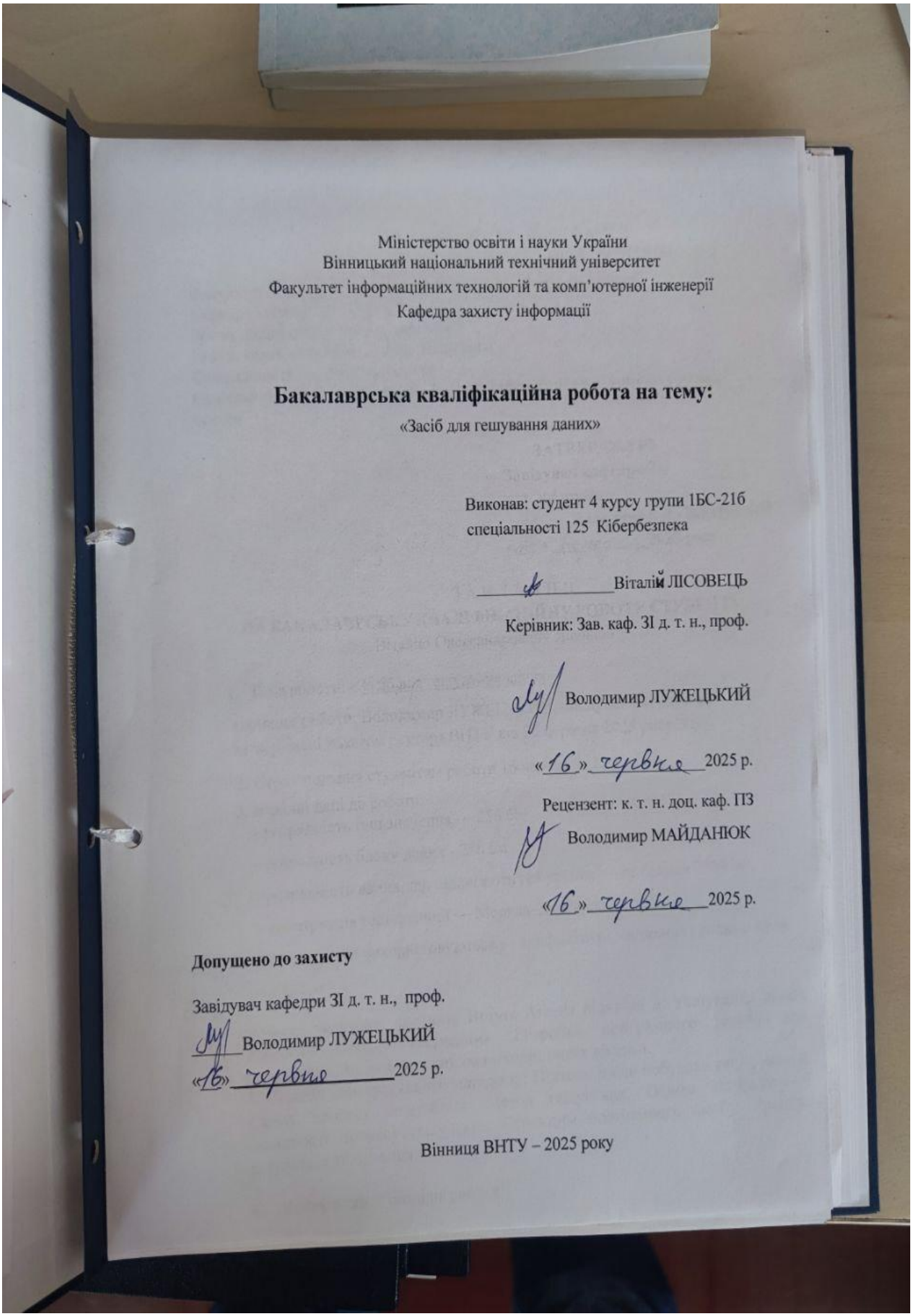


Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації

Бакалаврська кваліфікаційна робота на тему:

«Засіб для гешування даних»

Виконав: студент 4 курсу групи 1БС-21б
спеціальності 125 Кібербезпека

Віталій ЛІСОВЕЦЬ

Керівник: Зав. каф. ЗІ д. т. н., проф.

Володимир ЛУЖЕЦЬКИЙ

«16» червня 2025 р.

Рецензент: к. т. н. доц. каф. ПЗ

Володимир МАЙДАНЮК

«16» червня 2025 р.

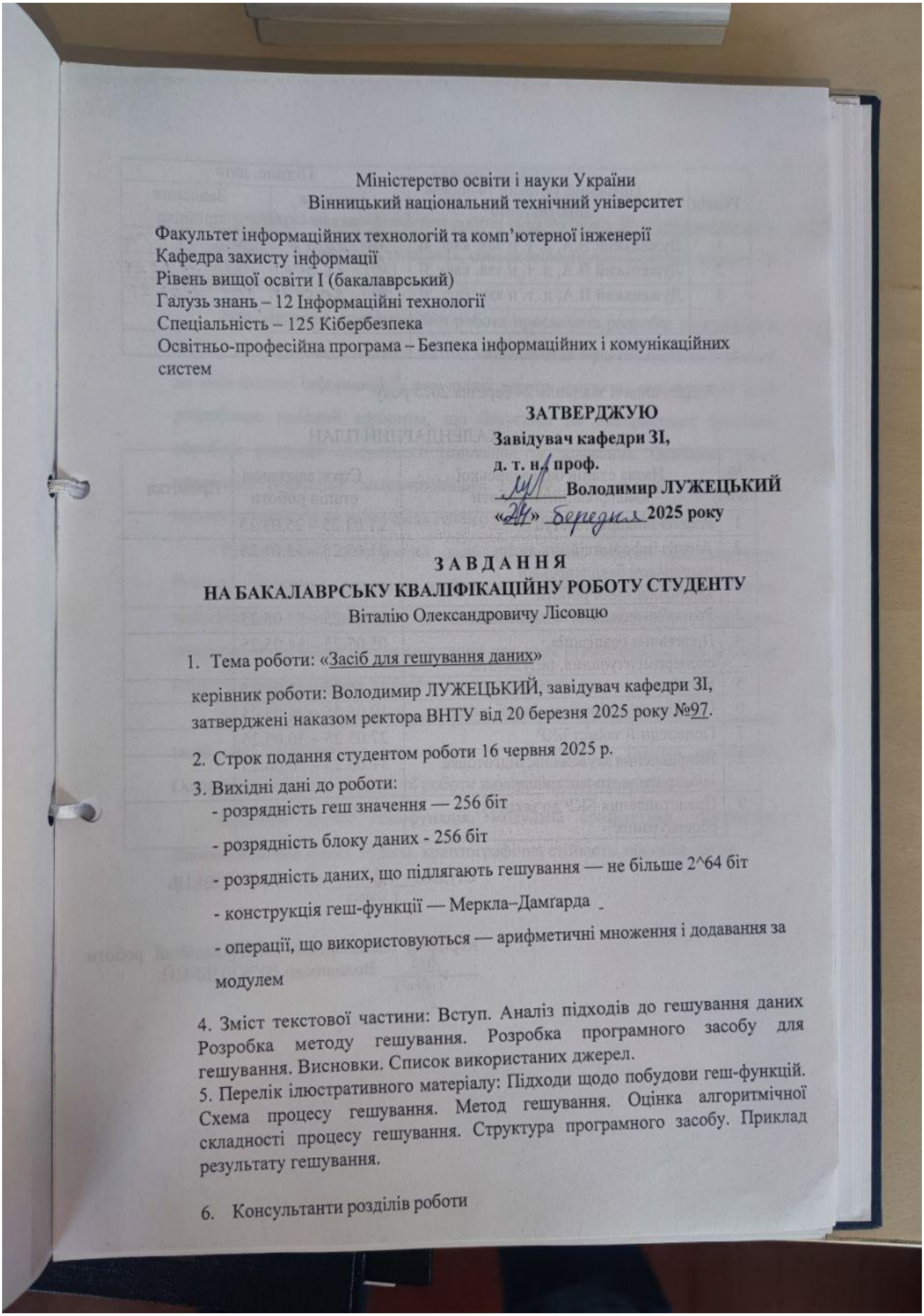
Допущено до захисту

Завідувач кафедри ЗІ д. т. н., проф.

Володимир ЛУЖЕЦЬКИЙ

«16» червня 2025 р.

Вінниця ВНТУ – 2025 року



Міністерство освіти і науки України
Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації
Рівень вищої освіти І (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 125 Кібербезпека
Освітньо-професійна програма – Безпека інформаційних і комунікаційних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри ЗІ,
д. т. н., проф.
Володимир ЛУЖЕЦЬКИЙ
«24» березня 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Віталію Олександровичу Лісовцю

1. Тема роботи: «Засіб для гешування даних»
керівник роботи: Володимир ЛУЖЕЦЬКИЙ, завідувач кафедри ЗІ, затверджені наказом ректора ВНТУ від 20 березня 2025 року №97.
2. Строк подання студентом роботи 16 червня 2025 р.
3. Вихідні дані до роботи:
   - розрядність геш значення — 256 біт
   - розрядність блоку даних - 256 біт
   - розрядність даних, що підлягають гешування — не більше 2^64 біт
   - конструкція геш-функції — Меркла–Дамгарда
   - операції, що використовуються — арифметичні множення і додавання за модулем
4. Зміст текстової частини: Вступ. Аналіз підходів до гешування даних Розробка методу гешування. Розробка програмного засобу для гешування. Висновки. Список використаних джерел.
5. Перелік ілюстративного матеріалу: Підходи щодо побудови геш-функцій. Схема процесу гешування. Метод гешування. Оцінка алгоритмічної складності процесу гешування. Структура програмного засобу. Приклад результату гешування.
6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Лужецький В.А, д. т. н зав. каф. ЗІ	24.03.25	11.04.25
2	Лужецький В.А, д. т. н зав. каф. ЗІ	16.05.25	11.05.25
3	Лужецький В.А, д. т. н зав. каф. ЗІ	30.05.25	11.05.25

7. Дата видачі завдання 24 березня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання. Вступ	21.03.25 – 25.03.25	
2	Аналіз інформаційних джерел за напрямком бакалаврської кваліфікаційної роботи	31.03.25 – 13.04.25	
3	Розробка моделей, алгоритмів	14.04.25 – 04.05.25	
4	Практична реалізація, експериментування, результати	05.05.25 – 14.05.25	
5	Висновки	15.05.25 – 18.05.25	
6	Оформлення пояснювальної записки	19.05.25 – 26.05.25	
7	Попередній захист БКР	27.05.25 – 30.05.25	
8	Виправлення зауважень, підготовка ілюстративного матеріалу	31.06.25 – 10.06.25	
9	Представлення БКР до захисту, рецензування	11.06.25 – 13.06.25	

Студент _____ Віталій ЛІСОВЕЦЬ
(підпис)

Керівник бакалаврської кваліфікаційної роботи
_____ Володимир ЛУЖЕЦЬКИЙ
(підпис)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 50 сторінок формату А4, на яких є 24 рисунків, 1 таблиць, список використаних джерел містить 19 найменувань.

Бакалаврська кваліфікаційна робота присвячена розробці програмного засобу для гешування даних з метою забезпечення ефективності та стійкості до змін вхідної інформації. У результаті аналізу існуючих геш-функцій було розроблено власний алгоритм, що базується на використанні блокової обробки, операцій модульного множення та додавання. Особливу увагу приділено реалізації лавиноподібного ефекту, що дозволяє забезпечити високу чутливість до змін навіть одного біта у вхідному повідомленні.

Розроблений програмний засіб реалізовано мовою програмування Python. Програма дозволяє здійснювати поблочне гешування даних, модифікацію повідомлень та аналіз зміни геш-значень. Додатково реалізовано функцію порівняння гешів для двох різних файлів, що дозволяє оцінити криптографічну стійкість запропонованого підходу.

У ході роботи розроблено, реалізовано та протестовано програмний засіб, що підтверджує коректність та ефективність роботи алгоритму. Основні задачі кваліфікаційної роботи виконано в повному обсязі.

Ключові слова: геш-функція, модульна арифметика, гешування, лавиноподібний ефект, Python, криптографічна стійкість, обробка даних

АНОТАЦІЯ

The bachelor’s qualification thesis consists of 50 A4-sized pages, including 24 figures, 1 tables, and a list of 19 references.

This thesis is dedicated to the development of a software tool for data hashing aimed at ensuring efficiency and sensitivity to changes in the input. Based on the analysis of existing hash functions, a custom algorithm was designed using block processing, modular multiplication, and addition. Special attention was paid to implementing the avalanche effect, which ensures high sensitivity to the alteration of even a single bit in the input message.

The developed software tool was implemented in the Python programming language. The application supports block-based data hashing, message modification, and analysis of hash changes. Additionally, a function was implemented for comparing the hash values of two different files, allowing the cryptographic strength of the algorithm to be evaluated.

The developed tool was implemented, tested, and verified to confirm the correctness and effectiveness of the algorithm. All objectives of the qualification thesis have been fully achieved.

Keywords: hash function, modular arithmetic, hashing, avalanche effect, Python, cryptographic strength, data processing.

Зміст

ВСТУП

На сьогоднішній день питання інформаційної безпеки набуло особливої актуальності. Постійна еволюція інформаційно-телекомунікаційних систем вимагає впровадження нових підходів і засобів для захисту даних. Серед ключових завдань інформаційної безпеки особливо важливими є забезпечення цілісності, достовірності та автентичності інформації.[1]

Ці завдання ефективно вирішуються за допомогою цифрових підписів, до складу яких обов'язково входять геш-функції. Окрім цього, гешування широко використовується в інших аспектах кіберзахисту — зокрема, при зберіганні паролів, захисті даних у мережах та верифікації повідомлень. Геш-функції також застосовуються в алгоритмах перевірки цілісності файлів, створенні цифрових відбитків, а також у блокчейн-технологіях, де вони виконують роль базового механізму безпеки.[7]

Однією з головних переваг геш-функцій є здатність перетворювати вхідні дані будь-якої довжини у вихід фіксованого розміру, що дозволяє оптимізувати подальші криптографічні операції. Також геш-функції зменшують надмірність інформації, що позитивно впливає на криптостійкість перетворених даних. Важливо й те, що добре сконструйована геш-функція має властивість стійкості до колізій, тобто унеможливлює генерацію однакових геш-значень для різних вхідних повідомлень, що критично для інформаційної безпеки.[2]

Однак, поряд із перевагами, класичні геш-алгоритми мають і низку обмежень, пов'язаних із зниженням стійкості до сучасних атак або обмеженою продуктивністю в певних умовах. У зв'язку з цим, актуальним є завдання пошуку нових способів побудови геш-функцій, які відповідали б сучасним вимогам як до безпеки, так і до швидкодії.[2]

З огляду на це, дослідження геш-функцій, аналіз їхніх недоліків і пошук нових підходів до побудови набувають практичної цінності.

Особливо важливим є створення ефективного програмного інструменту гешування, придатного для використання в прикладних системах, який дозволив би забезпечити баланс між криптостійкістю, швидкістю обробки та простотою реалізації в умовах сучасних IT-інфраструктур.

Об’єкт дослідження — процес гешування даних.

Предмет дослідження — метод і засіб для виконання гешування.

Мета роботи — пришвидшення процесу гешування данних шляхом використання арифметичних операцій за модулем.

Для досягнення мети необхідно розв’язати такі завдання:

-провести аналіз існуючих методів побудови геш-функцій;

-розробити метод гешування, що базується на використанні арифметичних операцій за модулем;

-створити програмний засіб, що реалізує запропонований метод.

Наукова новизна полягає у методі гешування, який на відміно від відомих передбачає використання лише арифметичних операцій за модулем, що забезпечує пришвидшення процесу гешування.

