

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет радіоелектроніки та телекомунікацій
(повне найменування інституту, назва факультету)

Кафедра захисту інформації
(повна назва кафедри)

Пояснювальна записка

до дипломного проекту (роботи)
бакалавр
(ступінь вищої освіти)

на тему АНАЛІЗ СТАНДАРТУ ХЕШУВАННЯ УКРАЇНИ ДСТУ 7564:2014
ANALYSIS OF THE UKRAINIAN HASHING STANDARD DSTU 7564:2014

Виконав: студент 4 курсу, групи РТ-819

Спеціальності 125 Кібербезпека
(код і назва напрямку підготовки, спеціальності)

Освітня програма (спеціалізація)

Безпека інформаційних і
телекомунікаційних систем

КОВАЛЬОВ І.Є.
(прізвище та ініціали)

Керівник КОЗИНА Г.Л.
(прізвище та ініціали)

Рецензент БОРЮ С.Ю.
(прізвище та ініціали)

2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет радіоелектроніки та телекомунікацій
Кафедра захисту інформації
Ступінь вищої освіти бакалавр
Спеціальність 125 Кібербезпека
(код і найменування)
Освітня програма (спеціалізація) Безпека інформаційних і телекомунікаційних систем
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ЗІ, к.т.н, доц.

В.О. ВОСКОБОЙНИК

“01” “06”

2023 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

КОВАЛЬОВА Іллі Євгеновича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Аналіз стандарту хешування України ДСТУ 7564:2014

Analysis of the Ukrainian hashing standard DSTU 7564:2014

керівник проєкту (роботи) к.ф-м.н., доцент КОЗІНА Галина Леонідівна,

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові,)

затверджені наказом закладу вищої освіти від “04” квітня 2023 року № 93

2. Строк подання студентом проєкту (роботи) 01 червня 2023 року

3. Вихідні дані до проєкту (роботи) виконати порівняльний аналіз хеш-функції Купина ДСТУ 7564:2014 із іншими актуальними хеш-функціями

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Теоретичні відомості про хешування та стандарт «Купина» ДСТУ 7564:2014, програмна реалізація хеш-функції «Купина» ДСТУ 7564:2014, Порівняльний аналіз властивостей хеш-функції «Купина» ДСТУ 7564:2014 із її аналогами

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
Розділи 1-3	КОЗІНА Г.Л., к.ф-м.н. доцент		
Нормоконтроль	КОРОЛЬКОВ Р. Ю., к.т.н. ст.викл.		

7. Дата видачі завдання « 01 » березня 2023 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Програмна реалізація функції хешування Купина ДСТУ 7564:2014 на мові C#	2 тиждень	Розділ 2
4	Порівняльний аналіз ДСТУ 7564:2014 із аналогами	3-4 тижні	Розділ 3
5	Оформлення пояснювальної записки та відповідної документації	6-7 тиждень	Додатки
7	Нормоконтроль та рецензування.	8 тиждень	
8	Захист дипломної роботи.	9 тиждень	

Студент(ка)

_____ Ілля КОВАЛЬОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

_____ Галина КОЗІНА
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра:
89 с., 8 табл., 12 рис., 3 дод., 24 джерела.

ДСТУ 7564:2014, КУПИНА, МОВА ПРОГРАМУВАННЯ С#,
ТЕСТУВАННЯ ФУНКЦІЙ ХЕШУВАННЯ, ХЕШ-ФУНЦІЯ, MD5, SHA256

Об'єкт дослідження – функція хешування Купина ДСТУ 7564:2014.

Предмет дослідження – порівняльний аналіз функції хешування Купина ДСТУ 7564:2014 із аналогами.

Мета роботи – порівняльний аналіз функції Купина з метою виокремлення її переваг та недоліків у порівнянні із популярними аналогами, такими як MD5 та SHA256/512.

Для реалізації програмної частини використовувалася мова програмування С# (.Net-Framework 4.8) та середовище розробки Visual Studio 2023.

В результаті було розроблено бібліотеку DLL, яку можна використовувати у будь-якому додатку на будь-якій мові платформи .Net-Framework. Було проведено порівняльний аналіз за наступними показниками: стійкість до колізій, швидкодія та показник середньої відстані Хеммінга у тестуванні на аваланшний ефект.

ABSTRACT

Explanatory note to the bachelor's thesis: 88 p., 8 tables, 12 figures, 3 appendix, 24 sources.

C# PROGRAMMING LANGUAGE, DSTU 7564:2014, HASH FUNCTION, KUPYNA, MD5, SHA256, TESTING OF HASHING FUNCTIONS

The object of research is the Kupyna hash function of DSTU 7564:2014.

The subject of the study is a comparative analysis of the Kupyna hashing function of DSTU 7564:2014 with analogues.

The purpose of the study is a comparative analysis of the Kupyna function in order to highlight its advantages and disadvantages in comparison with popular analogues such as MD5 and SHA256/512.

The C# programming language (.Net-Framework 4.8) and the Visual Studio 2023 development environment were used to implement the software part.

As a result, a DLL library was developed that can be used in any application in any language of the .Net-Framework platform. A comparative analysis was carried out for the following indicators: collision resistance, performance, and average Hamming distance in avalanche testing.

ЗМІСТ

Перелік скорочень	7
Вступ.....	8
1 Теоретичні відомості про хешування та стандарт «Купина» ДСТУ 7564:2014	10
1.1 Застосування хеш-функцій в інформаційній безпеці	10
1.2 Властивості хеш-функції «Купина» ДСТУ 7564:2014.....	12
1.3 Алгоритм роботи хеш-функції «Купина» ДСТУ 7564:2014.....	13
2 Програмна реалізація хеш-функції «Купина» ДСТУ 7564:2014.....	19
2.1 Обґрунтування вибору мови програмування	19
2.2 Розробка програмного забезпечення хеш-функції «Купина» ДСТУ 7564:2014.....	21
2.3 Тестування програми на відповідність із варіантами із ДСТУ 7564:2014	30
2.4 Методика роботи з програмою	32
3 Порівняльний аналіз властивостей хеш-функції «Купина» ДСТУ 7564:2014 із її аналогами	35
3.1 Аналіз функцій на стійкість до колізій	35
3.2 Аналіз функцій на аваланшний ефект із вимірюванням середньої відстані Хеммінга.....	47
3.3 Аналіз швидкості обчислення хеш-образу	50
Висновки	60
Перелік джерел посилання	61
Додаток А Код реалізації функції хешування «Купина».....	64
Додаток Б Код реалізації графічного інтерфейсу	76
Додаток В Презентація	83

ПЕРЕЛІК СКОРОЧЕНЬ

ДСТУ – Державний стандарт України;

AES-NI (Advanced Encryption Standard New Instructions) – нові інструкції розширеного стандарту шифрування;

AVX (Advanced Vector Extensions) – просунуті векторні розширення;

DLL (Dynamic Link Library) - бібліотека динамічних посилань;

GF (Galois field) – поле Галуа;

IL (Intermediate Language) - проміжна мова;

MD5 (Message Digest 5) – хеш-функція;

RIPEMD (RACE Integrity Primitives Evaluation Message Digest) – хеш-функція;

SHA (Secure Hash Algorithm) – безпечний алгоритм хешування;

SIMD (Single instruction, multiple data) – одиночний потік команд, множинний потік даних;

SSE (Streaming SIMD Extensions) – потокові SIMD розширення;

UWP (Universal Windows Platform) – універсальна платформа Windows

WPF (Windows Presentation Foundation) – графічна підсистема Windows;

ВСТУП

У сучасному світі інформаційних технологій та цифрової безпеки захист даних стає все більш важливим. Хешування є одним з основних криптографічних методів, що забезпечує цілісність даних у різних застосуваннях, таких як блокчейн, цифрові підписи, аутентифікація користувачів тощо.

У 2014 році Україна розробила свій власний стандарт хешування – «Купина», який описано в ДСТУ 7564:2014. Цей стандарт було створено з метою застосування в системах електронного документообігу та інших сферах, де необхідний високий рівень безпеки інформації.

Актуальність теми цієї дипломної роботи визначається важливістю аналізу стандарту хешування «Купина» з метою виявлення його переваг та недоліків, а також порівняння його характеристик з іншими відомими стандартами хешування.

Метою цієї дипломної роботи є проведення детального аналізу стандарту хешування «Купина», виявлення його властивостей та характеристик, а також порівняння його з іншими популярними стандартами хешування.

Завданнями дослідження є:

- вивчити теоретичні основи хешування та основні принципи хеш-функцій;
- розглянути стандарт хешування України та його алгоритм;
- проаналізувати характеристики та властивості стандарту;
- провести порівняльний аналіз ДСТУ 7564:2014 з іншими відомими стандартами хешування;
- оглянути програмне забезпечення, що підтримує стандарт;
- розробити програмне забезпечення для демонстрації роботи стандарту;

– виконати практичні випробування стандарту хешування та проаналізувати отримані результати.

Методи дослідження включають аналітичний огляд наукової літератури та нормативних документів, теоретичний аналіз стандарту хешування, програмну реалізацію, порівняльний аналіз різних стандартів хешування, та проведення практичних випробувань.

У цій дипломній роботі будуть висвітлені основні аспекти стандарту хешування «Купина», аналіз його характеристик та властивостей, а також проведення порівняльного аналізу за іншими стандартами хешування, що допоможе визначити перспективи його подальшого застосування та розвитку.

Структура дипломної роботи буде включати такі розділи: вступ, теоретичні відомості про хешування та стандарт «Купина» ДСТУ 7564:2014, програмна реалізація хеш-функції «Купина» ДСТУ 7564:2014, порівняльний аналіз властивостей хеш-функції «Купина» ДСТУ 7564:2014 із її аналогами та висновки по результатам аналізу.

В ході виконання роботи буде реалізовано функцію хешування «Купина» у вигляді бібліотеки, зроблено графічний інтерфейс для взаємодії із бібліотекою та виконано ряд тестів, зокрема: тестування на стійкість до колізій, виявлення середньої відстані Хеммінга в тестуванні на аваланшний ефект та швидкодію алгоритму.

Виконання цієї дипломної роботи допоможе отримати повноцінний аналіз стандарту хешування «Купина», виявити його сильні та слабкі сторони, а також визначити потенціал його застосування в різних сферах інформаційної безпеки.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ХЕШУВАННЯ ТА СТАНДАРТ «КУПИНА» ДСТУ 7564:2014

1.1 Застосування хеш-функцій в інформаційній безпеці

Хешування є фундаментальним поняттям в інформаційних технологіях та комп'ютерних науках [1]. Хеш-функції перетворюють дані будь-якої довжини в рядок фіксованої довжини, що має ряд властивостей, які роблять їх корисними для різних застосувань. Функції, що здійснюють таке перетворення, називаються хеш-функціями або функціями згортання, а результати їхньої роботи – хешами або хеш-сумами. Загальна схема обчислення відображена на рисунку 1.1. На цій схемі можна побачити, як відбувається колізія, коли на різні входні дані обчислюється однаковий хеш-образ повідомлення

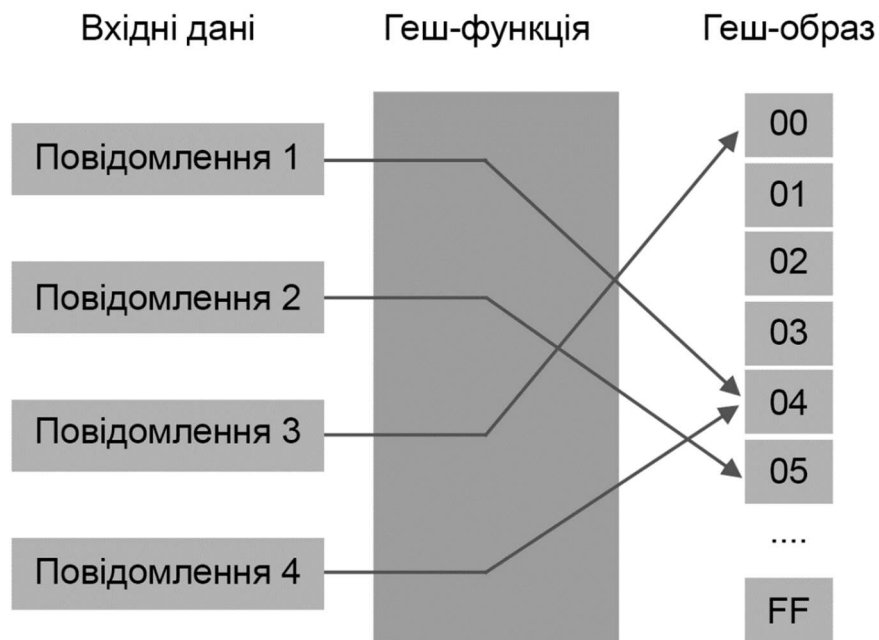


Рисунок 1.1 – Схематичне відображення принципу роботи хеш-функції

Хеш-функції повинні мати наступні основні властивості:

- визначеність: при однакових вхідних даних хеш-функція повинна повертати однаковий хеш;
- швидкість: хеш-функції повинні працювати швидко та ефективно, забезпечуючи високу продуктивність;
- аваланшний ефект: навіть незначна зміна вхідних даних має призводити до значної зміни хешу, що ускладнює знаходження зв'язку між схожими даними та їх хешами;
- відсутність колізій: хоча ідеальна хеш-функція має гарантувати відсутність колізій, на практиці абсолютна відсутність колізій неможлива через обмеженість простору хешів.

Основною метою є мінімізація колізій та зробити їхнє виявлення та експлуатацію вкрай складними. Криптографічні хеш-функції дозволяють перевіряти, чи відповідають вхідні дані заданому значенню хешу, але при невідомих вхідних даних відновлення вхідного значення (або еквівалентної альтернативи) за збереженим значенням хеш-функції стає складним завданням.

Криптографічні хеш-функції мають ще одну важливу властивість: однобічність. Це означає, що відновлення вхідного значення за значенням хеш-функції повинно бути надзвичайно складним завданням. У контексті криптографії, «складним» мається на увазі, що це вимагає непрактично великої кількості обчислювальних ресурсів.

Хеш-функції часто асоціюються з контрольними сумами, контрольними цифрами, відбитками пальців, функціями які обчислюють випадкове значення, кодами для виправлення помилок та шифрами. Хоча ці концепції мають певні спільні риси, кожна з них має свою власну область застосування.

Хешування відіграє важливу роль в різних сферах інформаційної безпеки, включаючи аутентифікацію, цифрові підписи, перевірку цілісності

даних та технологію блокчейн. Це дозволяє забезпечити безпеку інформації та довіру до переданих даних між користувачами і системами.

Наприклад, при аутентифікації користувачів хеш-функції допомагають перевірити, чи відповідає введений пароль хешу, збереженому на сервері, без необхідності передачі та зберігання відкритого тексту пароля. Це поліпшує безпеку аутентифікації та захищає дані користувачів.

У цифрових підписах хеш-функції використовуються для підписання дайджестів повідомлень, що дозволяє перевірити авторство та цілісність переданих даних. Вони також використовуються в блокчейн-технологіях, де хешування допомагає забезпечити взаємозв'язок блоків та можливість відстежувати транзакції.

1.2 Властивості хеш-функції «Купина» ДСТУ 7564:2014

Ітеративна хеш-функція «Купина», яка представлена в ДСТУ 7564:2014 є сімейством криптографічних хеш-функцій, розроблених в Україні в «Інституті інформаційних технологій», який введено в дію від 2 грудня 2014 року та чинний від 1 квітня 2015 року [2].

Як вказано в ДСТУ 7564:2014: «цей стандарт установлює алгоритм обчислення хеш-значення для послідовностей двійкових символів, що застосовують у криптографічних методах захисту, для забезпечення цілісності та автентичності інформації під час її передавання, обробки і зберігання, зокрема під час використання електронного цифрового підпису, що визначений у ДСТУ 4145».

«Купина» має на меті забезпечити стійкість до атак на MD5 і SHA-1, старіші хеш-функції, які виявились вразливими до ряду атак. Проблема в тому, що алгоритми MD5 та SHA-1 дуже давно вже не є безпечним і існує багато алгоритмів швидкого зламу цих хеш-функцій. Наприклад, для MD5 існує метод Ван Сяююня й Юй Хунбо, який побудовано на ітераційному методі Меркла-Дамгора і є різновидом диференційної атаки. Здійснення цієї атаки

дозволяє знайти колізію для MD4 менш ніж за секунду, а також її можна застосувати для RIPEMD та HAVAL. Що до SHA-1, через свою блокову та ітеративну структуру, а також відсутність обробки в кінці хешування, всі функції сімейства SHA вразливі для атак «подовження повідомлення» і колізіями при частковому хешування повідомлення.

Автори ДСТУ 7564:2014 запевняють, що диференціальні атаки та rebound-атаки (від англ. «rebound» - відскік – це криптоаналітична атака, яка використовується для злому хеш-функцій, зокрема, функцій, що використовують ітеративні конструкції з внутрішніми станами, таких як циклічні блокові шифри) неефективні вже після 4 ітерацій функцій перестановок. [3]

«Купина» включає декілька різних варіантів хеш-функцій, зокрема Купина-256, Купина-384 та Купина-512. Ці варіанти відрізняються вихідною довжиною хешу (256, 384 або 512 біт) і різними параметрами безпеки. Функція «Купина-256» рекомендується для більшості застосувань, де потрібна криптографічна стійкість.

«Купина» використовується в різних застосуваннях, таких як аутентифікація, цифрові підписи, перевірка цілісності даних та технології блокчейн. Використовується в українських електронних цифрових підписах, системах електронного документообігу та різних додатках для забезпечення конфіденційності і цілісності даних

1.3 Алгоритм роботи хеш-функції «Купина» ДСТУ 7564:2014

Як вже було зазначено, це ітеративна хеш-функція, а з цього можна зрозуміти, що обробка даних відбувається в декілька етапів (ітерацій). Кожна ітерація функції має відповідний вплив на вихідне значення, а результат попередньої ітерації використовується як вхід для наступної ітерації. Такий підхід допомагає забезпечити кращу криптографічну стійкість.

Спочатку вхідне повідомлення M доповнюється до довжини, кратної розміру блоку, кожне повідомлення доповнюється незалежно від його довжини. У кінець повідомлення додається одиничний біт і далі інший простір заповнюється нульовими бітами у кількості d (1.1)

$$d = (-N - 97) \bmod I, \quad (1.1)$$

де N – довжина повідомлення M ,

I – розмір внутрішнього стану $\in \{512, 1024\}$

Після виконання цієї операції, додається ще 96 біт, в яких записано значення N (максимальна довжина повідомлення, що може бути оброблена, складає $2^{96}-1$ біт. На цьому етапі обробки вхідного повідомлення закінчено, і після цього повідомлення розбивається на блоки $m_1, m_2 \dots m_k$ довжиною l . Залежність розміру внутрішнього стану l , кількості ітерацій t та кількості колонок матриці c від розміру хеш-значення n наведено у таблиці 1.1

Таблиця 1.1 – Таблиця розміру внутрішнього стану

Розмір хеш-значення	Розмір внутрішнього стану l	Кількість ітерацій перетворення t	Кількість колонок у матриці c
$8 \leq n \leq 256$	512	10	8
$256 \leq n \leq 512$	1024	14	16

З цього етапу починається обчислення хеш-функції за наступною ітеративною процедурою (1.2):

$$\begin{aligned}
 h_0 &= IV, \\
 h_v &= T_I^{\oplus}(h_{v-1} \oplus m_v) \oplus T_I^+(m_v) \oplus h_{v-1}, v = 1, 2, \dots, k \quad (1.2) \\
 H(IV, M) &= R_{I,n}(T_I^{\oplus}(h_k) \oplus h_k),
 \end{aligned}$$

де $IV = \begin{cases} 1 \ll 510 & \text{для } I = 512 \\ 1 \ll 1023 & \text{для } I = 1024 \end{cases}$ – вектор ініціалізації довжиною I біт;

$T_I^\oplus, T_I^+$ - бієктивні перетворення, що виконують відображення вхідного блоку довжиною I біт у вихідній такої самої довжини

$R_{I,n}(x)$ - функція, що повертає n старших біт з вхідного блоку x довжиною I біт ($n < I$), де результат записується в молодші n біт обчисленого значення.

Структурну схему обчислення функції хешування наведено на рисунку 1.2.

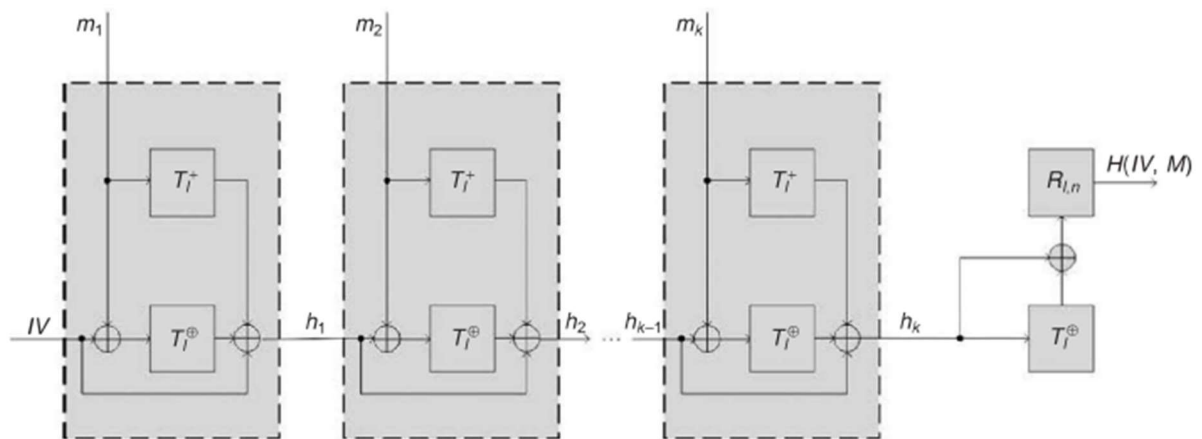


Рисунок 1.2 – Структурна схема функції хешування «Купина»

Власне, бієктивні перетворення $T_I^\oplus, T_I^+$, представлені у вигляді ітеративного застосування низки функцій, що оброблюють вхідний аргумент $x \in V_I$, як матрицю розміром $8 \times c$ байтів, що містить елементи поля $GF(2^8)$.

Матриця внутрішнього стану позначена як $G \in (g_{i,j}), g_{i,j} \in GF(2^8)$ де $i = \overline{0,7}, j = \overline{0,c-1}$. Записуються байти $V_1, V_2, \dots, V_{I/8}$ перетворень $T_I^\oplus, T_I^+$ до матриці та зчитують з неї за колонками дані (приклад на рисунку 1.3).

