити іншу бібліотеку, або створити схему з нуля, щоб мати можливість заповнити відкритий текст. Отже, незалежне використання бібліотеки Crypto було класифіковано як Середній.

Процес впровадження був простим, якщо використовувати разом із документацією GO, яка також вказує на одну з функцій GO, фрагменти. У GO ви

створюєте блоковий шифр, використовуючи ключ і випадкове число в певному діапазоні. Нарешті, ви вибираєте режим роботи для блокового шифру та шифруєте відкритий текст за допомогою нього. Завдяки модульній конструкції для алгоритмів блокового шифрування, точніше об'єктно-орієнтованій конструкції, реалізація 3DES була неймовірно простою. Реалізація 3DES ідентична реалізації AES, за винятком того, що алгоритм було переключено на 3DES, це був прагматичний весь процес.

Впровадження шифру RSA було процесом проб і помилок. Це пов'язано з тим, що існувала певна невизначеність щодо використання приватного та відкритого ключів для шифрування та дешифрування. Документація щодо реалізації RSA була хорошою, але все одно потрібен був час, щоб зрозуміти та переконатися, що всі типи змінних правильні для кожного кроку процесу шифрування та дешифрування. Наприклад, щоб використовувати відкритий ключ для функцій, нам потрібно було перемикатися між використанням екземпляра відкритого ключа та використанням покажчика на цей відкритий ключ. Тим не менш, загальний час розробки для трьох реалізацій був швидким, кожна з них займала менше чотирьох годин, що дало їй оцінку д о б р е .

Результати за критерієм «Простота впровадження» для GO зведені в таблицю 3.2.

Таблиця 3.2 – Результати GO для простоти впровадження

Criteria	Criteria Value
Ease of Implementation	*Good*, *Medium* or *Bad*
Ease-of-Use: ➔ How easy it was to read and write the code.	**Medium**
Independent Usability: ➔ The library could independently fulfill the encryption process without dependencies on other libraries for implementation.	**Medium**
Development Time: ➔ Evaluate whether the implementations were completed quickly.	**Good**

3.4 Криптографічні бібліотеки

Цей підрозділ охоплює зібрані дані щодо криптографічних бібліотек. У кожному розділі описано, що пропонують бібліотеки кожної мови програмування з точки зору доступних алгоритмів шифрування, режимів блокового шифрування та інших функцій і аспектів, які впливають на якість.

На сьогоднішній день Python не має стандартної бібліотеки для шифрування, яка супроводжується під час встановлення Python, але Python має інші криптографічні функції, зокрема хеш-функції. Зважаючи на це, шифрування в Python залежить від зовнішніх бібліотек. Щоб зрозуміти здатність Python шифрувати дані, було досліджено такі три бібліотеки: PyCryptodome, PyNaCi та Cryptography [46, 47, 48]. Усі бібліотеки підтримують криптографію з відкритим ключем (асиметричні алгоритми) та криптографію з закритим ключем (симетричні алгоритми), які обидві накопичуються в 22 різних алгоритмах шифрування. Усі бібліотеки включають п’ять стандартних режимів роботи блокового шифру (ECB, CBC, CTR, OFB, CFB). Деякі бібліотеки включають інші режими роботи, що дає загалом 14 різних режимів. За винятком того, що всі бібліотеки Python разом підтримують різні схеми шифрування.

Вони надають багато додаткових можливостей, що робить можливим використання більшості цих бібліотек незалежно від інших бібліотек. Ці особливості:

– аутентифіковане шифрування;

– ключові похідні функції;

– алгоритми обміну ключами;

– коди аутентифікації повідомлень;

– хеш-функції;

– цифровий підпис;

– функції постійного часу;

– обгортка ключів;

– схеми прокладки;

– двофакторна аутентифікація;

– серіалізація відкритого ключа.

AES має додаткові блокові режими роботи, призначені для автентифікації, зокрема CCM, GCM, SIV і OCB (враховані в попередньому номері для блокових режимів).

Технічне обслуговування також є важливою темою для бібліотеки, яка суттєво допомагає користувачеві під час розробки. Наразі найстаріший випуск для будь-якої зі згаданих бібліотек вийшов 7 січня 2022 року, тоді як інші бібліотеки внесли оновлення протягом цього року (2024). Кількість людей, які користуються цими бібліотеками, коливається щонайменше від 90,7 тис. до 647 тис. осіб. Користувачі PyNaCi в цьому питанні не були виявлені, тому надані номери стосуються інших бібліотек [49, 50, 44]. Нарешті, бібліотеки добре задокументовані на своїх оригінальних веб-сайтах, надаючи інформацію та додаткові приклади коду або фрагменти коду для ідеального використання алгоритмів у відповідних місцях [46, 47, 48]. Це призвело до а д о б р е оцінка якості документації.

Результати для критерію «Криптографічні бібліотеки» для Python зведені в таблицю 3.3.

Таблиця 3.3 – Результати Python для криптографічних бібліотек

Criteria	Criteria Value
Cryptographic libraries	Number, *Good*, *Medium*, or *Bad*
Number of encryption algorithms:	22
Number of block cipher modes:	14
Documentation quality: ➜ Assessing satisfaction with the instructions in the libraries	**Good**
Number of features ➜ List the number of features for additional functionality supplied by the libraries	11

У GO є стандартна бібліотека, яка включає алгоритми шифрування. Стандартна бібліотека називається crypto і включає основні алгоритми шифрування, зокрема DES/3DES, AES, RSA тощо. Загальна кількість алгоритмів шифрування (симетричних і асиметричних) становить 9 різних алгоритмів. Загалом включено 5 режимів блокового шифрування (CBC, CFB, GCM, CTR, OFB), за винятком режиму ECB. Не так багато інших добре відомих бібліотек для шифрування в GO було створено, однак однією помітною бібліотекою є бібліотека x/crypto з додатковими криптографічними функціями. Хоча x/crypto є нестандартною бібліотекою, вона відповідає тим самим суворим стандартам якості, що й стандартна бібліотека, оскільки команда GO розробляє її, але вони мають менші вимоги до сумісності, оскільки вона не є частиною основного дерева GO [51]. x/crypt також містить 5 нових алгоритмів шифрування, які дозволяють використовувати загалом 14 різних алгоритмів шифрування. І crypt, і x/crypt також мають додаткові функції, серед яких:

– аутентифіковане шифрування;

– ключові похідні функції;

– алгоритми обміну ключами;

– повідомлення автентифікації повідомлень;

– хеш-функції;

– цифрові підписи;

– функції постійного часу;

– реалізації сервер/клієнт SSH;

– підтримка TLS/SSL;

– сертифікати X.509;

– генератор випадкових чисел.

Бібліотеки GO обслуговуються самою GO та оновлюються під час нових випусків GO. З огляду на модель порівняння, використання цих бібліотек не можна помітити, оскільки, принаймні крипто, завжди включено в GO. Нові випуски GO за своєю суттю вказуватимуть на новий випуск криптобібліотеки, поточний випуск було опубліковано 3 квітня 2024 року. Хоча використання

бібліотеки неможливо визначити, документацію бібліотеки можна обговорити. Документація GO розміщена на веб-сайті GO і є легкодоступною, однак якість документації можна покращити. Документація надає гарну інформацію про те, як використовувати алгоритми шифрування, і існують приклади коду, але не для кожної функції. Криптобібліотека розміщує більшість прикладів коду в «режимах блочного шифрування» та не містить прикладів на кожній відповідній сторінці документації алгоритму шифрування. Це призвело до "Середній" оцінка якості документації.