1. АНАЛІЗ ПІДХОДІВ ДО ПОБУДОВИ ГЕШ-ФУНКЦІЇ

1.1 Вимоги, що висуваються до функції гешування

Геш-функція — це особливий тип математичної функції, яка перетворює інформацію довільної довжини у коротке числове значення сталої довжини. Такий процес називається гешуванням і широко використовується в комп'ютерних науках для забезпечення безпеки даних та їх обробки. На відміну від звичайних функцій, які можуть повертати різні обсяги результату, геш-функції завжди дають вихід фіксованої довжини, незалежно від розміру вхідних даних. Це дозволяє ефективно використовувати їх у ситуаціях, де потрібна швидка перевірка автентичності або цілісності інформації. Отримане після гешування значення називається геш-значенням або дайджестом повідомлення. Воно є своєрідним "відбитком" початкових даних: навіть незначна зміна вхідної інформації призводить до повністю іншого гешу. Завдяки цій властивості геш-функції широко застосовуються в криптографії, для створення цифрових підписів, збереження паролів у зашифрованому вигляді, а також для перевірки цілісності файлів при передаванні їх через мережу [7].

Головною характеристикою геш-функцій є те, що вони формують вихідне значення фіксованої довжини, яке називають геш-значенням або дайджестом повідомлення. Це значення отримують із вхідних даних будь-якого розміру — від кількох байтів до гігабайтів. Такий процес трансформації відомий як гешування. Зазвичай довжина гешу значно менша за обсяг вхідних даних, тому геш-функції іноді ще називають функціями стиснення. Їхнє завдання — створити компактне, але унікальне представлення великих обсягів інформації. Наприклад, геш-функція з виходом довжиною n біт має назву n-бітної геш-функції. Найпоширеніші сучасні алгоритми гешування генерують геші довжиною від 160 до 512 біт.

Ще одна важлива перевага геш-функцій — висока швидкість обчислення. Для будь-якого заданого значення x, обчислення гешу $h(x)$ є надзвичайно швидкою операцією. У порівнянні з алгоритмами симетричного шифрування, гешування є значно ефективнішим з точки зору обчислювальних витрат. Щоб геш-функція вважалася надійним криптографічним інструментом, вона повинна відповідати низці ключових вимог, які забезпечують її стійкість до різноманітних атак [7].

Для того щоб геш-функцію можна використовуватися в криптографічних цілях, вона повинна відповідати низці важливих вимог. Основними з них є стійкість до першого образу, стійкість до другого образу та колізійна стійкість. Ці властивості забезпечують надійність збереження та перевірки даних, а також унеможливлюють їх підробку або підміну [1], [2].

Колізійна стійкість означає, що має бути майже неможливо знайти дві різні вхідні послідовності x і x', для яких $h(x) = h(x')$. Якщо цю властивість зламати, можливо створити шкідливе повідомлення з тим самим гешем, що й справжнє, — наприклад, підробити цифровий підпис або підмінити фінансову транзакцію. Колізійна стійкість є найскладнішою з трьох властивостей, адже вона охоплює одночасно і перший, і другий образи [2], [3].

Сучасні криптографічні алгоритми гешування, такі як SHA-2 та SHA-3, були розроблені з урахуванням усіх трьох вищезазначених властивостей і наразі вважаються надійними для використання в системах інформаційної безпеки [3], [4].

Ця властивість має ключове значення в застосуваннях, пов’язаних із перевіркою цілісності даних, цифровими підписами та блокчейн-технологіями. Важливо зазначити, що сильна стійкість до колізій також забезпечує стійкість до другого образу, тобто підвищує загальний рівень захисту даних [2].

Слабка (або одностороння) опірність до колізій означає, що навіть

маючи конкретне вхідне повідомлення x та його геш-значення $h(x)$, зловмисникові має бути вкрай складно знайти інше повідомлення y (де $x \neq y$), яке при гешуванні дає такий самий результат, тобто $h(y) = h(x)$ [1]. Ця властивість критично важлива для захисту даних від підміни: якщо хтось має доступ до легітимного документа й його гешу, він не повинен мати змоги створити інший документ, який матиме ідентичний геш і таким чином пройде перевірку автентичності.

Основою будь-якого гешування є спеціальна математична функція, яка обробляє два блоки даних фіксованого розміру з метою отримання геш-коду. Така функція є ключовим елементом алгоритму гешування, однак сам алгоритм — це складніший процес, що включає кілька повторюваних етапів (раундів), подібних до циклів у блокових шифрах [1].

На кожному раунді обробляється поєднання нового блоку даних із результатом попереднього обчислення. Таким чином, вихід одного раунду використовується як вхід для наступного, забезпечуючи послідовну трансформацію всієї вхідної інформації. Цей принцип багаторазового впливу кожного байта даних на результат називається ефектом лавини. Він полягає в тому, що навіть незначна зміна у вхідному повідомленні (наприклад, один біт) призводить до суттєво відмінного результату гешування, що унеможливлює передбачення структури початкових даних за отриманим гешем [1].

Варто чітко розмежовувати поняття геш-функції та алгоритму гешування. Геш-функція — це окрема операція, що виконує обчислення геш-коду на основі вхідних блоків фіксованої довжини. Натомість алгоритм гешування — це повна процедура, яка описує, як обробляється увесь обсяг повідомлення: від розбиття даних на блоки до послідовної взаємодії між результатами кожного раунду [1].

Розмір блоків, з якими працюють геш-функції, залежить від конкретного алгоритму. Як правило, він коливається в межах від 128 до 512 біт(рис. 1.1). Наприклад, в алгоритмі SHA-256 використовується блок 512

біт, тоді як для SHA-1 — 512 біт, а для SHA-3 -1088 або 1152 біти, залежно від конфігурації [2].

довжина останнього блоку тексту n=512 біт

512 біт	...	512 біт	448 біт	64 біт

довжина останнього блоку тексту n=448 біт

512 біт	...	512 біт	448 біт	64 біт

довжина останнього блоку тексту n<448 біт

512 біт	...	512 біт			64 біт

довжина останнього блоку тексту $448 < n < 512$ біт

512 біт	...			448 біт	64 біт

Рисунок 1.1 - Структура розширеного повідомлення

Геш-функція повинна відповідати низці базових вимог, які забезпечують її надійність та ефективність у криптографічних застосуваннях. Загальний принцип її роботи полягає у перетворенні довільного за довжиною вхідного повідомлення *M* у вихідне значення фіксованої довжини *h*, тобто [3]:

$$h = H(M)$$

де *H* — це геш-функція, *M* — початкове повідомлення, а *h* — результат її обчислення, так званий геш-код або дайджест повідомлення [4].

Це перетворення має забезпечувати односторонність, тобто бути обчислювально незворотним: за значенням *h* має бути практично неможливо визначити оригінальне повідомлення *M*. Крім того, функція повинна демонструвати лавинний ефект — навіть мінімальні зміни у *M* мають призводити до кардинально іншого значення *h*. Ще одна ключова вимога — унікальність результату, що означає низьку ймовірність того, що два різних повідомлення M_1 та M_2 матимуть однаковий геш-код: $H(M_1) = H(M_2)$ [3].

Ці властивості — незворотність, лавинність, колізійна стійкість — є базовими для забезпечення надійності цифрових підписів, контролю цілісності даних, а також для роботи криптографічних протоколів у цілому.

Для забезпечення надійності в інформаційно-комунікаційних системах

геш-функція повинна відповідати певним фундаментальним вимогам, які роблять її придатною до використання в криптографічних механізмах. Зокрема, функція гешування H повинна мати такі властивості [4]:

1. Бути застосовної до блоку даних будь-якої довжини.
2. Давати на виході значення фіксованої довжини.
3. Значення H(x) повинне обчислюватися відносно легкодля будь-якого заданого x, а алгоритм обчисленняповинний бути практичним з погляду як апаратної, так іпрограмної реалізації
4. Односторонність. Для заданого значення гешу h має бути обчислювально неможливо знайти таке вхідне повідомлення x, для якого H(x) = h. Ця властивість гарантує захист від зловмисників, які, маючи лише геш-значення, намагаються підібрати відповідне повідомлення(рис1.2).

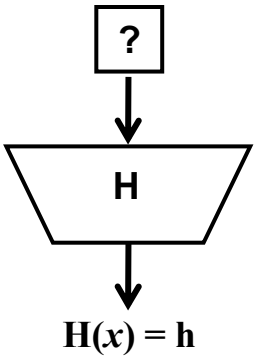


Рисунок 1.2 - Стійкість до пошуку прообразу

5. Стійкість до першого образу та стійкість до колізій першого роду — це ключові криптографічні властивості геш-функцій. Перша означає, що за заданим геш-значенням h(x) обчислювально складно знайти будь-яке повідомлення x, яке дало це значення, тобто неможливо відновити початкові дані лише з результату гешування. Друга — що для будь-якого заданого вхідного значення x складно підібрати інше повідомлення y, яке відрізняється від x, але має той самий геш H(x)=H(y). Іншими словами, геш-функція повинна унеможливлювати підбір альтернативного повідомлення з ідентичним гешем, що є критично важливим для захисту паролів, токенів доступу та іншої конфіденційної інформації (рис.1.3).

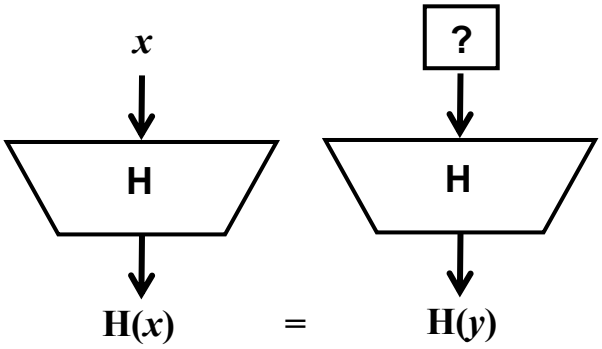


Рисунок 1.3. Сильна опірність колізіям

6. Стійкість до другого образу та стійкість до колізій другого роду (сильна опірність колізіям) — це важливі характеристики надійної геш-функції. Перша передбачає, що, маючи певне повідомлення x, обчислювально складно знайти інше повідомлення x′≠x, для якого h(x)=h(x′). Друга полягає в тому, що має бути надзвичайно важко знайти будь-яку пару різних повідомлень x і y, які мають однаковий геш: h(x)=h(y). Інакше кажучи, геш-функція повинна унеможливлювати як підміну конкретного повідомлення іншим, так і створення будь-яких двох різних повідомлень з однаковим геш-кодом. Це критично для забезпечення цілісності й достовірності даних(рис. 1.4).

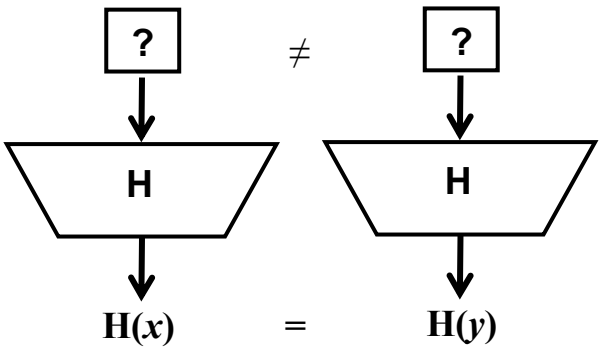


Рисунок 1.4. Сильна опірність колізіям

Під ефективністю геш-функції в криптографічному контексті розуміють її здатність виявляти навіть незначні зміни у вхідних даних. Мається на увазі ймовірність того, що при зміні (наприклад, помилковому спотворенні) хоча б одного біта у вхідному повідомленні вихідне геш залишиться незмінним [3].

Ця ймовірність позначається як P, і для якісної геш-функції вона

повинна бути надзвичайно низькою. Якщо довжина гешу становить n біт, то: $P=2^{-n}$ для випадкових (довільних) вхідних даних. Це означає, що ймовірність збігу гешу при наявності помилки в даних експоненціально зменшується зі збільшенням розрядності гешу. Для структурованих або форматованих даних, які мають певну внутрішню закономірність, ця ймовірність може бути ще меншою: $P<2^{-n}$ [3].

Таким чином, ефективна геш-функція гарантує лавиноподібний ефект: навіть мінімальна зміна у вхідному повідомленні призводить до кардинальної зміни геш-значення, що практично унеможливлює виявлення зіткнень або помилкових збігів гешів [3].

Приклад ітераційної функція гешування на рис 1.5

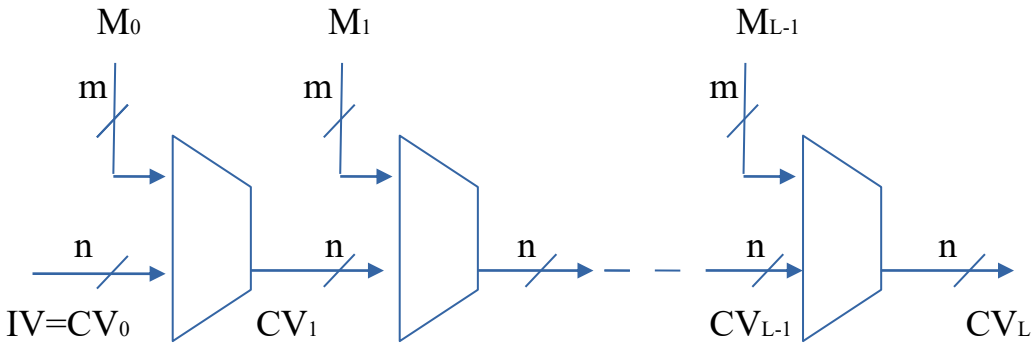


Рисунок 1.5. Ітераційна функція гешування

IV - початкове значення; CV^i – проміжне геш-значення; M^i - і-й блок, що вводиться; F - алгоритм ущільнення; L – кількість блоків, що вводяться; *n* – довжина геш-коду; *m* - довжина блоку, що вводиться.

1.2 Конструкція геш-функцій із використанням блокових шифрів

Існує низка методів побудови геш-функцій, і одним із найбільш відомих та ефективних є той, що використовує принципи блокового шифрування [5]

Приклад підходи до побудови геш-функцій на рис. 1.6

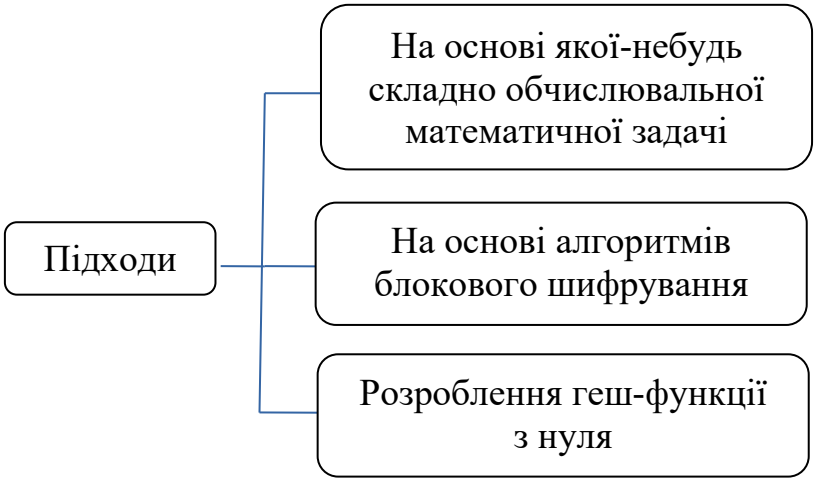


Рисунок 1.6. Підходи до побудови геш-функцій

Один із головних недоліків абсолютно криптостійкого шифрування, зокрема методу одноразового блокнота, полягає в необхідності використання ключа, довжина якого дорівнює або перевищує довжину самого повідомлення. Така вимога робить цей підхід малопридатним для широкого практичного застосування [4].

Щоб створити криптостійкий, але водночас практичний шифр, пропонуються такі концепції:

1. Режими шифрування. Для підвищення стійкості блочного шифрування часто використовують спеціальні режими, у яких під час обробки кожного наступного блоку враховується результат шифрування попереднього [3].