$T_I^\oplus, T_I^+$ визначено формулами (1.3) та (1.4) відповідно.

$$T_I^{\oplus} = \prod_{v=0}^{t-1} (\psi \circ \tau^{(I)} \circ \pi' \circ \kappa_v^{(I)}), \quad (1.3)$$

$$T_I^{+} = \prod_{v=0}^{t-1} (\psi \circ \tau^{(I)} \circ \pi' \circ \eta_v^{(I)}), \quad (1.4)$$

де $\kappa_v^{(I)}$ - функція додавання констант ітерацій за модулем 2;

$\eta_v^{(I)}$ - функція додавання констант ітерацій за модулем 2^{64} ;

π' - шар нелінійного бієктивного відображення, який виконує байтову підстановку елементів матриці внутрішнього стану $G = (g_{i,j})$;

$\tau^{(I)}$ - перестановка елементів $g_{i,j} \in GF(2^8)$ внутрішнього стану (циклічний зсув у разі матричного подавання);

Ψ - лінійне перетворення (множення вектора на матрицю над скінченним полем).

Вхідна послідовність							
B_1	B_9	B_{17}	B_{25}	B_{33}	B_{41}	B_{49}	B_{57}
B_2	B_{10}	B_{18}	B_{26}	B_{34}	B_{42}	B_{50}	B_{58}
B_3	B_{11}	B_{19}	B_{27}	B_{35}	B_{43}	B_{51}	B_{59}
B_4	B_{12}	B_{20}	B_{28}	B_{36}	B_{44}	B_{52}	B_{60}
B_5	B_{13}	B_{21}	B_{29}	B_{37}	B_{45}	B_{53}	B_{61}
B_6	B_{14}	B_{22}	B_{30}	B_{38}	B_{46}	B_{54}	B_{62}
B_7	B_{15}	B_{23}	B_{31}	B_{39}	B_{47}	B_{55}	B_{63}
B_8	B_{16}	B_{24}	B_{32}	B_{40}	B_{48}	B_{56}	B_{64}



Внутрішній стан функції гешування							
$g_{0,0}$	$g_{0,1}$	$g_{0,2}$	$g_{0,3}$	$g_{0,4}$	$g_{0,5}$	$g_{0,6}$	$g_{0,7}$
$g_{1,0}$	$g_{1,1}$	$g_{1,2}$	$g_{1,3}$	$g_{1,4}$	$g_{1,5}$	$g_{1,6}$	$g_{1,7}$
$g_{2,0}$	$g_{2,1}$	$g_{2,2}$	$g_{2,3}$	$g_{2,4}$	$g_{2,5}$	$g_{2,6}$	$g_{2,7}$
$g_{3,0}$	$g_{3,1}$	$g_{3,2}$	$g_{3,3}$	$g_{3,4}$	$g_{3,5}$	$g_{3,6}$	$g_{3,7}$
$g_{4,0}$	$g_{4,1}$	$g_{4,2}$	$g_{4,3}$	$g_{4,4}$	$g_{4,5}$	$g_{4,6}$	$g_{4,7}$
$g_{5,0}$	$g_{5,1}$	$g_{5,2}$	$g_{5,3}$	$g_{5,4}$	$g_{5,5}$	$g_{5,6}$	$g_{5,7}$
$g_{6,0}$	$g_{6,1}$	$g_{6,2}$	$g_{6,3}$	$g_{6,4}$	$g_{6,5}$	$g_{6,6}$	$g_{6,7}$
$g_{7,0}$	$g_{7,1}$	$g_{7,2}$	$g_{7,3}$	$g_{7,4}$	$g_{7,5}$	$g_{7,6}$	$g_{7,7}$

Рисунок 1.3 – Заповнення матриці внутрішнього стану для $I = 512$ та $c = 8$

У вище приведених функціях вхідний аргумент $x \in V_I$, вихідне значення $\chi(x) \in V_I$ розгуляється як матриця розміром $8 \times c$ байтів.

Розглянемо більш детально вище наведені функції.

Функції додавання констант ітерацій представлено двома функціями:

– $\kappa_v^{(I)}$ - здійснює додавання за модулем 2 до кожної колонки G_j матриці внутрішнього стану $G = (g_{i,j})$ вектора $\omega_j^{(v)} \in V_{64}$, де v – номер ітерації, $\omega_j^{(v)} = ((j \ll 4) \oplus v, 0, 0, 0, 0, 0, 0, 0)^T$;

– $\eta_v^{(I)}$ – здійснює додавання за модулем 2^{64} до кожної колонки G_j матриці внутрішнього стану $G = (g_{i,j})$ вектора $\zeta_j^{(v)} \in V_{64}$, де v – номер ітерації, $\zeta_j^{(v)} = (0xF3, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0), ((c-1)-j) \ll 4 \oplus v)^T$, при цьому під час операції додавання $0xF3$ – молодші 8 біт вектора $\zeta_j^{(v)}$, $g_{0,j}$ – молодші 8 біт вектора G_j .

Шар нелінійного бієктивного відображення представлено функцією π' , яка виконує заміну кожного $g_{i,j}$ елемента матриці внутрішнього стану G на $\pi_{j \bmod 4}(g_{i,j})$, де $\pi_s V_8 \rightarrow V_8, s \in \{0, 1, 2, 3\}$, простіше кажучи, це заміна за допомогою таблиці підстановки Sbox (до речі, яка співпадає з таблицями для блочного симетричного шифру ДСТУ 7624:2014 «Калина»).

Перетворення елементів представлено функцією $\tau^{(I)}$, яка виконує циклічний зсув праворуч рядків матриці стану $G = (g_{i,j})$. Рядки з номерами $i = 0, 1, 2, \dots, 6$ матриці зсуваються на i елементів, а рядок з номером 7 зсувається на 7 елементів для $I = 512$ і на 11 елементів для $I = 1024$.

Лінійне перетворення представлено функцією Ψ , під час обчислення якої кожен елемент $g_{i,j}$ матриці внутрішнього стану $G = (g_{i,j})$ розглядають як елемент скінченного поля $GF(2^8)$, що утворене незвідним поліномом $\vartheta(x) = x^8 + x^4 + x^3 + x^2 + 1$, або *0x11d* у шістнадцятковому поданні.

В результаті цих операцій отримується результуюча матриця стану $U = (u_{i,j})$ як результат множення векторів довжини 8 над скінченним полем $GF(2^8)$ за формулою (1.5).

$$u_{i,j} = (v \ggg i) \oplus G_j \quad (1.5)$$

де $v = (0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04)$ – вектор, що утворює цикрулянтну матрицю МДР-коду (клас лінійних блокових кодів, які використовуються для виявлення та корекції помилок в передачі даних) і складається з послідовності байтових констант у шістнадцятковому поданні, які інтерпретують як елементи поля $GF(2^8)$, при цьому циклічний зсув виконується відносно елементів вектора.

Саме за таким алгоритмом відбувається процес хешування повідомлення, якщо казати більш загально, то хешування відбувається в три етапи:

- доповнення повідомлення та перетворення в блоки;
- перетворення блоків за $T_I^\oplus$ та T_I^+ які включають в себе послідовне виконання функцій множення вектора на матрицю над скінченним полем, циклічний зсув, підстановка елементів матриці за таблицею заміни, додавання за модулем 2 та 2^{64} відповідно;
- поєднання та вивід результатів.

2 ПРОГРАМНА РЕАЛІЗАЦІЯ ХЕШ-ФУНКЦІЇ «КУПИНА» ДСТУ 7564:2014

2.1 Обґрунтування вибору мови програмування

Для програмної реалізації хеш-функції «Купина» ДСТУ 7564:2014, було обрано мову програмування C# з наступних причин [4].

1. Висока продуктивність: C# є мовою програмування зі швидким часом виконання завдяки оптимізованому управлінню пам'яттю та компіляції Just-In-Time (JIT). Це забезпечує достатній рівень продуктивності для обробки та генерації хешів.

2. Об'єктно-орієнтований підхід: C# є об'єктно-орієнтованою мовою програмування, що дозволяє створювати модульні та гнучкі програми з використанням класів, наслідування, інтерфейсів та поліморфізму. Це спрощує процес розробки, тестування та підтримки коду.

3. Зручність використання: C# має простий та зрозумілий синтаксис, який сприяє швидкому розумінню та вивченню мови. Завдяки інтуїтивному синтаксису, процес розробки програми на C# відбувається швидше та ефективніше.

4. Безпека: C# відомий своєю високою безпекою завдяки статичній типізації, автоматичному управлінню пам'яттю та обробці виняткових ситуацій. Ці властивості забезпечують надійну роботу програми та зменшують вірогідність виникнення потенційних помилок.

5. Широка підтримка .NET Framework: C# є основною мовою програмування для .NET Framework, який надає доступ до широкому спектру бібліотек та фреймворків. Це значно спрощує розробку програм, оскільки розробник може використовувати перевірені та надійні компоненти, які вже доступні в рамках платформи .NET.

6. Платформонезалежність: Завдяки підтримці .NET Core та .NET 5, програми, написані на C#, можуть працювати на різних операційних системах,

таких як Windows, Linux та macOS. Це забезпечує гнучкість та платформонезалежність програм, що базуються на хеш-функції «Купина» (ДСТУ 7564:2014).

7. Спільнота та підтримка: C# має велику спільноту розробників та широку кількість ресурсів для навчання та підтримки. Це полегшує процес вивчення мови та отримання допомоги у разі виникнення проблем.

8. Сумісність з іншими технологіями: Розроблені на C# бібліотеки можуть легко інтегруватися з іншими технологіями, такими як вебсервіси, бази даних, хмарні сервіси та різноманітні API. Це робить C# відмінним вибором для реалізації хеш-функції «Купина» у широкому спектрі застосувань.

Наразі офіційно існують наступні бібліотеки:

Kyryna-reference [5] – код референсної реалізації на мові C, яка виконана одним із розробників хеш-функції кандидатом технічних наук, Романом Олійниковим.

Cppcrypto [6] – бібліотека з відкритим вихідним кодом на C++, яка реалізує різноманітні функції хешування, блокового шифру, потокового шифру, зокрема функцію хешування «Купина» та блоковий шифр «Калина». До того ж бібліотека включає зразки інструментів командного рядка: «digest» - для обчислення та перевірки контрольної(их) сум(и) файлу за допомогою будь-якого з підтримуваних алгоритмів хешування (подібних до md5sum або RHash) та «cryptor» - для шифрування файлів за алгоритмом Serpent-256 в режимі AEAD.

Cryptonite [7] – бібліотека криптографічних перетворень від PrivatBank з відкритим програмним кодом на C, має експертний висновок UA.14360570.00001-01 90 01-1 за результатами державної експертизи у галузі КЗІ і зазначає, що ця бібліотека може бути використана для криптографічного захисту інформації з обмеженим доступом (крім службової інформації та інформації, що становить державну таємницю) та відкритої інформації, вимога щодо захисту якої встановлена законом.

Це перевірені, відкриті бібліотеки, які можна досить легко використовувати у власних розробках.

Але, еталонною версією реалізації хеш-функції «Купина» є саме Кирупа-reference, через це її було взято за основу для реалізації на мові C#.

2.2 Розробка програмного забезпечення хеш-функції «Купина» ДСТУ 7564:2014

Спираючись на референсну версію реалізації алгоритму на мові програмування C, виділимо з реалізації наступні функції:

- КирупаInit конструктор, який встановлює параметри хешування за таблицею 1.1;
- SubBytes (π'), яка виконує заміну матриці для кожного елемента $g_{i,j}$ за таблицею підстановки;
- ShiftBytes ($\tau^{(I)}$), яка виконує циклічний зсув i -строки;
- MixColumns, яка кожний стовпець матриці множить на матрицю MDS;
- AddRoundConstantP ($\kappa_v^{(I)}$) та AddRoundConstantQ ($\eta_v^{(I)}$);
- функції P та Q що відповідають перетворенню $T_I^{\oplus}$ та T_I^{+} ;
- MultiplyGF – Перемножує два байти у полі Галуа (2^8) і повертає результат;

Це основні функції, які визначені в стандарті, а також допоміжні функції:

- Pad – розбиває вхідні дані на блоки;
- Digest – обробляє вхідні дані та оновлює внутрішній стан;
- Trunc - обрізає внутрішній стан для генерації фінального хеш-коду;
- OutputTransformation – застосовує вихідне перетворення до внутрішнього стану і викликає функцію Trunc для генерації хеш-коду;

- `КирунаHash` – основна функція, яка обчислює хеш Кируна. Вона ініціалізує внутрішній стан, підставляє вхідні дані, обробляє дані та застосовує перетворення на виході;

- `КирунаKmac` – обчислює код автентифікації ключового повідомлення на основі Кируна (КМАС) для заданих ключа, даних та розміру digest. Повертає 0 у разі успіху або -1 у разі помилки;

- `OutputTransformation` – виконує вихідне перетворення для завершення обчислення хешу. Спочатку вона застосовує Р-перетворення до тимчасового стану, потім виконує XOR перетвореного стану з початковим станом. Нарешті, вона викликає `Trunc()`, щоб скоротити кінцевий хеш-код до потрібної довжини;

- `КирунаHash` – розпочинає процес хешування.

А також код програми включає файл, який містить константи матриць `sbox` та `mds_matrix`.

Метою розробки бібліотеки на платформі .Net Framework є забезпечення наступних умов [8]:

- а) Об'єктно-орієнтований підхід, який забезпечить безпеку реалізації алгоритму та при підтриманні принципу відкритості та закритості даних унеможливить втручання в процес виконання алгоритму

- б) Зручність використання функції та зручне під'єднання бібліотеки до будь яких C# та .Net проєктів, незалежність від платформи

Код програми буде розроблюватися в якості бібліотечної функції, яка приймає в себе бітову послідовність повідомлення та число типу `int`, яке буде визначати бітовий розмір хешу (256 або 512). В якості середовища розробки (IDE) було обрано Visual Studio 2023.

Основний клас бібліотеки – `КирунаHash` в просторі імен `КирунаHashFunction` з доступом `internal` - містить такі поля:

- константи для роботи з блоками та раундами;
- розмір хешу, розмір блоку, кількість стовпців, кількість раундів;

- внутрішній стан хеш-функції;
- масив для додавання блоків.

Всі поля, методи та функції в цьому класі з модифікатором доступу `private` або `internal`. Модифікатор доступу `private` надає доступ лише в межах класу, а модифікатор `internal` – в усій збірці або просторі імен, але недоступні при використанні бібліотеки в інших збірках.

Для надання доступу для роботи з бібліотекою реалізовано публічний (`public`) клас `Куруна` який реалізує всього одну публічну функцію `Hash`, через яку і здійснюється запуск процесу хешування та повернення результату.

Код реалізації бібліотеки та класу `Куруна` через який здійснюється робота з функцією хешування наведено в Додатку А.

Цю бібліотеку було зібрано та запаковано в DLL-файл `КурунаHash.dll` [9]. DLL-файл представляє собою файл з бібліотекою, який містить вже скомпільований код: функції, класи, поля та всі інші ресурси, які можуть бути використані іншими програмами.

При використанні DLL-файлів програма забезпечує роздільність коду, що дозволяє кільком програмам використовувати одну й ту ж бібліотеку, забезпечуючи модульність та забезпечення більш легкої можливості оновлення окремих частин програми без необхідності перекомпілювати та знов розгортати весь проєкт. Використання DLL також сприяє зменшенню розміру виконуваних файлів, оскільки бібліотеки можуть бути розділені між кількома програмами.

Щоб використовувати DLL-файл у проєкті `C#`, його необхідно додати як посилання (`reference`), що дозволяє використовувати код, класи, функції та ресурси з цієї бібліотеки у проєкті.

Для взаємодії з бібліотекою було реалізовано наступний код програми:

```
using System;
using System.Linq;
using KupynaHashFunction;

namespace TestKupynaDll
{
    class Program
    {
        public static byte[] StringToByteArray(string hex)
        {
            return Enumerable.Range(0, hex.Length)
                             .Where(x => x % 2 == 0)
                             .Select(x => Convert.ToByte(hex.Substring(x, 16)))
                             .ToArray();
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Hex message: ");
            byte [] input = StringToByteArray(Console.ReadLine());
            Console.WriteLine($"Hash: {BitConverter.ToString(Kupyna.Hash(input, 256)).Replace("-", "«»")});
            Console.ReadLine();
        }
    }
}
```

Детально розглянемо вище наведений код програми:

1. Директиви using вказують на залежності проекту від зовнішніх бібліотек. У даному випадку ми використовуємо стандартні бібліотеки System та System.Linq для роботи з колекціями та конвертації типів, а також власну бібліотеку KupynaHashFunction.
2. Клас Program містить визначення програми.
3. Метод StringToByteArray конвертує рядок шістнадцяткових чисел (hex) у масив байтів. Він робить це, використовуючи LINQ для фільтрації та вибірки кожного парного символу, а потім конвертує їх у відповідні байти.
4. Метод Main є точкою входу програми. Він робить наступне:
 - Виводить повідомлення «Hex message:» та чекає вводу користувача;
 - Конвертує введений рядок шістнадцяткових чисел у масив байтів, використовуючи метод StringToByteArray;
 - Обчислює 256-бітний хеш від отриманого масиву байтів за допомогою методу Kupyna.Hash;

- Виводить обчислений хеш у форматі рядка шістнадцяткових чисел;

- Очікує на натискання клавіші Enter перед закриттям програми.

Для більш зручного користування бібліотекою можна створити графічний інтерфейс користувача. Виділимо функції, які повинен надавати інтерфейс користувачеві:

- хешування натисканням однієї кнопки;
- зручна зміна довжини хешу
- зміна кодування (ASCII, UTF-8, HEX)
- викриття файлів, запис результату хешування в файл

Однією з найсучасніших графічних бібліотек в Windows є UPW (Universal Platform for Windows) [10] та WPF (Windows Presentation Foundation) [11] - це дві різні технології, які використовуються для розробки застосунків на платформі Windows. Обидві технології мають свої переваги та недоліки, порівняємо їх:

а) мета;

- 1) UPW створений для того, щоб допомогти розробникам створювати застосунки, які можуть працювати на всіх пристроях Windows з мінімальними змінами в коді;
- 2) WPF (Windows Presentation Foundation) - це технологія для створення графічного інтерфейсу користувача для настільних застосунків Windows. Вона використовує векторну графіку та апаратне прискорення для забезпечення гладкої роботи та високої продуктивності;

б) архітектура;

- 1) UPW використовує нову архітектуру, яка дозволяє застосункам працювати на різних пристроях Windows з мінімальними змінами в коді.

2) WPF використовує архітектуру .NET Framework, що забезпечує відмінну підтримку настільних додатків.

в) WPF має більшу кількість доступних бібліотек та компонентів, оскільки ця технологія існує довгий час. Це дозволяє розробникам швидше створювати складні застосунки з підтримкою багатофункціональних елементів. UPW має меншу кількість доступних компонентів, але спільнота продовжує розвивати нові бібліотеки та покращувати вже існуючі.

г) UPW розроблений спеціально для підтримки всіх пристроїв Windows 11 та Windows 10. WPF же є сумісним зі старими версіями Windows, починаючи від Windows XP, що робить його більш універсальним для настільних застосунків.

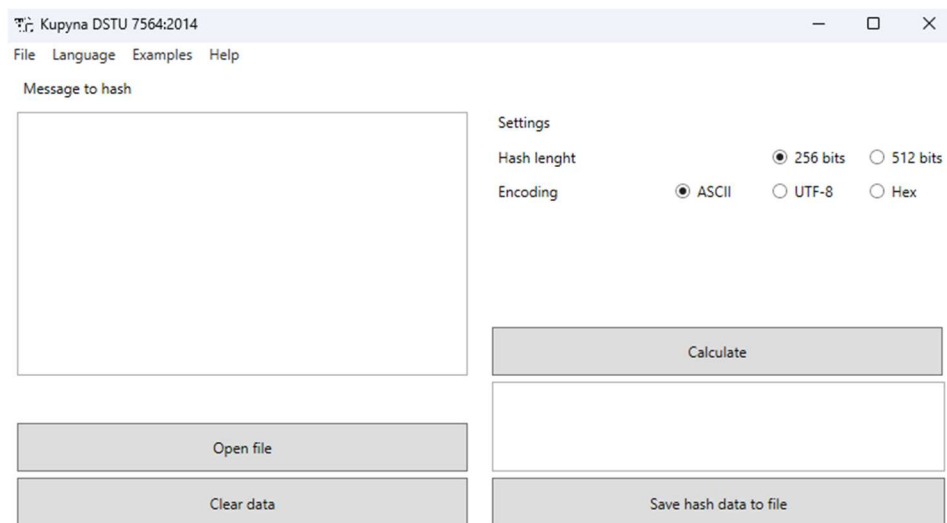


Рисунок 2.1 – Інтерфейс додатку

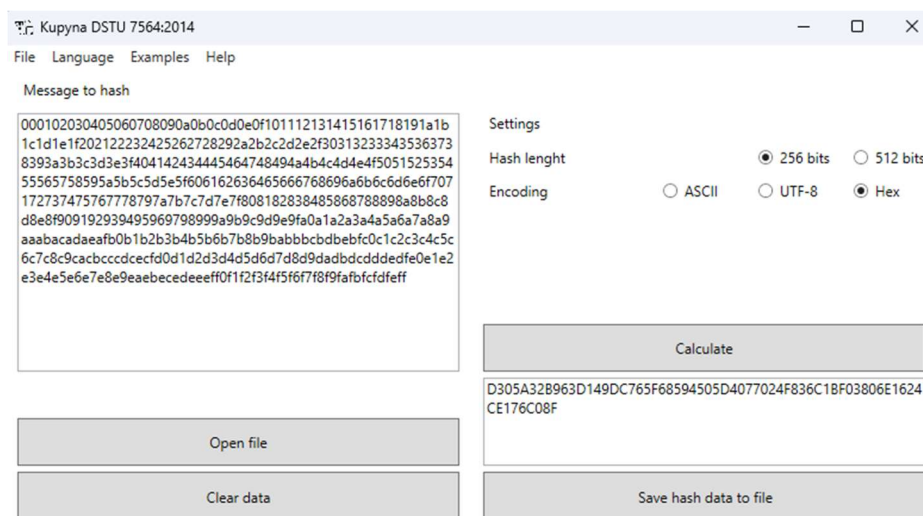


Рисунок 2.2 – Процес виконання програми

З огляду на ці характеристики, було зроблено вибір в сторону WPF. Додаток на WPF пишеться за допомогою мови розмітки XAML (код представлено в додатку Б), але для більш зручного процесу створення в середовищі розробки Visual Studio присутній візуальний конструктор.

