МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування факультету)

Кафедра «Інформаційна безпека та наноелектроніка»
(повне найменування кафедри)

Пояснювальна записка

до дипломної роботи
магістр
(ступінь вищої освіти)

на тему Дослідження стандарту цифрового підпису SM2
(назва теми)

Виконав: студент 2 курсу, групи БК-813м
Спеціальності 125 Кібербезпека та захист інформації
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Безпека інформаційних і комунікаційних систем
КОВАЛЬОВ І.Є.
(ПРІЗВИЩЕ та ініціали)
Керівник КОЗІНА Г.Л.
(ПРІЗВИЩЕ та ініціали)
Рецензент МОРОЗ Г.В.
(ПРІЗВИЩЕ та ініціали)

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
Кафедра інформаційної безпеки та наноелектроніки
Ступінь вищої освіти магістр
Спеціальність 125 Кібербезпека та захист інформації
(код і найменування)
Освітня програма (спеціалізація) безпека інформаційних і комунікаційних систем
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІБтаН, к. ф.-м. н., доцент
Андрій КОРОТУН
«_____» _____2024 р.

З А В Д А Н Н Я

НА ДИПЛОМНИЙ РОБОТУ СТУДЕНТА

КОВАЛЬОВА Іллі Євгеновича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження стандарту цифрового підпису SM2 (Research of the SM2 digital signature standard)
керівник проєкту (роботи) к.ф-м.н., доцент КОЗІНА Галина Леонідівна ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові,)
затверджені наказом закладу вищої освіти від «5» грудня 20 24 року № 507
2. Строк подання студентом проєкту (роботи) 24.12.2024р.
3. Вихідні дані до проєкту (роботи) документація стандарту SM2

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Програмна реалізація SM2, порівняльна характеристика SM2 з аналогами, реалізація сліпого підпису SM2

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Презентація PowerPoint (18 слайдів)

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-3	КОЗІНА Г.Л., доцент кафедри ІБтаН	01.10.2024	16.12.2024
Нормоконтроль	КОРОЛЬКОВ Р.Ю., доцент кафедри ІБтаН	20.12.2024	20.12.2024

7. Дата видачі завдання »_01_» _жовтня_ 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	01.10.2024	Виконано
2	Аналіз предметної області	01.10-06.10.2024	Виконано
3	Програмна реалізація функцій SM2	07.10-03.11.2024	Виконано
4	Порівняльна характеристика SM2 із аналогами	04.11-01.12.2024	Виконано
5	Оформлення пояснювальної записки та відповідної документації	02.12-15.12.2024	Виконано
6	Нормоконтроль та рецензування.	16.12-20.12.2024	Виконано
7	Захист дипломної роботи	24.12.2024	Виконано

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

Керівник проєкту (роботи) _____ Галина КОЗІНА
(підпис) (Ім'я ПРІЗВИЩЕ)

АНОТАЦІЯ

Пояснювальна записка до магістерської роботи: 105 с., 7 табл., 15 рис., 2 дод., 47 джерел.

ЕЛЕКТРОННИЙ ЦИФРОВИЙ ПІДПИС, МОВА ПРОГРАМУВАННЯ C#, СЛІПИЙ ПІДПИС, ХЕШ-ФУНЦІЯ, SM2, SM3

Об’єкт дослідження – алгоритм цифрового підпису SM2.

Предмет дослідження – порівняльний аналіз реалізацій алгоритмів підпису із аналогами.

Мета роботи – порівняльний аналіз реалізацій алгоритму підпису SM2 з метою виокремлення його переваг та недоліків у порівнянні із аналогами.

Для реалізації програмної частини використовувалася мова програмування C# (.Net-Framework 4.8) та середовище розробки Visual Studio 2024.

Було проведено порівняльний аналіз за загальними характеристиками, а також виконано тести на швидкодію та вживання пам’яті. Розроблено програму, за допомогою якої можна виконувати сліпий підпис файлів, здійснювати перевірку підпису.

ABSTRACT

Explanatory note to the master's thesis: 105 p., 7 tables, 15 figures, 2 appendixes, 47 sources.

BLIND SIGNATURE, C# PROGRAMMING LANGUAGE, ELECTRONIC DIGITAL SIGNATURE, HASH FUNCTION, SM2, SM3

The object of research is the SM2 digital signature algorithm.