2. Багатораундова обробка. Ще одним способом посилення стійкості шифру є багаторазове (раундове) шифрування кожного блоку. Головна умова – використання нелінійного перетворення як шифрувальної функції. Це створює складні залежності між вхідними та вихідними даними, що ускладнює зворотний аналіз [5].

У блокових алгоритмах обробка даних здійснюється поетапно: вхідна послідовність розділяється на окремі фрагменти фіксованої довжини, які називають блоками. Найчастіше розмір таких блоків становить 64 біти (рис. 1.7).

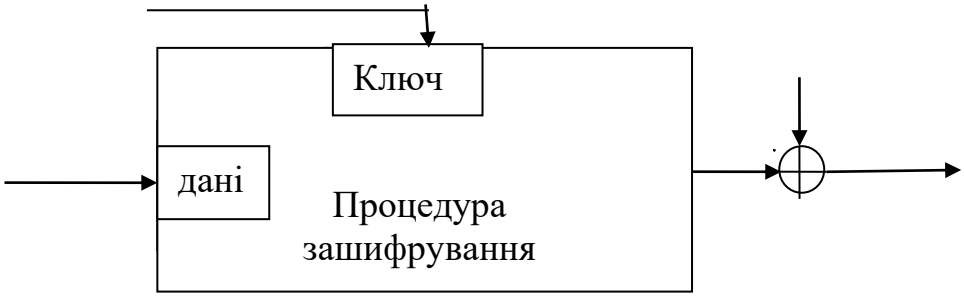


Рисунок 1.7. Узагальнена схема ітерації геш-функції

У тих випадках, коли довжина відкритого тексту не є кратною довжині блоку, застосовується процедура доповнення (padding). Це необхідно для того, щоб останній блок мав відповідний розмір для обробки. Доповнення може здійснюватися шляхом додавання нулів, спеціального службового байта або випадкової послідовності символів — усе залежить від обраного алгоритму та його специфікацій [3].

У блокових шифрах кожен блок проходить через послідовність математичних перетворень, які можуть повторюватися кілька разів (раундів). Саме багатораундова обробка одного і того ж блоку даних забезпечує підвищену стійкість до криптоаналізу. Кожне наступне перетворення ускладнює зворотну інтерпретацію даних, роблячи шифр більш захищеним від атак. Завдяки цьому криптографічна міцність системи зростає, а загальне перетворення виявляється значно надійнішим, ніж застосування лише одного раунду (рис. 1.8).

$$h_i = E_{h_{i-1}}(M_i) \oplus M_i$$

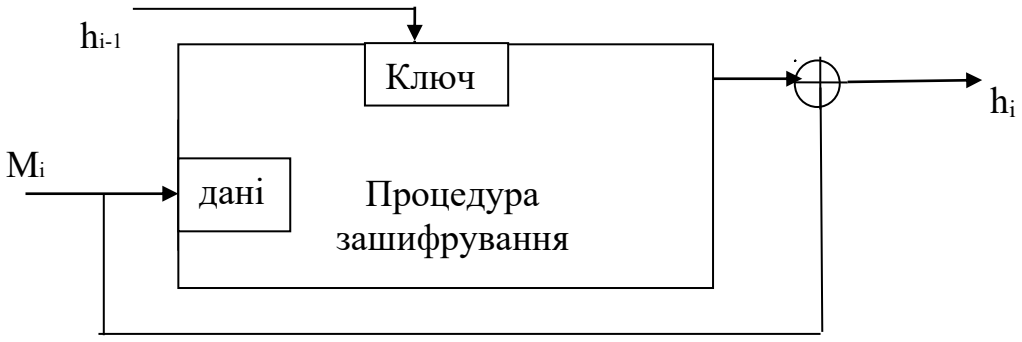


Рисунок 1.8. Схема ітерації безпечної геш-функції

Метою використання криптографічних перетворень у блокових алгоритмах є забезпечення максимальної взаємозалежності: кожен біт зашифрованого блоку повинен залежати від кожного біта як відкритого тексту, так і ключа шифрування. Такий підхід підвищує криптостійкість системи, ускладнюючи її злам [4].

Щоб забезпечити контекстну зв’язність блоків та захист від атак типу «вставка» або «видалення» окремих блоків, блоки шифротексту можуть використовуватись при обробці наступних блоків. Це дозволяє зберегти залежність між усіма частинами повідомлення. Такі способи організації шифрування визначаються як режими шифрування [4].

Режим шифрування — це спосіб застосування блокового шифру, за якого процес шифрування блоку повідомлення може включати попередньо зашифровані дані. Це необхідно для покращення криптографічного захисту.

1. На сьогодні блокові шифри є основним інструментом для захисту інформації в криптографії. Вони мають низку переваг:

2. Висока продуктивність при шифруванні та розшифруванні.

3. Гарантована стійкість, яка може бути формально підтверджена математично.

4. Гнучкість реалізації, включаючи можливість ефективного впровадження в програмне або апаратне забезпечення [4].

1.3 Побудова геш-функцій із використанням важко обчислюваних математичних проблем

Одним із надійних підходів до побудови геш-функцій є використання складнообчислювальних математичних задач, зокрема операцій модульного піднесення до степеня. Такий підхід забезпечує високу криптографічну стійкість завдяки важкості зворотного обчислення, що лежить в основі більшості алгоритмів асиметричної криптографії [6].

На першому етапі повідомлення розбивається на послідовність блоків

довжиною mmm бітів: $M_0,M_1,...,M_{L-1}$ [3].

Для обчислення геш-коду G використовується ітеративний процес, у якому проміжні значення H_i залежать від попередніх результатів. Початкове значення гешу задається як H_0, а кінцевий результат G визначається як: $G=H_{L-1}$

Відомі способи побудови:

Спосіб 1:

$$H_i = H_{i-1}^{M_{i-1}} \bmod p$$

У цьому варіанті попередній геш підноситься до степеня, рівного відповідному блоку повідомлення. Модуль p — велике просте число.

Спосіб 2:

$$H_i = M_{i-1}^{H_{i-1}} \bmod p$$

Навпаки, блок повідомлення виступає основою, а попередній геш — показником степеня.

$$H_i = g^{H_{i-1} \oplus M_{i-1}} \bmod p$$

Спосіб 3: У цьому способі застосовується примітивний елемент g у полі за модулем p, а також операція побітового XOR між попереднім гешем і блоком повідомлення.

1.4 Геш-функції, розроблені з нуля

Одним із базових методів реалізації геш-функції є застосування операції побітового виключного «АБО» (XOR) для поєднання всіх блоків вхідних даних. Такий підхід дозволяє швидко обчислити контрольну суму, яка може виступати у ролі простого геш-коду.

Геш-код обчислюється за наступною формулою:

$$C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im},$$

де: C_i — i-й біт сформованого геш-коду,

t — кількість n-бітних блоків вхідного повідомлення,

b_{ij} — i-й біт у j-му блоці,

$\oplus$ — побітова операція XOR.

Такий підхід є швидким у реалізації, але характеризується низькою криптографічною стійкістю, оскільки легко піддається атакам.

Удосконалений варіант простої геш-функції

З метою підвищення стійкості і ускладнення передбачення результату, базовий метод може бути розширений шляхом запровадження додаткових операцій. Один з удосконалених варіантів передбачає поетапну обробку блоків даних за таким алгоритмом:

1. Ініціалізація: проміжне значення геш-функції встановлюється як n-бітове нульове слово.
2. Обробка блоків: кожен n-бітовий блок обробляється по черзі:

проміжне значення геш-функції піддається циклічному зсуву вліво на один біт; результат XOR-у з поточним блоком обчислюється та зберігається як нове проміжне значення гешу.

Узагальнено, така процедура дозволяє отримати більш складний геш-код, який уже чутливіший до змін вхідних даних. Завдяки циклічному зсуву підвищується лавинний ефект — незначна зміна у вхідному повідомленні призводить до суттєвих змін у результаті.

Приклад удосконаленої проста функція гешування наведено на рис. 1.9

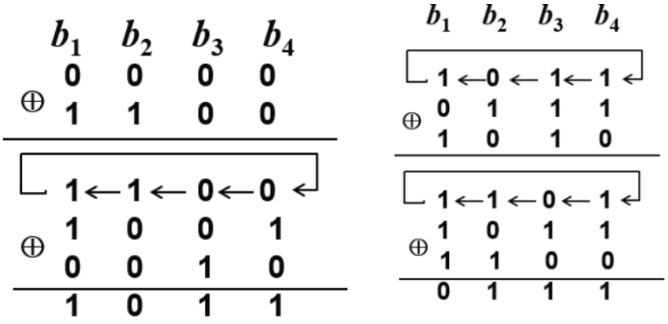


Рисунок 1.9. Удосконалена проста функція гешування

Однією з найвідоміших геш-функцій свого часу була MD5 (Message Digest 5), яка протягом тривалого періоду активно використовувалась у різних галузях інформаційної безпеки (рис. 1.5).

Завдяки високій швидкості та простоті реалізації, MD5 знайшла широке застосування у програмному забезпеченні, зокрема для контролю цілісності даних (рис. 1.6). Наприклад, при розповсюдженні програм або великих файлів через інтернет, сервери часто надають попередньо обчислений MD5-геш, який користувач може використати для перевірки правильності завантаженого файлу [14].

Це дозволяє виявити будь-які пошкодження або несанкціоновані зміни у даних, що можуть виникнути під час передавання або зберігання файлу.

Однак, незважаючи на свою популярність у минулому, MD5 вважається недостатньо надійною з точки зору криптостійкості через виявлені вразливості, пов’язані з можливістю знаходження колізій [14].

Посилання обробка 512-бітного блоку в алгоритмі MD-5 на рис 1.10

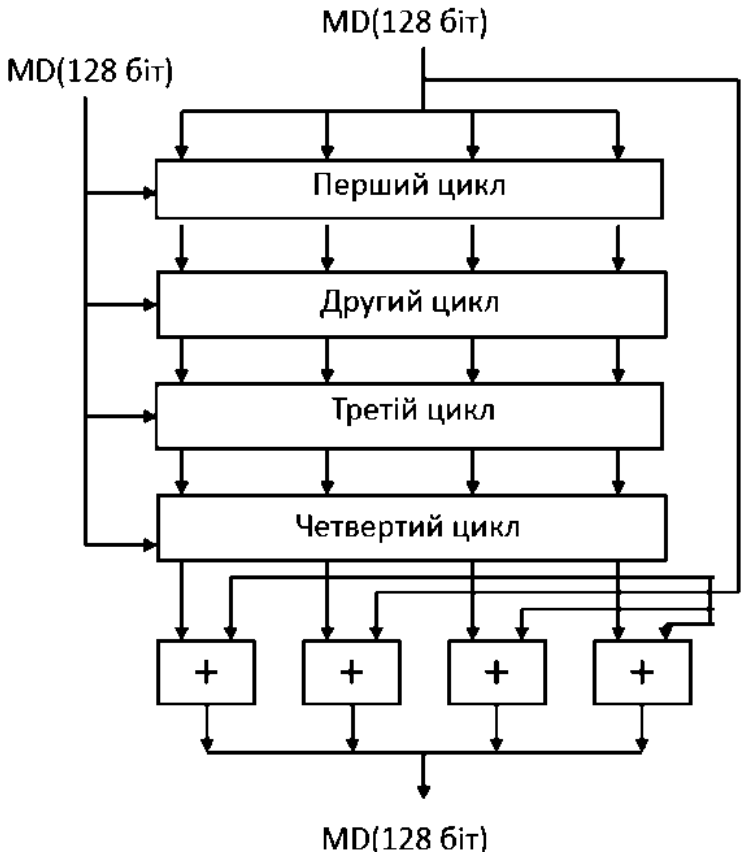


Рисунок 1.10 - Обробка 512-бітного блоку в алгоритмі MD-5

MD5 (Message Digest 5) — одна з найвідоміших геш-функцій, розроблена Р. Рівестом у 1991 році як покращення MD4. Алгоритм працює з блоками по 512 біт і формує геш-значення довжиною 128 біт. Його

ключовими перевагами тривалий час були висока швидкість та простота реалізації, що сприяло широкому впровадженню в різних програмних продуктах [5].

У зв’язку з цим використання MD5 у сфері безпеки сьогодні не рекомендується. Він поступився сучаснішим алгоритмам, таким як SHA-2 або SHA-3, які забезпечують вищу криптостійкість(рис. 1 .11).

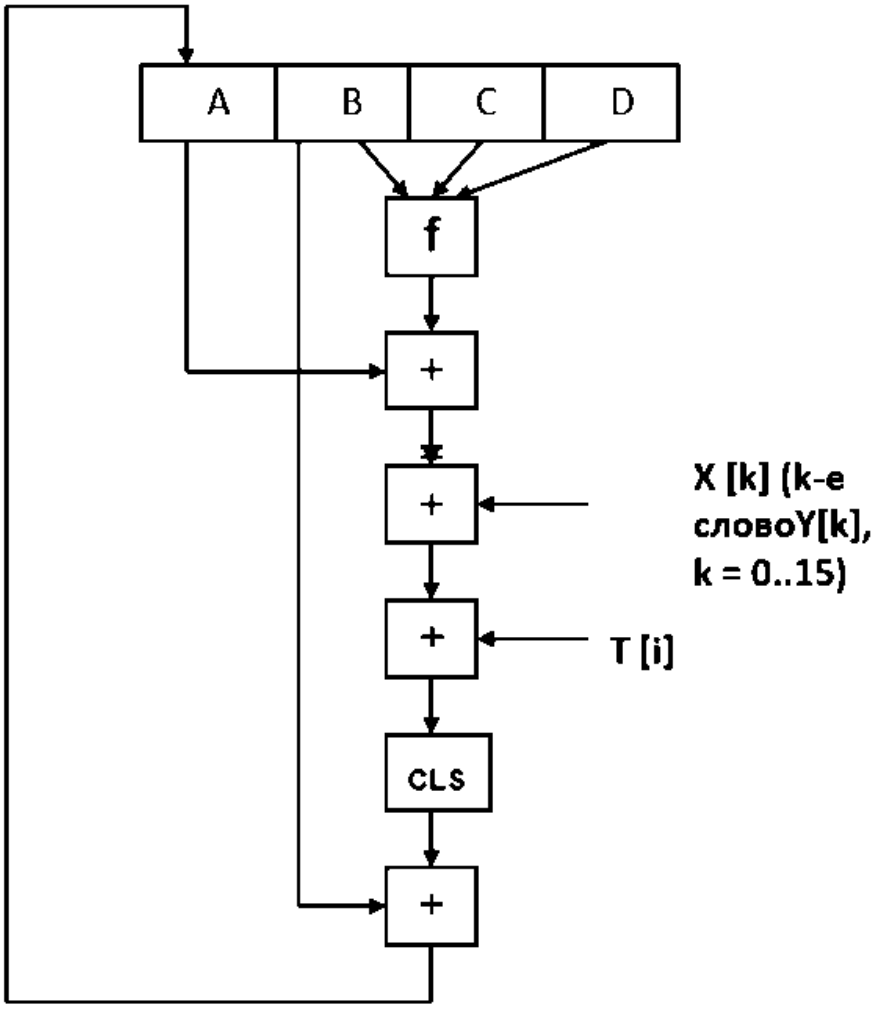


Рисунок 1.11. Логіка виконання окремого кроку MD-5

Дослідники з Китаю вперше продемонстрували ефективну атаку на геш-функцію MD5, згенерувавши колізії — різні повідомлення, що мають однакове геш-значення. Для цього знадобилось менше години роботи комп’ютерного кластера. Надалі вдалося розробити ще ефективніші методи зламу, що зробило MD5 непридатним для використання у сфері безпеки,

зокрема для цифрових підписів та сертифікатів. Попри це, MD5 ще використовується в деяких несекретних сценаріях, наприклад, для перевірки цілісності даних, де криптостійкість не є критично важливою [14].