За його допомогою було створено інтерфейс користувача з наступними компонентами (рисунок 2.1):

- Window: Основне вікно додатка з налаштуваннями розміру, заголовка та іншими параметрами;
- Grid: Основний контейнер для розміщення елементів інтерфейсу користувача. Він має дві колонки та два рядки;
- TextBox (hash_output): Компонент текстового поля для відображення результату хешування. Властивість IsReadOnly встановлена на «True», що робить це поле тільки для читання;
- Button (Save_Hash): Кнопка для збереження хешу в файл. Обробник подій Click встановлений на «Save_Hash_Click»;
- Label: Елементи текстової мітки, які відображають підписи «Довжина хешу» та «Кодування»;
- Button (OpenFile): Кнопка для відкриття файлу. Обробник подій Click встановлений на «OpenFile_Click»;
- Button (ClearFile): Кнопка для очищення даних. Обробник подій Click встановлений на «ClearFile_Click»;
- Button (Hash_Data): Кнопка для хешування даних. Обробник подій Click встановлений на «Hash_Data_Click»;
- RadioButton: Компоненти радіокнопок для вибору довжини хешу (256 або 512 біт) та кодування (ASCII, UTF-8 або Hex). Вони мають спільні групи імен («hash» та «code») для забезпечення взаємовиключення;
- TextBox (Input_Text): Текстове поле для введення даних користувачем. Властивості ScrollViewer дозволяють відображати вертикальні та горизонтальні смуги прокрутки;

- `TextBlock`: Компонент для відображення тексту, який відключений за допомогою властивості `IsEnabled=«False»`;

- `TextBox (TextBox_FilePath)`: Текстове поле для відображення шляху до відкритого файлу. Властивості `IsEnabled` та `IsReadOnly` встановлені на «False» та «True» відповідно;

В межах простору імен `КурпунаUI` реалізовано статичний клас `DataEncoding`. Він містить статичні методи для роботи з різними типами кодування даних:

- `getBytesASCII(string msg)`: Приймає рядок `msg` та конвертує його в масив байтів за допомогою кодування ASCII. Повертає масив байтів.

- `getBytesUTF8(string msg)`: Приймає рядок `msg` та конвертує його в масив байтів за допомогою кодування UTF-8. Повертає масив байтів.

- `getBytesHex(string msg)`: Приймає рядок `msg` у форматі шістнадцяткового кодування, видаляє пробіли та дефіси, а потім конвертує рядок у масив байтів. Повертає масив байтів.

- `ByteArrayToString(byte[] ba)`: Приймає масив байтів `ba` та конвертує його в рядок у форматі шістнадцяткового кодування. Повертає рядок.

Основним класом, який реалізує методи обробки подій, які викликаються під час взаємодії з компонентами є клас «`MainWindow`», який включає в себе:

- `MainWindow()`: Конструктор класу, який викликає метод `InitializeComponent()` для ініціалізації компонентів вікна;

- `Hash_Data_Click()`: Обробник події `Click` для кнопки «Хешувати». Він перевіряє, який `RadioButton` кодування обраний, конвертує введений текст в масив байтів відповідно до вибраного кодування, а потім викликає асинхронний метод `RunHashProcessAsync()` для хешування повідомлення;

- `RunHashProcessAsync()`: Асинхронний метод для хешування повідомлення. Він відключає інтерфейс, викликає асинхронний метод `HashMessageAsync()` для хешування, а потім включає інтерфейс назад;
- `HashMessageAsync()`: Асинхронний метод для виконання хешування повідомлення за допомогою функції `КурьнаHashFunction.Курьна.Hash()` та заданої довжини хешу;
- `TurnInterface()`: Метод для включення або відключення інтерфейсу. Він змінює властивість `IsEnabled` для всіх компонентів інтерфейсу;
- `OpenFile_Click()`: Обробник події `Click` для кнопки «Відкрити файл». Він викликає асинхронний метод `OpenFileProcAsync()` для відкриття файлу;
- `OpenFileProcAsync()`: Асинхронний метод для відкриття файлу. Він відображає вікно `OpenFileDialog`, зчитує вміст файлу, а потім виводить шлях до файлу та його вміст у відповідних текстових полях;
- `ClearFile_Click()`: Обробник події `Click` для кнопки «Очистити дані». Він очищує текстове поле для введення даних;
- `Save_Hash_Click()`: Обробник події `Click` для кнопки «Зберегти в файл». Він відображає вікно `SaveFileDialog` та зберігає результат хешування в файл.

Важливо відмітити, що методи хешування виклику процесу зчитування файлу виконується асинхронно. Якщо виконувати такі довгі операції в головному потоці, це може заблокувати взаємодію користувача з додатком (наприклад, вікно може «зависнути» або перестати реагувати на дії користувача). Асинхронне виконання операцій дозволяє головному потоку продовжувати обробляти інші події, такі як взаємодія користувача з іншими елементами інтерфейсу.

Метод `HashMessageAsync()` використовує `Task.Run()` для запуску хешування в окремому потоці, відмінному від головного потоку додатка. Це

дозволяє головному потоку продовжувати обслуговувати інші події та запити користувача, поки операція хешування виконується паралельно. Коли операція хешування завершиться, результат повертається до головного потоку за допомогою `await`, і він може бути відображений у вікні додатка.

2.3 Тестування програми на відповідність із варіантами із ДСТУ 7564:2014

Для перевірки програми, було виконано введення прикладів повідомлень із ДСТУ 7564:2014. В документі всі повідомлення вказані у вигляді шістнадцяткового коду. Також було перевірено хешування з використанням довжини хешу в 512 біт. Було здійснено наступні перевірки.

Для $N = 0$ (пуста стрічка):

Hex message:

Hash: 656B2F4CD71462388B64A37043EA55DBE445D452AECD46C3298343314EF0

4019BCFA3F04265A9857F91BE91FCE197096187CEDA78C9C1C021C294A0689198538

Для $N = 8$, Input = «FF»:

Hex message: FF

Hash: 871B18CF754B72740307A97B449ABEB32B64444CC0D5A4D65830AE5456837A

72D8458F12C8F06C98C616ABE11897F86263B5CB77C420FB375374BEC52B6D0292

Для $N = 512$, Input = «000102030405060708090A0B0C0D0E0F10111213
1415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F3
03132333435363738393A3B3C3D3E3F»:

Hex message: 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1

C1D1E1F202122232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F

Hash: 3813E2109118CDB5A6D5E72F7208DCCC80A2DFB3AFDFB02F46992B5EDBE536
B3560DD1D7E29C6F53978AF58B444E37BA685C0DD910533BA5D78EFFFFC13DE62A

Для N = 2048, Input = «000102030405060708090A0B0C0D0E0F101112
131415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2
F303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E
4F505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6
E6F707172737475767778797A7B7C7D7E7F808182838485868788898A8B8C8D
8E8F909192939495969798999A9B9C9D9E9FA0A1A2A3A4A5A6A7A8A9AA
ABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBBCBDBEBFC0C1C2C3C4C
5C6C7C8C9CACBCCCDCECFD0D1D2D3D4D5D6D7D8D9DADBDCDDDED
FE0E1E2E3E4E5E6E7E8E9EAEBECEDEEEFF0F1F2F3F4F5F6F7F8F9FAFBF
CFDFEFF»:

Hex message: 000102030405060708090A0B0C0D0E0F101112131415161718191A1B
1C1D1E1F202122232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F40414
2434445464748494A4B4C4D4E4F505152535455565758595A5B5C5D5E5F606162636465666768
696A6B6C6D6E6F707172737475767778797A7B7C7D7E7F808182838485868788898A8B8C8D8E8
F909192939495969798999A9B9C9D9E9FA0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5
B6B7B8B9BABBBBCBDBEBFC0C1C2C3C4C5C6C7C8C9CACBCCCDCECFD0D1D2D3D4D5D6D7D8D9DADB
DCDDDEDFE0E1E2E3E4E5E6E7E8E9EAEBECEDEEEFF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF

Hash: D305A32B963D149DC765F68594505D4077024F836C1BF03806E1624CE176C08F

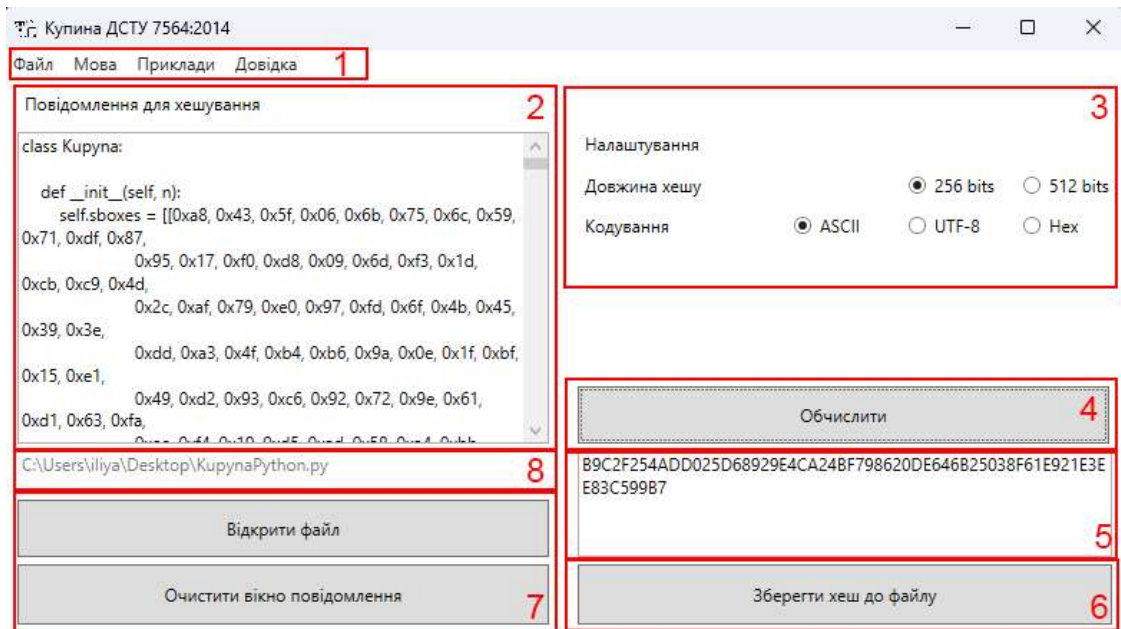
Результати співпадають з результатами хешування, які приведені в ДСТУ. При цьому, якщо такі самі вхідні дані задати, але в кодуванні ASCII, то результат буде відрізнятися:

ASCII message: 000102030405060708090A0B0C0D0E0F101112131415161718191
A1B1C1D1E1F202122232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D
3E3F

Hex encoding: 30303031303230333034303530363037303830393041304230433044

30453046313031313132313331343135313631373138313931413142314331443145314
 63230323132323233323432353236323732383239324132423243324432453246333033313332
 33333334333533363337333833393341334233433344334533460A

Hash: 2E42EED8978E87823BC17A217FF1F55257BF0726644FF9CC5341EFB92C614E84
 1E98CD0F4A48F4407DD4B59558C0FD01563B070702883EF7CF74CC3A69BDEE16



1 – загальне меню, 2 – вікно для введення повідомлення для хешування, 3 – область налаштувань, 4 – кнопка, яка розпочинає процес хешування, 5 – поле, в яке виводиться результат хешування, 6 – кнопка, яка відкриває вікно зберігання результату хешування, 7 – кнопки керування вмістом поля для введення повідомлення для хешування, 8 – назва відкритого файлу

Рисунок 2.3 – Вікно довідки

На основі отриманих даних зроблено висновок, що додаток працює коректно.

2.4 Методика роботи з програмою

Користувачеві надається графічний інтерфейс, який дозволяє йому зручно вводити повідомлення для хешування будь-якого типу із різними варіантами кодування.

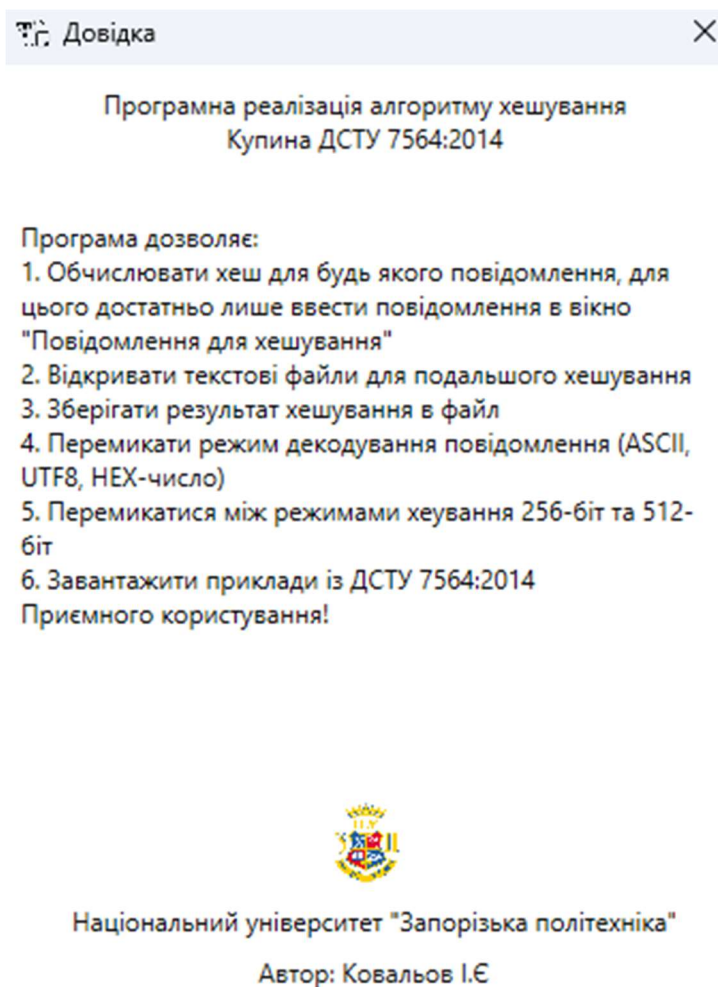


Рисунок 2.4 – Вікно довідки

Інтерфейс програми (рисунок 2.3) складає головне меню, поле для введення повідомлення для хешування, область налаштування хеш-функції та кодування, кнопки відкриття файлу та очистки вмісту поля для введення, а також кнопки для початку процесу хешування та для зберігання результату у файл. Сам хеш виводиться у відповідному полі. Головне меню дублює деякі основні кнопки додатку, такі як відкриття або зберігання повідомлення та хешу, очистка даних, а також надає користувачеві можливість змінити мову, відкрити вікно із прикладами повідомлень із ДСТУ 7564:2014 та довідку, в якій розписані можливості застосунку (рисунок 2.4).

Принцип роботи із застосунком для всіх випадків однаковий: користувач за допомогою поля введення (рисунок 2.3 – 1) вводить своє повідомлення, натискає кнопку «обчислити» (рисунок 2.3 – 4) і отримує результат до поля виводу (рисунок 2.3 – 5). У випадку, якщо користувачеві

потрібно буде змінити довжину хешу (у бітах) або те, як буде розпізнаватися вхідне повідомлення (доступні ASCII або UTF-8 кодування та розпізнання 16-річного коду).

Натиснувши на пункт меню «Приклади» (рисунок 2.5) користувач отримує список повідомлень, за допомогою яких здійснювалося тестування функції хешування в ДСТУ 7564:2014, що допоможе швидко перевірити працездатність програми.

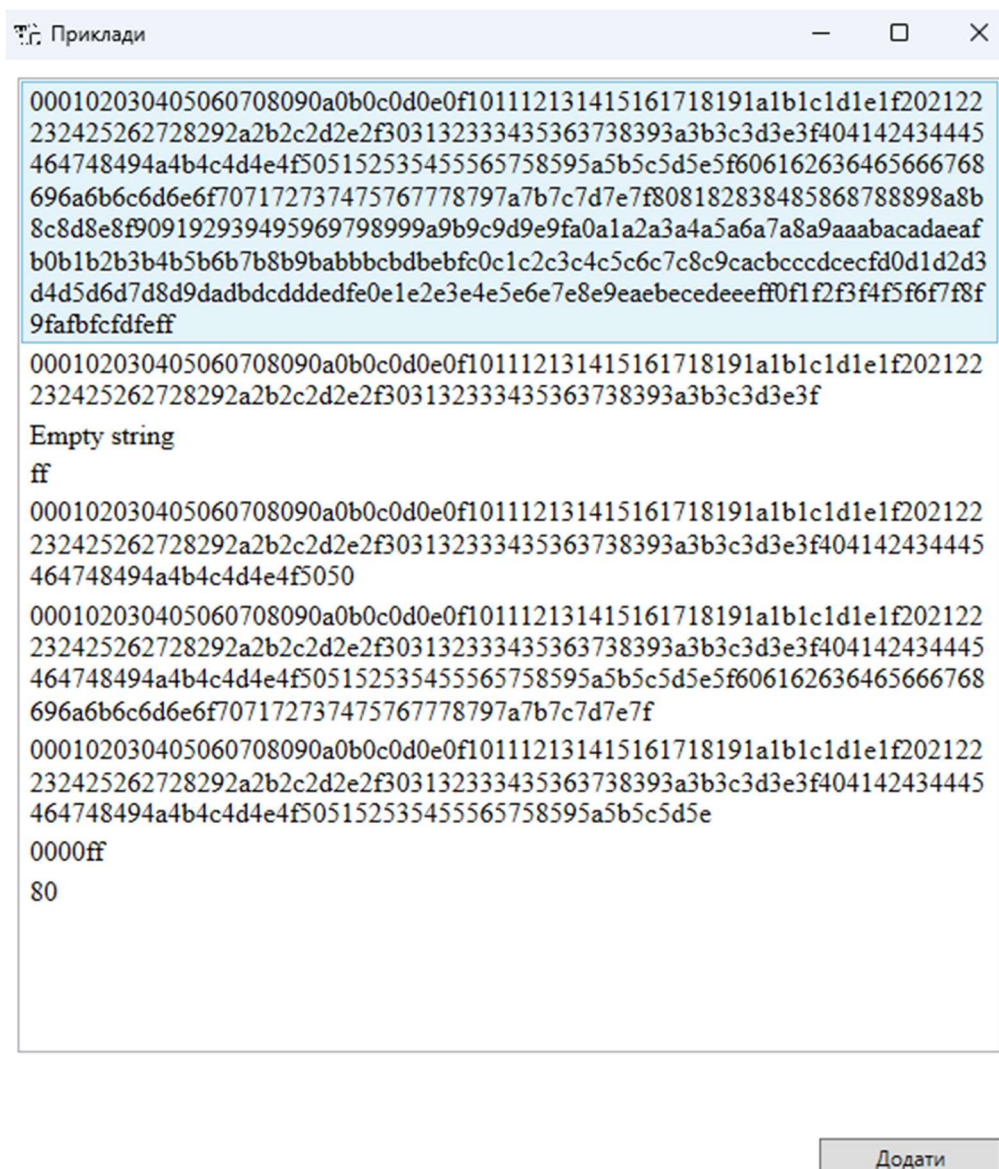


Рисунок 2.5 – Вікно прикладів

3 ПОРІВНЯЛЬНИЙ АНАЛІЗ ВЛАСТИВОСТЕЙ ХЕШ-ФУНКЦІЇ «КУПИНА» ДСТУ 7564:2014 ІЗ ЇЇ АНАЛОГАМИ

3.1 Аналіз функцій на стійкість до колізій

Основною характеристикою для хеш функції є витривалість на колізії. Для аналізу на витривалість до колізій, розширимо додаток, додавши кнопку «Тестування на колізії». При натисканні на кнопку викликається асинхронний метод `Collisson_test_ClickAsync()`, який запускає процес пошуку колізії `PerformBirthdayAttackAsync()` та скасовує процес за допомогою `_cancellationTokenSource.Dispose()`. Процес пошуку відображено на рисунку 3.1.

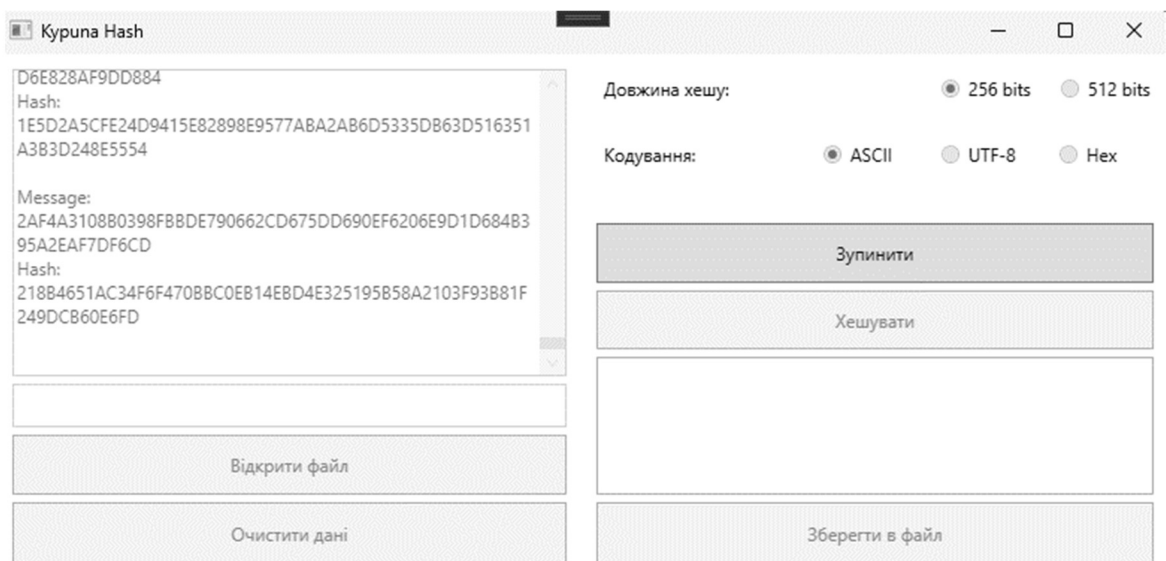


Рисунок 3.1 – Процес пошуку колізії

Для пошуку колізії застосовано метод атаки «Днів народження» [12]. Атака днів народження (Birthday Attack) – це метод криптоаналізу, спрямований на виявлення колізій в хеш-функціях. Імовірність колізії зростає внаслідок того, що хеш-функція перетворює повідомлення різної довжини на фіксовану кількість бітів. Це явище відоме як парадокс днів народження, оскільки імовірність того, що дві людини на випадковій групі мають однаковий день народження, зростає набагато швидше, ніж очікується. Атака

днів народження базується на статистичній властивості, що відома як принцип днів народження. Він стверджує, що вибірковий простір розміром n з ймовірністю в 50% містить хоча б одну пару з однаковими значеннями, коли ми випадково обираємо $\sqrt{2 \cdot n}$ значень. Наприклад, у просторі хеш-функцій розміром 2^{128} , атака днів народження може виявити колізію з ймовірністю в 50% після випробування близько 2^{64} випадкових повідомлень.