Таблиця 3.4 – Результати GO для криптографічних бібліотек

Criteria	Criteria Value
Cryptographic libraries	Number, *Good*, *Medium*, or *Bad*
Number of encryption algorithms:	14
Number of block cipher modes:	5
Documentation quality: ➔ Assessing satisfaction with the instructions in the libraries	**Medium**
Number of features ➔ List the number of features for additional functionality supplied by the libraries	11

3.5 Аналіз ефективності

Пропускна здатність різних алгоритмів, представлених у розділі були надані як індикатор ефективності алгоритму для кожної мови програмування. Було очевидно, що жодна мова не була переважно найефективнішою. Для AES GO має найвищу пропускну здатність, у результаті чого GO становить 64. На 7% ефективніше, ніж Python. Для RSA Python має найвищу пропускну здатність, у результаті чого Python становить 40. На 6% ефективніше, ніж GO для шифрування. Нарешті, для 3DES пропускна здатність дуже схожа з GO лише 10. На 8% ефективніше, ніж Python для шифрування. Тому і Python, і GO показали хороші результати для різних алгоритмів.

Аналіз простоти впровадження.

З точки зору читабельності, Python дав більше, ніж GO. Python дуже схожий на англійські тексти з низьким використанням спеціальних символів і спрощує написання коду за допомогою використання таких функцій, як висновок типу. Тим не менш, GO добре обізнаний у прийнятті шаблону розробки, подібного до мови C, що може виявитися корисним, коли потрібен більше контролю та вищий рівень абстракції.

GO та Python спільно використовують як алгоритми шифрування, так і режими блокового шифрування, але, здається, GO не реалізував жодних функцій доповнення чи серіалізації. Хоча серіалізація не є вимогою, доповнення завжди необхідно для алгоритмів блокового шифрування.

Розробка всіх реалізацій зайняла менше 4 годин, і складність розробки, здається, однакова для обох мов. Реалізація RSA дещо відрізняється складністю, але обидві реалізації були складними по-своєму. Для реалізації двох симетричних алгоритмів шифрування, AES і 3DES, для обох мов використовувався, здавалося б, однаковий підхід. Здається, це пов’язано з подібним дизайном бібліотек щодо алгоритмів блокового шифрування та хороших прикладів коду в обох бібліотеках.

Аналіз криптографічних бібліотек.

Для криптографічних бібліотек дані щодо їх якості та загальної кількості бібліотек, призначених для розробки алгоритмів шифрування, були надані для кожної мови програмування. Python має багато бібліотек, які зосереджені на різних областях криптографії, і GO також надає можливість працювати в цій області. Однак використання реалізацій шифрування в GO в основному покладається на стандартну бібліотеку, а не на зовнішні бібліотеки. Для Python також потрібні деякі зовнішні бібліотеки. Python пропонує вісім більше алгоритмів шифрування, ніж GO, хоча ці додаткові алгоритми реалізують слабкіші алгоритми шифрування. Режими блокового шифрування суттєво відрізняються між мовами, де Python надає ще 9 режимів блокового шифрування. Бібліотеки GO зосереджені на забезпеченні очікуваних режимів роботи, таких як

CBC, тоді як бібліотеки Python також надають спеціальні режими роботи для AES для автентифікації.

GO та Python містять функції, важливі для цієї мети. Python і GO мають 5 спільних функцій, але решта функцій унікальні для кожної мови. GO вирішив розширити протоколи, такі як SSH і TLS, і надає функції для роботи з сертифікатами x509. Python робить більший внесок із відповідними механізмами для алгоритмів шифрування, такими як схеми заповнення та серіалізація відкритих ключів. У цьому плані Python надає більше функціональних можливостей для полегшення роботи з алгоритмами шифрування, тоді як GO надає більше функціональних можливостей для роботи з різними протоколами.

Важливість функцій заповнення очевидна при роботі з симетричними алгоритмами шифрування. Алгоритми симетричного шифрування, які є блоковими шифрами, наприклад AES, працюють з блоками. Кожен блок у блочному шифрі має бути заповненим перед шифруванням, що означає, що доповнення є важливим для алгоритмів симетричного блочного шифру. Але це не властиво досліджуваним бібліотекам GO. На відміну від Python, GO потребує додаткових бібліотек для правильної роботи з блоковими шифрами, якщо це не реалізовано розробником.

Нарешті, технічне обслуговування бібліотек Python здійснюється зовнішніми сторонами, які самі не пов’язані з Python і випускають нерегулярні випуски. Але жодні випуски не були старші 3 років. Бібліотеки GO обслуговуються командою GO та випускають оновлення разом із новими випусками GO. Це ненавмисно призводить до того, що оновлення бібліотек GO рідше, ніж бібліотеки Python, але стандарти якості порівнюються з будь-якою стандартною бібліотекою в GO. Кількість користувачів цих бібліотек було виявлено лише для Python, оскільки це єдині зовнішні бібліотеки. Бібліотеки в GO не надали жодних даних щодо цієї статистики, тому їх не можна справедливо порівнювати з бібліотеками Python.

Важливим результатом було те, що реалізація RSA не могла використовувати стільки відкритих текстових файлів різного розміру, ніж AES і

3DES. Це пояснюється тим, що RSA може шифрувати дані лише до максимальної кількості, що дорівнює розміру ключа, за вирахуванням будь-яких доповнень і даних заголовка. Оскільки ми хотіли, щоб алгоритми шифрування відповідали типовому використанню, ми обмежилися розміром ключа 4096 біт, оскільки ключ RSA зазвичай має розмір від 2048 до 4096 біт. Отже, можна стверджувати, що результати тестів RSA менш точні порівняно з результатами, отриманими з AES і 3DES. Це пояснюється тим, що останні методи шифрування мали більший пул файлів для тестування, а розбіжності в розмірах файлів також були більшими.

Можна перевірити більші розміри файлів, розділивши файл відкритого тексту на менші сегменти та надіславши їх один за одним. Однак цей підхід відхилятиметься від стандартизованого варіанту використання, оскільки вимагатиме кількох операцій шифрування для одного повідомлення, на відміну від AES і 3DES, які шифрують усе повідомлення за одну операцію. Отже, це рішення не буде типовим для стандартного використання RSA.

Час шифрування для тестів RSA постійно зменшувався з кожним поступовим збільшенням розміру файлу в обох мовах, за винятком випадків, які спостерігалися в GO під час тестування розмірів файлів 100 і 200 байтів. Ця аномалія може виникнути через складність точного вимірювання тривалості часу під час шифрування дуже малих файлів, оскільки процес відбувається швидко. Тому дуже важливо з обережністю підходити до оцінки ефективності шифрування RSA, оскільки точність вимірювань може бути скомпрометована. Варто зазначити, що ці спостереження були помічені під час процесу тестування, що спонукало усвідомити потенційні обмеження цих тестів.