Для вирішення проблем, пов’язаних із застарілими алгоритмами, такими як MD5, були розроблені безпечніші альтернативи, серед яких — сімейство RIPEMD. Його назва розшифровується як RACE Integrity Primitives Evaluation Message Digest — це результат роботи європейської дослідницької групи RACE у 1990-х роках [13].

Найбільш відомими представниками цього сімейства є:

1. RIPEMD-128 — початкова версія з довжиною гешу 128 біт, яка забезпечує помірний рівень безпеки, але вже не відповідає сучасним вимогам(рис. 1.12).

2. RIPEMD-160 — вдосконалений варіант, розроблений як більш безпечна альтернатива MD4/MD5. Його структура містить дві паралельні лінії обробки, що підвищує стійкість до атак. Саме ця версія сьогодні вважається найбільш надійною серед RIPEMD-функцій(рис 1.13).

3. Також існують RIPEMD-256 та RIPEMD-320, які забезпечують більшу довжину гешу, зменшуючи ймовірність випадкових колізій, однак не гарантують вищої криптостійкості у порівнянні з RIPEMD-160 [13].

Хоча RIPEMD-160 не має широкого розповсюдження на рівні з SHA-2 чи SHA-3, він усе ще застосовується у деяких криптографічних системах, зокрема в блокчейн-технологіях, таких як Bitcoin, де використовується в парі з SHA-256 для створення адрес [13].

Загалом, RIPEMD-160 залишається безпечною геш-функцією, але в нових системах зазвичай надають перевагу більш сучасним стандартам, затвердженим NIST, зокрема SHA-2 та SHA-3.

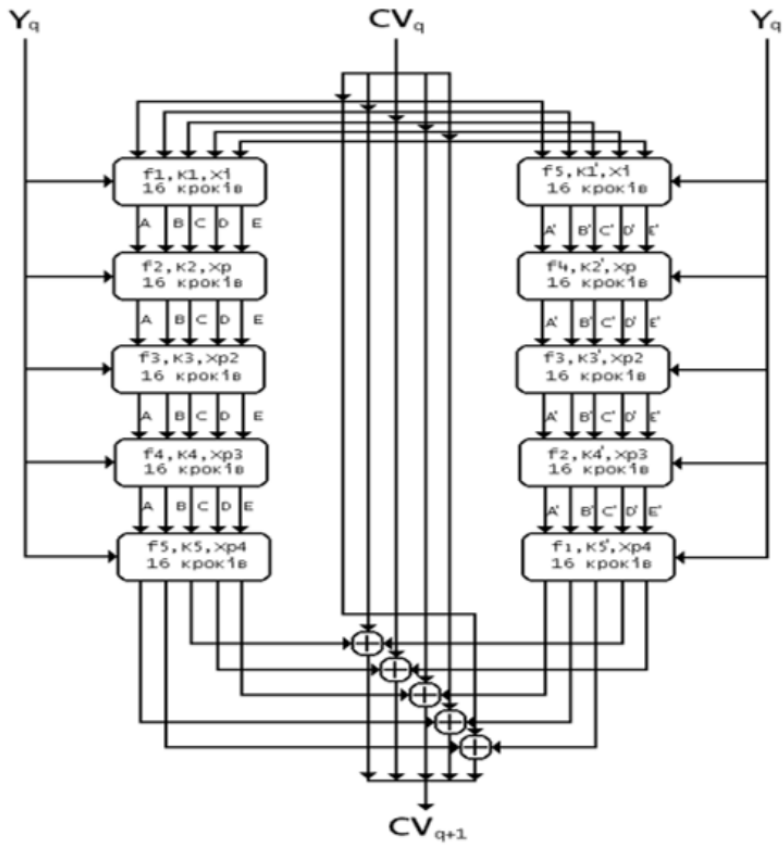


Рисунок 1.12. Обробка 512-бітного блоку алгоритму RIPEMD-160

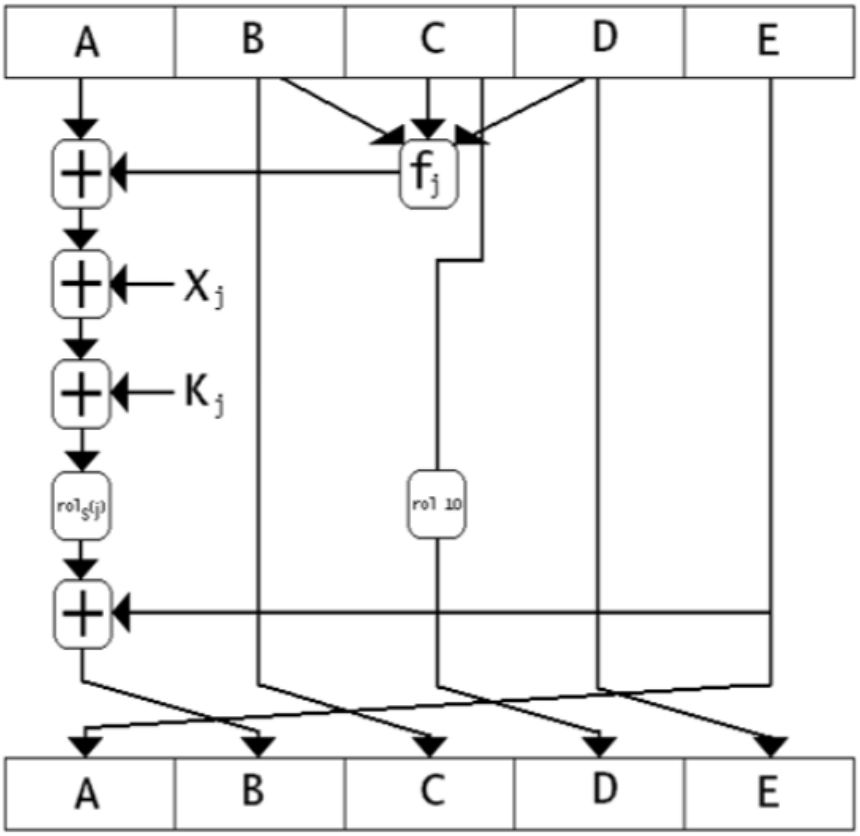


Рисунок 1.13. Логіка виконання окремого кроку RIPEMD-160

Сімейство SHA (Secure Hash Algorithm) включає кілька криптографічних геш-функцій, розроблених Національним інститутом стандартів і технологій США (NIST). До цього сімейства належать SHA-0, SHA-1, SHA-2 та SHA-3. Попри те, що всі алгоритми входять до однієї групи, вони відрізняються за структурною побудовою та рівнем криптостійкості [15].

Перший варіант, SHA-0, було представлено в 1993 році як 160-бітна геш-функція, однак незабаром після появи виявились її криптографічні вразливості, що зумовило обмежене використання. У 1995 році її було замінено на SHA-1, який виправляв деякі структурні недоліки попередника.

SHA-1 став популярним у різноманітних протоколах безпеки, таких як SSL, TLS, IPsec та цифрові підписи. Проте вже у 2005 році дослідники змогли підібрати колізії для SHA-1 протягом практичного часу, що поставило під сумнів його подальше використання. У 2017 році компанія Google спільно з CWI Amsterdam офіційно продемонструвала перший публічний приклад колізії для SHA-1, що остаточно підтвердило його вразливість [15].

У відповідь на виявлені недоліки SHA-1 було запропоновано нову серію алгоритмів — SHA-2. Ця серія включає кілька варіантів з різною довжиною гешу: SHA-224, SHA-256, SHA-384 та SHA-512. Алгоритми SHA-2 суттєво покращено як за структурою, так і за рівнем безпеки. Вони залишаються надійними навіть сьогодні, і на момент написання не було виявлено ефективних крипто аналітичних атак, які б загрожували їхній стійкості [13 15 18].

Попри високу безпеку SHA-2, для подальшого розвитку було ініційовано відкритий конкурс NIST для створення нового покоління геш-функцій. У 2012 році на його основі було обрано SHA-3, побудований на абсолютно іншій структурі — так званій губкоподібній (sponge) конструкції. SHA-3 не є прямим замінником SHA-2, але використовується як додаткова альтернатива, особливо в умовах, що потребують

максимальної безпеки або іншої архітектури гешування.

На сьогодні SHA-2 та SHA-3 вважаються основними стандартами для захищених систем. SHA-3 вирізняється продуктивністю при обробці великих даних і гнучкістю, зокрема можливістю задавати довжину гешу, що робить його придатним для хмарних технологій, крипто валют і ресурс залежних пристроїв [13, 15].

Однією з головних переваг SHA-3 є висока стійкість до різних типів атак, включно з диференційним аналізом і атаками на довжину повідомлення. Завдяки основі в алгоритмі Keccak, SHA-3 може застосовуватись не лише для гешування, а й для створення MAC-кодів та генераторів випадкових чисел, що забезпечує його універсальність у криптографії [13, 15].

У сучасних умовах зростання обчислювальних потужностей і появи квантових комп’ютерів усе більше уваги приділяється дослідженням квантостійких криптографічних алгоритмів. Хоча SHA-3 наразі вважається безпечним і витривалим до класичних атак, наукове співтовариство продовжує аналіз можливих ризиків у контексті квантових обчислень. Це стимулює створення нових моделей геш-функцій, які зможуть відповідати майбутнім вимогам до інформаційної безпеки, зокрема — з урахуванням постквантової криптографії(рис. 1.14).

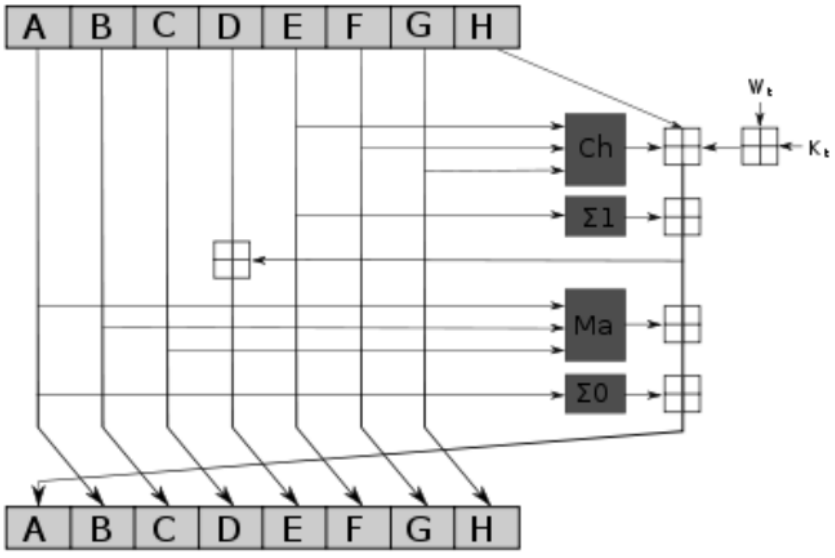


Рисунок 1.14 Схема однієї ітерації алгоритмів SHA-2

Попри високу стійкість SHA-2, його архітектура має спільні риси з попереднім стандартом SHA-1, що викликало занепокоєння у криптографічної спільноти щодо потенційних вразливостей у майбутньому. Зважаючи на це, Національний інститут стандартів і технологій США (NIST) ініціював відкритий конкурс на розробку нової криптографічної геш-функції, яка б відповідала сучасним вимогам до безпеки та продуктивності [13].

Висновок до розділу:

Аналіз наявних методів гешування даних свідчить, що всі вони базуються на ітераційних процедурах. Основним їхнім недоліком є можливість компрометації геш-функції шляхом поступового відновлення попередніх ітерацій, починаючи з фінального геш-значення. Це зумовлює необхідність забезпечення високої стійкості кожного окремого кроку, що ускладнює перетворення на кожному етапі та, відповідно, негативно впливає на швидкість гешування. Тому проблема підвищення ефективності процесу гешування є надзвичайно актуальною.

2.РОЗРОБКА МЕТОДУ ГЕШУВАННЯ

У межах цієї бакалаврської кваліфікаційної роботи було поставлено задачу розробити власний метод гешування, що перетворює повідомлення довільної довжини у фіксований вихід довжиною 256 біт. Такий алгоритм має відповідати базовим вимогам криптографії, зокрема забезпечувати незворотність, стійкість до колізій, лавиноподібну реакцію на зміну вхідних даних, а також бути достатньо ефективним для застосування в реальних програмних системах. Окрему увагу було приділено простоті реалізації алгоритму без втрати його криптографічної надійності.

В основі розробленого методу лежить класична схема Меркла–Дамґарда, яка є перевіреною часом конструкцією для побудови геш-функцій. Головна її перевага — це здатність обробляти повідомлення довільної довжини шляхом розбиття його на блоки фіксованого розміру і поетапного обчислення гешу. Запропонований метод модифікує цю схему, вводячи нові правила обробки блоків та механізми, що посилюють складність аналізу оберненості функції. Однією з особливостей стало використання множення на попереднє значення гешу з подальшим розділенням результату на частини та обробкою цих частин модульною арифметикою.

Такий підхід поєднує детермінованість та складну залежність між блоками вхідного повідомлення, забезпечуючи тим самим бажану властивість лавиноподібності.

2.1 Метод гешування

Підготовка повідомлення.

На першому етапі вхідне повідомлення розбивається на блоки по 256 біт. Якщо довжина повідомлення не кратна 256, застосовується паддінг — доповнення повідомлення до потрібної довжини спеціальною послідовністю бітів, що включає одиницю та необхідну кількість нулів. Цей прийом

дозволяє уникнути неоднозначностей при обробці повідомлень різної довжини.

Початкове значення

Початкове геш-значення (h_0) встановлюється рівним нулю. Це стандартна практика, яка дозволяє забезпечити відтворюваність результатів і простоту перевірки алгоритму.

Основний цикл обробки

Для кожного блоку m_i виконується послідовність дій:

1. Множення блоку даних на попереднє геш-значення;

$$h_i^* = m_i \times h_{i-1}$$

2. Результат h_i^* розділяється на дві рівні частини L і R (по 256 біт);

3. Обчислюється геш-значення:

$$h_i = (L + R) \bmod 2^{256}.$$

4. Якщо $h_i = 0$, то

$$h_i = h_i + (101010...10).$$

Такий підхід гарантує вплив кожного блоку на результат, а також створює внутрішню складну залежність між даними. Кожен блок не лише самостійно змінює поточне геш-значення, але й взаємодіє з попереднім контекстом, що унеможливлює ізольований вплив окремих частин повідомлення. Навіть незначна зміна в одному блоці спричиняє зміну всього ланцюга обчислень, оскільки всі наступні значення залежать від попередніх через операцію множення та подальше додавання частин. У результаті формується ефект лавини, при якому зміна одного біта у вхідному повідомленні призводить до значних змін у всьому геш-коді. Це робить алгоритм стійким до аналізу структури вхідних даних і ускладнює спроби знайти колізії або підібрати повідомлення з однаковим гешем. Таким чином, забезпечується не лише математична складність, а й криптографічна надійність.