Атака днів народження проводиться за таким алгоритмом:

- випадково генеруються повідомлення;
- обчислюється хеш-функція для кожного повідомлення;
- хеші зберігаються у структурі даних (наприклад, словник);
- перевірка на наявність колізії (дві різні повідомлення з однаковим хешем).

Було реалізовано метод PerformBirthdayAttackAsync:

```
TurnInterface(false);

int outputLength = hashSize / 8;
var hashToMessageMap = new Dictionary<string, byte[]>();
var random = new Random();

StringBuilder logBuilder = new StringBuilder();

while (true)
{
    cancellationToken.ThrowIfCancellationRequested();
    byte[] message = new byte[outputLength];
    random.NextBytes(message);
    string hash =
        BitConverter.ToString(KupynaHashFunction.Kupyna.Hash(message,
            hashSize)).Replace("-", "");
    logBuilder.AppendLine($"Message:
        {BitConverter.ToString(message).Replace("-", "")}");
    logBuilder.AppendLine($"Hash: {hash}");
    logBuilder.AppendLine();
    if (hashToMessageMap.ContainsKey(hash))
    {
        byte[] collidingMessage = hashToMessageMap[hash];
        logBuilder.AppendLine("Collision found!");
        logBuilder.AppendLine($"Message 1:
        {BitConverter.ToString(collidingMessage).Replace("-", "")}");
        logBuilder.AppendLine($"Message 2:
        {BitConverter.ToString(message).Replace("-", "")}");
        logBuilder.AppendLine($"Hash: {hash}");
        break;
    }
    else
    {
        hashToMessageMap.Add(hash, message);
    }
    if (hashToMessageMap.Count % 100 == 0)
    {
        Input_Text.Text = logBuilder.ToString();
        await Task.Delay(100, cancellationToken);
    }
    Input_Text.ScrollToEnd();
}
Input_Text.Text = logBuilder.ToString();
TurnInterface(true);
```

Основні кроки методу PerformBirthdayAttackAsync.

1. Відключення інтерфейсу користувача, викликаючи TurnInterface(false).
2. Ініціалізація змінних, таких як outputLength, hashToMessageMap, random та logBuilder.
3. Виконання циклу while (true), що продовжується, поки не буде знайдено зіткнення хешів.
4. Перевірка на скасування операції за допомогою cancellationToken.ThrowIfCancellationRequested().
5. Генерація випадкового повідомлення message.
6. Хешування повідомлення за допомогою функції Кірпуна та отримання результату hash.
7. Додавання повідомлення та хешу до logBuilder.
8. Перевірка на наявність зіткнення хешів. Якщо зіткнення знайдено, додавання деталей про зіткнення до logBuilder та переривання циклу.
9. Якщо зіткнення не знайдено, додавання хешу та повідомлення до словника hashToMessageMap.
10. Оновлення вмісту текстового поля Input_Text зі змістом logBuilder через кожні 100 повторів циклу. Затримка в 100 мс перед продовженням операції за допомогою await Task.Delay(100, cancellationToken).
11. Завершення атаки, оновлення текстового поля Input_Text зі змістом logBuilder та повернення інтерфейсу користувача в активний стан, викликаючи TurnInterface(true).

Але, слід зазначити, що обчислення такого довгого повідомлення може зайняти досить значний час, бо для пошуку колізії хешу 256 біт може знадобитися до 2^{128} спроб, а для 512 біт до 2^{256} спроб.

Для підтвердження було проведено аналогічний тест, але для коротших повідомлень, і за отриманими результатами вимірювань дані буде екстрапольовано. Атаки проводилися в декілька раундів, за результатами яких виводилось середнє значення.

За результатами аналізу було отримано показники, які відображено в таблиці 3.1 і побудовано графік (рисунок 3.2), на який було накладено лінію тренду.

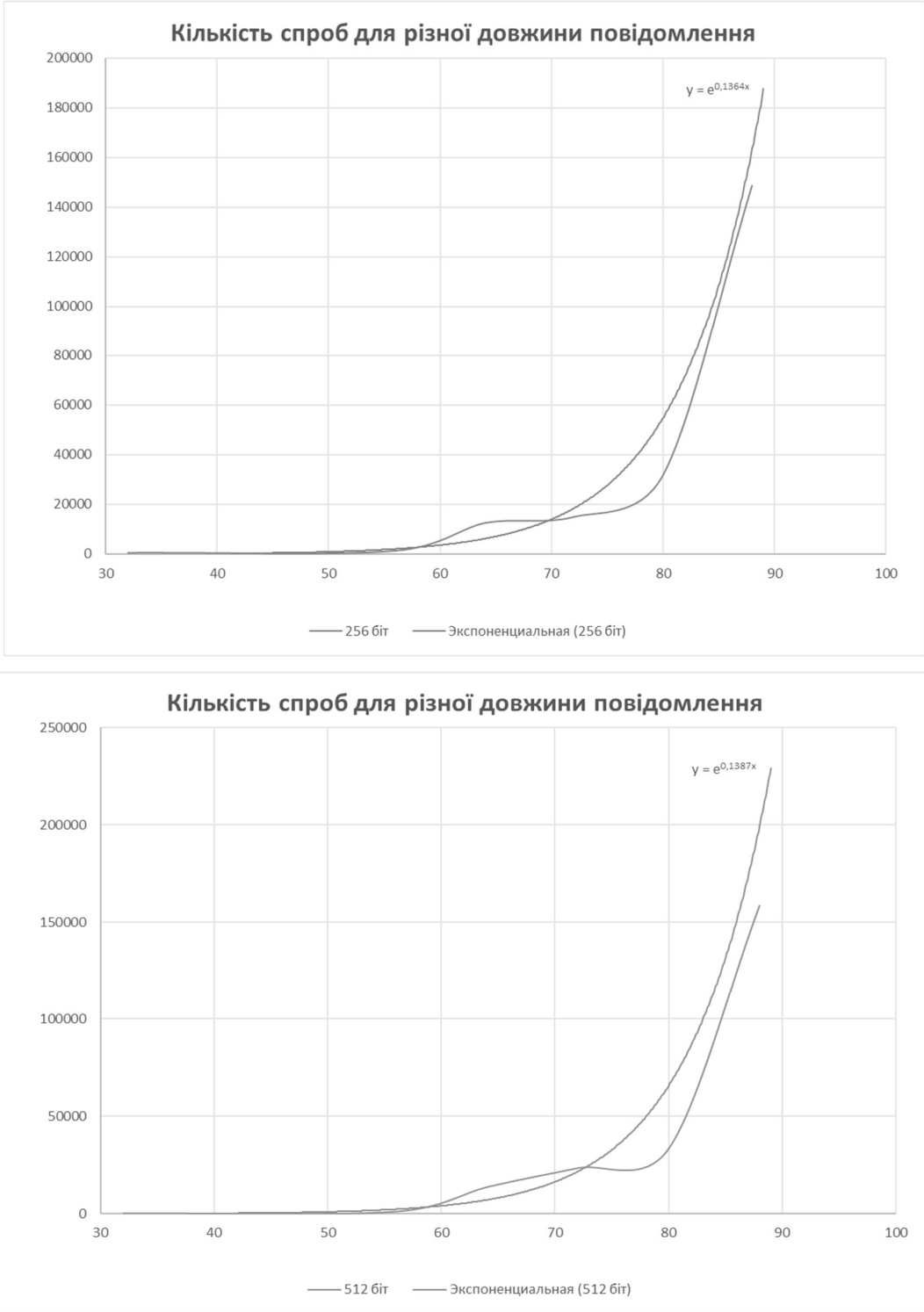


Рисунок 3.2 – Діаграми результатів за таблицею 3.1

Таблиця 3.1 – Кількість спроб перед колізією

Куруна-256		Куруна-512	
Кількість біт повідомлення	Спроб перед колізією	Кількість біт повідомлення	Спроб перед колізією
32	420	32	471
56	1224	56	1378
64	12401	64	13943
72	14865	72	23661
80	31578	80	33578
88	148545	88	158545

Слід зазначити, що дані приблизні, атака «днів народження» цілком спирається на псевдовипадкові значення, через що результати можуть значно відрізнятися один від одного. Дані було екстрапольовано (таблиця 3.2).

Таблиця 3.2 – Екстрапольовані дані

Куруна-256		Куруна-512	
Кількість біт повідомлення	Спроб перед колізією	Кількість біт повідомлення	Спроб перед колізією
96	487380	96	515040
104	1451633	104	1541089
112	4323605	112	4611206
120	12877601	120	13797528
128	38355174	128	41284596
136	114238623	136	123530667
144	340252994	144	369625171
152	1013423459	152	1105982592
160	3018421954	160	3309291649
168	8990191620	168	9901974315
176	26776755071	176	29628423769
184	79752984412	184	88653380336
192	237539556444	192	265266283019
200	707497547481	200	793722705666
208	2107239683300	208	2374955936048
216	6276289011438	216	7106279885786
224	18693556346378	224	21263221371242
232	55677654142180	232	63623244559616
240	165832606344961	240	190371777522259
248	493922629300027	248	569625361420002
256	1471119396309630	256	1704417832285730

Нас цікавить саме кількість спроб при 128 та 256 бітовому повідомленні: 38355174 та 1704417832285730 спроб відповідно, що менше за 2^{128} та 2^{256} . В таблиці 3.3 представлений отриманий час під час експерименту та екстрапольований час.

Таблиця 3.3 – Відповідність кількості спроб до часових витрат

Кількість біт повідомлення	Купина-256		Купина-512	
	Спроб перед першою колізією	Час на обчислення	Спроб перед першою колізією	Час на обчислення
32	420	1:18	471	1:26
64	12401	19:15	13943	24:15
80	31578	59:54	33578	1:00:12
88	148545	4:49:18	158545	5:09:18
Екстрапольовані дані				
96	487380	16:00:00	518454	16:40:00
128	38355174	1248:00:00	41284596	1360:00
256	147111939630963	47 867 257	17044178322	56 147 049 20
	0	152:00:00	85730	0:00:00

Але, слід розуміти, що це приблизні розрахунки, а враховуючи те, що для здійснення атаки в 88 біт знадобилась 5 годин, а на 80 біт 1 годину, з чого можна зробити екстраполяцію даних, і побачити, що для пошуку однієї колізії для ключа довжиною 128 біт може знадобитися 1248 і більше годин, для ключа довжиною 256 біт може знадобитися більше 56 147 049 20 годин (при використанні референсної реалізації хеш-функції), що робить цей вид атаки дуже не ефективним як з точки зору витрат часу, так і витрат на пам'ять, яку потрібно виділити для подібних атак.

Але й існують інші атаки на функції хешування, наприклад rebound-атака [13]. Основна ідея такої атаки полягає у створенні великої кількості розв'язків з низькою складністю на одне рішення під час вхідної фази, і

розповсюдженні цих розв'язків у ймовірнісний спосіб під час так званої вихідної фази. В її основі лежить статистична атака з використанням ротаційного і диференціального криптоаналізу для знаходження колізій і вразливостей функцій. Цей метод криптоаналізу, спеціально розроблений для хеш-функцій, заснованих на конструкції Меркла-Дамгорда.

Однак важливо зазначити, що проведення атаки на хеш-функцію з відскоком є складним процесом і вимагає глибоких знань криптографічної структури, зазвичай для цього застосовуються наступні кроки:

- вибір внутрішнього стану: Виберіть внутрішній стан (значення ланцюжка) хеш-функції після певної кількості раундів;
- згенеруйте вхідну фазу: З обраного внутрішнього стану рухайтесь вперед і назад через операції хеш-функції. Ви хочете знайти дійсне повідомлення, яке призводить до внутрішнього стану при виконанні операцій у прямому напрямку, і дійсний стан, який досягає внутрішнього стану при виконанні операцій у зворотному напрямку. Ця фаза називається вхідною фазою, оскільки вона сходиться до обраного внутрішнього стану;
- згенеруйте вихідну фазу: Відштовхуючись від дійсного повідомлення і стану, знайденого на вхідній фазі, виконайте операції хеш-функції в прямому і зворотному напрямках, щоб знайти дійсне повідомлення і стан для решти раундів хеш-функції. Ця фаза називається вихідною, оскільки вона відрізняється від обраного внутрішнього стану;
- зіставлення ланцюгових значень: Якщо значення ланцюжка в прямому і зворотному напрямках збігаються, ви знайшли дійсне повідомлення, яке створює зіткнення.

Через високу схожість «Купини» з Гростлем, основою атак слід використовувати аналіз Гростля. В роботі технічного університету Граца, який знаходиться в Австрії було проведено подібну атаку [14]. На основі атак з відскоком на Гростля ми побудували атаки на зіткнення для до 5 раундів хеш-функції Курупа-256 та до 7 раундів функції стиснення Курупа-256. В

результаті було з'ясовано, що Купина було здійснено невдалий вибір круглих констант для модульного додавання ще більше знижує ефективність додавань. У поєднанні з відсутністю інших контрзаходів (таких як різні константи обертання), це робить «Купину» легшою мішенню для відскоку, ніж схожа за іншими параметрами хеш-функція Грьостля. Аналіз цього університету показав, що модульні додавання всередині перестановки не є оптимальним вибором для урізноманітнення схожих будівельних блоків, а також створюють нові проблеми.

За результатами атаки встановили, що можна провести атаку тільки на перші 5 раундів, і складність знаходження колізії для скороченої до 5 раундів функції «Купина» 256 складає 2^{120} .

Для проведення порівняльного аналізу з функціями MD5 та SHA256 на колізії не буде враховуватися час на пошук через те, що бібліотечні реалізації цих функцій хешування мають швидшу реалізацію алгоритму з апаратною підтримкою всіх функцій процесора (наприклад інструкцій SSE, SSE2, AVX). Результати аналізу було зведено до таблиці 3.4.

Таблиця 3.4 – Порівняльна таблиця результатів тестування хеш функцій

Кількість біт повідомлення	Функція хешування			
	Купина-256	Купина-512	MD5	SHA256
32	420	471	148	157
128	38355174	41284596	137729200	155289240
256	147111939\ 6309630	1704417832\ 285730	18969332\ 668809200	241147480\ 37387800

Аналіз було проведено аналогічним способом, тобто було виконано тестування для коротких повідомлень, далі дані було екстрапольовано. Тут також слід зазначити, що на теперішній час для алгоритмів MD5 та SHA256 існують вразливості, які дозволяють набагато швидше досягнути колізії. За отриманими даними можна зробити висновки, що «Купина» більш ефективна

для коротких повідомлень, а MD5 та SHA256 більш стійкий при більшому повідомленні.

Функції хешування тісно пов'язані із системою автентифікації користувачів за допомогою ім'я користувача та паролю. Хеш-функції використовуються для зберігання паролів [22] з метою забезпечення безпеки та конфіденційності.

Специфічні властивості хеш-функцій допомагають забезпечити наступні переваги:

- одностороння трансформація: хеш-функції легко обчислюються, але практично неможливо відновити вихідні дані з хешу. Це означає, що навіть якщо хеш потрапить у неправильні руки, зломисникам буде важко отримати вихідний пароль;

- відповідність: навіть незначні зміни у вхідних даних призводять до значної зміни хеш-коду, що робить його важким для зломисників згадати або знайти зразки;

- унікальність: хеш-функція, яка працює коректно, створює унікальний хеш для кожного унікального набору вхідних даних, що зменшує ризик колізій (однакових хешів для різних вхідних даних);

Для підвищення рівня безпеки також слід застосовувати «сіль» – це випадково згенерований набір даних, який використовується разом з паролем для створення ще більш безпечного хеша. Використання солі має наступні переваги:

- Захист від атак зі словниками та таблицями веселкового перегину (rainbow tables): «сіль» додається до пароля перед хешуванням, що ускладнює використання таких атак зломисниками, оскільки їм потрібно буде перебрати значно більше комбінацій.

- Унікальність хешів: «сіль» робить хеші унікальними навіть для однакових паролів, що забезпечує додатковий захист від атак «пошуку за зразком» та спостереження за користувачами з однаковими паролями. Це

забезпечує анонімність користувачів, оскільки зловмисники не зможуть визначити, чи використовують двоє користувачів однакові паролі.

– Запобігання атакам на швидкість: Оскільки «сіль» додається до паролів, зловмисникам потрібно буде витратити більше часу та ресурсів на підбір пароля для кожного окремого користувача. Це ускладнює масові атаки на паролі та змушує зловмисників витратити значно більше зусиль на злам кожного акаунта.

Стійкість такого методу захисту ґрунтується саме на стійкості до колізій, тому, щоб переконатися в стійкості такого методу, спробуємо знаючи хеш, знайти потрібний пароль методом грубої атаки (brute-force) [23].

Для цього в існуючий код, який здійснює просту модель автентифікації користувача, додаймо окрему функцію для перебору паролей:

```
private static readonly char[] Charset =
    «abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789».ToCharArray();

private static IEnumerable<string> BruteForce(char[] charset, int maxLength)
{
    for (int length = 1; length <= maxLength; length++)
    {
        foreach (var combination in Combinations(charset, length))
        {
            yield return new string(combination.ToArray());
        }
    }
}

length private static IEnumerable<IEnumerable<char>> Combinations(char[] charset, int
{
    if (length == 1)
    {
        foreach (var c in charset)
        {
            yield return new[] { c };
        }
    }
    else
    {
        foreach (var c in charset)
        {
            foreach (var innerCombination in Combinations(charset, length -
1))
            {
                yield return new[] { c }.Concat(innerCombination);
            }
        }
    }
}
```

Цей код здійснює перебір паролів і намагається отримати потрібне хеш-значення. Цей код додає метод грубої сили для генерації комбінацій символів до вказаної максимальної довжини. Після того, як користувач спробує увійти в систему, почнеться атака грубої сили, яка намагатиметься вгадати пароль, використовуючи згенеровані комбінації.

Слід зазначити, що цей приклад досить простий і може бути неефективним для довгих паролів [24]. На практиці слід розглянути можливість використання більш просунутих методів, таких як словникові атаки, райдужні таблиці та інші оптимізації для прискорення процесу.

Складність атаки грубої сили залежить від розміру набору символів (N) і максимальної довжини пароля (L). Кількість можливих комбінацій можна обчислити за наступною формулою 3.1.

$$C = N^1 + N^2 + \dots + N^L, \quad (3.1)$$

Це сума можливих комбінацій для кожної довжини пароля від 1 до L . Щоб знайти час, необхідний для перебору всіх комбінацій, потрібно розділити загальну кількість комбінацій C на кількість спроб в секунду A , які може виконати система.

Для атаки було створено акаунт із довжиною пароля в 4 символи (для швидшого обчислення). Обчислимо час, який теоретично може знадобитися для зламу. У нас пароль складається з $L = 4$ символів, кількість символів $N = 62$ символи, примусимо що комп'ютер може обчислити 10000 хешів за секунду:

$$C = 62^1 + 62^2 + 62^3 + 62^4 = 15018570$$

$$T = 15018570 / 10000 = 23.03 \text{ хв}$$

Як бачимо, теоретичний час навіть для настільки короткого пароля вже досить довгий, виконуємо код, щоб переконатися в правильності розрахунків. Слід зауважити, що реальний час буде більше, ніж очікуваний через меншу кількість операцій за секунду. Процес перебору можна побачити на рисунку 3.3.

```

C:\Users\iliya\source\re
Trying password: pasi
Trying password: pasj
Trying password: pask
Trying password: pasl
Trying password: pasm
Trying password: pasn
Trying password: paso
Trying password: pasp
Trying password: pasq
Trying password: pasr
Trying password: pass
Password found: pass
Elapsed time: 00:41:46.4213529

```

Рисунок 3.3 – процес «грубої атаки»

Процес пошуку пароля зайняв 41 хвилину та 46 секунд, що майже у два рази більше. Для більш точного прогнозу часу заміряємо, скільки хешів обчислюється за одну секунду. Вирахуємо середню швидкість обчислення хешу із сіллу за допомогою наступного коду:

```

static void MeasureHashingSpeed()
{
    Console.WriteLine(«Measuring hashing speed...»);
    var random = new Random();
    var salt = GenerateSalt(16);
    var buffer = new byte[32];

    int hashCount = 0;
    var sw = Stopwatch.StartNew();
    while (sw.Elapsed.TotalSeconds < 1)
    {
        random.NextBytes(buffer);
        HashPasswordWithSalt(buffer, salt);
        hashCount++;
    }
    sw.Stop();
    Console.WriteLine($»Hashes computed in one second: {hashCount}»);
}

```

В ході вимірювання було отримано результат в 1711 хешів в секунду. Порахуємо, скільки приблизно часу знадобиться для середньостатистичного пароля довжиною у 8 символів та з використанням спец символів та чисел:

Нехай $N = 72$, $L = 8$, $A = 1711$

$$C = 72^1 + 72^2 + 72^3 + 72^4 + 72^5 + 72^6 + 72^7 + 72^8 = 732376025552520$$

$$T = 732376025552520 / 1711 = 428039757774,71 \text{ секунд}$$

$$428039757774,71 = 118899932,7152 \text{ годин}$$

Як ми бачимо, «груба атака» не є ефективною. Навіть якщо запустити процес хешування на всіх доступних потоках процесора (наприклад, для 16 потоків, доступних для AMD Ryzen 7 5800x):

```
var parallelOptions = new ParallelOptions { MaxDegreeOfParallelism = 16 };
Parallel.For(0, 16, parallelOptions, _ =>
{
    while (sw.Elapsed.TotalSeconds < 1)
    {
        random.NextBytes(buffer);
        HashPasswordWithSalt(buffer, salt);
        Interlocked.Increment(ref hashCount);
    }
});
```

То результат буде складати 18074 хешів на секунду, а в годинах це:

$$T = 732376025552520 / 18074 = 40520970762,01 \text{ секунд}$$

$$40520970762,01 = 40520970762,01 \text{ годин}$$

що теж є занадто довгим процесом. Тому такий метод можна вважати надійним при умові, що всі паролі користувачів будуть задані відповідно до правил безпеки.

Якби не було «солі», тоді ця задача назвалась «атакою на знаходження першовзору». Згідно з документацією стійкість алгоритму до таких атак є 2^{256} для Купина-256 та 2^{512} для Купина-512, що вважається досить надійним показником.

3.2 Аналіз функцій на аваланшний ефект із вимірюванням середньої відстані Хеммінга

Аваланшний (або лавинний) ефект [15] – це властивість криптографічних хеш-функцій, коли невелика зміна на вході повинна призводити до значної зміни на виході:

- генеруються пари вхідних даних: створюються пари вхідних повідомлень, які відрізняються на один біт. Наприклад, перевернувши один біт у кожній парі, залишивши інші біти незмінними
- далі пари хешуються;
- обчислюється відстань Хеммінга, яка являє собою кількість різних бітів між двома хешованими виходами.