Між бібліотеками також була велика різниця. GO мала стандартну бібліотеку для шифрування, включену в інсталяцію мови, тоді як Python головним чином включав хеш-функції в інсталяцію мови. Обмежуючись лише двома мовами, було важко відзначити щось, що загалом пов’язане зі стандартними бібліотеками чи зовнішніми бібліотеками. Проте в цій дисертації можна було знайти позитивні та негативні сторони, засновані на криптографічних бібліотеках і простоті впровадження.

Аналіз валідності.

У цій роботі аналізуються чотири загрози.

1. Достовірність. Ця загроза стосується достовірності звіту. Наші контрольні тести проводилися на реалізаціях, створених за допомогою звичайних бібліотек програмування, до яких може отримати доступ кожен. Оскільки ці бібліотеки або є стандартними для мов, або за ними стоять великі спільноти, було б розумно оцінити, що реалізації в цих бібліотеках є дійсними алгоритмами шифрування. Це сприяє більшій довірі до нашого дослідження.

2. Можливість передачі. Коли ми говоримо про можливість перенесення, ми маємо на увазі можливість застосувати наші висновки до різних ситуацій. Оскільки наше дослідження в основному зосереджено на перевагах використання Python і GO для шифрування, важко застосувати ці висновки до інших мов або алгоритмів шифрування через їхні властиві відмінності. Однак створені алгоритми шифрування були створені з типовим розміром ключа, методом доповнення та режимом роботи для кожного відповідного алгоритму. Роблячи це, ми прагнули продемонструвати ефективність кожної мови в реалізації узагальнених версій алгоритмів шифрування. Цей підхід підвищує потенційну релевантність наших висновків для подібних досліджень із використанням різних мов або методів шифрування, оскільки забезпечує основу для порівняння в різних контекстах.

3. Надійність. Надійність означає послідовність і повторюваність заходів. Ми перевірили алгоритми шифрування, реалізовані бібліотеками програмування, тобто ці результати можна відтворити за допомогою тих самих реалізацій. Крім того, вимірювання проводилися кілька разів, а потім із них брали середнє значення. Це підвищує надійність дослідження. Однак, оскільки частина цього дослідження базувалася на нашому власному досвіді, буде важко точно відтворити ці результати.

4 Можливість підтвердження. Можливість підтвердження гарантує, що упередження дослідника не вплинуть на результат дослідження. Наша модель порівняння частково ґрунтувалася на нашій суб’єктивній думці, тому деяким

думкам було дуже важко не бути упередженими, можливо, навіть невідомими нам самим. Щоб вирішити цю проблему, ми створили три критерії для використання як вимірювання, щоб мінімізувати ризик того, що наші особисті уподобання вплинуть на наші результати та спотворять висновки дослідження на користь однієї мови.

Криптографічні профілі комп'ютерних систем є основою забезпечення безпеки в сучасних інформаційних технологіях. Вони являють собою комплексне поєднання алгоритмів, протоколів, методів управління ключами та програмних реалізацій, спрямованих на захист даних від несанкціонованого доступу, забезпечення їх цілісності, автентичності та неспростовності. З інженерної точки зору, криптографічний профіль — це структурована система, яка враховує вимоги до безпеки, продуктивності, масштабованості та сумісності з апаратними і програмними компонентами.

У контексті комп'ютерних систем криптографічні профілі застосовуються для вирішення широкого спектра завдань:

– захист конфіденційних даних у базах даних і файлових системах;

– забезпечення безпеки мережевих комунікацій, включаючи протоколи TLS/SSL для веб-додатків;

– аутентифікація користувачів і пристроїв у розподілених системах;

– гарантія цілісності програмного забезпечення та даних під час оновлень;

– захист транзакцій у фінансових системах і блокчейнах.

Інженерний підхід до розробки криптографічних профілів передбачає ретельний аналіз вимог до безпеки, вибір оптимальних алгоритмів, оцінку їх продуктивності та стійкості до сучасних загроз, таких як квантові обчислення чи атаки на побічні канали. Крім того, важливим є забезпечення сумісності з апаратними модулями безпеки (HSM), які використовуються для зберігання ключів і виконання криптографічних операцій.

Основними викликами при реалізації криптографічних профілів є:

– Баланс між безпекою та продуктивністю: високий рівень безпеки часто супроводжується зниженням швидкості обробки даних.

– Управління ключами: безпечне генерування, зберігання, обмін і ротація ключів є складним завданням, особливо в розподілених системах.

– Стійкість до нових загроз: поява квантових комп'ютерів може зробити деякі алгоритми (наприклад, RSA) вразливими.

– Інтеграція з існуючими системами: криптографічні профілі повинні бути сумісними з різними платформами, операційними системами та протоколами.

Ця документація розширює аналіз програмних аспектів криптографічних профілів, надаючи детальний огляд їх компонентів, інженерних принципів реалізації та перспектив розвитку.

Симетрична криптографія базується на використанні одного ключа для шифрування та дешифрування даних. Вона є основою для захисту великих обсягів інформації завдяки високій швидкості обробки та ефективності. Найпоширенішим стандартом симетричного шифрування є Advanced Encryption Standard (AES), який підтримує ключі довжиною 128, 192 або 256 біт і використовується в різних режимах роботи, таких як CBC (Cipher Block Chaining), GCM (Galois/Counter Mode) і CTR (Counter Mode).

З інженерної точки зору, симетрична криптографія має кілька ключових особливостей:

Висока продуктивність: Алгоритми, такі як AES, оптимізовані для апаратного прискорення (наприклад, через набір інструкцій AES-NI на процесорах Intel), що дозволяє обробляти великі обсяги даних у реальному часі.

Простота реалізації: Симетричні алгоритми мають меншу обчислювальну складність порівняно з асиметричними, що полегшує їх інтеграцію в програмне забезпечення.

Вимоги до ключів: Безпечне управління ключами є критичним, оскільки компрометація ключа призводить до втрати конфіденційності всіх зашифрованих даних.

Основні режими роботи AES мають різні інженерні застосування:

– CBC: Забезпечує ланцюгову залежність блоків, що підвищує безпеку, але вимагає унікального вектора ініціалізації (IV) для кожного шифрування.

– GCM: Поєднує шифрування з аутентифікацією, що робить його ідеальним для захисту мережевих даних.

– CTR: Перетворює блочний шифр у поточний, що корисно для потокового шифрування.

Інженерні виклики включають:

– Управління IV: Повторне використання IV у режимі CBC може призвести до витоку інформації. Рекомендується генерувати унікальний IV для кожного сеансу шифрування.

– Padding: Використання доповнення (наприклад, PKCS#5/PKCS#7) для вирівнювання розміру даних може створювати вразливості, такі як padding oracle атака.

– Оптимізація**:** Для великих даних необхідно використовувати апаратне прискорення або паралельну обробку.

Симетрична криптографія широко застосовується в системах зберігання даних, захисту мережевих комунікацій і шифрування дисків. Однак її головним недоліком є проблема безпечного обміну ключами, що вирішується за допомогою асиметричної криптографії.

Криптографічний профіль визначає набір алгоритмів, параметрів і процедур, які використовуються для захисту даних у комп'ютерній системі. Основні компоненти включають:

1. Симетрична криптографія: Використовує один ключ для шифрування та дешифрування. Приклади: AES, ChaCha20.

Переваги: Висока швидкість, ефективність для великих обсягів даних.

Недоліки: Проблема безпечного обміну ключами.

2. Асиметрична криптографія: Використовує пару ключів (публічний і приватний). Приклади: RSA, ECDSA.