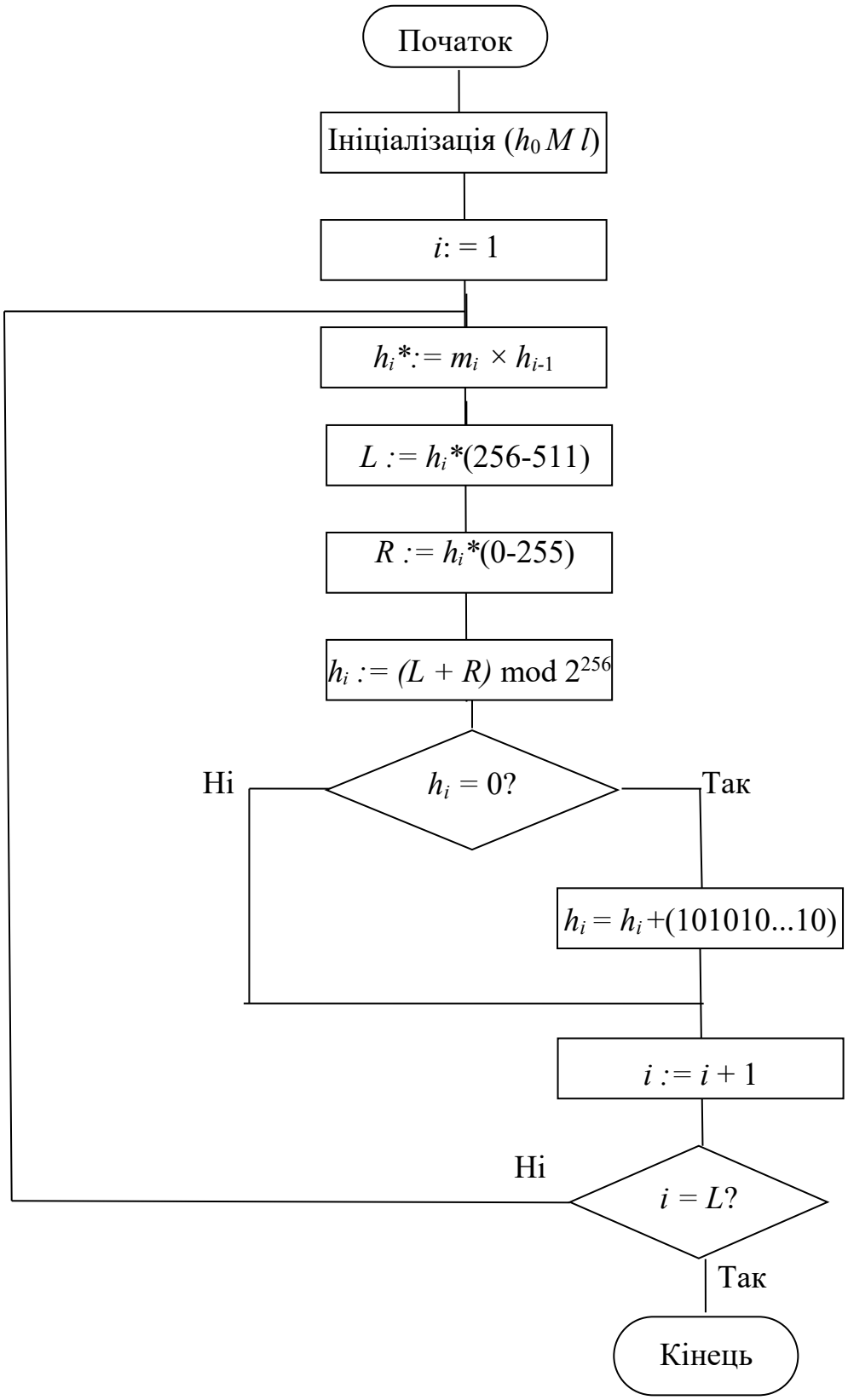


Рисунок 2.1 — Блок-схема алгоритму

Ця блок-схема описує алгоритм обчислення геш-функції, яка працює за принципом ітеративної обробки повідомлення з використанням попереднього гешу, модульних операцій і корекції. Нижче — детальний опис кожного етапу:

Початок

Ініціалізація (h_0, M, l)

h_0 — початкове геш-значення.(256 біт)

M — повідомлення, що підлягає гешуванню

l — кількість блоків повідомлення довжиною 256 біт

$i := 1$

Цикл (ітерація)

1. $h_i := m_i \times h_{i-1}$

m_i — і-й блок повідомлення.

Виконується множення блоку повідомлення на попереднє геш-значення.

Результат має 512 біт.

2. Розділення h_i на частини L і R

$L := h_i$* (біти 256–511) — старші 256 біт.

$R := h_i$* (біти 0–255) — молодші 256 біт.

Це розбиття дає два 256-бітних числа для подальшої обробки.

3. $h_i := (L + R) \bmod 2^{256}$

Додаються L і R за модулем 2^{256} (тобто зберігаються тільки 256 молодших бітів).

4. Перевірка: $h_i = 0$?

Якщо результат дорівнює 0 — це вважається некоректним або небажаним значенням гешу.

5. Корекція: $h_i := h_i + (101010\ldots10)$

Якщо $h_i = 0$?, до результату додається фіксована бітова маска (наприклад, чергування 1 і 0: 101010...10).

Мета — уникнути нульового гешу, який може порушити

криптографічну стійкість або викликати колізії.

6. Збільшення лічильника: $i := i + 1$

Переходимо до наступного блоку повідомлення.

7. Умова завершення: $i = L$?

Якщо всі блоки повідомлення оброблено ($i = L$?), алгоритм завершується.

Кінець

2.2 Аналіз властивостей алгоритму

Розглянемо на прикладі процес гешування.

Початкові дані:

$$h_0 = 10101010 \; m = 10011101$$

Крок 1: Обчислення добутку h_1*

$$10011101 \times 10101010 = 01101000\ 01000010$$

Крок 2: Поділ на частини

$$L = 01101000$$
$$R = 01000010$$

Крок 3: Додавання за модулем

$$h_1 = (L + R) \bmod 2^8 = (01101000 + 01000010) \bmod 2^8 = 10101010$$

Крок 4: Зміна одного біта в m - m = 10010101

Повторне обчислення:

$$h_1^* = m \times h_0 = 10010101 \times 10101010 = 01100010\ 11110010$$

$$L = 01100010$$

$$R = 11110010$$

$$h_1 = (L + R) \bmod 2^8 = (01100010 + 11110010) \bmod 2^8 = 01010100$$

$$h_1 = 01010100$$

$$10101010 \oplus 01010100 = 11111110$$

Зміна одного біта повідомлення m привела до зміни 7 біт геш-значення, що вказує на гарний лавиноподібний ефект.

Початкові дані:

$$h_0 = 10101010,\ m = 10011101$$

Крок 1: Обчислення добутку h_1*

$$h_1* = m \times h_0 = 10011101 \times 10101010 = 01101000\ 01000010$$

Крок 2: Поділ на частини

$$L = 01101000$$

$$R = 01000010$$

Крок 3: Додавання за модулем

$$h_1 = (L + R) \bmod 2^8 = 10101010$$

Крок 4: Зміна одного біта в $m = 10011100$

Повторне обчислення:

$$h_1* = m \times h_0 = 10011100 \times 10101010 = 01100111\ 10011000$$

$$L = 01100111$$

$$R = 10011000$$

$$h_1 = (L + R) \bmod 2^8 = 11111111$$

$$10101010 \oplus 11111111 = 01010101$$

Зміна одного біта повідомлення m привела до зміни 4 біт геш-значення, що вказує на гарний лавиноподібний ефект.

Початкові дані:

$$h_0 = 11001100,\ m = 10011101$$

Крок 1: Обчислення добутку h_1*

$$h_1^* = m \times h_0 = 10011101 \times 11001100 = 01111101\ 00011100$$

Крок 2: Поділ на частини

$$L = 01111101$$

$$R = 00011100$$

Крок 3: Додавання за модулем

$$h_1 = (L + R) \bmod 2^8 = 10011001$$

Крок 4: Зміна одного біта в $m = 10111101$

Повторне обчислення:

$$h_1^* = m \times h_0 = 10111101 \times 11001100 = 10010110\ 10011100$$

$$L = 10010110$$

$$R = 10011100$$

$$h_1 = (L + R) \bmod 2^8 = 00110010$$

$10011001 \oplus 00110010 = 10101011$

Зміна одного біта повідомлення m привела до зміни 5 біт геш-значення, що вказує на гарний лавиноподібний ефект.

2.3 Оцінка складності алгоритму гешування

У цьому підрозділі виконується детальний аналіз складності та ефективності розробленого алгоритму гешування. Зокрема розглянуто кількість базових операцій, необхідних для обробки одного блоку даних, а також проведено порівняння з найпоширенішими сучасними алгоритмами: MD5, SHA-1, SHA-256, RIPEMD-160 та SHA-3.

Запропонований алгоритм базується на модифікованій конструкції Меркла–Дамґарда та використовує виключно арифметичні операції: множення та додавання за модулем. Це дозволяє уникнути використання складних побітових перетворень і забезпечити простоту реалізації на більшості платформ, включаючи мікроконтролери та вбудовані системи.

Блок вхідного повідомлення і проміжне геш-значення мають довжину 256 біт. Обробка таких даних за допомогою 64-розрядного процесора передбачає розбиття їх на 4 частини по 64 біта.

Для кожного блоку вхідного повідомлення виконується послідовність таких дій:

1. Множення блоку m_i на попереднє геш-значення h_{i-1}.

Нехай блок повідомлення подано в вигляді таких 4 складових $a_4\ a_3\ a_2\ a_1$, а проміжне геш-значення в вигляді $b_4\ b_3\ b_2\ b_1$.

Тоді множення виконується таким чином:

$$
\begin{array}{rl}
 & a_4\ a_3\ a_2\ a_1 \\
\times & \\
 & \underline{b_4\ b_3\ b_2\ b_1} \\
 & a_1 b_1 \\
+ & \\
 & a_2 b_1
\end{array}
$$

$$
\begin{array}{r}
+ \\
a_3b_1 \\
+ \\
a_4b_1 \\
\hline
c_5c_4c_3c_2c_1 \\
a_1b_2 \\
+ \\
a_2b_2 \\
+ \\
a_3b_2 \\
+ \\
a_4b_2 \\
\hline
d_5d_4d_3d_2d_1 \\
a_1b_3 \\
+ \\
a_2b_3 \\
+ \\
a_3b_3 \\
+ \\
a_4b_3 \\
\hline
e_5e_4e_3e_2e_1 \\
a_1b_4 \\
+ \\
a_2b_4 \\
+ \\
a_3b_4 \\
+ \\
a_4b_4 \\
\hline
f_5f_4f_3f_2f_1
\end{array}
$$

Множення на b_1, b_2, b_3 і b_4 потребує по 4 операції множення кожне. Таким чином потрібно виконати 16 операцій множення.

Для отримання проміжного результату $c_5c_4c_3c_2c_1$ потрібно виконати 3 операції. Аналогічно для отримання проміжних результатів $d_5d_4d_3d_2d_1$, $e_5e_4e_3e_2e_1$ і $f_5f_4f_3f_2f_1$ потрібно виконати по 5 операції додавання.

Додавання проміжних результатів

$$
\begin{array}{r}
c_5c_4c_3c_2c_1 \\
+ \\
d_5d_4d_3d_2d_1
\end{array}
$$

потребує 5 додавань.

Додавання проміжних результатів

$$\begin{array}{r} e_5e_4e_3e_2e_1 \\ + \\ f_5f_4f_3f_2f_1 \end{array}$$

також потребує 5 додавань.

І нарешті остаточний результат множення отримується виконанням 5 операцій додавання.

Таким чином множення блоку m_i на попереднє геш-значення h_{i-1} потребує виконання 16 операцій множення і 25 операцій додавання, тобто разом 41 операція.

2. Поділ результату h_i* на частини L і R (по 256 біт):

h_i* має розрядність 512 біт, тобто складається з 8 частин по 64 біти. Для отримання кожної групи потрібно виконати 2 операції.
Таким чином поділ результату h_i* на частини L і R потребує виконання 16 операцій.

3. Обчислення $h_i = (L + R) \bmod 2^{256}$:

L та R — це дві 256-бітні частини, кожна з яких складається з 4 блоків по 64 біти.

Щоб обчислити суму $L + R$, потрібно виконати додавання відповідних 64-бітних блоків між L і R:

$$L = (L_3, L_2, L_1, L_0)$$
$$R = (R_3, R_2, R_1, R_0)$$

Обчислення:

$$h_0 = L_0 + R_0$$
$$h_1 = L_1 + R_1$$
$$h_2 = L_2 + R_2$$
$$h_3 = L_3 + R_3$$

Кожна операція — це 64-бітне додавання з урахуванням переносу, що може потребувати обробки переносу на наступний розряд. Проте для загальної оцінки кількості арифметичних операцій вважаємо, що для кожної пари блоків виконується 1 додавання.

Таким чином, для обчислення $h_i = (L + R) \bmod 2^{256}$ необхідно виконати 4 операції додавання.

4. Перевірка на нульове значення та корекція:

Після обчислення $h_i = (L + R) \bmod 2^{256}$ потрібно переконатися, що результат не дорівнює нулю, оскільки значення гешу $h_i = 0$ вважається некоректним або небажаним.

Геш h_i складається з 4 блоків по 64 біти:

$$h_i = (h_3, h_2, h_1, h_0)$$

Для перевірки нульового значення необхідно порівняти всі 4 блоки з нулем. Це потребує 4 операції порівняння.

Якщо усі блоки дорівнюють нулю (тобто $h_i = 0$), то виконується корекція шляхом додавання ненульового коду довжини 256 біт.

Оскільки цей код також має 256 біт (4 × 64-бітні блоки), додавання виконується по 64 біти — аналогічно до звичайного 256-бітного додавання. Це потребує ще 4 операції додавання.

Усього:

4 операції порівняння для перевірки нульового значення

У разі виявлення нуля — 4 операції додавання для корекції

Таким чином, перевірка та корекція потребує 4 операції порівняння, а в разі необхідності — ще 4 операції додавання.

У середньому для кожного блоку виконується 69 арифметичні операції, що є дуже невисоким показником порівняно з іншими алгоритмами.

Обчислимо продуктивність:

Розмір блоку: 256 біт = 32 байти

Операцій на біт: 69 / 256 = 0.27

Операцій на байт: 69 / 32 = 2.2

Таблиця 2.1 – Порівняльні оцінки криптографічних геш-функцій

Алгоритм	Розрядність блоку (біт)	Розрядність геш-значення (біт)	Операцій на блок	Операцій на байт	Особливості
Розроблений	256	256	69	2.2	Множення + додавання за модулем
MD5 [14]	512	128	744	11.62	Швидкий, але небезпечний
SHA-1 [19] [5]	512	160	900	14.06	Застарілий, зламаний
SHA-256 [15]	512	256	1100	17.18	Сучасний стандарт NIST
RIPEMD-160 [13]	512	160	1600	25	Середня продуктивність
SHA-3 [15]	1088	256	123	13	Безпечний, інноваційна структура Keccak

Як бачимо з таблиці, запропонований алгоритм демонструє кращі показники швидкості на байт порівняно з SHA-1, SHA-2 та SHA-3. Його перевагою є простота обчислень, що не вимагають логічних операцій, побітових зсувів або використання S-блоків. Це робить його придатним для реалізації у мікропроцесорних системах з обмеженою обчислювальною потужністю.

Окрім швидкодії, важливим показником є лавиноподібний ефект. Проведене тестування показало, що зміна одного біта вхідного повідомлення

змінює щонайменше 4–7 біт у результаті гешування. Такий рівень лавинності цілком прийнятний для багатьох практичних застосувань, особливо у внутрішніх контрольних сумах, фільтрах даних або в захищеному зберіганні паролів.

Водночас слід зазначити, що алгоритм не має формально доведеної стійкості до всіх типів атак, тому не може прямо конкурувати з SHA-3 у контексті високо захищених застосувань. Проте, з огляду на ефективність, може бути використаний як допоміжний алгоритм, наприклад, у попередній обробці, гешуванні чи скороченому підписуванні структур даних.

Таким чином, алгоритм забезпечує збалансовану комбінацію швидкодії та простоти реалізації, демонструє адекватну реакцію на зміну вхідних даних, і може бути рекомендованим для застосування у практичних задачах інформаційної безпеки, що не потребують максимального рівня криптостійкості.

3. РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ДЛЯ ГЕШУВАННЯ

3.1 Обґрунтування вибору мови програмування

Для реалізації програмного засобу, що виконує операції гешування даних за модульною схемою, було обрано мову програмування Python, оскільки вона забезпечує низку переваг, критично важливих у контексті криптографічних досліджень і розробки.