The subject of the study is a comparative analysis of signature algorithm implementations with analogs.

The purpose of the study is a comparative analysis of implementations of the SM2 signature algorithm in order to identify its advantages and disadvantages in comparison with analogues.

To implement the software part, the C# programming language (.Net-Framework 4.8) and the Visual Studio 2024 development environment were used.

A comparative analysis was carried out in terms of general characteristics, as well as tests for performance and memory consumption. A program has been developed that can be used to blindly sign files and verify the signature.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DH (Diffie-Hellman) – Діффі-Гелман;

ECC (Elliptic Curve Cryptography) – еліптична криптографія;

ECDH (Elliptic curve Diffie–Hellman) - Протокол Діффі-Геллмана на еліптичних кривих;

ECDSA (Elliptic Curve Digital Signature Algorithm) – алгоритм з відкритим ключем для створення цифрового підпису;

GF (Galois field) – поле Галуа;

HTTPS (HyperText Transfer Protocol Secure) – захищенний протокол гіперпосилань;

MD5 (Message Digest 5) – хеш-функція;

PGP (Pretty Good Privacy) – бібліотека що дозволяє забезпечити «досить хорошу конфіденційність»;

RSA (Rivest, Shamir та Adleman) – криптографічний алгорим;

S/MIME (Secure/Multipurpose Internet Mail Extensions) – стандарт для шифрування і підпису в електронній пошті за допомогою відкритого ключа;

SHA (Secure Hash Algorithm) – безпечний алгоритм хешування;

SM (Shangmi) – криптографічний стандарт;

SSH (Secure Shell) – протокол віддаленого адміністрування;

SSL (Secure Sockets Layer) – рівень захищених сокетів;

TLS (Transport Layer Security) – захист на транспортному рівні;

WPF (Windows Presentation Foundation) – графічна підсистема Windows.

ВСТУП

У сучасному світі інформаційних технологій та цифрової безпеки цифровий підпис стає ключовим інструментом для забезпечення цілісності та достовірності даних. Ця технологія особливо корисна в конфіденційних системах, таких як документообіг, розповсюдження програмного забезпечення, електронне голосування чи анонімні фінансові транзакції, тощо.

Актуальність теми цієї дипломної роботи визначається зростаючими вимогами до безпеки інформації, криптографічних технологій, зокрема необхідністю надійних та швидких електронно-цифрових підписів, які важливі в питаннях конфіденційності, цілісності та достовірності даних.

Метою цієї дипломної роботи є проведення детального аналізу стандарту підпису SM2, його властивостей та характеристик, а також порівняння з іншими популярними алгоритмами, такими як RSA та ECDSA.

Завданнями дослідження є:

– вивчити теоретичні основи цифрового підпису;

– розглянути стандарт SM2;

– проаналізувати характеристики та властивості стандарту;

– провести порівняльний аналіз SM2 з іншими відомими стандартами;

– розробити програмне забезпечення для демонстрації роботи стандарту у варіанті використання сліпого підпису.

Методи дослідження включають аналітичний огляд наукової літератури та нормативних документів, теоретичний аналіз стандарту SM2, програмну реалізацію, порівняльний аналіз різних стандартів, та проведення практичних випробувань.

У цій дипломній роботі будуть висвітлені основні характеристики стандарту SM2, аналіз його властивостей, а також проведення порівняльного

аналізу із іншими алгоритмами підпису, що допоможе визначити перспективи його подальшого застосування.

Структура дипломної роботи буде включати такі розділи: вступ, теоретичні відомості про стандарт SM2, стандарт хешування SM3 та порівняльна характеристика SM2 із аналогами, а також аналіз стандарту на швидкодію та вживання пам’яті у порівнянні із аналогами, програмна реалізація алгоритму сліпого підпису на стандарту SM2 та висновки.

В ході виконання роботи буде реалізовано програми для здійснення аналізу швидкодії та об’єму вживаної пам’яті, а також програму із графічним інтерфейсом, яка виконує сліпе підписання та перевірку підпису на основі SM2, проведено аналіз його анонімності.