Переваги: Безпечний обмін ключами, підтримка цифрових підписів.

Недоліки: Низька швидкість, обмеження на розмір даних.

3. Хеш-функції: Генерують фіксований дайджест для перевірки цілісності. Приклади: SHA-256, SHA-3.

Застосування: Перевірка цілісності файлів, аутентифікація повідомлень.

4. Цифрові підписи: Забезпечують автентичність і неспростовність. Приклади: ECDSA, EdDSA.

5. Обмін ключами: Протоколи для безпечного узгодження ключів (наприклад, Diffie-Hellman).

6. Управління ключами: Генерація, зберігання, ротація та знищення ключів.

Інженерний підхід до реалізації криптографічних профілів передбачає:

– аналіз вимог до безпеки (наприклад, стійкість до квантових атак);

– оптимізацію продуктивності (наприклад, використання апаратного прискорення);

– тестування на вразливості (наприклад, атаки на IV, padding oracle).

Симетрична криптографія: Аналіз AES.

Симетрична криптографія є основою для захисту великих обсягів даних завдяки високій швидкості обробки та ефективності. Найпоширенішим стандартом у цій категорії є Advanced Encryption Standard (AES), який підтримує ключі довжиною 128, 192 або 256 біт і використовується в різних режимах роботи, таких як Cipher Block Chaining (CBC), Galois/Counter Mode (GCM) і Counter Mode (CTR). З інженерної точки зору, AES є оптимальним вибором для сценаріїв, що вимагають швидкого шифрування, таких як захист файлів, баз даних або мережевих комунікацій. У режимі CBC AES забезпечує високу безпеку за рахунок ланцюгової залежності блоків і використання випадкового вектора ініціалізації (IV), який додає унікальність кожному шифруванню.

Інженерна реалізація AES вимагає ретельного вибору режиму роботи залежно від потреб системи. Наприклад, режим CBC підходить для шифрування даних, де важлива цілісність і конфіденційність, але він потребує унікального IV для кожного сеансу, щоб уникнути витоку інформації через повторне використання. Режим GCM, окрім шифрування, забезпечує аутентифікацію даних, що робить його ідеальним для мережевих протоколів, таких як TLS. Управління IV є критичним аспектом, оскільки повторне використання може

призвести до серйозних вразливостей, таких як атаки на основі аналізу шифротексту. Захист ключів від витоку також є ключовим завданням: ключі повинні зберігатися в захищеному середовищі, наприклад, у апаратному модулі безпеки (HSM), щоб запобігти їх компрометації.

Продуктивність AES значною мірою залежить від апаратного забезпечення. Сучасні процесори, такі як Intel і AMD, підтримують набір інструкцій AES-NI, який значно прискорює операції шифрування та дешифрування. Для великих обсягів даних рекомендується використовувати апаратне прискорення або паралельну обробку, щоб мінімізувати затримки. Однак інженери повинні враховувати потенційні вразливості, такі як padding oracle атаки, які виникають через неправильне використання доповнення в режимі CBC. Для їх мінімізації рекомендується використовувати режими, такі як GCM, які забезпечують вбудовану аутентифікацію.

Безпека AES залежить від якості управління ключами та вибору параметрів. Наприклад, ключ довжиною 256 біт забезпечує високий рівень стійкості до атак грубої сили, але його безпечне зберігання і передача є складним завданням. У реальних системах AES часто використовується в гібридних схемах, де ключ шифрується асиметричним алгоритмом, таким як RSA, для безпечного обміну між сторонами. Таким чином, інженерна реалізація AES вимагає комплексного підходу, що поєднує вибір оптимального режиму, захист ключів, оптимізацію продуктивності та захист від відомих вразливостей.

Код реалізації AES

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64

def aes_encrypt(plain_text, key):
    """
    Шифрування тексту за допомогою AES-256 у режимі CBC.
    :param plain_text: Текст для шифрування (str)
    :param key: Ключ шифрування (bytes, 32 байти для AES-256)
    :return: Закодований у base64 шифротекст (str)
    """
    plain_text_bytes = plain_text.encode('utf-8')
    iv = get_random_bytes(AES.block_size)
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padding_length = AES.block_size - len(plain_text_bytes) %
AES.block_size
    plain_text_bytes += bytes([padding_length]) * padding_length
    cipher_text = cipher.encrypt(plain_text_bytes)
    return base64.b64encode(iv + cipher_text).decode('utf-8')

def aes_decrypt(cipher_text, key):
    """
    Дешифрування тексту, зашифрованого AES-256 у режимі CBC.
    :param cipher_text: Закодований у base64 шифротекст (str)
    :param key: Ключ шифрування (bytes, 32 байти для AES-256)
    :return: Дешифрований текст (str)
    """
    cipher_text_bytes = base64.b64decode(cipher_text)
    iv = cipher_text_bytes[:AES.block_size]
    cipher_text_bytes = cipher_text_bytes[AES.block_size:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plain_text_bytes = cipher.decrypt(cipher_text_bytes)
    padding_length = plain_text_bytes[-1]
    plain_text_bytes = plain_text_bytes[:-padding_length]
    return plain_text_bytes.decode('utf-8')