Однією з ключових причин такого вибору є високий рівень читабельності та зрозумілості коду. Python вирізняється синтаксисом, що наближений до природної мови, і дозволяє легко описувати складні алгоритмічні структури без перевантаження деталями низького рівня. Це спрощує процес супроводу та верифікації коду, що особливо актуально в криптографії, де навіть незначна помилка може призвести до серйозних порушень безпеки. Крім того, така якість коду важлива в умовах навчального або дослідницького середовища, коли пріоритет надається розумінню логіки роботи алгоритму.

Ще однією важливою перевагою Python є підтримка чисел довільної довжини на рівні мови. Це особливо важливо при реалізації алгоритмів, які потребують опрацювання великих цілих чисел або оперують із блоками розміром у сотні бітів. На відміну від мов із жорсткою типізацією, таких як C чи Java, Python автоматично обробляє арифметику великих чисел без додаткових бібліотек або розширень. Це дозволяє суттєво зменшити ризики переповнення та похибок при виконанні модульних обчислень, що є базовими для геш-функцій.

Python також надає високу гнучкість при тестуванні та прототипуванні. Завдяки простоті створення і запуску нових версій алгоритму, розробник може швидко змінювати структуру геш-функції, експериментувати з параметрами (розмір блоків, функція змішування, механізми доповнення) та вивчати їх вплив на такі характеристики як лавиноподібність, стійкість до колізій та ефективність. Такий підхід дозволяє сконцентруватися саме на ідеї

алгоритму, а не на технічних деталях реалізації.

Окремо варто згадати вбудовані інструменти Python для аналізу безпеки та налагодження. Мова має потужний стек бібліотек для логування, профілювання, тестування, що значно полегшує виявлення вразливостей або недоліків у логіці гешування. У поєднанні з відкритим кодом програми це створює сприятливі умови для незалежної перевірки та сертифікації програмного продукту.

Незважаючи на те, що Python не є мовою з максимальною швидкістю виконання, її модульна структура дозволяє за потреби винести критичні для продуктивності частини в C/C++ або використовувати такі рішення, як Cython, NumPy або Numba для прискорення обчислень. Це робить Python не лише зручним для початкової реалізації, але й потенційно масштабованим для продуктивного використання.

Варто відзначити і платформну незалежність: написана на Python програма може бути запущена на будь-якій сучасній операційній системі без необхідності значних змін у коді. Це забезпечує універсальність і зручність розгортання геш-засобу у середовищах з різною архітектурою.

Також Python має розвинену екосистему та активну спільноту, що сприяє швидкому пошуку рішень, бібліотек або прикладів реалізації суміжних задач. У сфері криптографії існує низка пакетів (наприклад, hashlib, pycryptodome), які можуть бути використані для тестування або порівняння результатів авторських алгоритмів з існуючими стандартами.

З огляду на зазначене, мову Python можна розглядати як оптимальний вибір для реалізації геш-функцій на етапі дослідження та розробки, а також як платформу для створення навчального або дослідницького прототипу з можливістю подальшого вдосконалення і перенесення в продуктивне середовище.

3.2 Розробка програмного засобу

Для реалізації програмного засобу, що обчислює геш-значення, потрібно розробити модулі, які виконують послідовне зчитування змісту файлу, розбиття його на блоки, падінг обчислення геш-значення та зберігання його в вигляді файлу. Схема алгоритму цього процесу наведена на рис. 3.1.

Рисунок 3.1 – Алгоритм роботи програмного засобу

Першим етапом роботи програми є зчитування вхідного файлу. Для цього використовується функція read_file_as_binary(), яка відкриває файл у бінарному режимі (режим rb). Весь вміст файлу зчитується у вигляді байтів, після чого кожен байт перетворюється на відповідне 8-бітне двійкове

представлення. Усі біти об’єднуються в єдиний бітовий рядок, який надалі використовується для обчислення гешу.

```
def read_file_as_binary(filepath):
    with open(filepath, "rb") as f:
        byte_data = f.read()
        return ''.join(format(b, '08b') for b in byte_data)
```

Ця функція гарантує, що незалежно від типу вхідного файлу (текстовий чи бінарний), результат буде поданий у вигляді одного бітового рядка, що уніфікує вхідні дані для подальших етапів гешування.

Розбиття на блоки та доповнення (паддінг)

Після того як дані зчитано з файлу у вигляді єдиного бітового рядка, необхідно підготувати їх до гешування шляхом доповнення (паддінгу) та розбиття на блоки фіксованої довжини. Для цього реалізовано функцію to_blocks(), яка працює за принципом, подібним до алгоритмів сімейства SHA.

Згідно з логікою доповнення:

1. До кінця повідомлення додається один біт зі значенням 1, який слугує маркером завершення основного повідомлення;

2. Далі додається така кількість нулів, щоб після фінального етапу довжина всього повідомлення (включно з 64-бітовим представленням початкової довжини) стала кратною розміру блоку — 256 біт;

3. Наприкінці додається 64-бітне число, що представляє початкову довжину повідомлення.

Приклади доповнення наведено рис 3.2

довжина останнього блоку тексту n=256 біт

256 біт	...	256 біт	192 біт	64 біт

довжина останнього блоку тексту n=192 біт

256 біт	...	256 біт	192 біт	64 біт

довжина останнього блоку тексту n<192 біт

256 біт	...	256 біт			64 біт

довжина останнього блоку тексту $192 < n < 256$ біт

256 біт	...			192 біт	64 біт

Рисунок 3.2 - Структура розширеного повідомлення

Таким чином гарантується, що після доповнення повідомлення може бути поділено на блоки фіксованого розміру без залишку. Це забезпечує правильну роботу алгоритму на наступних етапах.

```
def to_blocks(binary_str):
    padded = binary_str + '1'  # додавання біта "1"
    length_bits = bin(len(binary_str))[2:].zfill(64)
    padded += '0' * ((BLOCK_SIZE - (len(padded) + 64) % BLOCK_SIZE) %
BLOCK_SIZE)
    padded += length_bits
    return [int(block, 2) for block in textwrap.wrap(padded,
BLOCK_SIZE)]
```

Результатом виконання функції є список блоків у вигляді цілих чисел, кожен із яких має довжину 256 біт.

Ключовий етап обробки даних — це обчислення гешу на основі підготовлених блоків. За це відповідає функція custom_hash(). Спочатку встановлюється початкове значення гешу (h_0) як 256-бітне число, побудоване шляхом повторення 8-бітного шаблону 10101010 (двійкове представлення числа 170) тридцять два рази.

Далі для кожного блоку вхідного повідомлення виконується наступна послідовність дій:

1. Поточний блок множиться на попереднє значення гешу (h_prev).
2. Результат множення (h_star) — 512-бітне число — розбивається на дві

рівні частини: старшу (L) та молодшу (R) по 256 біт кожна.

3. Частини додаються за модулем 2^{256}, і таким чином утворюється нове проміжне геш-значення.
4. Якщо у результаті утворюється нульове значення, воно замінюється на результат додавання з наперед заданою константою FALLBACK_MASK. Це дозволяє уникнути потенційного ослаблення гешу через нульові значення.

```
def custom_hash(binary_data, h0_bin):
    h_prev = int(h0_bin, 2)
    blocks = to_blocks(binary_data)
    steps = []
    for i, m in enumerate(blocks):
        h_star = m * h_prev
        h_star_bin = format(h_star, '0512b')
        L = int(h_star_bin[:256], 2)
        R = int(h_star_bin[256:], 2)
        h_i = (L + R) % MODULO
        if h_i == 0:
            h_i = (h_i + FALLBACK_MASK) % MODULO
        steps.append({
            'i': i + 1,
            'm': format(m, '0256b'),
            'h_prev': format(h_prev, '0256b'),
            'h_star': h_star_bin,
            'L': format(L, '0256b'),
            'R': format(R, '0256b'),
            'h_i': format(h_i, '0256b'),
            'h_i_int': h_i
        })
        h_prev = h_i
    return h_prev, steps
```

Цей алгоритм має виражену нелінійність і забезпечує сильну залежність гешу від кожного блоку вхідного повідомлення. Перевагою такого підходу є його простота в реалізації при одночасному забезпеченні достатньої стійкості до колізії для демонстраційних та дослідницьких задач.

Для перевірки лавинного ефекту розроблено програмний засіб алгоритм якого наведено на рис. 3.3.

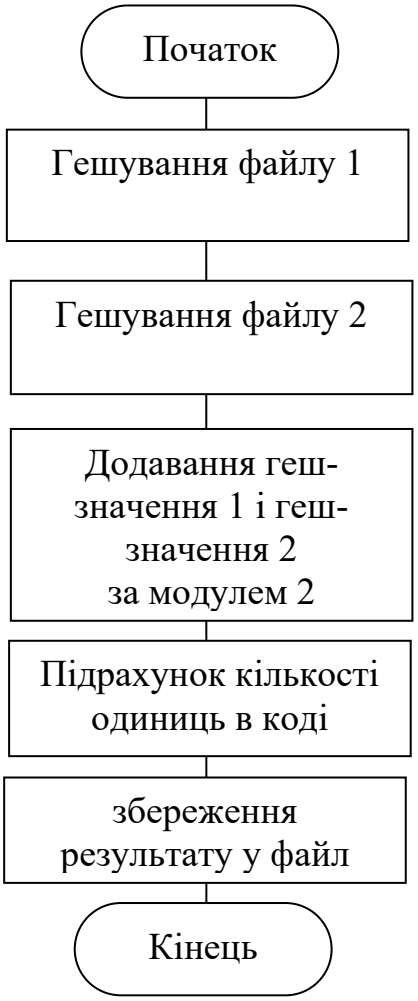


Рисунок 3.3 – Алгоритм роботи програмного засобу для обчислення характеристики лавинного ефекту

Зміст файл 2 відрізняється від змісту файлу 1 лише 1 символом.

В алгоритмі реалізовано функцію flip_middle_bit(). Вона змінює рівно один біт — той, що знаходиться посередині вхідного бітового рядка. Це дозволяє наочно перевірити, наскільки сильно змінюється геш-функція при незначному втручанні у вхідні дані.

```
def flip_middle_bit(binary_str):
    bits = list(binary_str)
    idx = len(bits) // 2
    bits[idx] = '1' if bits[idx] == '0' else '0'
```

```
        return ''.join(bits)
```

Наступна функція avalanche_effect() використовується для аналізу змін у вихідному геші. Вона порівнює два значення гешу (до і після зміни одного біта) побітово та рахує кількість відмінностей:

```
    def avalanche_effect(h1, h2):
        bin1 = format(h1, '0256b')
        bin2 = format(h2, '0256b')
        xor = ''.join('1' if a != b else '0' for a, b in zip(bin1, bin2))
        diff = xor.count('1')
        return bin1, bin2, xor, diff
```

Завдяки цій реалізації можна оцінити, наскільки ефективно геш-функція реагує на навіть найменші зміни — що є ключовим критерієм криптографічної стійкості.

Щоб результати роботи алгоритму зберігались у файл, реалізовано окрему функцію save_output(). Вона створює вихідну директорію (за потреби) і записує результати у вказаний текстовий файл.

```
    def save_output(text):
        os.makedirs(OUTPUT_DIR, exist_ok=True)
        with open(os.path.join(OUTPUT_DIR, OUTPUT_FILE), "w", encoding="utf-8") as f:
            f.write(text)
```

Завершальним етапом є головна функція main(), яка координує виконання всіх кроків — від зчитування вхідного файлу до аналізу лавинного ефекту та збереження результатів. Вона є точкою входу для запуску всієї програми.

```
    def main():
        path = input("Введіть шлях до файлу: ").strip()
        binary_data = read_file_as_binary(path)
        h0_bin = '10101010' * 32
        original_hash, steps_original = custom_hash(binary_data,
h0_bin)
        modified_data = flip_middle_bit(binary_data)
        modified_hash, steps_modified = custom_hash(modified_data,
h0_bin)
        orig_bin, mod_bin, xor_bin, diff_bits =
avalanche_effect(original_hash, modified_hash)
        output = f"""Оригінальний геш (h):\n{orig_bin}\n
    Змінений геш (h'):\n{mod_bin}\n
    XOR (h ⊕ h'):\n{xor_bin}\n
    Кількість змінених бітів: {diff_bits} з {BIT_LENGTH}\n
    Це свідчить про {'хороший' if diff_bits > 100 else 'слабкий'}
лавиноподібний ефект.
    """
        print(output)
        save_output(output)
```

Цей блок коду інтегрує всі частини гешування: зчитування файлу, обчислення гешів до і після змін, перевірку лавинного ефекту, а також вивід результатів на екран і до файлу.

3.3 Тестування програмного засобу для гешування

З метою перевірки працездатності та коректності реалізованого програмного засобу було проведено тестування, яке дозволило оцінити поведінку алгоритму гешування при обробці різних вхідних даних. Основна мета полягає у демонстрації лавинного ефекту — коли незначна зміна у

повідомленні призводить до суттєвих змін у геш-значенні.

Створено текстові файли з змістом, що вказано на (рис.3.1 – рис.3.2).

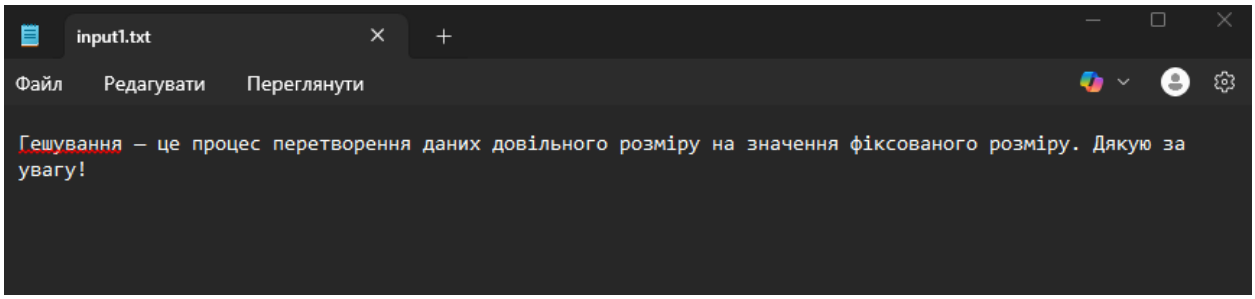


Рисунок 3.4 Приклад вхідного файлу з повідомленням.

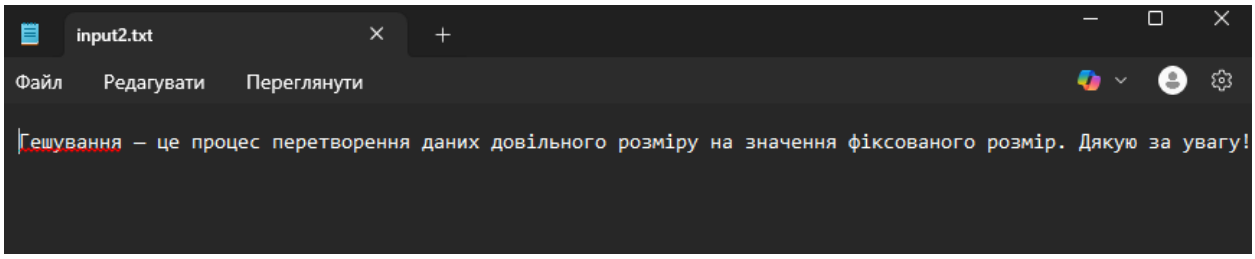


Рисунок 3.5 Приклад вхідного файлу з зміненим повідомленням.