Хороша хеш-функція повинна демонструвати лавиноподібний ефект, тобто середня відстань Хеммінга повинна бути близькою до 50% від розміру хешу в бітах (наприклад, 128 біт для 256-бітового хешу).

Для реалізації було використано наступний код:

```
public static void TestAvalancheEffect(KupynaHash kupynaHash, int numberOfPairs)
{
    Random random = new Random();
    int totalHammingDistance = 0;

    for (int i = 0; i < numberOfPairs; i++)
    {
        Console.WriteLine($"Iteration: {i}");
        byte[] input = new byte[kupynaHash.GetByteLength()];
        random.NextBytes(input);
        byte[] inputFlipped = new byte[input.Length];
        Array.Copy(input, inputFlipped, input.Length);
        int byteToFlip = random.Next(input.Length);
        int bitToFlip = random.Next(8);
        inputFlipped[byteToFlip] ^= (byte)(1 << bitToFlip);
        byte[] hash1 = new byte[kupynaHash.GetDigestSize()];
        byte[] hash2 = new byte[kupynaHash.GetDigestSize()];
        kupynaHash.BlockUpdate(input, 0, input.Length);
        kupynaHash.DoFinal(hash1, 0);
        kupynaHash.BlockUpdate(inputFlipped, 0,
inputFlipped.Length);
        kupynaHash.DoFinal(hash2, 0);
        Console.WriteLine($"Hash 1:
{BitConverter.ToString(hash1).Replace("-", "<<")}
Hash 2:
{BitConverter.ToString(hash2).Replace("-", "<<")}");
        int hammingDistance = 0;
        for (int j = 0; j < hash1.Length; j++)
        {
            hammingDistance += CountBits((byte)(hash1[j] ^
hash2[j]));
        }
        totalHammingDistance += hammingDistance;
    }
    double averageHammingDistance = (double)totalHammingDistance /
numberOfPairs;
    Console.WriteLine($"Average Hamming distance:
{averageHammingDistance}");
}

private static int CountBits(byte value)
{
    int count = 0;
    for (int i = 0; i < 8; i++)
    {
        count += (value >> i) & 1;
    }
    return count;
}
```

`TestAvalancheEffect(KupynaHash kupynaHash, int numberOfPairs)` – запускає лавинний тест на класі `KupynaHash` для вказаної кількості вхідних пар. Генерує випадкові вхідні пари з різницею в один біт, хешує їх за допомогою класу `KupynaHash`, обчислює відстань Хеммінга між виходами хешу та обчислює середню відстань Хеммінга для всіх вхідних пар.

`CountBits` (базове значення) – метод утиліти для підрахунку кількості встановлених бітів у байтовому значенні.

Після здійснення тесту в 100 000 ітерацій для алгоритму Купина-256 було встановлено середню відстань Хеммінга в 127,97657 біт, що відповідає умові. Для аналогічного тесту в Купина-512 результат становить 255,95849 біт, що також відповідає умові. З цих тестів можна зробити висновок, що «Купина» є досить хорошим алгоритмом хешування.

Проведемо аналогічний тест для SHA256 та SHA512 та MD5, результати буде зведено до таблиці 3.5. Для цього перевантажимо метод `TestAvalancheEffect`, замінивши параметр `KupynaHash` на об'єкт відповідної функції та замінимо функцію, яка викликає процес хешування.

Таблиця 3.5 – Результат вимірювання відстані Хеммінга

Функція хешування	Довжина хешу (біт)	Середня відстань Хеммінга (біт)
Купина-256	256	127,97657 / 128
SHA256	256	128,00681 / 128
Купина-512	512	255,95849 / 256
SHA512	512	255,99733 / 256
MD5	128	63,67954 / 64

За отриманими даними можна зробити висновок, що за показниками вимірювання аваланшного ефекту функція «Купина» має майже однакові результати із перевіреним часом та досить популярним SHA256 та SHA512, відхилення в результатах складає ~0,03 біти.

3.3 Аналіз швидкості обчислення хеш-образу

Тестування на швидкодію алгоритму хешування буде проведено на референсній версії реалізації алгоритму на мові C, саме ця реалізація є найшвидшою у виконанні через «низькорівневе» програмування та більшу гнучкість у керуванні пам'яттю. Але аналогічне тестування буде проведено і для бібліотеки, яку реалізовано на C#.

Для повноти аналізу алгоритм буде порівняно з алгоритмами MD5, SHA-256. Всі обчислення виконуються на Windows 11 версії 22H2, процесорі AMD Ryzen 7 5800X (8 ядер, які працюють на частоті 3.8 ГГц), з 32 гігабайтами оперативної пам'яті.

Бібліотеку `sys/time.h` було під'єднано до `main.c` референсної реалізації алгоритму хешування «Купина» та було виміряно час початку та кінця виконання обчислення, значення яких записано в змінні типу `struct timeval`. Результат обчислюється та виводиться за допомогою наступного коду:

```
long seconds = (end.tv_sec - start.tv_sec);
long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);

printf("The elapsed time is %d seconds and %d micros\n", seconds,
micros);
```

Для того, щоб переконатися, що код працює було за допомогою функції `sleep()` встановлено затримку на 5 секунд та заміряно час виконання. Було виведено наступний результат:

```
The elapsed time is 5 seconds and 5017529 micros
```

Тепер обчислило хеш для 2048 бітного повідомлення:

```
uint8_t test[2048 / 8] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
    0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F,
    0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5A, 0x5B, 0x5C, 0x5D, 0x5E, 0x5F,
    0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
    0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x7B, 0x7C, 0x7D, 0x7E, 0x7F,
    0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8A, 0x8B, 0x8C, 0x8D, 0x8E, 0x8F,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9D, 0x9E, 0x9F,
    0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6, 0xA7, 0xA8, 0xA9, 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF,
    0xB0, 0xB1, 0xB2, 0xB3, 0xB4, 0xB5, 0xB6, 0xB7, 0xB8, 0xB9, 0xBA, 0xBB, 0xBC, 0xBD, 0xBE, 0xBF,
    0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7, 0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF,
    0xD0, 0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7, 0xD8, 0xD9, 0xDA, 0xDB, 0xDC, 0xDD, 0xDE, 0xDF,
    0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7, 0xE8, 0xE9, 0xEA, 0xEB, 0xEC, 0xED, 0xEE, 0xEF,
    0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
};
```

Обчислення було здійснено за 0.0346 секунди або за 34600 мікросекунд.

Для більш великого повідомлення розміром в 16 384 біт час обчислення значно не збільшився і склав 0.03449 секунди (34490 мікросекунд). Для створення більшого об'єму даних було створено наступний код:

```
#include <stdio.h>
#include <memory.h>
#include «kupyna.h»
#include <sys/time.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

void print(int data_len, uint8_t data[]);

void generate_random_data(uint8_t *data, size_t length) {
    size_t i;
    for (i= 0; i < length; i++) {
        data[i] = rand() % 256;
    }
}

int main(int argc, char **argv) {
    kupyna_t ctx;
    uint8_t hash_code[512 / 8];

    int num_tests = 1000;
    size_t message_length = 1000;

    struct timeval start, end;
    long total_micros = 0;

    srand(time(NULL));
```

```

printf(«=====\\nKupyna-256\\n\\n»);
KupynaInit(256, &ctx);
    int t;
    for (t = 0; t < num_tests; t++) {
        uint8_t message[message_length];

        generate_random_data(message, message_length);

        gettimeofday(&start, NULL);
        KupynaHash(&ctx, message, message_length * 8, hash_code);
        gettimeofday(&end, NULL);

        long seconds = (end.tv_sec - start.tv_sec);
        long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
        total_micros += micros;
    }

    printf(«Total elapsed time: %ld micros\\n», total_micros);
    printf(«Average time per hash operation: %lf micros\\n», (double)total_micros /
num_tests);

    return 0;
}

void print(int data_len, uint8_t data[]) {
    int i = 0;
    int data_size = data_len / BITS_IN_BYTE;
    for (i = 0; i < data_size; i++) {
        if (!(i % 16)) printf(«    »);
        printf(«%02X», (unsigned int)data[i]);
        if (((i + 1) % 16)) printf(«\\n»);
    };
    if (data_len % BITS_IN_BYTE != 0) {
        if (!(i % 16)) printf(«    »);
        printf(«%02X», (unsigned int)((data[i]) & (~(1 << (BITS_IN_BYTE - (data_len %
BITS_IN_BYTE)))) - 1));
        if (((i + 1) % 16)) printf(«\\n»);
    };
    printf(«\\n»);
}

```

За допомогою цього коду створюється довільний набір даних заданої довжини, і за ними проводиться в декілька етапів тестування, час якого обчислюється та складається, щоб далі отримати середній час. Для повідомлення довжиною в 262144 біт (або 32 768 ASCII-символів) середній час обчислення склав 2,47 секунди (2 477 298 мікросекунд). Для повідомлення 2^{19} – 524 288 (65 536 ASCII-символів) час обчислення склав 4,95 секунд (4 954 556 мікросекунд), для 800 000 біт (100 000 ASCII-символів) час обчислення склав 7,59 секунд (7 595 742 мікросекунд). Для повідомлення в 8000000 біт алгоритм на мові C результат не зміг обчислити.

Далі проведемо аналогічні тестування для C# реалізації, MD5 та SHA256 (також на мові C#). Було складено аналогічний код, але на мові C#.

```

public static void Main()
{
    KupynaHash kupynaHash = new KupynaHash(512);
    int numTests = 10;
    int messageLength = 16384;

    Stopwatch stopwatch = new Stopwatch();
    long totalMicros = 0;

    Random random = new Random();

    Console.WriteLine(«=====»);
    Console.WriteLine();

    for (int t = 0; t < numTests; t++)
    {
        byte[] message = new byte[messageLength];
        GenerateRandomData(message, random);
        stopwatch.Start();
        byte[] hash = new byte[kupynaHash.GetDigestSize()];
        kupynaHash.BlockUpdate(message, 0, message.Length);
        kupynaHash.DoFinal(hash, 0);
        stopwatch.Stop();
        long micros = stopwatch.ElapsedTicks / (Stopwatch.Frequency / (1000 *
1000));

        totalMicros += micros;
        stopwatch.Reset();
    }

    Console.WriteLine($»Total elapsed time: {totalMicros} micros»);
    Console.WriteLine($»Average time per hash operation: {(double)totalMicros /
numTests} micros»);
    Console.ReadLine();
}

static void GenerateRandomData(byte[] data, Random random)
{
    random.NextBytes(data);
}

```

Результати обчислення для 16384 біт – 0,33 секунд (333 753,8 мікросекунд), для 262144 біт – 5,18 секунд (5 188 105,4 мікросекунд), а для 524 288 біт – 10,3 секунди (10 307 448,9 мікросекунд), для 800 000 біт час склав 15,7 секунд (15 788 036 мікросекунд), також C# на відміну від версії на C зміг

дати результат для повідомлення довжиною в 8 000 000 біт – 157,6 секунд (157 628 228 мікросекунд).

Таблиця 3.6 – Час обчислення хешу (у мікросекундах) для повідомлень різної довжини

Функція хешування	16 384 біт	262 144 біт	524 288 біт	800 000 біт	8 000 000 біт
Куруна-256 Reference	34 490	2 477 298	4 954 556	7 595 742	-
Куруна-256 C#	333 753,8	5 188 105,4	10 307 448,9	15 788 036	157 628 228
MD5	7109	7210	7210	7135	7281
SHA-256	7147	7251	7251	7178	7357

Таблиця 3.7 - Результати тестів на швидкою від розробників [2].

Функція гешування	Розмір блоку, що обробляється					
	1 byte	8 bytes	32 bytes	64 bytes	2048 bytes	4096 bytes
Куруна-256	3.9820	31.8547	126.8001	155.2432	378.7078	387.4621
Куруна-512	1.5440	12.2717	49.1410	98.0369	272.5307	285.0508
Groestl-256	1.5974	12.7639	50.8011	66.7458	189.1179	195.1246
Groestl-512	1.0712	8.5942	34.3238	68.7215	260.0773	278.0338
Stribog (256,512)	0.9261	7.3959	29.4322	42.4140	139.0854	144.4780
GOST 34.311-95	2.4893	19.8379	79.3996	125.0525	292.7297	299.0621
SHA2-256	14.1968	112.7476	444.6520	474.7652	980.8201	997.5488
SHA2-512	4.4543	35.5932	141.9562	283.1616	571.1605	589.6123
SHA3-256	1.5863	12.7119	51.0108	101.6447	258.3613	268.9411
SHA3-512	1.5782	12.7096	50.6990	101.6891	209.9916	217.1344

Intel Core i5-2300 2,8 ГГц; Windows Server 2008 R2 x64; Microsoft Visual Studio 2010/Visual C++

Було проведено аналогічні тести для алгоритмів SHA-256 та MD5, загальні результати було зведено до таблиці 3.6. Внаслідок аналізу можна побачити, що бібліотечні функції, що вбудовані в System.Security.Cryptography [16] значно швидші за аналогічні реалізації на C# та на C, таке відбувається через ряд причин:

- оптимізована реалізація: Функції хешування реалізуються за допомогою високооптимізованих алгоритмів, які використовують переваги низькорівневої оптимізації апаратного та програмного забезпечення. Це гарантує, що функції хешування добре працюють на широкому спектрі платформ;

- оптимізація для конкретної платформи: Функції хешування можуть використовувати оптимізацію апаратного забезпечення або набору інструкцій для конкретної платформи, наприклад, AES-NI (Advanced Encryption Standard New Instructions) від Intel або Crypto Extensions від ARM. Ці апаратні оптимізації можуть значно прискорити криптографічні операції;

- нативний код: Функції хешування в просторі імен System.Security.Cryptography часто реалізуються у власному коді, а не в керованому. Нативний код компілюється безпосередньо в машинний код для конкретної платформи, що може забезпечити кращу продуктивність, ніж керований код, який компілюється проміжною мовою (IL) і виконується середовищем виконання .NET;

- ефективне управління пам'яттю: Функції хешування призначені для ефективного управління використанням пам'яті, що може допомогти зменшити накладні витрати і підвищити продуктивність. Наприклад, вони можуть використовувати пул пам'яті, кешування або інші методи для зменшення виділення та звільнення пам'яті.

Для оптимізації виконання цієї функції до рівня бібліотечних MD5 та SHA256 – це окрема велика тема. Для оптимізації продуктивності цього коду можна розглянути можливість використання інструкцій SIMD [17] (Single Instruction, Multiple Data), таких як SSE2 [18] або AVX2 [19], які доступні на сучасних процесорах x86 і x64. Інструкції SIMD можуть обробляти декілька елементів даних одночасно, що призводить до покращення продуктивності для певних операцій. Однак, слід розуміти, що використання таких функцій здебільшого потребує «небезпечного» програмування, для використання

якого потрібно мати досить глибоке розуміння того, як працювати з пам'яттю. Такий вид програмування передбачає використання прямого доступу до пам'яті. Небезпечний код може бути швидшим у деяких випадках, особливо при роботі з низькорівневими операціями, оскільки дозволяє обійти певні перевірки часу виконання і безпосередньо маніпулювати адресами пам'яті. Однак це відбувається за рахунок потенційного пошкодження пам'яті, переповнення буферів або інших проблем, якщо писати код неправильно.

Але, якщо говорити про аналогічні реалізації інших функцій хешування, то «Купина» є досить швидким алгоритмом (таблиця 3.7). Враховуючи те, що «Купина» має головну ціль замінити застарілі та досить ненадійні ГОСТ 34.311-95 та Stribog (ГОСТ 34.11-2018), з головною задачею він впорався. Також, дивлячись на цю таблицю, можна побачити, що алгоритм хешування SHA-256 має майже втричі більшу швидкість обчислення навіть в референсній реалізації.

Одним з основних напрямків, де потрібні функції хешування є деякі стандарти цифрового підпису і дуже важливою властивістю є швидкодія алгоритму. Алгоритм ECDSA [20] є як раз одним із них. В ньому функція хешування потребується для хешування повідомлення m за формулою 3.2.

$$e = H(m), \quad (3.2)$$

де H – хеш-функція

Для цього, спираючись на документацію стандарту, а також бібліотеку C# System.Security.Cryptography реалізуємо просту програму, яка здійснює підпис повідомлення та його перевірку. Для демонстрації працездатності алгоритму спочатку підпис буде перевірено без змін вхідних даних, а після буде змінено перший байт повідомлення та здійснено перевірку ще раз.

Наступний код реалізує процедуру підпису за стандартом ECDSA:

```
using KupynaHashFunction;
using System;
using System.Security.Cryptography;
using System.Text;

public class CustomHashAlgorithm : HashAlgorithm
{
    byte[] message;
    protected override void HashCore(byte[] array, int ibStart, int cbSize) {
message = array; }
    protected override byte[] HashFinal() { return Kupyna.Hash(message,256); }
    public override void Initialize() { return; }
}

public class ECDSaCustomHash : ECDSa
{
    private ECDSa _ecdsa;

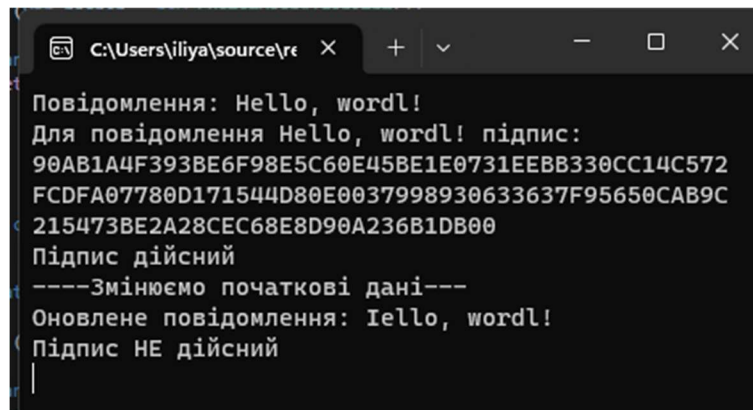
    public ECDSaCustomHash(ECDSa ecdsa)
    {
        _ecdsa = ecdsa;
    }

    public override byte[] SignHash(byte[] hash)
    {
        using (var hasher = new CustomHashAlgorithm())
        {
            var customHash = hasher.ComputeHash(hash);
            return _ecdsa.SignHash(customHash);
        }
    }

    public override bool VerifyHash(byte[] hash, byte[] signature)
    {
        using (var hasher = new CustomHashAlgorithm())
        {
            var customHash = hasher.ComputeHash(hash);
            return _ecdsa.VerifyHash(customHash, signature);
        }
    }
}

public static class Program
{
    public static void Main()
    {
        using (var ecdsa = ECDSa.Create(ECCurve.NamedCurves.nistP256))
        {
            var ecdsaCustomHash = new ECDSaCustomHash(ecdsa);
            Console.WriteLine("Повідомлення: ");
            string message = Console.ReadLine();
            byte[] data = Encoding.ASCII.GetBytes(message);
            byte[] signature = ecdsaCustomHash.SignHash(data);
            Console.WriteLine($"Для повідомлення {message} підпис:
\n{BitConverter.ToString(signature).Replace("-", " «»")}");
            bool isValid = ecdsaCustomHash.VerifyHash(data, signature);
            Console.WriteLine("Підпис « + (isValid ? «дійсний» : «НЕ
дійсний»));
            Console.WriteLine("----Змінюємо початкові дані----");
            data[0] = (byte)(data[0] + 1 % 255);
            Console.WriteLine($"Оновлене повідомлення:
{System.Text.Encoding.Default.GetString(data)}");
            isValid = ecdsaCustomHash.VerifyHash(data, signature);
            Console.WriteLine("Підпис « + (isValid ? «дійсний» : «НЕ
дійсний»));
            Console.ReadLine();
        }
    }
}
```

В результаті виконання програми було отримано результат, що відображений на рисунку 3.5.

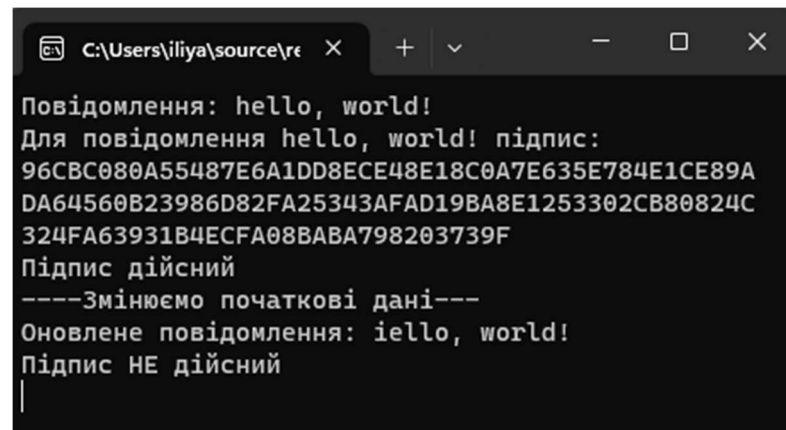


```

C:\Users\iliya\source\re x + - □ x
Повідомлення: Hello, wordl!
Для повідомлення Hello, wordl! підпис:
90AB1A4F393BE6F98E5C60E45BE1E0731EEBB330CC14C572
FCDF407780D171544D80E0037998930633637F95650CAB9C
215473BE2A28CEC68E8D90A236B1DB00
Підпис дійсний
----Змінюємо початкові дані---
Оновлене повідомлення: Iello, wordl!
Підпис НЕ дійсний
|

```

Рисунок 3.5 – процес здійснення цифрового підпису повідомлення



```

C:\Users\iliya\source\re x + - □ x
Повідомлення: hello, world!
Для повідомлення hello, world! підпис:
96CVC080A55487E6A1DD8ECE48E18C0A7E635E784E1CE89A
DA64560B23986D82FA25343AFAD19BA8E1253302CB80824C
324FA63931B4ECFA08BABA798203739F
Підпис дійсний
----Змінюємо початкові дані---
Оновлене повідомлення: iello, world!
Підпис НЕ дійсний
|

```

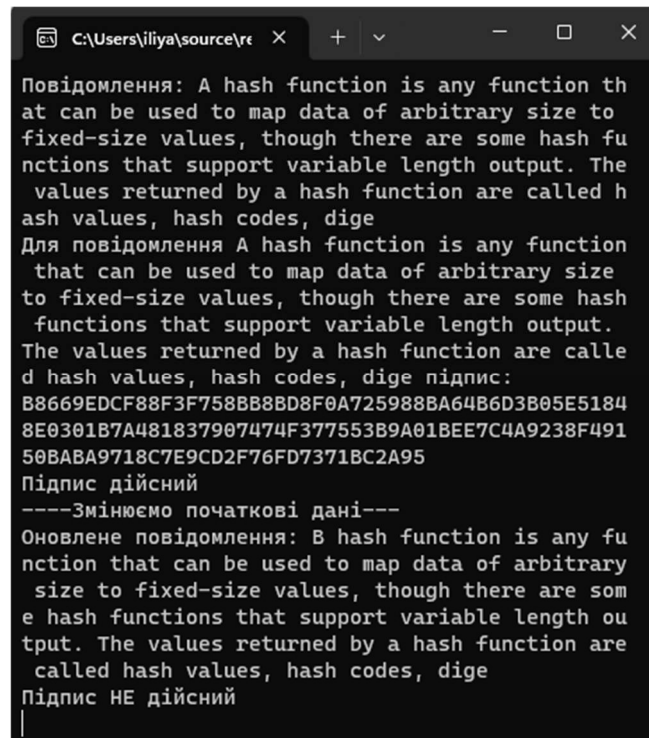
Рисунок 3.6 – процес здійснення цифрового підпису для схожого повідомлення, що демонструє чутливість алгоритму

Для того, щоб переконатися в чутливості та працездатності коду та функції хешування, здійснимо процес для ідентичного повідомлення, але з замінивши один символ тим же, але в нижньому регістрі. Як видно з результату на рисунку 3.6, підпис досить сильно відрізняється. При цьому час виконання програми склав менше 0,2 секунд (203 548 мікросекунди). Для повідомлення більшої довжини результат відображено на рисунку 3.7.