if __name__ == "__main__":
    key = get_random_bytes(32)
    original_text = "Конфіденційне повідомлення для AES"
    encrypted_text = aes_encrypt(original_text, key)
    decrypted_text = aes_decrypt(encrypted_text, key)
    print(f"Оригінал: {original_text}")
    print(f"Зашифровано: {encrypted_text}")
    print(f"Дешифровано: {decrypted_text}")
```

Асиметрична криптографія: Аналіз RSA.

Асиметрична криптографія базується на використанні пари ключів — публічного та приватного, що дозволяє вирішити проблему безпечного обміну ключами та забезпечити цифрові підписи. Алгоритм RSA, який ґрунтується на математичній складності факторизації великих чисел, є одним із найпоширеніших у цій категорії. RSA використовується для шифрування невеликих обсягів даних, таких як ключі для симетричних алгоритмів, а також для створення цифрових підписів, які гарантують автентичність і цілісність повідомлень.

З інженерної точки зору, реалізація RSA вимагає ретельного вибору довжини ключа, яка визначає рівень безпеки. Ключі довжиною 2048 біт є компромісом між безпекою та продуктивністю, тоді як ключі довжиною 4096 біт

рекомендуються для довгострокового захисту, особливо в умовах потенційних квантових атак. Використання безпечного доповнення, такого як Optimal Asymmetric Encryption Padding (OAEP), є обов'язковим для захисту від атак на основі вибору шифротексту, які можуть розкрити зашифровані дані. Захист приватного ключа є ще одним критичним аспектом: він повинен зберігатися в захищеному середовищі, такому як HSM або TPM, щоб запобігти його компрометації.

Продуктивність RSA є значно нижчою порівняно з симетричними алгоритмами, такими як AES, що обмежує його використання для шифрування великих даних. У реальних системах RSA часто застосовується в гібридних схемах, де він шифрує ключ для симетричного алгоритму, який, у свою чергу, шифрує основний обсяг даних. Такий підхід поєднує безпеку асиметричної криптографії з високою швидкістю симетричної. Інженери повинні враховувати, що RSA вразливий до квантових атак, таких як алгоритм Шора, який може ефективно факторизувати великі числа, що вимагає підготовки до переходу на постквантови алгоритми.

Безпека RSA залежить від якості реалізації та захисту ключів. Наприклад, неправильне використання доповнення або слабкий генератор псевдовипадкових чисел може призвести до вразливостей. Крім того, інженери повинні регулярно оновлювати реалізації RSA, щоб враховувати нові атаки та рекомендації стандартів, таких як NIST SP 800-57. У контексті комп'ютерних систем RSA застосовується для захисту мережевих протоколів (наприклад, TLS), аутентифікації користувачів і забезпечення безпеки оновлень програмного забезпечення.

Код реалізації RSA

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

def generate_rsa_keys():
    """
    Генерація пари ключів RSA (2048 біт).
    :return: (private_key, public_key)
    """
```

```
    key = RSA.generate(2048)
    private_key = key
    public_key = key.publickey()
    return private_key, public_key

def rsa_encrypt(plain_text, public_key):
    """
    Шифрування тексту за допомогою RSA з OAEP.
    :param plain_text: Текст для шифрування (str)
    :param public_key: Публічний ключ RSA
    :return: Закодований у base64 шифротекст (str)
    """
    cipher = PKCS1_OAEP.new(public_key)
    cipher_text = cipher.encrypt(plain_text.encode('utf-8'))
    return base64.b64encode(cipher_text).decode('utf-8')

def rsa_decrypt(cipher_text, private_key):
    """
    Дешифрування тексту, зашифрованого RSA.
    :param cipher_text: Закодований у base64 шифротекст (str)
    :param private_key: Приватний ключ RSA
    :return: Дешифрований текст (str)
    """
    cipher = PKCS1_OAEP.new(private_key)
    cipher_text_bytes = base64.b64decode(cipher_text)
    plain_text = cipher.decrypt(cipher_text_bytes)
    return plain_text.decode('utf-8')

if __name__ == "__main__":
    private_key, public_key = generate_rsa_keys()
    original_text = "Секретне повідомлення RSA"
    encrypted_text = rsa_encrypt(original_text, public_key)
    decrypted_text = rsa_decrypt(encrypted_text, private_key)
    print(f"Оригінал: {original_text}")
    print(f"Зашифровано: {encrypted_text}")
    print(f"Дешифровано: {decrypted_text}")
```

Хеш-функції та аутентифікація: Аналіз SHA-256 і HMAC.

Хеш-функції відіграють ключову роль у забезпеченні цілісності даних, генеруючи фіксований дайджест для вхідних даних довільної довжини. SHA-256, що належить до сімейства SHA-2, є однією з найпоширеніших хеш-функцій, яка створює 256-бітний дайджест, стійкий до колізій. З інженерної точки зору, SHA-256 є ефективним інструментом для перевірки цілісності файлів, програмного забезпечення та повідомлень, а також для зберігання паролів у хешованому вигляді з використанням солі.

HMAC (Hash-based Message Authentication Code) поєднує хеш-функцію, таку як SHA-256, з ключем, щоб забезпечити не лише цілісність, а й автентичність даних. HMAC-SHA256 є стандартом для аутентифікації API-запитів, захисту сесій і перевірки повідомлень у мережевих протоколах. Інженерна реалізація HMAC вимагає захисту ключів від витоку, оскільки компрометація ключа дозволяє зловмиснику підробити повідомлення. Крім того, ключ HMAC повинен бути достатньо довгим (не менше 128 біт), щоб забезпечити стійкість до атак грубої сили.

Продуктивність SHA-256 і HMAC залежить від апаратного забезпечення та обсягу даних. Для великих файлів рекомендується використовувати потокову обробку, щоб мінімізувати використання пам'яті. Хоча SHA-256 є стійким до колізій, він повільніший за новіші алгоритми, такі як BLAKE2, які можуть бути кращим вибором у системах із високими вимогами до швидкості. Інженери повинні враховувати, що SHA-256 вразливий до атак на довжину розширення, якщо використовується без HMAC, що робить HMAC обов'язковим для аутентифікації.

Застосування SHA-256 і HMAC охоплює перевірку цілісності оновлень програмного забезпечення, аутентифікацію API-запитів, захист сесій у веб-додатках і забезпечення безпеки транзакцій у фінансових системах. Безпека цих механізмів залежить від якості реалізації та захисту ключів, а також від регулярного оновлення бібліотек для врахування нових вразливостей.

Код реалізації SHA-256 і HMAC

```
from Crypto.Hash import SHA256, HMAC
from Crypto.Random import get_random_bytes

def sha256_hash(data):
    """
    Обчислення хешу SHA-256.
    :param data: Дані для хешування (str)
    :return: Хеш у шістнадцятковому форматі (str)
    """
    h = SHA256.new()
    h.update(data.encode('utf-8'))
    return h.hexdigest()
```

```
def hmac_sha256(key, data):
    """
    Обчислення HMAC-SHA256.
    :param key: Ключ для HMAC (bytes)
    :param data: Дані для хешування (str)
    :return: HMAC у шістнадцятковому форматі (str)
    """
    h = HMAC.new(key, digestmod=SHA256)
    h.update(data.encode('utf-8'))
    return h.hexdigest()

if __name__ == "__main__":
    data = "Дані для хешування"
    key = get_random_bytes(16)
    sha256_result = sha256_hash(data)
    hmac_result = hmac_sha256(key, data)
    print(f"SHA-256: {sha256_result}")
    print(f"HMAC-SHA256: {hmac_result}")
```

Цифрові підписи: Аналіз ECDSA

Цифрові підписи є важливим інструментом для забезпечення автентичності та неспростовності даних. ECDSA (Elliptic Curve Digital Signature Algorithm), що базується на еліптичних кривих, є ефективнішим за RSA завдяки меншому розміру ключів і швидшій обробці. Крива P-256, яка забезпечує 128 біт безпеки, є оптимальним вибором для більшості застосувань, поєднуючи високу безпеку з продуктивністю.

З інженерної точки зору, ECDSA вимагає обережного вибору параметрів, зокрема кривої, щоб уникнути вразливостей, пов’язаних із слабкими кривими. Продуктивність ECDSA є вищою порівняно з RSA, особливо для створення та перевірки підписів, що робить його ідеальним для сценаріїв, таких як аутентифікація транзакцій у блокчейнах або захист оновлень програмного забезпечення. Однак ECDSA вразливий до атак на генератори псевдовипадкових чисел (PRNG), зокрема до повторного використання nonce, що може призвести до компрометації приватного ключа. Для мінімізації цього ризику використовуються криптографічно безпечні PRNG, такі як ті, що надаються бібліотекою Crypto.Random.

Застосування ECDSA охоплює захист транзакцій у фінансових системах, аутентифікацію в розподілених мережах і забезпечення безпеки вбудованих систем. Інженери повинні враховувати, що безпека ECDSA залежить від якості реалізації, захисту ключів і регулярного оновлення бібліотек для врахування нових атак.