Після опрацювання програмним засобом тесту файлу, отримано загешовані повідомлення (рис.3.3)

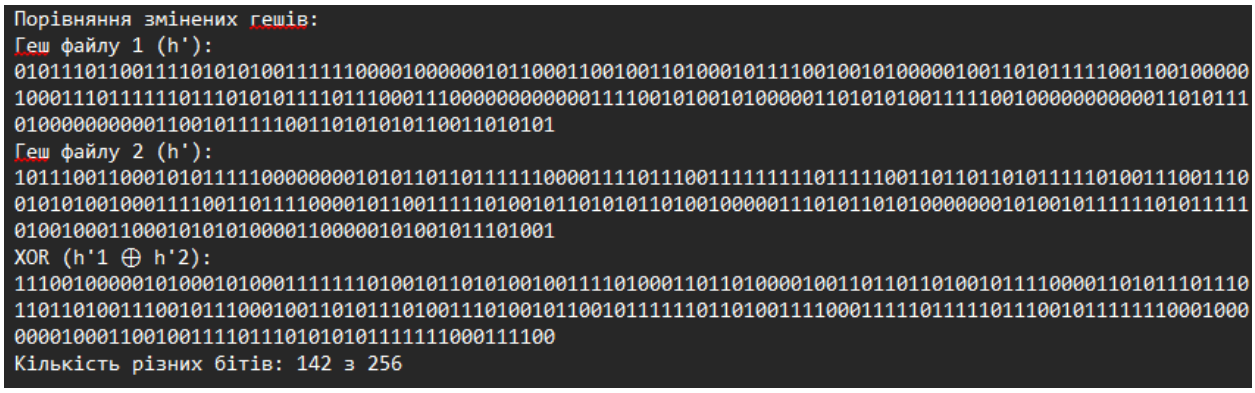


Рисунок 3.6 Приклад зміни вхідного файлу з повідомленням.

Наведено, ще один приклад повідомлення (рис 3.6)

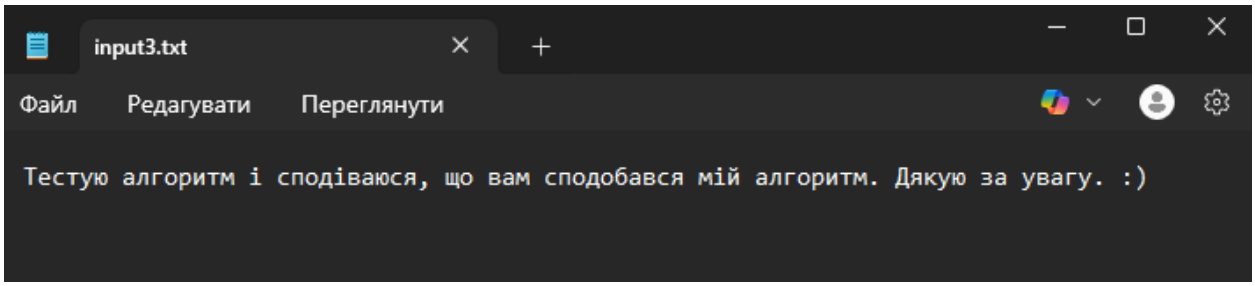


Рисунок 3.7 Приклад вхідного файлу з повідомленням.

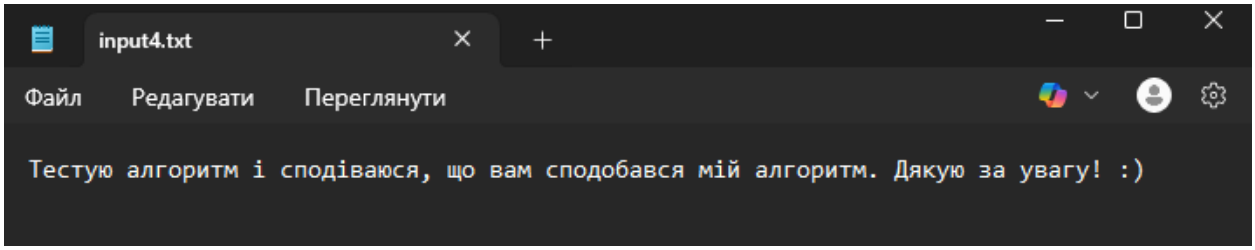


Рисунок 3.8 Приклад вхідного файлу з зміненим повідомленням.

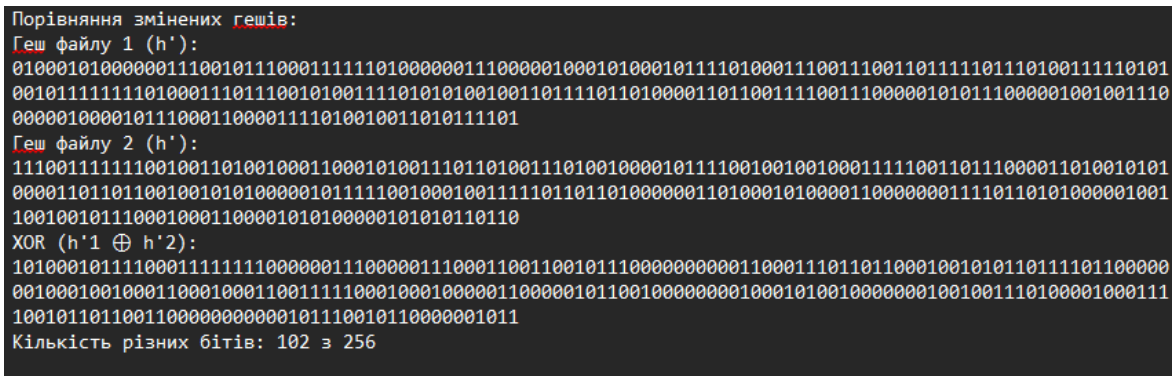


Рисунок 3.9. Геш-значення повідомлень

Як показано, геш-значеннння повідомлення набув зовсім нових значень навіть при зміні одного символа.

ВИСНОВКИ

У першому розділі бакалаврської кваліфікаційної роботи було проведено аналіз сучасних методів гешування даних, зокрема розглянуто основні принципи побудови геш-функцій, їх вимоги, сфери застосування та криптографічні властивості. Особливу увагу приділено властивості лавиноподібного ефекту, що є ключовою для забезпечення надійності гешування. Проаналізовано традиційні підходи до побудови геш-функцій, які використовуються у практиці інформаційної безпеки.

У другому розділі було запропоновано власний підхід до побудови геш-функції, яка базується на використанні модульного множення та додавання, а також розбиття вхідного повідомлення на блоки фіксованої довжини. Розроблено формальний алгоритм обробки даних, включаючи початкову ініціалізацію, поблочну обробку та генерацію фінального гешу.

У третьому розділі реалізовано програмний засіб для гешування даних на мові програмування Python. Програма забезпечує зчитування вхідного файлу, перетворення його у двійкову форму, поблочну обробку та побудову фінального гешу. Особливу увагу приділено експериментальному аналізу лавиноподібного ефекту — реалізовано інверсію одного біта у середині повідомлення та порівняння отриманих гешів. Додатково розширено функціональність програми: користувач має можливість ввести два різні файли та порівняти отримані геш-значення, що дозволяє оцінити чутливість алгоритму до змін у вхідних даних.

Результати тестування підтвердили, що зміна навіть одного біта призводить до суттєвих змін у геш-коді (більше 100 змінених бітів зі 256), що свідчить про наявність сильного лавиноподібного ефекту. У результаті аналізу двох окремих файлів встановлено, що запропонований алгоритм здатен чітко фіксувати навіть незначні відмінності між файлами.

Таким чином, усі поставлені у роботі задачі були успішно реалізовані.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Технології захисту інформації. Ю. А. Тарнавський. 121-124 с. Режим доступу: https://ela.kpi.ua/bitstream/123456789/23896/1/TZI_book.pdf

2. Характеристика основних варіантів SHA-2 1-5 с. Режим доступу: https://studwood.net/1590657/informatika/visnovki#465

3. Козлов Д.В., Красюк В.С. Основи криптографії та захисту інформації. — Київ: КНЕУ, 2020. — 328 с.

4. Петрик В.Я., Бондаренко О.В. Методи та засоби захисту інформації: Навчальний посібник. — Харків: НТУ “ХПІ”, 2018. — 240 с.

5. Пономаренко В.С., Тищенко О.Г. Оцінка ефективності алгоритмів гешування в IT-системах // Сучасні інформаційні технології. — 2021. — №2(44). — С. 45–52

6. Криптографія і що таке геш-функція, блоки Режим доступу: https://www.gate.io/uk/learn/articles/what-is-cryptographic-hash-function/2131

7. Криптографія і що таке геш-функція, блоки [Електронний ресурс]. – Режим доступу: https://www.gate.io/uk/learn/articles/

8. Криптографія і що таке геш-функція, блоки. – Режим доступу: https://www.gate.io/uk/learn/articles/what-is-cryptographic-hash-function/2131

9. Ю. В. Баришев, В. А. Лужецький Методи та засоби швидкого багатоканального гешування даних в комп’ютерних системах 7-8 с. Режим доступу: https://press.vntu.edu.ua/index.php/vntu/

10. П. В. Кравчук, І. Д. Горбенко, А. І. Пушкарьов Аналіз показав, що, під час розробки технологій Blockchain 147-148 с. Режим доступу: https://nure.ua/wp-content/uploads/2018/Scientific_editions/are_2018_19

11. Dhananjoy Dey. Cryptographic Hash Functions: Design, Analysis & Applications Режим доступу: https://iiitl.ac.in/wp-content

12. FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Secure Hash Standard Режим доступу: https://nvlpubs.nist.gov/nistpubs

13. Antoon Bosselaers or Bart Preneel The hash function RIPEMD-160. Режим

доступу - https://homes.esat.kuleuven.be/~bosselae/ripemd160.html

14. R. Rivest The MD5 Message-Digest Algorithm. Режим доступу: https://datatracker.ietf.org/doc/html/rfc1321

15. National Institute of Standards and Technology (NIST). (2015). *SHA-3* Standard: Permutation-Based Hash and Extendable-Output Functions. Режим доступу: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

16. Preneel, B. (1999). Analysis and Design of Cryptographic Hash Functions. Katholieke Universiteit Leuven. Режим доступу: https://homes.esat.kuleuven.be/~preneel/phd.html

17. Rogaway, P., & Shrimpton, T. (2014). Cryptographic Hash-Function Basics: Definitions, Implications, and Separations. In Fast Software Encryption. Режим доступу: https://eprint.iacr.org/2004/276.pdf

18. Biryukov, A., & Perrin, L. (2017). State of the Art in Lightweight Symmetric Cryptography. Режим доступу: https://eprint.iacr.org/2017/511.pdf

19. Wang, X., Yin, Y. L., & Yu, H. (2005). Finding Collisions in the Full *SHA-1*. In *CRYPTO 2020*. Режим доступу: https://crypto.stanford.edu/

Додаток В

ІЛЮСТРАТИВНА ЧАСТИНА

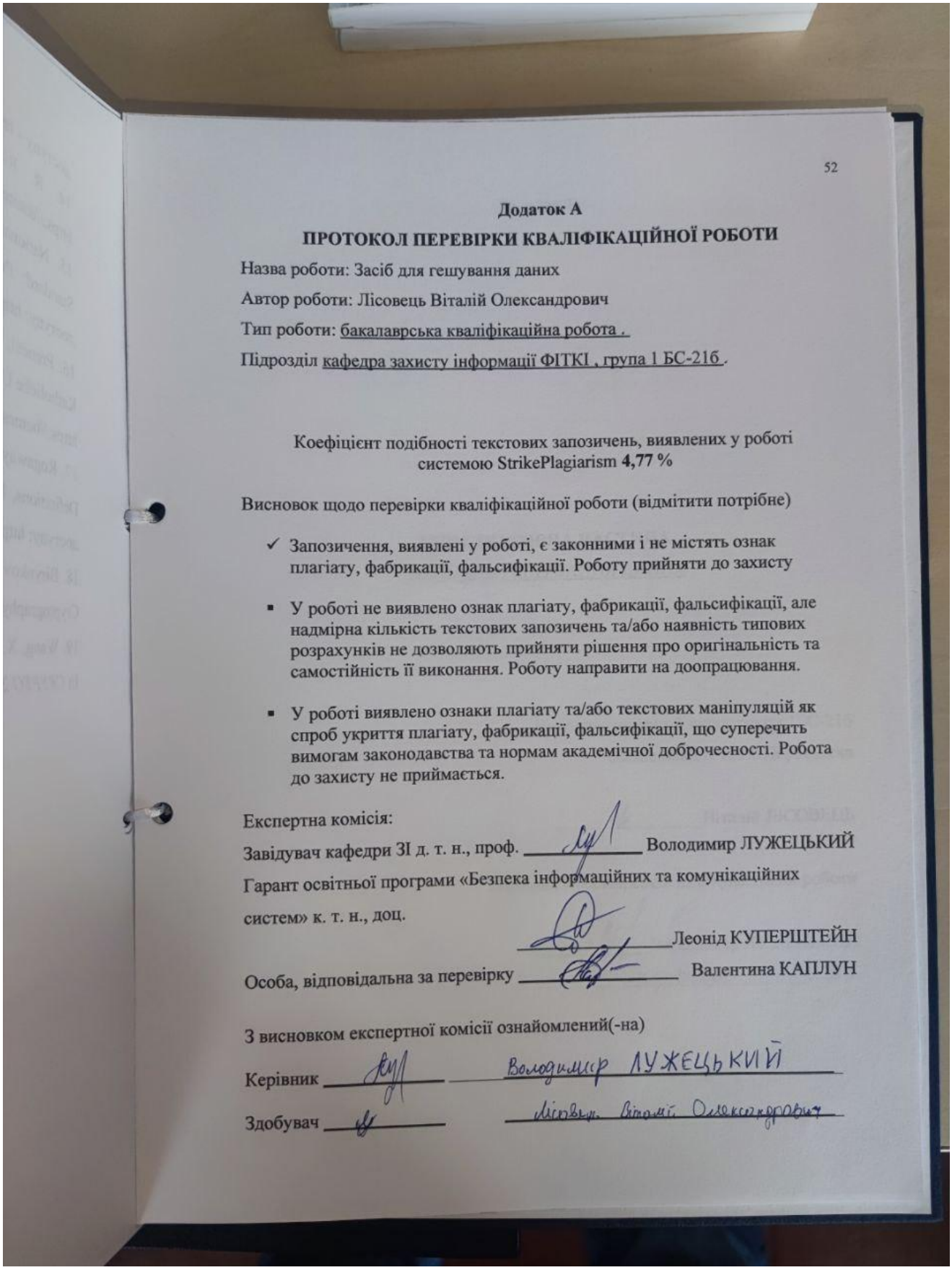
ЗАСІБ ДЛЯ ГЕШУВАННЯ ДАНИХ

Виконав: студент 4 курсу групи 1БС-21б
спеціальності 125 Кібербезпека

_____ Віталій ЛІСОВЕЦЬ

Керівник бакалаврської кваліфікаційної роботи

_____ Володимир ЛУЖЕЦЬКИЙ
«16» червня 2025 р.



Додаток А

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Засіб для гешування даних

Автор роботи: Лісовець Віталій Олександрович

Тип роботи: бакалаврська кваліфікаційна робота .

Підрозділ кафедра захисту інформації ФІТКІ , група 1 БС-21б .