За допомогою функції File.ReadAllText на підписання було завантажено .docx файл, який містить в собі 1 491 264 символів, що дорівнює майже 700

сторінок тексту. Процес підписання зайняв всього 11,8 секунд (11 118 000 мікросекунд), а перевірка 11,08 секунд (11 080 000 мікросекунд).

Спираючись на отримані результати, можна стверджувати, що використання функції хешування «Купина», як основної функції хешування в цифрових підписах є цілком доцільним та швидким рішенням.



```

C:\Users\iliya\source\rc x + - □ x
Повідомлення: A hash function is any function th
at can be used to map data of arbitrary size to
fixed-size values, though there are some hash fu
nctions that support variable length output. The
values returned by a hash function are called h
ash values, hash codes, dige
Для повідомлення A hash function is any function
that can be used to map data of arbitrary size
to fixed-size values, though there are some hash
functions that support variable length output.
The values returned by a hash function are calle
d hash values, hash codes, dige підпис:
B8669EDCF88F3F758BB8BD8F0A725988BA64B6D3B05E5184
8E0301B7A481837907474F377553B9A01BEE7C4A9238F491
50BABA9718C7E9CD2F76FD7371BC2A95
Підпис дійсний
----Змінюємо початкові дані---
Оновлене повідомлення: B hash function is any fu
nction that can be used to map data of arbitrary
size to fixed-size values, though there are som
e hash functions that support variable length ou
tput. The values returned by a hash function are
called hash values, hash codes, dige
Підпис НЕ дійсний

```

Рисунок 3.7 – процес здійснення підпису для більш великого повідомлення

ВИСНОВКИ

В ході проведення дослідження було реалізовано функцію хешування «Купина» на мові C#, а також проведено аналіз властивостей функції хешування, використовуючи референсну та власну реалізацію. Було виконано базовий аналіз функції і тестування її на колізії, аваланшний ефект та швидкодію.

В ході аналізу було встановлено, що при здійсненні однієї із найпростіших атак «днів народження» на функцію було з'ясовано, що «Купина» більш ефективна для коротких повідомлень, а MD5 та SHA256 більш стійкий при більшому повідомленні.

В ході аналізу на аваланшний ефект було здійснено обчислення та порівняння показників. Також, було встановлено, що за показниками вимірювання аваланшного ефекту функція «Купина» відповідає сучасним стандартам та має хороші результати відносно SHA256 та SHA512. Аналіз на швидкодію, користуючись даними з офіційних досліджень було встановлено, що основного конкурента, на заміну якого прийшла ця функція – ГОСТ 34.311-95 та Stribog (ГОСТ 34.11-2018), за показниками швидкості він їх обійшов, але все ж ця функція потребує сучасної реалізації з використанням всіх апаратних можливостей.

Також було наведено приклади використання функції хешування в реальних ситуаціях та проведено порівняльний аналіз, в ході якого було використано «Купину» в системі цифрового підпису ECDSA та системі автентифікації користувача за допомогою паролю. В ході роботи встановлено що функція хешування «Купина» є цілком конкурентоздатною в порівнянні із популярними SHA256 та MD5 та може використовуватися в сучасних системах та програмному забезпеченні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Хешування даних в інформаційних системах : монографія / С. П. Євсєєв, О. Ю. Йохов, О. Г. Король. – Х. : Вид. ХНЕУ, 2013. – 312 с.
2. ДСТУ ISO 7564:2014 Інформаційні технології. Криптографічний захист інформації. Функція хешування [Чинний від 2015-04-01]. Київ, 2015. 39 с. (Інформація та документація).
3. Стаття: «Купина (хеш-функція)» – Електронний ресурс. Режим доступу: <https://en.wikipedia.org/wiki/Купина> (дата звернення 17.03.2023)
4. C# Programming in Easy Steps / Mike McGrath, 2 видання, - Вид. Easy Steps, 2020. – 192 с.
5. «The Kupyna hash function reference implementation», GitHub – Електронний ресурс. Режим доступу: <https://github.com/Roman-Oliynykov/Kupyna-reference> (дата звернення 20.03.2023)
6. «C++ cryptographic library (modern hash functions, ciphers, KDFs)», SourceForge – Електронний ресурс. Режим доступу: <https://sourceforge.net/projects/cppcrypto/> (дата звернення 20.03.2023)
7. «Cryptonite», GitHub – Електронний ресурс. Режим доступу: <https://github.com/privat-it/cryptonite> (дата звернення 20.03.2023)
8. «Що таке Net Framework і навіщо він потрібен?», Intelserv – Електронний ресурс. Режим доступу: <https://intelserv.net.ua/blog/material/id/247> (дата звернення 22.03.2023)
9. Стаття: «Що таке DLL бібліотека?» Microsoft Learn – Електронний ресурс – Режим доступу: <https://learn.microsoft.com/ru-ru/troubleshoot/windows-client/deployment/dynamic-link-library> (дата звернення 18.03.2023)

10. Стаття: «Що таке UWP додаток?» Microsoft Learn – Електронний ресурс – Режим доступу: <https://learn.microsoft.com/ru-ru/windows/uwp/get-started/universal-application-platform-guide> (дата звернення 18.03.2023)
11. Стаття: «Створення WPF додатку за допомогою .NET» Microsoft Learn – Електронний ресурс – Режим доступу: <https://learn.microsoft.com/ru-ru/dotnet/desktop/wpf/get-started/create-app-visual-studio?view=netdesktop-7.0> (дата звернення 18.03.2023)
12. Стаття: «Атака днів народження» Юніопедія – Електронний ресурс – Режим доступу: https://uk.unionpedia.org/i/Атака_«днів_народження» (дата звернення 19.03.2023)
13. «The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl» Springer - Електронний ресурс – Режим доступу: https://link.springer.com/chapter/10.1007/978-3-642-03317-9_16 (дата звернення 22.03.2023)
14. «Analysis of the Kupyna-256 Hash Function» / Christoph Dobraunig, Maria Eichlseder, and Florian Mendel, Graz University of Technology, Austria, - 16 с.
15. «Demonstrate the Avalanche Effect of a Hash Function» Wolfram Language, - Електронний ресурс – Режим доступу: <https://www.wolfram.com/language/12/cryptography/demonstrate-the-avalanche-effect-of-a-hash-function.html?product=language> (дата звернення 24.03.2023)
16. «System.Security.Cryptography Namespace» Microsoft Learn, - Електронний ресурс – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=net-8.0> (дата звернення 26.03.2023)
17. «Використання числових типів з прискоренням SIMD» Microsoft Learn, - Електронний ресурс – Режим доступу: <https://learn.microsoft.com/ru-ru/dotnet/standard/simd> (дата звернення 27.03.2023)

18. «SSE2 (Streaming SIMD Extensions 2)» Chaynikam, - Електронний ресурс – Режим доступу: https://www.chaynikam.info/ukr/cpu_sse2.html (дата звернення 27.03.2023)

19. «AVX (Advanced Vector Extensions)» Chaynikam, - Електронний ресурс – Режим доступу: https://www.chaynikam.info/ukr/cpu_avx.html (дата звернення 27.03.2023)

20. «ECDSA: Elliptic Curve Signatures», - Електронний ресурс – Режим доступу: <https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages> (дата звернення 01.04.2023)

21. «ECDSA: Elliptic Curve Signatures», - Електронний ресурс – Режим доступу: <https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages> (дата звернення 01.04.2023)

22. «Що таке хеші паролів і як їх НЕ можна зберігати», - Електронний ресурс – Режим доступу: <https://yura.stryi.com/2012-10-15/plain-text-passwords/> (дата звернення 02.04.2023)

23. «What Is a Brute Force Attack?», Fortinet - Електронний ресурс – Режим доступу: <https://www.fortinet.com/resources/cyberglossary/brute-force-attack> (дата звернення 03.04.2023)

24. «Основні методи злому пароля і як від них захиститися», Passgenerator - Електронний ресурс – Режим доступу: <https://passgenerator.com.ua/osnovni-metody-zlomu-parolia-i-yak-vid-nykh-zakhystytysia> (дата звернення 05.04.2023)

ДОДАТОК А

Код реалізації функції хешування «Купина»

Hash.cs

```
using System;

namespace KupynaHashFunction
{
    /// <summary>
    /// Calculating Kupyna Hash (256 or 512)
    /// </summary>
    public static class Kupyna
    {
        public static byte[] Hash (byte[] input, int nbytes)
        {
            byte[] hash = new byte[nbytes / 8];
            KupynaHash kupynaHash = new KupynaHash(nbytes);
            kupynaHash.BlockUpdate(input, 0, input.Length);
            kupynaHash.DoFinal(hash, 0);
            return hash;
        }
    }
}
```

Kupyna.cs

```
using System;

namespace KupynaHashFunction
{
    internal sealed class Pack
    {
        internal static void UInt16_To_BE(ushort n, byte[] bs)
        {
            bs[0] = (byte)(n >> 8);
            bs[1] = (byte)(n);
        }

        internal static void UInt16_To_BE(ushort n, byte[] bs, int off)
        {
            bs[off] = (byte)(n >> 8);
            bs[off + 1] = (byte)(n);
        }

        internal static ushort BE_To_UInt16(byte[] bs)
        {
            uint n = (uint)bs[0] << 8
                | (uint)bs[1];
            return (ushort)n;
        }

        internal static ushort BE_To_UInt16(byte[] bs, int off)
        {
            uint n = (uint)bs[off] << 8
                | (uint)bs[off + 1];
            return (ushort)n;
        }
    }
}
```

```

}

internal static byte[] UInt32_To_BE(uint n)
{
    byte[] bs = new byte[4];
    UInt32_To_BE(n, bs, 0);
    return bs;
}

internal static void UInt32_To_BE(uint n, byte[] bs)
{
    bs[0] = (byte)(n >> 24);
    bs[1] = (byte)(n >> 16);
    bs[2] = (byte)(n >> 8);
    bs[3] = (byte)(n);
}

internal static void UInt32_To_BE(uint n, byte[] bs, int off)
{
    bs[off] = (byte)(n >> 24);
    bs[off + 1] = (byte)(n >> 16);
    bs[off + 2] = (byte)(n >> 8);
    bs[off + 3] = (byte)(n);
}

internal static byte[] UInt32_To_BE(uint[] ns)
{
    byte[] bs = new byte[4 * ns.Length];
    UInt32_To_BE(ns, bs, 0);
    return bs;
}

internal static void UInt32_To_BE(uint[] ns, byte[] bs, int off)
{
    for (int i = 0; i < ns.Length; ++i)
    {
        UInt32_To_BE(ns[i], bs, off);
        off += 4;
    }
}

internal static uint BE_To_UInt32(byte[] bs)
{
    return (uint)bs[0] << 24
        | (uint)bs[1] << 16
        | (uint)bs[2] << 8
        | (uint)bs[3];
}

internal static uint BE_To_UInt32(byte[] bs, int off)
{
    return (uint)bs[off] << 24
        | (uint)bs[off + 1] << 16
        | (uint)bs[off + 2] << 8
        | (uint)bs[off + 3];
}

internal static void BE_To_UInt32(byte[] bs, int off, uint[] ns)
{
    for (int i = 0; i < ns.Length; ++i)
    {
        ns[i] = BE_To_UInt32(bs, off);
        off += 4;
    }
}

internal static byte[] UInt64_To_BE(ulong n)
{

```

```

        byte[] bs = new byte[8];
        UInt64_To_BE(n, bs, 0);
        return bs;
    }

    internal static void UInt64_To_BE(ulong n, byte[] bs)
    {
        UInt32_To_BE((uint)(n >> 32), bs);
        UInt32_To_BE((uint)(n), bs, 4);
    }

    internal static void UInt64_To_BE(ulong n, byte[] bs, int off)
    {
        UInt32_To_BE((uint)(n >> 32), bs, off);
        UInt32_To_BE((uint)(n), bs, off + 4);
    }

    internal static byte[] UInt64_To_BE(ulong[] ns)
    {
        byte[] bs = new byte[8 * ns.Length];
        UInt64_To_BE(ns, bs, 0);
        return bs;
    }

    internal static void UInt64_To_BE(ulong[] ns, byte[] bs, int off)
    {
        for (int i = 0; i < ns.Length; ++i)
        {
            UInt64_To_BE(ns[i], bs, off);
            off += 8;
        }
    }

    internal static ulong BE_To_UInt64(byte[] bs)
    {
        uint hi = BE_To_UInt32(bs);
        uint lo = BE_To_UInt32(bs, 4);
        return ((ulong)hi << 32) | (ulong)lo;
    }

    internal static ulong BE_To_UInt64(byte[] bs, int off)
    {
        uint hi = BE_To_UInt32(bs, off);
        uint lo = BE_To_UInt32(bs, off + 4);
        return ((ulong)hi << 32) | (ulong)lo;
    }

    internal static void BE_To_UInt64(byte[] bs, int off, ulong[] ns)
    {
        for (int i = 0; i < ns.Length; ++i)
        {
            ns[i] = BE_To_UInt64(bs, off);
            off += 8;
        }
    }

    internal static void UInt16_To_LE(ushort n, byte[] bs)
    {
        bs[0] = (byte)(n);
        bs[1] = (byte)(n >> 8);
    }

    internal static void UInt16_To_LE(ushort n, byte[] bs, int off)
    {
        bs[off] = (byte)(n);
        bs[off + 1] = (byte)(n >> 8);
    }

```

```

internal static ushort LE_To_UInt16(byte[] bs)
{
    uint n = (uint)bs[0]
        | (uint)bs[1] << 8;
    return (ushort)n;
}

internal static ushort LE_To_UInt16(byte[] bs, int off)
{
    uint n = (uint)bs[off]
        | (uint)bs[off + 1] << 8;
    return (ushort)n;
}

internal static byte[] UInt32_To_LE(uint n)
{
    byte[] bs = new byte[4];
    UInt32_To_LE(n, bs, 0);
    return bs;
}

internal static void UInt32_To_LE(uint n, byte[] bs)
{
    bs[0] = (byte)(n);
    bs[1] = (byte)(n >> 8);
    bs[2] = (byte)(n >> 16);
    bs[3] = (byte)(n >> 24);
}

internal static void UInt32_To_LE(uint n, byte[] bs, int off)
{
    bs[off] = (byte)(n);
    bs[off + 1] = (byte)(n >> 8);
    bs[off + 2] = (byte)(n >> 16);
    bs[off + 3] = (byte)(n >> 24);
}

internal static byte[] UInt32_To_LE(uint[] ns)
{
    byte[] bs = new byte[4 * ns.Length];
    UInt32_To_LE(ns, bs, 0);
    return bs;
}

internal static void UInt32_To_LE(uint[] ns, byte[] bs, int off)
{
    for (int i = 0; i < ns.Length; ++i)
    {
        UInt32_To_LE(ns[i], bs, off);
        off += 4;
    }
}

internal static uint LE_To_UInt32(byte[] bs)
{
    return (uint)bs[0]
        | (uint)bs[1] << 8
        | (uint)bs[2] << 16
        | (uint)bs[3] << 24;
}

internal static uint LE_To_UInt32(byte[] bs, int off)
{
    return (uint)bs[off]
        | (uint)bs[off + 1] << 8
        | (uint)bs[off + 2] << 16
        | (uint)bs[off + 3] << 24;
}

```

```

internal static void LE_To_UInt32(byte[] bs, int off, uint[] ns)
{
    for (int i = 0; i < ns.Length; ++i)
    {
        ns[i] = LE_To_UInt32(bs, off);
        off += 4;
    }
}

internal static void LE_To_UInt32(byte[] bs, int bOff, uint[] ns, int nOff, int count)
{
    for (int i = 0; i < count; ++i)
    {
        ns[nOff + i] = LE_To_UInt32(bs, bOff);
        bOff += 4;
    }
}

internal static byte[] UInt64_To_LE(ulong n)
{
    byte[] bs = new byte[8];
    UInt64_To_LE(n, bs, 0);
    return bs;
}

internal static void UInt64_To_LE(ulong n, byte[] bs)
{
    UInt32_To_LE((uint)(n), bs);
    UInt32_To_LE((uint)(n >> 32), bs, 4);
}

internal static void UInt64_To_LE(ulong n, byte[] bs, int off)
{
    UInt32_To_LE((uint)(n), bs, off);
    UInt32_To_LE((uint)(n >> 32), bs, off + 4);
}

internal static ulong LE_To_UInt64(byte[] bs)
{
    uint lo = LE_To_UInt32(bs);
    uint hi = LE_To_UInt32(bs, 4);
    return ((ulong)hi << 32) | (ulong)lo;
}

internal static ulong LE_To_UInt64(byte[] bs, int off)
{
    uint lo = LE_To_UInt32(bs, off);
    uint hi = LE_To_UInt32(bs, off + 4);
    return ((ulong)hi << 32) | (ulong)lo;
}
}
internal class KupynaHash
{
    private const int ROWS = 8;
    private const int REDUCTION_POLYNOMIAL = 0x011d;
    private const int BITS_IN_BYTE = 8;
    private const int NB_512 = 8;
    private const int NB_1024 = 16;
    private const int NR_512 = 10;
    private const int NR_1024 = 14;
    private const int STATE_BYTE_SIZE_512 = ROWS * NB_512;
    private const int STATE_BYTE_SIZE_1024 = ROWS * NB_1024;
    private int hashSize;
    private int blockSize;
    private int columns;
    private int rounds;
    private byte[] padded_;

```

```

private byte[][] state_;

internal KupynaHash(int hashSizeBits)
{
    if (hashSizeBits == 256 || hashSizeBits == 384 || hashSizeBits == 512)
    {
        this.hashSize = hashSizeBits / 8;
    }
    else
    {
        throw new ArgumentException("Hash size is not recommended. Use 256 or 384 or
512 size");
    }

    if (hashSizeBits > 256)
    {
        this.blockSize = 1024 / 8;
        this.columns = NB_1024;
        this.rounds = NR_1024;
        this.state_ = new byte[STATE_BYTE_SIZE_1024][];

    }
    else
    {
        this.blockSize = 512 / 8;
        this.columns = NB_512;
        this.rounds = NR_512;
        this.state_ = new byte[STATE_BYTE_SIZE_512][];

    }

    for (int i = 0; i < state_.Length; i++)
    {
        this.state_[i] = new byte[columns];
    }

    this.state_[0][0] = (byte)state_.Length;

    this.hashSize = hashSizeBits / 8;

    this.padded_ = null;
}

internal virtual void BlockUpdate(byte[] input, int inOff, int length)
{
    padded_ = Pad(input, inOff, length);

    int paddedOff = 0;
    int paddedLength = padded_.Length;
    while (paddedLength != 0)
    {
        ProcessBlock(padded_, paddedOff);
        paddedLength -= blockSize;
        paddedOff += blockSize;
    }
}

internal byte[] Pad(byte[] input, int inOff, int length)
{
    byte[] padded;
    if (length % blockSize == 0)
    {
        padded = new byte[length + blockSize];
    }
    else
    {
        int blocks = length / blockSize;
        padded = new byte[(blocks * blockSize) + blockSize];
    }
}

```

```

    }

    Array.Copy(input, inOff, padded, 0, length);
    padded[length] = 0x80;
    intToBytes(length * 8, padded, padded.Length - 12);

    return padded;
}
internal void ProcessBlock(byte[] input, int inOff)
{
    byte[][] temp1 = new byte[STATE_BYTE_SIZE_1024][];
    byte[][] temp2 = new byte[STATE_BYTE_SIZE_1024][];

    for (int i = 0; i < state_.Length; i++)
    {
        temp1[i] = new byte[ROWS];
        temp2[i] = new byte[ROWS];
    }

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < columns; ++j)
        {
            temp1[j][i] = (byte)(state_[j][i] ^ input[j * ROWS + i + inOff]);
            temp2[j][i] = input[j * ROWS + i + inOff];
        }
    }

    P(temp1);
    Q(temp2);

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < columns; ++j)
        {
            state_[j][i] ^= (byte)(temp1[j][i] ^ temp2[j][i]);
        }
    }
}
public int DoFinal(byte[] output, int outOff)
{
    byte[] stateLine = new byte[ROWS * columns];

    byte[][] temp = new byte[STATE_BYTE_SIZE_1024][];
    for (int i = 0; i < state_.Length; i++)
    {
        temp[i] = new byte[ROWS];
        Array.Copy(state_[i], temp[i], ROWS);
    }

    P(temp);

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < columns; ++j)
        {
            state_[j][i] ^= temp[j][i];
        }
    }

    int stateLineIndex = 0;
    for (int j = 0; j < columns; ++j)
    {

```

```

        for (int i = 0; i < ROWS; ++i)
        {
            stateLine[stateLineIndex] = state_[j][i];
            stateLineIndex++;
        }
    }

    Array.Copy(stateLine, stateLine.Length - hashSize, output, outOff, hashSize);

    return hashSize;
}
private void Reset()
{
    for (int i = 0; i < state_.Length; i++)
    {
        state_[i] = new byte[columns];
    }

    state_[0][0] = (byte)state_.Length;

    Array.Clear(padded_, 0, padded_.Length);
}
private int GetDigestSize()
{
    return hashSize;
}
private int GetByteLength()
{
    return blockSize;
}
private void Update(byte input)
{
    throw new NotImplementedException();
}
private void SubBytes(byte[][] state)
{
    int i, j;
    for (i = 0; i < ROWS; ++i)
    {
        for (j = 0; j < columns; ++j)
        {
            state[j][i] = sBoxes[i % 4][state[j][i]];
        }
    }
}
private void ShiftBytes(byte[][] state)
{
    int i, j;
    byte[] temp = new byte[NB_1024];
    int shift = -1;
    for (i = 0; i < ROWS; ++i)
    {
        if ((i == ROWS - 1) && (columns == NB_1024))
        {
            shift = 11;
        }
        else
        {
            ++shift;
        }
        for (j = 0; j < columns; ++j)
        {
            temp[(j + shift) % columns] = state[j][i];
        }
        for (j = 0; j < columns; ++j)
        {