Код реалізації ECDSA

```
from Crypto.PublicKey import ECC
from Crypto.Signature import DSS
from Crypto.Hash import SHA256
import base64

def generate_ecdsa_keys():
    """
    Генерація пари ключів ECDSA (крива P-256).
    :return: (private_key, public_key)
    """
    private_key = ECC.generate(curve='P-256')
    public_key = private_key.public_key()
    return private_key, public_key

def ecdsa_sign(message, private_key):
    """
    Підписання повідомлення за допомогою ECDSA.
    :param message: Повідомлення (str)
    :param private_key: Приватний ключ ECDSA
    :return: Підпис у base64 (str)
    """
    h = SHA256.new(message.encode('utf-8'))
    signer = DSS.new(private_key, 'fips-186-3')
    signature = signer.sign(h)
    return base64.b64encode(signature).decode('utf- on')
```

Код реалізації ECDSA (продовження)

```
def ecdsa_verify(message, signature, public_key):
    """
    Перевірка підпису ECDSA.
    :param message: Повідомлення (str)
    :param signature: Підпис у base64 (str)
    :param public_key: Публічний ключ ECDSA
    :return: True, якщо підпис валідний
    """
    h = SHA256.new(message.encode('utf-8'))
    verifier = DSS.new(public_key, 'fips-186-3')
    try:
        verifier.verify(h, base64.b64decode(signature))
        return True
    except ValueError:
        return False
```

```
if __name__ == "__main__":
    private_key, public_key = generate_ecdsa_keys()
    message = "Повідомлення для підписання"
    signature = ecdsa_sign(message, private_key)
    is_valid = ecdsa_verify(message, signature, public_key)
    print(f"Повідомлення: {message}")
    print(f"Підпис: {signature}")
    print(f"Підпис валідний: {is_valid}")
```

Обмін ключами: Аналіз Diffie-Hellman.

Протокол Diffie-Hellman дозволяє двом сторонам узгодити спільний секретний ключ через незахищений канал, що робить його основою для гібридного шифрування та захисту мережевих комунікацій. З інженерної точки зору, Diffie-Hellman вимагає використання великих простих чисел або еліптичних кривих, щоб забезпечити стійкість до атак, таких як дискретне логарифмування. Параметри, такі як 2048-бітний ключ, є стандартними для забезпечення безпеки, але інженери повинні перевіряти їх на відповідність стандартам, щоб уникнути атак на малі підгрупи.

Спільний секрет, отриманий за допомогою Diffie-Hellman, часто використовується як ключ для симетричного шифрування, наприклад, AES. Однак протокол вразливий до атак "людина посередині", якщо не застосовується додаткова аутентифікація, наприклад, за допомогою цифрових підписів. У реальних системах Diffie-Hellman інтегрується з протоколами, такими як TLS, для забезпечення безпечного обміну ключами.

Інженерна реалізація Diffie-Hellman вимагає ретельного вибору параметрів, захисту ключів і забезпечення сумісності з іншими компонентами системи. Продуктивність протоколу залежить від розміру ключів і типу обчислень, але сучасні реалізації оптимізуються за допомогою апаратного прискорення або еліптичних кривих, які є ефективнішими за класичний підхід.

Код реалізації Diffie-Hellman

```
from Crypto.PublicKey import DH
from Crypto.Random import get_random_bytes

def generate_dh_keys():
```

```
    """
    Генерація ключів Diffie-Hellman.
    :return: (private_key, public_key)
    """
    key = DH.generate(2048)
    return key, key.publickey()

def dh_shared_secret(private_key, other_public_key):
    """
    Обчислення спільного секрету Diffie-Hellman.
    :param private_key: Приватний ключ DH
    :param other_public_key: Публічний ключ іншої сторони
    :return: Спільний секрет (bytes)
    """
    return private_key.shared_secret(other_public_key)

if __name__ == "__main__":
    # Сторона A
    private_a, public_a = generate_dh_keys()
    # Сторона B
    private_b, public_b = generate_dh_keys()
    # Обмін публічними ключами та обчислення спільного секрету
    secret_a = dh_shared_secret(private_a, public_b)
    secret_b = dh_shared_secret(private_b, public_a)
    print(f"Спільний секрет A: {secret_a.hex()[:32]}...")
    print(f"Спільний секрет B: {secret_b.hex()[:32]}...")
    assert secret_a == secret_b, "Спільні секрети не співпадають!"
```

Аналіз безпеки та вразливостей.

Безпека криптографічних профілів залежить від стійкості алгоритмів, якості їх реалізації та захисту ключів. Кожен алгоритм має потенційні вразливості, які інженери повинні враховувати при розробці систем. Наприклад, AES може бути вразливим до атак на побічні канали, таких як таймінгові атаки або аналіз енергоспоживання, що вимагає використання апаратного прискорення та захищених бібліотек, таких як pycryptodome або OpenSSL. Режим CBC у AES потребує унікального IV, щоб уникнути витоку інформації, а неправильне використання доповнення може призвести до padding oracle атак.

RSA вразливий до атак на основі вибору шифротексту, якщо не використовується безпечне доповнення, таке як OAEP. Крім того, ключі довжиною менше 2048 біт вважаються небезпечними, особливо в умовах потенційних квантових атак. SHA-256 є стійким до колізій, але без HMAC він вразливий до атак на довжину розширення, що робить HMAC обов'язковим для

аутентифікації. ECDSA залежить від якості PRNG, і повторне використання nonce може призвести до компрометації ключа. Diffie-Hellman вимагає аутентифікації для захисту від атак "людина посередині", що зазвичай досягається за допомогою цифрових підписів.

Інженерні заходи для підвищення безпеки включають використання перевірених бібліотек, регулярну ротацію ключів, аудит коду та тестування на вразливості, такі як fuzzing і статичний аналіз. Крім того, стандарти, такі як NIST SP 800-57, надають рекомендації щодо вибору ключів, режимів роботи та управління життєвим циклом ключів. У контексті сучасних загроз, таких як квантові обчислення, інженери повинні готуватися до переходу на постквантови алгоритми, такі як ґраткові алгоритми, які розробляються NIST.

Оптимізація продуктивності.

Продуктивність криптографічних алгоритмів є критичним аспектом їх застосування в реальних системах, особливо в умовах обмежених ресурсів або високих вимог до швидкості. AES є значно швидшим за RSA, що робить його основним вибором для шифрування великих даних. ECDSA перевершує RSA за продуктивністю при створенні та перевірці підписів, що робить його кращим для аутентифікації. Продуктивність залежить від апаратного забезпечення: сучасні процесори з підтримкою AES-NI або GPU можуть значно прискорити обчислення.

Для великих даних використовується гібридне шифрування, де RSA або ECDSA шифрує ключ AES, а AES обробляє основний обсяг даних. Це дозволяє поєднати безпеку асиметричної криптографії з високою швидкістю симетричної. Інженери повинні оптимізувати обробку даних, використовуючи потокову обробку для хеш-функцій і шифрування, а також паралельну обробку для багатоядерних систем. У вбудованих системах, таких як IoT, продуктивність може бути обмежена, що вимагає використання легких алгоритмів або апаратного прискорення.