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism **4,77 %**

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту
- ▪ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- ▪ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Завідувач кафедри ЗІ д. т. н., проф. _____ Володимир ЛУЖЕЦЬКИЙ

Гарант освітньої програми «Безпека інформаційних та комунікаційних систем» к. т. н., доц. _____ Леонід КУПЕРШТЕЙН

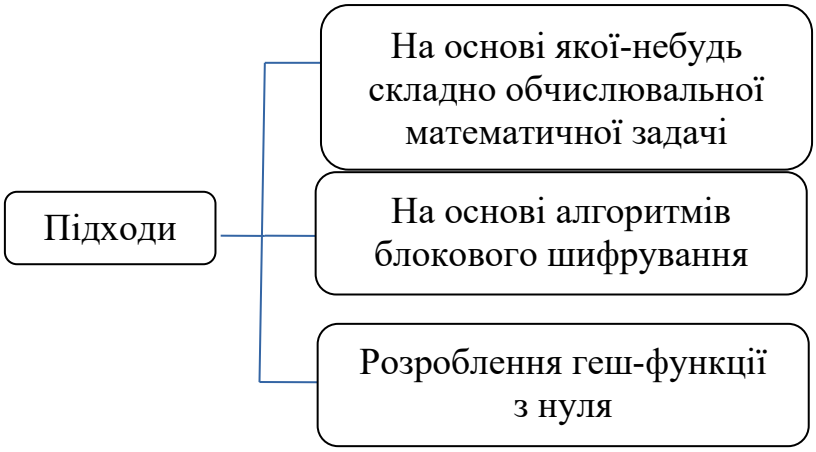
Особа, відповідальна за перевірку _____ Валентина КАПЛУН

З висновком експертної комісії ознайомлений(-на)

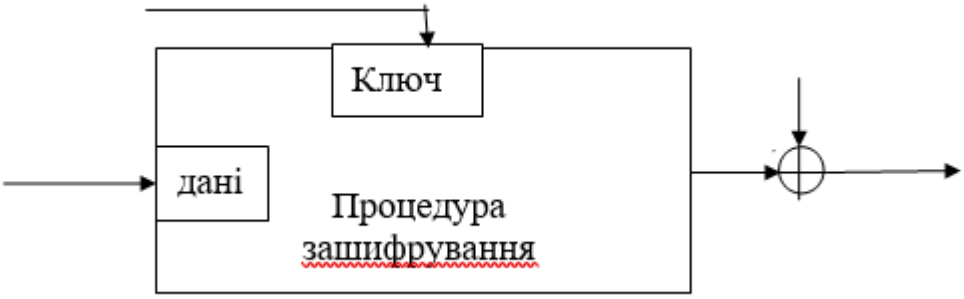
Керівник _____ Володимир ЛУЖЕЦЬКИЙ

Здобувач _____ Лісовець Віталій Олександрович

ПІДХОДИ ЩОДО ПОБУДОВИ ГЕШ-ФУНКЦІЇ



Геш-функції на основі блокового шифру



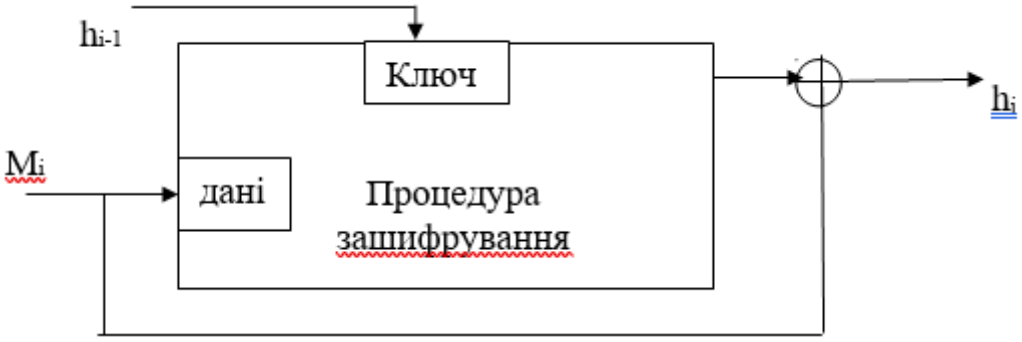
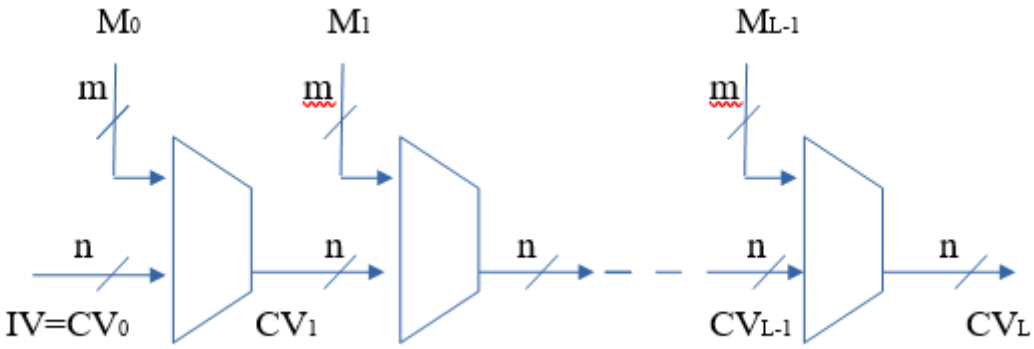


СХЕМА ПРОЦЕСУ ГЕШУВАННЯ

Ітераційна конструкція гешування Меркла–Дамґарда



Структура розширеного повідомлення

довжина останнього блоку тексту n=512 біт

512 біт	...	512 біт	448 біт	64 біт

довжина останнього блоку тексту n=448 біт

512 біт	...	512 біт	448 біт	64 біт

довжина останнього блоку тексту n<448 біт

512 біт	...	512 біт			64 біт

довжина останнього блоку тексту $448 < n < 512$ біт

512 біт	...			448 біт	64 біт

МЕТОД ГЕШУВАННЯ

Основний цикл обробки

1. Множення блоку даних на попереднє геш-значення;

$$h_i^* = m_i \times h_{i-1}$$

2. Результат h_i^* розділяється на дві рівні частини L і R (по 256 біт);

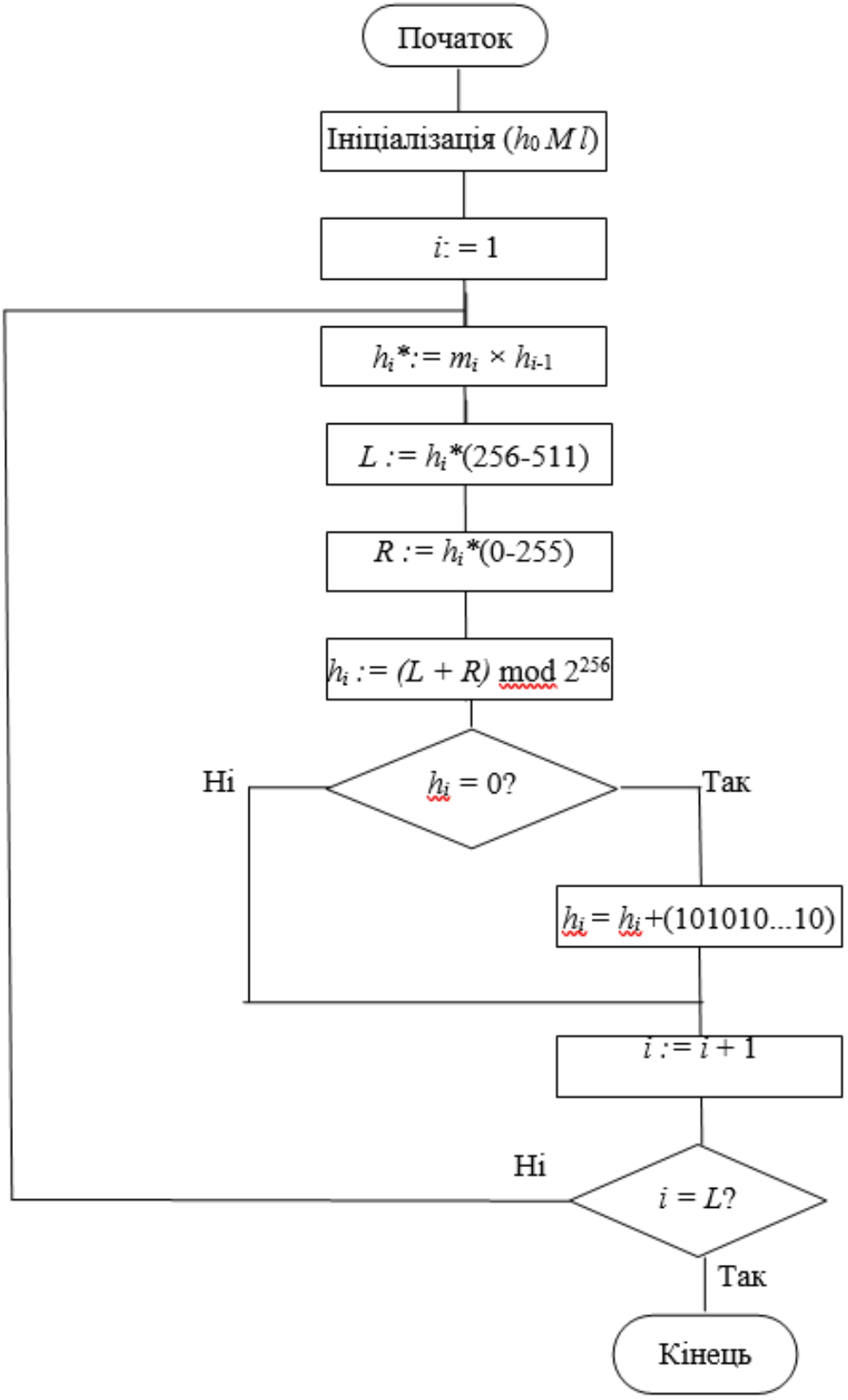
3. Обчислюється геш-значення:

$$h_i = (L + R) \bmod 2^{256}.$$

4. Якщо $h_i = 0$, то

$$h_i = h_i + (101010...10).$$

БЛОК-СХЕМА АЛГОРИТМУ

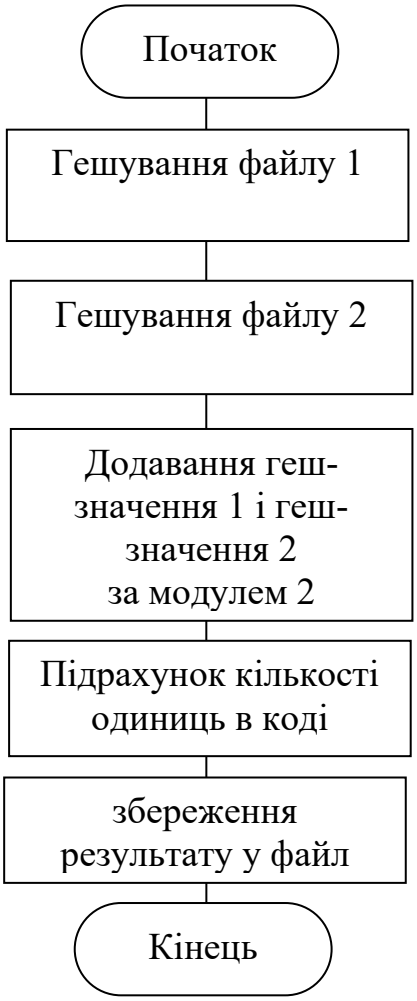


ПОРІВНЯЛЬНІ ОЦІНКИ КРИПТОГРАФІЧНИХ ГЕШ-ФУНКЦІЙ

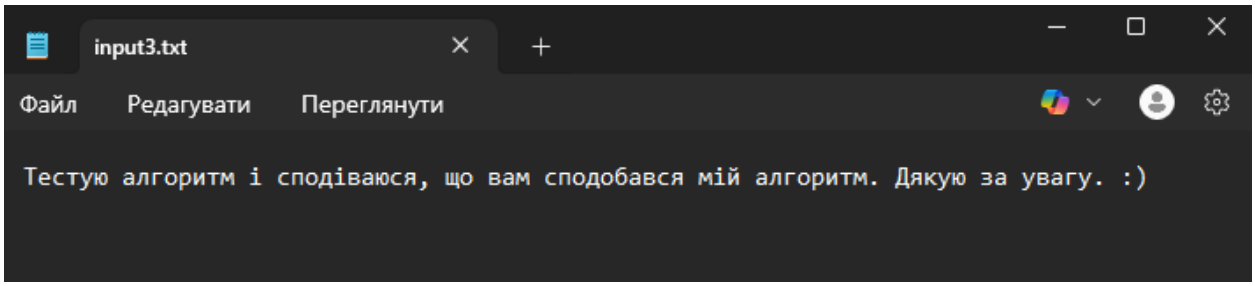
Алгоритм	Розрядність блоку (біт)	Розрядність геш-значення (біт)	Операцій на блок	Операцій на байт	Особливості
Розроблений	256	256	69	2.2	Множення + додавання за модулем
MD5 [14]	512	128	744	11.62	Швидкий, але небезпечний
SHA-1 [19] [5]	512	160	900	14.06	Застарілий, зламаний
SHA-256 [15]	512	256	1100	17.18	Сучасний стандарт NIST
RIPEMD-160 [13]	512	160	1600	25	Середня продуктивність
SHA-3 [15]	1088	256	123	13	Безпечний, інноваційна структура Keccak

СХЕМА АЛГОРИТМУ РОБОТИ ПРОГРАМНОГО ЗАСОБУ ДЛЯ ГЕШУВАННЯ

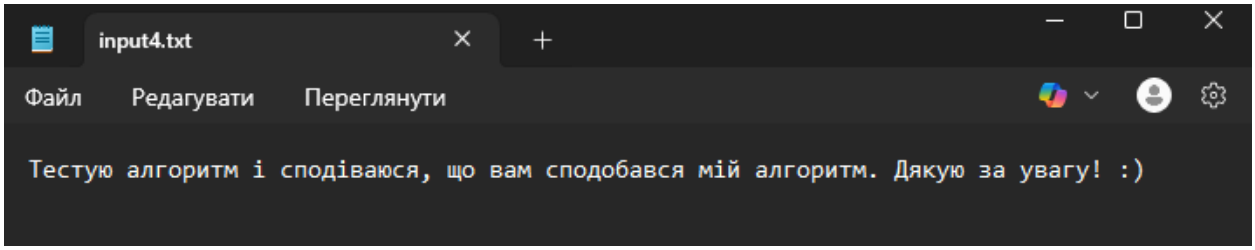
СХЕМА АЛГОРИТМУ РОБОТИ ПРОГРАМНОГО ЗАСОБУ ДЛЯ ОБЧИСЛЕННЯ ХАРАКТЕРИСТИКИ ЛАВИННОГО ЕФЕКТУ



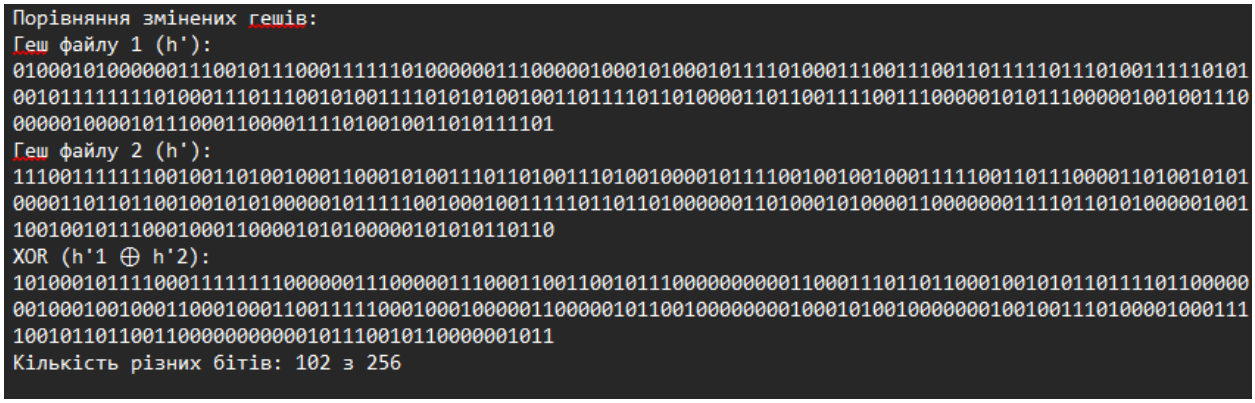
ПРИКЛАД РЕЗУЛЬТАТУ ГЕШУВАННЯ



Приклад вхідного файлу з повідомленням.



Приклад вхідного файлу з зміненим повідомленням.



Геш-значення повідомлень

Як показано, геш-значеннння повідомлення набув зовсім нових значень навіть при зміні одного символа.