```

```

        state[j][i] = temp[j];
    }
}
private byte MultiplyGF(byte x, byte y)
{
    int i;
    byte r = 0;
    byte hbit = 0;
    for (i = 0; i < BITS_IN_BYTE; ++i)
    {
        if ((y & 0x1) == 1)
        {
            r ^= x;
        }

        hbit = (byte)(x & 0x80);

        x <<= 1;

        if (hbit == 0x80)
        {
            x = (byte)((int)x ^ REDUCTION_POLYNOMIAL);
        }

        y >>= 1;
    }
    return r;
}
private void MixColumns(byte[][] state)
{
    int i, row, col, b;
    byte product;
    byte[] result = new byte[ROWS];

    for (col = 0; col < columns; ++col)
    {
        Array.Clear(result, 0, ROWS);
        for (row = ROWS - 1; row >= 0; --row)
        {
            product = 0;
            for (b = ROWS - 1; b >= 0; --b)
            {
                product ^= MultiplyGF(state[col][b], mds_matrix[row][b]);
            }
            result[row] = product;
        }
        for (i = 0; i < ROWS; ++i)
        {
            state[col][i] = result[i];
        }
    }
}
private void AddRoundConstantP(byte[][] state, int round)
{
    int i;
    for (i = 0; i < columns; ++i)
    {
        state[i][0] ^= (byte)((i * 0x10) ^ round);
    }
}
private void AddRoundConstantQ(byte[][] state, int round)
{
    int j;
    UInt64[] s = new UInt64[columns];

    for (j = 0; j < columns; j++)
    {

```

```

        s[j] = Pack.LE_To_UInt64(state[j]);

        s[j] = s[j] + (0x00F0F0F0F0F0F3UL ^ (((UInt64)(columns - j - 1) * 0x10UL) ^
(UInt64)round) << (7 * 8)));

        state[j] = Pack.UInt64_To_LE(s[j]);
    }
}
private void P(byte[][] state)
{
    int i;
    for (i = 0; i < rounds; ++i)
    {
        AddRoundConstantP(state, i);
        SubBytes(state);
        ShiftBytes(state);
        MixColumns(state);
    }
}
private void Q(byte[][] state)
{
    int i;
    for (i = 0; i < rounds; ++i)
    {
        AddRoundConstantQ(state, i);
        SubBytes(state);
        ShiftBytes(state);
        MixColumns(state);
    }
}

private static void intTobytes(
    int num,
    byte[] outBytes,
    int outOff)
{
    outBytes[outOff + 3] = (byte)(num >> 24);
    outBytes[outOff + 2] = (byte)(num >> 16);
    outBytes[outOff + 1] = (byte)(num >> 8);
    outBytes[outOff] = (byte)num;
}

#region CONSTANTS
private readonly byte[][] mds_matrix = new byte[][]
{
    new byte[] {0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04},
    new byte[] {0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07},
    new byte[] {0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06},
    new byte[] {0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08},
    new byte[] {0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01},
    new byte[] {0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05},
    new byte[] {0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01},
    new byte[] {0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01}
};
private readonly byte[][] sBoxes = new byte[][]
{
    new byte[]
    {
        0xa8, 0x43, 0x5f, 0x06, 0x6b, 0x75, 0x6c, 0x59, 0x71, 0xdf, 0x87, 0x95,
0x17, 0xf0, 0xd8, 0x09,
        0x6d, 0xf3, 0x1d, 0xcb, 0xc9, 0x4d, 0x2c, 0xaf, 0x79, 0xe0, 0x97, 0xfd, 0x6f,
0x4b, 0x45, 0x39,
        0x3e, 0xdd, 0xa3, 0x4f, 0xb4, 0xb6, 0x9a, 0x0e, 0x1f, 0xbf, 0x15, 0xe1, 0x49,
0xd2, 0x93, 0xc6,
        0x92, 0x72, 0x9e, 0x61, 0xd1, 0x63, 0xfa, 0xee, 0xf4, 0x19, 0xd5, 0xad, 0x58,
0xa4, 0xbb, 0xa1,
        0xdc, 0xf2, 0x83, 0x37, 0x42, 0xe4, 0x7a, 0x32, 0x9c, 0xcc, 0xab, 0x4a, 0x8f,
0x6e, 0x04, 0x27,

```

```

0xa9, 0x47, 0x41, 0x2e, 0xe7, 0xe2, 0x5a, 0x96, 0x16, 0x23, 0x2b, 0xc2, 0x65, 0x66, 0x0f, 0xbc,
0x7b, 0xc3, 0x1e, 0x34, 0x48, 0xfc, 0xb7, 0x6a, 0x88, 0xa5, 0x53, 0x86, 0xf9, 0x5b, 0xdb, 0x38,
0x9f, 0x08, 0x55, 0x22, 0x33, 0x24, 0x28, 0x36, 0xc7, 0xb2, 0x3b, 0x8e, 0x77, 0xba, 0xf5, 0x14,
0x1b, 0x89, 0xff, 0x9b, 0x4c, 0xfe, 0x60, 0x5c, 0xda, 0x18, 0x46, 0xcd, 0x7d, 0x21, 0xb0, 0x3f,
0x30, 0x91, 0xb1, 0xeb, 0x84, 0x69, 0x3a, 0x9d, 0xd7, 0xd3, 0x70, 0x67, 0x40, 0xb5, 0xde, 0x5d,
0x0b, 0xa2, 0x76, 0x78, 0x11, 0x01, 0xe5, 0x00, 0x68, 0x98, 0xa0, 0xc5, 0x02, 0xa6, 0x74, 0x2d,
0xaa, 0xf6, 0x26, 0xb3, 0xbe, 0xce, 0xbd, 0xae, 0xe9, 0x8a, 0x31, 0x1c, 0xec, 0xf1, 0x99, 0x94,
0x20, 0x3d, 0x82, 0x2f, 0xef, 0xe8, 0x8c, 0x35, 0x03, 0xd4, 0x7f, 0xfb, 0x05, 0xc1, 0x5e, 0x90,
0x62, 0xe3, 0xc8, 0xf7, 0xea, 0x0a, 0x0d, 0x7e, 0xf8, 0x50, 0x1a, 0xc4, 0x07, 0x57, 0xb8, 0x3c,
0xd6, 0x73, 0x8d, 0xac, 0x52, 0x64, 0x10, 0xd0, 0xd9, 0x13, 0x0c, 0x12, 0x29, 0x51, 0xb9, 0xcf,
0x8b, 0x56, 0x80 0x81, 0x54, 0xc0, 0xed, 0x4e, 0x44, 0xa7, 0x2a, 0x85, 0x25, 0xe6, 0xca, 0x7c,
    },
    new byte[]
    {
        0xce, 0xbb, 0xeb, 0x92, 0xea, 0xcb, 0x13, 0xc1, 0xe9, 0x3a, 0xd6, 0xb2,
0xd2, 0x90, 0x17, 0xf8, 0x42, 0x15, 0x56, 0xb4, 0x65, 0x1c, 0x88, 0x43, 0xc5, 0x5c, 0x36, 0xba, 0xf5,
0x57, 0x67, 0x8d, 0x31, 0xf6, 0x64, 0x58, 0x9e, 0xf4, 0x22, 0xaa, 0x75, 0x0f, 0x02, 0xb1, 0xdf,
0x6d, 0x73, 0x4d, 0x7c, 0x26, 0x2e, 0xf7, 0x08, 0x5d, 0x44, 0x3e, 0x9f, 0x14, 0xc8, 0xae, 0x54,
0x10, 0xd8, 0xbc, 0x1a, 0x6b, 0x69, 0xf3, 0xbd, 0x33, 0xab, 0xfa, 0xd1, 0x9b, 0x68, 0x4e, 0x16,
0x95, 0x91, 0xee, 0x4c, 0x63, 0x8e, 0x5b, 0xcc, 0x3c, 0x19, 0xa1, 0x81, 0x49, 0x7b, 0xd9, 0x6f,
0x37, 0x60, 0xca, 0xe7, 0x2b, 0x48, 0xfd, 0x96, 0x45, 0xfc, 0x41, 0x12, 0x0d, 0x79, 0xe5, 0x89,
0x8c, 0xe3, 0x20, 0x30, 0xdc, 0xb7, 0x6c, 0x4a, 0xb5, 0x3f, 0x97, 0xd4, 0x62, 0x2d, 0x06, 0xa4,
0xa5, 0x83, 0x5f, 0x2a, 0xda, 0xc9, 0x00, 0x7e, 0xa2, 0x55, 0xbf, 0x11, 0xd5, 0x9c, 0xcf, 0x0e,
0x0a, 0x3d, 0x51, 0x7d, 0x93, 0x1b, 0xfe, 0xc4, 0x47, 0x09, 0x86, 0x0b, 0x8f, 0x9d, 0x6a, 0x07,
0xb9, 0xb0, 0x98, 0x18, 0x32, 0x71, 0x4b, 0xef, 0x3b, 0x70, 0xa0, 0xe4, 0x40, 0xff, 0xc3, 0xa9,
0xe6, 0x78, 0xf9, 0x8b, 0x46, 0x80, 0x1e, 0x38, 0xe1, 0xb8, 0xa8, 0xe0, 0x0c, 0x23, 0x76, 0x1d,
0x25, 0x24, 0x05, 0xf1, 0x6e, 0x94, 0x28, 0x9a, 0x84, 0xe8, 0xa3, 0x4f, 0x77, 0xd3, 0x85, 0xe2,
0x52, 0xf2, 0x82, 0x50, 0x7a, 0x2f, 0x74, 0x53, 0xb3, 0x61, 0xaf, 0x39, 0x35, 0xde, 0xcd, 0x1f,
0x99, 0xac, 0xad, 0x72, 0x2c, 0xdd, 0xd0, 0x87, 0xbe, 0x5e, 0xa6, 0xec, 0x04, 0xc6, 0x03, 0x34,
0xfb, 0xdb, 0x59, 0xb6, 0xc2, 0x01, 0xf0, 0x5a, 0xed, 0xa7, 0x66, 0x21, 0x7f, 0x8a, 0x27, 0xc7,
0xc0, 0x29, 0xd7
    },
    new byte[]
    {
        0x93, 0xd9, 0x9a, 0xb5, 0x98, 0x22, 0x45, 0xfc, 0xba, 0x6a, 0xdf, 0x02,
0x9f, 0xdc, 0x51, 0x59, 0x4a, 0x17, 0x2b, 0xc2, 0x94, 0xf4, 0xbb, 0xa3, 0x62, 0xe4, 0x71, 0xd4, 0xcd,
0x70, 0x16, 0xe1, 0x49, 0x3c, 0xc0, 0xd8, 0x5c, 0x9b, 0xad, 0x85, 0x53, 0xa1, 0x7a, 0xc8, 0x2d,
0xe0, 0xd1, 0x72,

```

```

0xde, 0xb2, 0x90, 0xa6, 0x2c, 0xc4, 0xe3, 0x76, 0x78, 0xb7, 0xb4, 0x09, 0x3b, 0x0e, 0x41, 0x4c,
0x7d, 0xcb, 0x35, 0x25, 0xa5, 0xd7, 0x03, 0x11, 0x00, 0xc3, 0x2e, 0x92, 0xef, 0x4e, 0x12, 0x9d,
0x36, 0xe6, 0x48, 0x10, 0xd5, 0x4f, 0x9e, 0x4d, 0xa9, 0x55, 0xc6, 0xd0, 0x7b, 0x18, 0x97, 0xd3,
0x7c, 0xda, 0x38, 0x56, 0x81, 0x8f, 0x77, 0xcc, 0x9c, 0xb9, 0xe2, 0xac, 0xb8, 0x2f, 0x15, 0xa4,
0xaf, 0x5e, 0x33, 0x1e, 0x0b, 0x05, 0xd6, 0x14, 0x6e, 0x6c, 0x7e, 0x66, 0xfd, 0xb1, 0xe5, 0x60,
0xed, 0x80, 0x29, 0x87, 0xc9, 0xf0, 0x5d, 0x6d, 0x3f, 0x88, 0x8d, 0xc7, 0xf7, 0x1d, 0xe9, 0xec,
0x5b, 0x40, 0x83, 0x27, 0xcf, 0x99, 0xa8, 0x50, 0x0f, 0x37, 0x24, 0x28, 0x30, 0x95, 0xd2, 0x3e,
0xee, 0x31, 0xa2, 0xb3, 0x69, 0x57, 0x1f, 0x07, 0x1c, 0x8a, 0xbc, 0x20, 0xeb, 0xce, 0x8e, 0xab,
0x96, 0xdd, 0x43, 0x73, 0xf9, 0xca, 0x3a, 0x1a, 0xfb, 0x0d, 0xc1, 0xfe, 0xfa, 0xf2, 0x6f, 0xbd,
0x64, 0x84, 0x82, 0x52, 0xb6, 0x08, 0xf3, 0xae, 0xbe, 0x19, 0x89, 0x32, 0x26, 0xb0, 0xea, 0x4b,
0x65, 0xe8, 0x91, 0x6b, 0xf5, 0x79, 0xbf, 0x01, 0x5f, 0x75, 0x63, 0x1b, 0x23, 0x3d, 0x68, 0x2a,
0x06, 0x46, 0x44, 0xf6, 0xff, 0x13, 0x58, 0xf1, 0x47, 0x0a, 0x7f, 0xc5, 0xa7, 0xe7, 0x61, 0x5a,
0x0c, 0x74, 0x67, 0x42, 0x04, 0xa0, 0xdb, 0x39, 0x86, 0x54, 0xaa, 0x8c, 0x34, 0x21, 0x8b, 0xf8,
    },
    new byte[]
    {
        0x68, 0x8d, 0xca, 0x4d, 0x73, 0x4b, 0x4e, 0x2a, 0xd4, 0x52, 0x26, 0xb3,
0x54, 0x1e, 0x19, 0x1f, 0x22, 0x03, 0x46, 0x3d, 0x2d, 0x4a, 0x53, 0x83, 0x13, 0x8a, 0xb7, 0xd5, 0x25,
0x79, 0xf5, 0xbd, 0x58, 0x2f, 0x0d, 0x02, 0xed, 0x51, 0x9e, 0x11, 0xf2, 0x3e, 0x55, 0x5e, 0xd1,
0x16, 0x3c, 0x66, 0x70, 0x5d, 0xf3, 0x45, 0x40, 0xcc, 0xe8, 0x94, 0x56, 0x08, 0xce, 0x1a, 0x3a,
0xd2, 0xe1, 0xdf, 0xb5, 0x38, 0x6e, 0x0e, 0xe5, 0xf4, 0xf9, 0x86, 0xe9, 0x4f, 0xd6, 0x85, 0x23,
0xcf, 0x32, 0x99, 0x31, 0x14, 0xae, 0xee, 0xc8, 0x48, 0xd3, 0x30, 0xa1, 0x92, 0x41, 0xb1, 0x18,
0xc4, 0x2c, 0x71, 0x72, 0x44, 0x15, 0xfd, 0x37, 0xbe, 0x5f, 0xaa, 0x9b, 0x88, 0xd8, 0xab, 0x89,
0x9c, 0xfa, 0x60, 0xea, 0xbc, 0x62, 0x0c, 0x24, 0xa6, 0xa8, 0xec, 0x67, 0x20, 0xdb, 0x7c, 0x28,
0xdd, 0xac, 0x5b, 0x34, 0x7e, 0x10, 0xf1, 0x7b, 0x8f, 0x63, 0xa0, 0x05, 0x9a, 0x43, 0x77, 0x21,
0xbf, 0x27, 0x09, 0xc3, 0x9f, 0xb6, 0xd7, 0x29, 0xc2, 0xeb, 0xc0, 0xa4, 0x8b, 0x8c, 0x1d, 0xfb,
0xff, 0xc1, 0xb2, 0x97, 0x2e, 0xf8, 0x65, 0xf6, 0x75, 0x07, 0x04, 0x49, 0x33, 0xe4, 0xd9, 0xb9,
0xd0, 0x42, 0xc7, 0x6c, 0x90, 0x00, 0x8e, 0x6f, 0x50, 0x01, 0xc5, 0xda, 0x47, 0x3f, 0xcd, 0x69,
0xa2, 0xe2, 0x7a, 0xa7, 0xc6, 0x93, 0x0f, 0x0a, 0x06, 0xe6, 0x2b, 0x96, 0xa3, 0x1c, 0xaf, 0x6a,
0x12, 0x84, 0x39, 0xe7, 0xb0, 0x82, 0xf7, 0xfe, 0x9d, 0x87, 0x5c, 0x81, 0x35, 0xde, 0xb4, 0xa5,
0xfc, 0x80, 0xef, 0xcb, 0xbb, 0x6b, 0x76, 0xba, 0x5a, 0x7d, 0x78, 0x0b, 0x95, 0xe3, 0xad, 0x74,
0x98, 0x3b, 0x36, 0x64, 0x6d, 0xdc, 0xf0, 0x59, 0xa9, 0x4c, 0x17, 0x7f, 0x91, 0xb8, 0xc9, 0x57,
0x1b, 0xe0, 0x61
    }
};
#endregion
}
}

```

ДОДАТОК Б

Код реалізації графічного інтерфейсу

MainWindow.xaml

```
<Window x:Class="KupynaUI.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:res="clr-namespace:KupynaUI.Properties"
        mc:Ignorable="d"
        Title="{DynamicResource Title}" Height="452.099" Width="800" MinWidth="700"
        MinHeight="400">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="395*" />
            <ColumnDefinition Width="397*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="22" />
            <RowDefinition />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <TextBox x:Name="hash_output" Margin="10,0,10,60" TextWrapping="Wrap"
            IsReadOnly="True" Grid.Column="1" Height="74" VerticalAlignment="Bottom" Grid.Row="1" />
        <Button x:Name="Save_Hash" Content="{DynamicResource SaveToFile}" Grid.Column="1"
            Margin="10,0,10,13" Height="42" VerticalAlignment="Bottom" Click="Save_Hash_Click"
            Grid.Row="1" />
        <Label Content="{DynamicResource HashLenght}" Grid.Column="1"
            HorizontalAlignment="Left" Margin="10,59,0,0" VerticalAlignment="Top" Width="122"
            Grid.Row="1" Height="26" />
        <Button x:Name="OpenFile" Content="{DynamicResource OpenFile}"
            Margin="10,0,10,60" Click="OpenFile_Click" Height="40" VerticalAlignment="Bottom"
            Grid.Row="1" />
        <Button x:Name="ClearFile" Content="{DynamicResource ClearData}"
            Margin="10,0,10,13" Click="ClearFile_Click" Height="42" VerticalAlignment="Bottom"
            Grid.Row="1" />
        <Button x:Name="Hash_Data" Content="{DynamicResource Calculate}" Grid.Column="1"
            Margin="10,0,12,139" Click="Hash_Data_Click" Height="40" VerticalAlignment="Bottom"
            Grid.Row="1" />
        <Label Content="{DynamicResource Encoding}" Grid.Column="1"
            HorizontalAlignment="Left" Margin="10,87,0,0" VerticalAlignment="Top" Width="122"
            Grid.Row="1" Height="26" />
        <RadioButton x:Name="hash_256" Content="256 bits" Grid.Column="1"
            Margin="0,65,92,0" VerticalAlignment="Top" HorizontalAlignment="Right" Width="61"
            GroupName="hash" IsChecked="True" Grid.Row="1" Height="15" />
        <RadioButton x:Name="hash_512" Content="512 bits" Grid.Column="1"
            Margin="0,65,12,0" VerticalAlignment="Top" HorizontalAlignment="Right" Width="61"
            GroupName="hash" Grid.Row="1" Height="15" />
        <RadioButton x:Name="hash_ascii" Content="ASCII" Grid.Column="1"
            Margin="0,93,172,0" VerticalAlignment="Top" HorizontalAlignment="Right" Width="61"
            GroupName="code" IsChecked="True" Grid.Row="1" Height="15" />
        <RadioButton x:Name="hash_utf8" Content="UTF-8" Grid.Column="1"
            Margin="0,93,92,0" VerticalAlignment="Top" HorizontalAlignment="Right" Width="61"
            GroupName="code" Grid.Row="1" Height="15" />
    </Grid>
</Window>
```

```

        <RadioButton x:Name="hash_hex" Content="Hex" Grid.Column="1" Margin="0,93,12,0"
VerticalAlignment="Top" HorizontalAlignment="Right" Width="61" GroupName="code"
Grid.Row="1" Height="15"/>
        <TextBox
            x:Name="Input_Text"
            Margin="10,35,10,139"
            TextWrapping="Wrap"
            ScrollViewer.VerticalScrollBarVisibility="Auto"
            ScrollViewer.HorizontalScrollBarVisibility="Auto" Grid.Row="1"/>
        <TextBlock HorizontalAlignment="Left" Margin="10,136,0,0" TextWrapping="Wrap"
VerticalAlignment="Top" IsEnabled="False" Grid.Row="1" Height="16" Width="0"/>
        <TextBox x:Name="TextBox_FilePath" Margin="10,0,10,105" TextWrapping="Wrap"
IsEnabled="False" IsReadOnly="True" Height="29" VerticalAlignment="Bottom"
BorderBrush="{x:Null}" Grid.Row="1"/>
        <Label Content="{DynamicResource HashMessage}" HorizontalAlignment="Left"
Margin="10,2,0,0" VerticalAlignment="Top" Grid.Row="1" Height="26" Width="174"/>
        <Menu Grid.ColumnSpan="2" Height="22" VerticalAlignment="Top" Background="White">
            <MenuItem Name ="FileMenu" x:Uid="File" Header="{DynamicResource File}">
                <MenuItem Header="{DynamicResource OpenFile}" Click="MenuItem_Click_2"/>
                <MenuItem Header="{DynamicResource SaveMessageFile}"
Click="MenuItem_Click_3"/>
                <Separator/>
                <MenuItem Header="{DynamicResource SaveToFile}"
Click="MenuItem_Click_4"/>
                <Separator/>
                <MenuItem Header="{Binding Source={x:Static res:Resources.ClearData}}"
Click="MenuItem_Click_5"/>
            </MenuItem>
            <MenuItem x:Name="menuLanguage" Header="{DynamicResource Language}">
                <RadioButton Content="Українська" IsChecked="True"
Checked="Ukrainian_Checked" GroupName="language" />
                <RadioButton Content="English" Checked="English_Checked"
GroupName="language" />
            </MenuItem>
            <MenuItem Header="{DynamicResource Examples}" Click="MenuItem_Click_1"/>
            <MenuItem Header="{DynamicResource Help}" Click="MenuItem_Click"/>
        </Menu>
        <Label Content="{DynamicResource Settings}" HorizontalAlignment="Left"
Margin="10,30,0,0" VerticalAlignment="Top" Grid.Row="1" Height="26" Width="174"
Grid.Column="1"/>

    </Grid>
</Window>