Тест продуктивності AES vs RSA

```
import time
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
```

```
from Crypto.Random import get_random_bytes

def test_performance():
    """
    Порівняння продуктивності AES і RSA.
    """
    # Підготовка даних
    key_aes = get_random_bytes(32)
    key_rsa = RSA.generate(2048)
    public_rsa = key_rsa.publickey()
    data = "Тестові дані" * 1000  # 12 КБ даних

    # Тест AES
    start_time = time.time()
    cipher_aes = AES.new(key_aes, AES.MODE_CBC,
get_random_bytes(16))
    padded_data = data.encode('utf-8') + b' ' * (16 -
len(data.encode('utf-8')) % 16)
    cipher_aes.encrypt(padded_data)
    aes_time = time.time() - start_time

    # Тест RSA (обмежено до 190 байт через OAEP)
    start_time = time.time()
    cipher_rsa = PKCS1_OAEP.new(public_rsa)
    cipher_rsa.encrypt(data[:190].encode('utf-8'))
    rsa_time = time.time() - start_time

    print(f"Час AES: {aes_time:.6f} сек")
    print(f"Час RSA: {rsa_time:.6f} сек")

if __name__ == "__main__":
    test_performance()
```

Результати тестування показують, що AES значно швидший за RSA, особливо для великих даних. Гібридний підхід, де RSA шифрує ключ AES, є стандартним рішенням для поєднання безпеки та продуктивності.

Практичні сценарії застосування та управління ключами.

Криптографічні профілі застосовуються в численних сценаріях, які охоплюють захист даних, аутентифікацію та забезпечення цілісності. Шифрування файлів за допомогою AES є поширеним рішенням для захисту конфіденційних даних на диску або під час передачі. У цьому процесі дані розбиваються на блоки, шифруються з використанням унікального IV і зберігаються у зашифрованому вигляді. Такий підхід використовується в системах резервного копіювання, хмарних сховищах і захисті баз даних.

Аутентифікація API-запитів з використанням HMAC-SHA256 є стандартним механізмом для захисту веб-додатків. HMAC генерує підпис для запиту, який перевіряється сервером, щоб гарантувати, що запит не був змінений і походить від авторизованого клієнта. Цей механізм широко застосовується в фінансових системах, соціальних мережах і платформах електронної комерції.

Управління ключами є критично важливим для безпеки криптографічних профілів. Генерація ключів повинна виконуватися за допомогою криптографічно безпечних PRNG, таких як Crypto.Random, щоб уникнути передбачуваності. Зберігання ключів у захищених сховищах, таких як HSM або системний keyring, запобігає їх компрометації. Ротація ключів, тобто регулярна їх заміна, зменшує ризик довгострокового використання скомпрометованого ключа. Обмін ключами за допомогою Diffie-Hellman або RSA забезпечує безпечну передачу через незахищені канали.

Шифрування файлів

```
  def encrypt_file(input_file, output_file, key):
    """
    Шифрування файлу за допомогою AES-256.
    :param input_file: Шлях до вхідного файлу
    :param output_file: Шлях до вихідного файлу
    :param key: Ключ шифрування (bytes)
    """
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    with open(input_file, 'rb') as f_in, open(output_file, 'wb')
as f_out:
        f_out.write(iv)
        while True:
            chunk = f_in.read(1024)
            if len(chunk) == 0:
                break
            if len(chunk) % AES.block_size != 0:
                chunk += b' ' * (AES.block_size - len(chunk) %
AES.block_size)
            f_out.write(cipher.encrypt(chunk))

# Приклад використання
if __name__ == "__main__":
    key = get_random_bytes(32)
    encrypt_file("input.txt", "encrypted.bin", key)
```

Аутентифікація API-запитів з HMAC

```
def generate_api_signature(api_key, request_data):
    """
    Генерація HMAC-SHA256 для аутентифікації API-запиту.
    :param api_key: Ключ API (bytes)
    :param request_data: Дані запиту (str)
    :return: Підпис (str)
    """
    return hmac_sha256(api_key, request_data)

# Приклад використання
if __name__ == "__main__":
    api_key = get_random_bytes(16)
    request = "GET /api/resource?user=123"
    signature = generate_api_signature(api_key, request)
    print(f"API запит: {request}")
    print(f"Підпис: {signature}")
```

Управління ключами: Зберігання в HSM (псевдокод)

```
from hsm import HSMClient

def store_key_in_hsm(key, key_id):
    """
    Зберігання ключа в HSM.
    :param key: Ключ для збереження (bytes)
    :param key_id: Ідентифікатор ключа (str)
    """
    hsm = HSMClient()
    hsm.store_key(key_id, key)
    print(f"Ключ {key_id} збережено в HSM")
```

Тестування реалізації

Тестування криптографічних реалізацій є невід'ємною частиною інженерного підходу, що забезпечує їх коректність, безпеку та продуктивність. Функціональні тести перевіряють, чи правильно виконуються операції шифрування, дешифрування, підписання та перевірки. Тести безпеки оцінюють стійкість до вразливостей, таких як padding oracle атаки або повторне використання IV. Тести продуктивності порівнюють швидкість алгоритмів для різних обсягів даних, що дозволяє оптимізувати їх використання.

Приклад тестів

```
import unittest

class TestCrypto(unittest.TestCase):
```

```
        def test_aes(self):
            key = get_random_bytes(32)
            original = "Тест AES"
            encrypted = aes_encrypt(original, key)
            decrypted = aes_decrypt(encrypted, key)
            self.assertEqual(original, decrypted)

        def test_rsa(self):
            private_key, public_key = generate_rsa_keys()
            original = "Тест RSA"
            encrypted = rsa_encrypt(original, public_key)
            decrypted = rsa_decrypt(encrypted, private_key)
            self.assertEqual(original, decrypted)

        def test_ecdsa(self):
            private_key, public_key = generate_ecdsa_keys()
            message = "Тест ECDSA"
            signature = ecdsa_sign(message, private_key)
            self.assertTrue(ecdsa_verify(message, signature,
    public_key))

    if __name__ == "__main__":
        unittest.main()
```

Криптографічні профілі є основою безпеки сучасних комп'ютерних систем, забезпечуючи захист даних, аутентифікацію та цілісність у різноманітних сценаріях. Програмна реалізація на Python з використанням бібліотеки pycryptodome дозволяє створювати надійні рішення для шифрування, підписання та перевірки даних. Інженерний підхід передбачає ретельний вибір алгоритмів залежно від сценарію: AES для великих даних, RSA і ECDSA для ключів і підписів, SHA-256 і HMAC для цілісності та аутентифікації, Diffie-Hellman для обміну ключами.

Оптимізація продуктивності, включаючи апаратне прискорення та гібридне шифрування, є ключовим аспектом для забезпечення швидкості та масштабованості. Управління ключами, захист від вразливостей і ретельне тестування гарантують безпеку системи. У контексті сучасних загроз, таких як квантові обчислення, інженери повинні готуватися до переходу на постквантови алгоритми, які розробляються для забезпечення безпеки в майбутньому.

Перспективи розвитку включають інтеграцію з новими технологіями, такими як гомоморфне шифрування для хмарних обчислень, легкі алгоритми для

IoT і постквантови алгоритми для захисту від квантових атак. Ця документація надає всебічний аналіз програмних аспектів криптографічних профілів, який може бути розширений шляхом додавання аналізу нових алгоритмів, порівняння з іншими платформами або детальних сценаріїв інтеграції з мережевими протоколами.

ВИСНОВКИ

У нашому технологічно розвиненому світі кібербезпека стала критично важливою для захисту систем, мереж і програм від нових загроз. Криптографія, історичний наріжний камінь кібербезпеки, використовує математичні принципи для захисту інформації. Вибір мови програмування для реалізації алгоритмів шифрування збільшує складність у цій галузі. Оскільки програмування відіграє центральну роль у впровадженні шифрування, зростаюча різноманітність мов програмування створює проблему для вибору найбільш прийнятної мови для шифрування даних. Ця робота підкреслює актуальність мов Python і GO для підтримки трьох криптографічних алгоритмів, AES, RSA і 3DES. Робота присвячена продуктивності, простоті впровадження та підтримці криптографічних бібліотек. Розуміння сильних сторін цих мов для криптографічних завдань може допомогти розробникам програмного забезпечення приймати обґрунтовані рішення. Результати показують, що і Python, і GO мають переваги в окремих сферах. GO має хорошу продуктивність, тоді як Python має добре підтримувані криптографічні бібліотеки. Обидва вони прості у використанні для реалізації алгоритмів шифруванняі обидва вони мають мовні функції, які спрощують програмування. Різні мови мають різні переваги та недоліки, залежно від того, чому розробники віддають перевагу. У цій роботі ми виявили, що Python і GO легко вивчати та використовувати. Одна проблема щодо синтаксису виникла під час використання Python, і це була читабельність виведених типів. Серед інших відмінностей були продуктивність алгоритмів і поточна підтримка бібліотек. GO був швидшим для симетричних алгоритмів шифрування та підтримував різні протоколи, наприклад SSH. Хоча в GO не було структури бібліотечної документації, необхідна документація існувала з деякими функціями шифрування. З іншого боку, Python був швидшим для асиметричного шифрування та надавав додаткову функціональність для алгоритмів шифрування, наприклад функції заповнення. Однак через невизначеність щодо точності вимірювань у тестах RSA неможливо зробити висновок, що Python працює краще,

ніж GO, для асиметричних алгоритмів. Python був загалом повільнішим, але сприяв читабельності через його схожість з англійською мовою, і тому його було легше вивчити. Його бібліотеки були дуже добре адаптовані для виконання різноманітних завдань програмування. Основний висновок полягає в тому, що GO має кращу продуктивність, ніж Python для симетричних алгоритмів, тоді як Python має доступ до більшої кількості функцій, що дає користувачеві більше можливостей і допомоги для методів шифрування.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 2nd ed. Wiley, 1996. 784c. URL: https://www.wiley.com/en-us/Applied+Cryptography%3A+Protocols%2C+Algorithms%2C+and+Source+Code+in+C%2C+2nd+Edition-p-9780471117094 (дата звернення: травень 2025).

2. Stallings W. Cryptography and Network Security: Principles and Practice. 8th ed. Pearson, 2020. 832c. URL: https://www.pearson.com/store/p/cryptography-and-network-security-principles-and-practice/P100000907087 (дата звернення: травень 2025).

3. Murray R. Security Engineering: A Guide to Building Dependable Distributed Systems. 3rd ed. Wiley, 2020. 1232c/ URL: https://www.wiley.com/en-us/Security+Engineering%3A+A+Guide+to+Building+Dependable+Distributed+Systems%2C+3rd+Edition-p-9781119508571 (дата звернення: травень 2025).

4. Bishop M. 2nd ed. Addison-Wesley, 2018. 1440c. URL: https://www.pearson. com/store/p/computer-security-art-and-science/P100000906709

5. NIST Special Publication 800-57 Part 1 Rev. 5**.** Recommendation for Key Management: General. National Institute of Standards and Technology, 2020. 160. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf (дата звернення: травень 2025).

6. NIST Special Publication 800-130. A Framework for Designing Cryptographic Key Management Systems. National Institute of Standards and Technology, 2013. 116c. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-130.pdf (дата звернення: травень 2025).

7. Ferguson N., Schneier B., Kohno T**.** Cryptography Engineering: Design Principles and Practical Applications. Wiley, 2010. 384c. URL: https://www.wiley.com/enus/Cryptography+Engineering%3A+Design+Principles+and+Practical+Applications-p-9780470474242 (дата звернення: травень 2025).

8. Kocher P., Jaffe J., Jun B**.** Differential Power Analysis. Advances in Cryptology CRYPTO '99, Lecture Notes in Computer Science, Vol. 1666, 1999. C.

388–397. URL: https://link.springer.com/chapter/10.1007/3-540-48405-1_25 (дата звернення: травень 2025).

9. Menezes A. J., van Oorschot P. C., Vanstone S. A. Handbook of Applied Cryptography. CRC Press, 1996. 816 с. URL: https://www.crcpress.com/Handbook-of-Applied-Cryptography/Menezes-van-Oorschot-Vanstone/p/book/9780849385230 (дата звернення: травень 2025).

10. Anderson R. Security Engineering: A Guide to Building Dependable Distributed Systems. 2nd ed. Wiley, 2008. 1088с. URL: https://www.wiley.com/en-us/Security+Engineering%3A+A+Guide+to+Building+Dependable+Distributed+Systems%2C+2nd+Edition-p-9780470068526 (дата звернення: травень 2025).

11. OWASP Foundation. OWASP Key Management Cheat Sheet. 2021. URL: https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html (дата звернення: травень 2025).

12. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force (IETF), 2018. 160с. URL: https://tools.ietf.org/html/rfc8446 (дата звернення: травень 2025).

13. Dworkin M. Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38A, 2001. 66с. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf (дата звернення: травень 2025).

14. Barker E., Roginsky A. Recommendation for Cryptographic Key Generation. NIST Special Publication 800-133 Rev. 2, 2020. 48с. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-133r2.pdf (дата звернення: травень 2025).

15. Diffie W., Hellman M. New Directions in Cryptography. IEEE Transactions on Information Theory, 1976. Vol. 22, No. 6. с. 644–654. URL: https://ieeexplore.ieee.org/document/1055638 (дата звернення: травень 2025).

16. Atkinson R., Kent S. Security Architecture for the Internet Protocol. RFC 4301, Internet Engineering Task Force (IETF), 2005. 101с. URL: https://tools.ietf.org/html/rfc4301 (дата звернення: травень 2025).

17. Sarkar P., Chowdhury M. A. Comprehensive Survey of Cryptography Key Management Systems. Journal of Information Security and Applications, 2022. Vol.68. C. 103–124. URL: https://www.sciencedirect.com/science/article/pii/ S22142126 2200067X (дата звернення: травень 2025).

18. ISO/IEC 27001:2022**.** Information Security Management Systems Requirements. International Organization for Standardization, 2022. 36 с. URL: https://www.iso.org/standard/82875.html (дата звернення: травень 2025).

19. Boneh D. Twenty Years of Attacks on the RSA Cryptosystem. Notices of the AMS, 1999. Vol. 46, No. 2. C. 203–213. URL: https://www.ams.org/notices/199902/ boneh.pdf (дата звернення: травень 2025).

20. Katz J., Lindell Y. Introduction to Modern Cryptography. 3rd ed. CRC Press, 2020. 648с. URL: https://www.crcpress.com/Introduction-to-Modern-Cryptography/ Katz-Lindell/p/book/9780815354369 (дата звернення: травень 2025).

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: *Менеджмент криптографічних профілів користувачів розподілених комп'ютерних систем*

Обсяг пояснювальної записки 82 аркушів:

4 таблиць;

9 рисунків;

- додатки.

Дата завершення роботи*: 12 червня 2025р*.

Підпис студента- _____*Пиж Р.Ю.*