```

HelpWindows.xaml

```

<Window x:Name="HelpWindow1" x:Class="KupynaUI.HelpWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:res="clr-namespace:KupynaUI.Properties"
mc:Ignorable="d"
Title="{DynamicResource Help}" Height="476.033" Width="361.157"
ResizeMode="NoResize">
    <Grid Margin="0,0,0,0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
    </Grid>

```

```

        <Image Margin="156,0,158,55" Source="Resources/NUZP-GERB.svg.png" Height="49"
VerticalAlignment="Bottom" HorizontalAlignment="Center"/>
        <Label Content="Національний університет &quot;Запорізька політехніка&quot;;"
Margin="34,0,29,24" Height="26" VerticalAlignment="Bottom" Width="292"
HorizontalContentAlignment="Center" HorizontalAlignment="Center"/>
        <Label Content="Автор: Ковальов І.Є" Margin="34,0,29,0" Height="26"
VerticalAlignment="Bottom" Width="292" HorizontalContentAlignment="Center"
HorizontalAlignment="Center"/>
        <TextBlock TextWrapping="Wrap" Margin="34,10,37,0" Height="42"
VerticalAlignment="Top" HorizontalAlignment="Center" TextAlignment="Center" Width="284">
        Програма реалізація алгоритму хешування
        Купина ДСТУ 7564:2014
        </TextBlock>
        <TextBlock TextWrapping="Wrap" Margin="10,72,10,0" Height="266"
VerticalAlignment="Top"><Run Text="Програма дозволяє:

1. Обчислювати хеш для будь якого повідомлення, для цього достатньо лише ввести
повідомлення в вікно &quot;Повідомлення для хешування&quot;;

2. Відкривати текстові файли для подальшого хешування

3. Зберігати результат хешування в файл

4. Перемикає режим декодування повідомлення (ASCII, UTF8, HEX-число)

5. Перемикається між режимами хешування 256-біт та 512-біт

6. Завантажити приклади із ДСТУ 7564:2014

Приємного користування!"/></TextBlock>
    </Grid>
</Window>

```

Examples.xaml

```

<Window x:Name="Examples1" x:Class="KupynaUI.Examples"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:res="clr-namespace:KupynaUI.Properties"
mc:Ignorable="d"
Title="{ DynamicResource Examples}" Height="420.248" Width="312.81"
MinHeight="420" MinWidth="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="348*"/>
            <RowDefinition Height="43*"/>
        </Grid.RowDefinitions>
        <Button x:Name="AddExample" Content="{DynamicResource Add}" Margin="0,0,10,10"
Grid.Row="1" Height="23" VerticalAlignment="Bottom" HorizontalAlignment="Right"
Width="105" Click="AddExample_Click"/>
        <ListBox x:Name="Examples_list" Margin="10" FontFamily="Times New Roman"
FontSize="16"
ScrollViewer.HorizontalScrollBarVisibility="Disabled">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding}" TextWrapping="Wrap"/>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>

```

```

        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
</Window>

```

MainWindow.cs

```

using System;
using System.Threading.Tasks;
using System.IO;
using Microsoft.Win32;
using System.Windows;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;
using System.Globalization;
using System.Threading;
using System.Windows.Controls;

namespace KupynaUI
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            App.Loading();
            App.LanguageChanged += LanguageChanged;
            CultureInfo currLang = App.Language;
            menuLanguage.Items.Clear();
            foreach (var lang in App.Languages)
            {
                MenuItem menuLang = new MenuItem();
                menuLang.Header = lang.DisplayName;
                menuLang.Tag = lang;
                menuLang.IsChecked = lang.Equals(currLang);
                menuLang.Click += ChangeLanguageClick;
                menuLanguage.Items.Add(menuLang);
            }
        }
        private void LanguageChanged(Object sender, EventArgs e)
        {
            CultureInfo currLang = App.Language;
            foreach (MenuItem i in menuLanguage.Items)
            {
                CultureInfo ci = i.Tag as CultureInfo;
                i.IsChecked = ci != null && ci.Equals(currLang);
            }
        }
        private void ChangeLanguageClick(Object sender, EventArgs e)
        {
            MenuItem mi = sender as MenuItem;
            if (mi != null)
            {
                CultureInfo lang = mi.Tag as CultureInfo;
                if (lang != null)
                {
                    App.Language = lang;
                }
            }
        }
    }
}

```

```

    }
    private async void Hash_Data_Click(object sender, RoutedEventArgs e)
    {
        try
        {
            byte[] message = null;
            switch (true)
            {
                case bool b when b == (bool)hash_ascii.IsChecked:
                    message = DataEncoding.GetBytesASCII(Input_Text.Text);
                    break;
                case bool b when b == (bool)hash_utf8.IsChecked:
                    message = DataEncoding.GetBytesUTF8(Input_Text.Text);
                    break;
                case bool b when b == (bool)hash_hex.IsChecked:
                    message = DataEncoding.GetBytesHex(Input_Text.Text);
                    break;
            }
            await RunHashProcessAsync(message);
        }
        catch (Exception)
        {
            MessageBox.Show("Input error!", "Error!");
        }
    }
    private async Task RunHashProcessAsync(byte[] message)
    {
        TurnInterface(false);
        hash_output.Text = await HashMessageAsync(message,
        (bool)hash_256.IsChecked?256:512);
        TurnInterface(true);
    }

    private Task<string> HashMessageAsync(byte[] message, int v)
    {
        return Task.Run(() =>
        {
            string result;
            result = BitConverter.ToString(KupynaHashFunction.Kupyna.Hash(message,
v)).Replace("-", "");
            return result;
        });
    }
    private void TurnInterface(bool state)
    {
        Hash_Data.IsEnabled = state;
        Save_Hash.IsEnabled = state;
        OpenFile.IsEnabled = state;
        ClearFile.IsEnabled = state;
        Input_Text.IsEnabled = state;
        hash_256.IsEnabled = state;
        hash_512.IsEnabled = state;
        hash_ascii.IsEnabled = state;
        hash_hex.IsEnabled = state;
        hash_utf8.IsEnabled = state;
    }

    private void OpenFile_Click(object sender, RoutedEventArgs e)
    {
        _ = OpenFileProcAsync();
    }

    private async Task OpenFileProcAsync()

```

```

{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    if (openFileDialog.ShowDialog() == true)
    {
        if (File.Exists(openFileDialog.FileName))
        {
            try
            {
                using (StreamReader streamReader = new
StreamReader(openFileDialog.FileName))
                {
                    TurnInterface(false);
                    string message = await streamReader.ReadToEndAsync();
                    TextBox_FilePath.Text = openFileDialog.FileName;
                    Input_Text.Text = message;
                    TurnInterface(true);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error loading file: " + ex.Message, "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }
        else
        {
            MessageBox.Show("File not found: " + openFileDialog.FileName,
"Error", MessageBoxButton.OK, MessageBoxImage.Error);
        }
    }
}

private void ClearFile_Click(object sender, RoutedEventArgs e)
{
    Input_Text.Text = null;
    TextBox_FilePath.Text = null;
}

private void Save_Hash_Click(object sender, RoutedEventArgs e)
{
    SaveHashResult();
}

private void SaveHashResult()
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    saveFileDialog.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog.DefaultExt = ".txt";
    saveFileDialog.AddExtension = true;

    if (saveFileDialog.ShowDialog() == true)
    {
        File.WriteAllText(saveFileDialog.FileName, hash_output.Text);
    }
}

private void SaveInputData()
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    saveFileDialog.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog.DefaultExt = ".txt";
    saveFileDialog.AddExtension = true;

    if (saveFileDialog.ShowDialog() == true)
    {

```

```

        File.WriteAllText(saveFileDialog.FileName, Input_Text.Text);
    }
}

private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    if (App.Language.Name == "uk-UA")
    {
        HelpWindow helpWindow = new HelpWindow();
        helpWindow.Show();
    }
    else
    {
        HelpWindowEN helpWindowEN = new HelpWindowEN();
        helpWindowEN.Show();
    }
}

private void MenuItem_Click_1(object sender, RoutedEventArgs e)
{
    Examples examples = new Examples();
    examples.SetWindowObject(this);
    examples.Show();
}

private void MenuItem_Click_2(object sender, RoutedEventArgs e)
{
    _ = OpenFileProcAsync();
}

private void MenuItem_Click_3(object sender, RoutedEventArgs e)
{
    SaveInputData();
}

private void MenuItem_Click_4(object sender, RoutedEventArgs e)
{
    SaveHashResult();
}

private void MenuItem_Click_5(object sender, RoutedEventArgs e)
{
    Input_Text.Text = null;
}

private void Ukrainian_Checked(object sender, RoutedEventArgs e)
{
    //SetLanguage(new CultureInfo("uk-UA"));
}

private void English_Checked(object sender, RoutedEventArgs e)
{
    // SetLanguage(new CultureInfo("en-US"));
}
}
}

```

ДОДАТОК В

ПРЕЗЕНТАЦІЯ



Рисунок В.1 – Титульний слайд

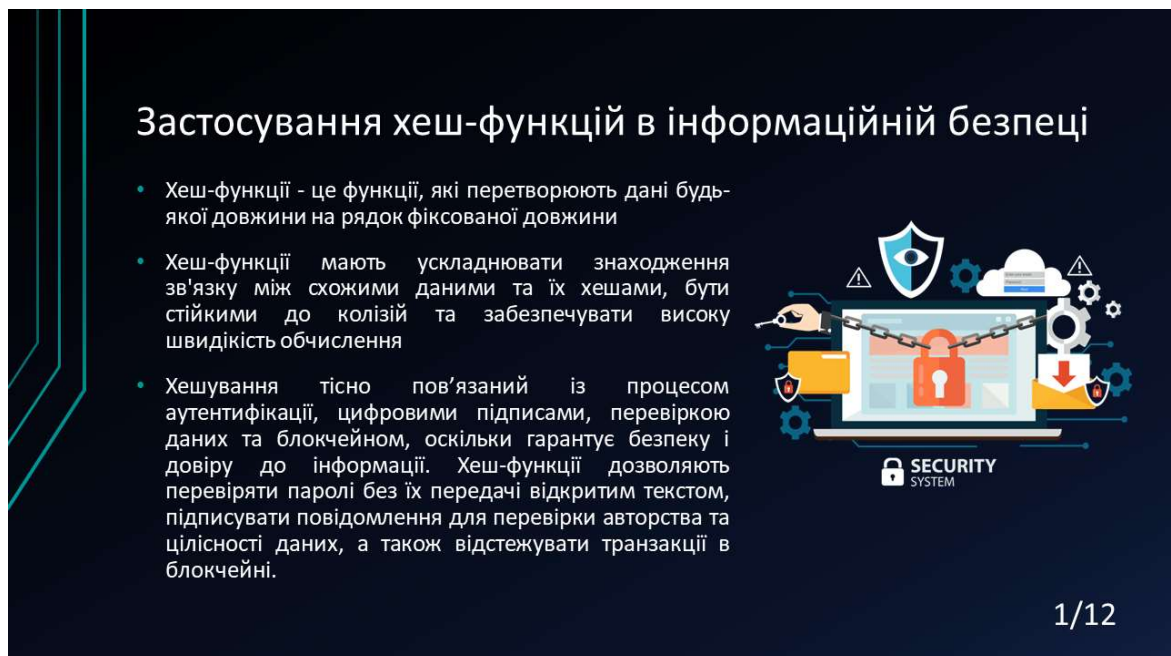


Рисунок В.2 – Слайд 1

Властивості хеш-функції «Купина» ДСТУ 7564:2014



- Ітеративна хеш-функція «Купина», яка представлена в ДСТУ 7564:2014 розроблена в Україні в «Інституті інформаційних технологій», який введено в дію від 2 грудня 2014 року та чинний від 1 квітня 2015 року як заміна ГОСТ Р 34.11-94 та «Стрибог»
- «Купина» пропонує декілька варіантів хеш-функцій, включаючи Купина-256, Купина-384 та Купина-512, які відрізняються довжиною хешу і параметрами безпеки. Для більшості криптографічних застосувань рекомендується «Купина-256»
- Автори ДСТУ 7564:2014 стверджують, що диференціальні та rebound-атаки, які використовуються для злому хеш-функцій, стають неефективними після 4-х ітерацій перестановок.

2/12

Рисунок В.3 – Слайд 2

ПРОГРАМНА РЕАЛІЗАЦІЯ ХЕШ-ФУНКЦІЇ «КУПИНА» ДСТУ 7564:2014 ВИБІР МОВИ ПРОГРАМУВАННЯ

- Для реалізації хеш-функції "Купина" ДСТУ 7564:2014 обрано C# через його високу продуктивність, об'єктно-орієнтований підхід, зручність використання, безпеку, широку підтримку .NET Framework, платформонезалежність, велику спільноту та сумісність з іншими технологіями.
- Програмний код буде створено як бібліотечну функцію, яка приймає бітову послідовність повідомлення та int число для визначення розміру хешу (256 або 512 біт). Обраним середовищем розробки є Visual Studio 2023.



3/12

Рисунок В.4 – Слайд 3

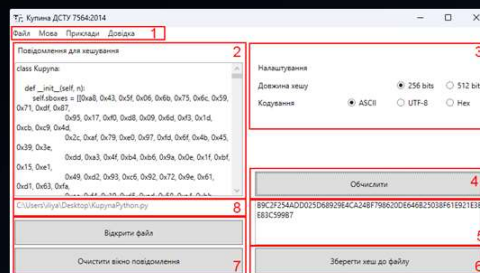
Готові бібліотечні реалізації

1. «Курыпа-reference» - це референсна реалізація на C, створена Романом Олійниковим, одним з розробників хеш-функції.
2. «Сррcrypto» - це відкрита бібліотека на C++, яка включає різні функції хешування та шифрування, включаючи "Купина" і "Калина". Вона також містить інструменти командного рядка для обчислення та перевірки хешів файлів та шифрування файлів.
3. «Cryptonite» - це відкрита бібліотека на C від PrivatBank для криптографічних перетворень, яка отримала державний сертифікат та може використовуватися для захисту інформації з обмеженим доступом та відкритою інформацією.

4/12

Рисунок В.5 – Слайд 4

Розробка програмного забезпечення хеш-функції «Купина» ДСТУ 7564:2014



1 – загальне меню, 2 – вікно для введення повідомлення для хешування, 3 – область налаштувань, 4 – кнопка, яка розпочинає процес хешування, 5 – поле, в яке виводиться результат хешування, 6 – кнопка, яка відкриває вікно зберігання результату хешування, 7 – кнопки керування вмістом поля для введення повідомлення для хешування, 8 – назва відкритого файлу

- Спираючись на референсну версію реалізації алгоритму на мові програмування C, було розроблено власну бібліотеку на мові C#
- Бібліотека представлена у вигляді DLL файлу, який можна під'єднати до будь-якого проекту на платформі .Net Framework починаючи із версії 4.
- Для зручності роботи із програмою було реалізовано графічний інтерфейс програми за допомогою графічної бібліотеки WPF (Windows Presentation Foundation)

Додаток працює із Windows 10/11 (а також більш старших версіях за наявності бібліотеки .Net Framework).

Додаток та бібліотека доступні за посиланням:
<https://github.com/IlliaKovalov/Kupyna-DSTU-7564-2014>

5/12

Рисунок В.6 – Слайд 5

Аналіз функцій на стійкість до колізій

- За допомогою методу «Днів народження» проведено пошук колізій. Метод «Днів народження» – це спосіб криптоаналізу, який має на меті знайти колізії у хеш-функціях. Ймовірність колізії підвищується через те, що хеш-функція переводить повідомлення різної довжини у фіксовану кількість бітів.
- Атака днів народження проводиться наступним чином:
 1. випадково генеруються повідомлення;
 2. обчислюється хеш-функція для кожного повідомлення;
 3. хеші зберігаються у структурі даних (наприклад, словник);
 4. перевірка на наявність колізії (дві різні повідомлення з однаковим хешем).



6/12

Рисунок В.7 – Слайд 6

Результати аналізу

Кількість біт повідомлення	Купина-256		Купина-512	
	Спроб перед першою колізією	Час на обчислення	Спроб перед першою колізією	Час на обчислення
32	420	1:18	471	1:26
64	12401	19:15	13943	24:15
80	31578	59:54	33578	1:00:12
88	148545	4:49:18	158545	5:09:18
Екстрапольовані дані				
96	487380	16:00:00	518454	16:40:00
128	38355174	1248:00:00	41284596	1360:00
256	147111939630963	47 867 257	17044178322	56 147 049 20
	0	152:00:00	85730	0:00:00

Кількість біт повідомлення	Функція хешування			
	Купина-256	Купина-512	MD5	SHA256
32	420	471	148	157
128	38355174	41284596	137729200	155289240
256	147111939	1704417832	18969332	241147480
	6309630	285730	668809200	37387800

За отриманими даними можна зробити висновки, що «Купина» більш ефективна для коротких повідомлень, а MD5 та SHA256 більш стійкий при більшому повідомленні.

7/12

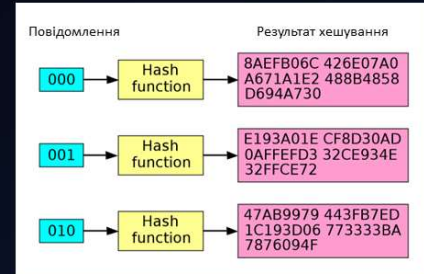
Рисунок В.8 – Слайд 7

Аналіз функцій на аваланшний ефект із вимірюванням середньої відстані Хеммінга

Аваланшний (або лавинний) ефект — це властивість криптографічних хеш-функцій, коли невелика зміна на вході повинна призводити до значної зміни на виході.

Показник вимірюється як відстань Хеммінга, яка являє собою кількість різних бітів між двома хешами.

Хороша хеш-функція повинна демонструвати середню відстань Хеммінга близькою до 50% від розміру хешу в бітах (наприклад, 128 біт для 256-бітового хешу)



8/12

Рисунок В.9 – Слайд 8

Результати аналізу

Функція хешування	Довжина хешу (біт)	Середня відстань Хеммінга (біт)
Курипа-256	256	127,97657
SHA256	256	128,00681
Курипа-512	512	255,95849
SHA512	512	255,99733
MD5	256	63,67954

За отриманими даними можна зробити висновок, що за показниками вимірювання аваланшного ефекту функція «Курипа» має майже однакові результати із перевіреним часом та досить популярним SHA256 та SHA512, відхилення в результатах складає ~0,03 біти.

9/12

Рисунок В.10 – Слайд 9

Результати аналізу

Функція хешування	16 384 Бит	262 144 Бит	524 288 Бит	800 000 Бит	8 000 000 Бит
Купина-256	34 490	2 477 298	4 954 556	7 595 742	-
Reference					
Купина-256 C#	333 753,8	5 188 105,4	10 307 448,9	15 788 036	157 628 228
MD5	7109	7210	7210	7135	7281
SHA-256	7147	7251	7251	7178	7357

Функція хешування	Розмір блоку, що обробляється				
	1 byte	8 bytes	32 bytes	64 bytes	2048 bytes
Купина-256	3.9820	31.8547	126.8081	155.2432	278.7078
Купина-512	1.5440	12.2717	49.1410	98.0369	272.5507
Good-256	1.3974	12.7639	50.8011	66.7458	189.1179
Good-512	1.0712	8.5942	34.3238	68.7215	260.0773
Stribog (256,512)	0.9261	7.3999	29.4322	42.4140	139.0854
ГОСТ 34.311-95	2.4099	19.8379	79.3996	125.0529	202.7207
SHA2-256	14.1968	112.7476	444.6520	474.7652	980.8201
SHA2-512	4.4543	35.5932	141.9562	283.1616	571.1605
SHA3-256	1.5863	12.7119	51.0108	101.6447	258.3613
SHA3-512	1.5762	12.7096	50.6990	101.6891	209.9916

Intel Core i5-2300 2,8 ГГц; Windows Server 2008 R2 x64; Microsoft Visual Studio 2010/Visual C++

Внаслідок аналізу можна побачити, що бібліотечні функції, що вбудовані в System.Security.Cryptography [16] значно швидші за аналогічні реалізації на C# та на C. Для оптимізації продуктивності цього коду можна розглянути можливість використання інструкцій SIMD (Single Instruction, Multiple Data), таких як SSE2 [18] або AVX2 [19], які доступні на сучасних процесорах x86 і x64.

Але, якщо говорити про аналогічні реалізації інших функцій хешування, то «Купина» є досить швидким алгоритмом. Враховуючи те, що «Купина» має головну ціль замінити застарілі та досить ненадійні ГОСТ 34.311-95 та Stribog (ГОСТ 34.11-2018), з головною задачею він впорався.

11/12

Рисунок В.10 – Слайд 11

Аналіз швидкості обчислення хеш-образу

Тестування на швидкість алгоритму хешування буде проведено на референсній версії реалізації алгоритму на мові C, саме ця реалізація є найшвидшою у виконанні через «низькорівневе» програмування та більшу гнучкість у керуванні пам'яттю. Але аналогічне тестування буде проведено і для бібліотеки, яку реалізовано на C#.

Для повноти аналізу алгоритм буде порівняно з алгоритмами MD5, SHA-256. Всі обчислення виконуються на Windows 11 версії 22H2, процесорі AMD Ryzen 7 5800X (8 ядер, які працюють на частоті 3.8 ГГц), з 32 гігабайтами оперативної пам'яті.



10/12

Рисунок В.12 – Слайд 10

Висновки

В ході роботи було реалізовано функцію «Купина» на C# і було проаналізовано її властивості за допомогою референсної та власної реалізації. Було перевірено функцію на колізії, аваланшний ефект та швидкодію. Виявлено, що «Купина» ефективніша для коротких повідомлень, ніж MD5 та SHA256. Також, «Купина» має хороші показники аваланшного ефекту і перевершує ГОСТ 34.311-95 та Stribog за швидкості.

12/12

Рисунок В.13 – Слайд 12

Дякую за увагу!

Рисунок В.14 – Фінальний слайд