Виконання цієї дипломної роботи допоможе отримати повноцінну порівняльну характеристику стандарту SM2, виявити його сильні та слабкі сторони, а також визначити потенціал його застосування в різних сферах інформаційної безпеки.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО СТАНДАРТ SM2

1.1 Застосування алгоритмів підпису в кібербезпеці

Цифровий підпис є основний інструмент сучасних криптографічних технологій, без якого неможливо забезпечити цілісність і достовірність даних. Його застосування охоплює широкий спектр задач – від електронного документообігу до забезпечення безпеки фінансових транзакцій та електронного голосування. Надійність цифрового підпису базується на використанні складних криптографічних алгоритмів. Наприклад, стандарт SM2 включає в себе алгоритми створення ключів, шифрування повідомлення та підпису. Виключимо із цього опису шифрування, тому що в даному контексті воно не потрібно, а зупинимося на створенні ключів та підписах. Створення ключів в SM2 реалізовано за допомогою асиметричної криптографії, яка використовуються в кібербезпеці для захисту даних, забезпечення автентичності, цілісності й конфіденційності в різноманітних галузях та ситуаціях, та в цілому в будь якій сфері, в якій потрібен захист даних [1].

Асиметрична криптографія – це методи шифрування, які використовують пару ключів: один для шифрування (відкритий ключ) і інший для дешифрування (закритий ключ).

У таких системах відкритий ключ не приховується та може вільно поширюватися, тоді як закритий ключ зберігається у таємниці та передається лише закритими каналами, або взагалі не передається нікому, при цьому кожен із учасників створення ключів знає тільки свій таємний ключ [2]. Тим самим, асиметрична криптографія вирішує проблему передачі ключів, характерну для симетричних систем, де той самий ключ використовується як для шифрування, так і для розшифрування.

Основна ідея полягає в тому, що інформація, зашифрована відкритим ключем, може бути розшифрована лише відповідним закритим ключем, і

навпаки. Це досягається за рахунок складних математичних функцій, які є важкими для зворотного розрахунку. Складність таких обчислень забезпечує високу безпеку.

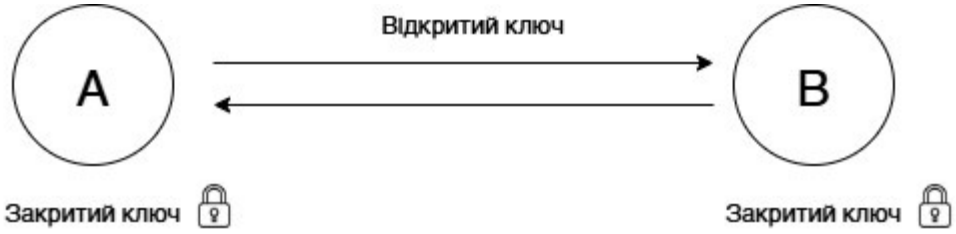


Рисунок 1.1 – Схематичне зображення принципу роботи криптографічного алгоритму із відкритим ключем

Основною механікою асиметричних ключів є генерація ключів – користувач генерує пару ключів:

– відкритий ключ – цей ключ може бути вільно поширений серед інших користувачів і використовується для шифрування повідомлень або для перевірки цифрового підпису;

– закритий ключ – цей ключ є секретним і зберігається в безпеці, який використовується для розшифрування повідомлень, зашифрованих відкритим ключем, або для створення цифрового підпису.

Основними алгоритмами є RSA, який є одним із найвідоміших алгоритмів криптографії з відкритим ключем, заснованим на факторизації великих чисел [3]. Безпека алгоритму базується на складності розкладання великих чисел на прості множники. Користувач генерує два великі прості числа та обчислює їх добуток. Цей добуток використовується для створення відкритого ключа, а інші параметри – для закритого. RSA широко використовується для шифрування даних, цифрових підписів та обміну ключами у веб-з'єднаннях HTTPS, VPN-з'єднаннях і в додатках електронної пошти (наприклад, PGP). RSA подібний до замка з ключем, де один ключ (відкритий) може закрити замок (зашифрувати дані), але тільки інший ключ (закритий) може його відкрити (розшифрувати дані).

В основі еліптичних кривих використовується властивості еліптичних кривих (далі ECC) над кінцевими полями. ECC дозволяє досягти аналогічного рівня безпеки з меншими розмірами ключів у порівнянні з RSA. ECC використовує властивості точок на еліптичною кривою для створення складних математичних відносин між відкритим і закритим ключем.

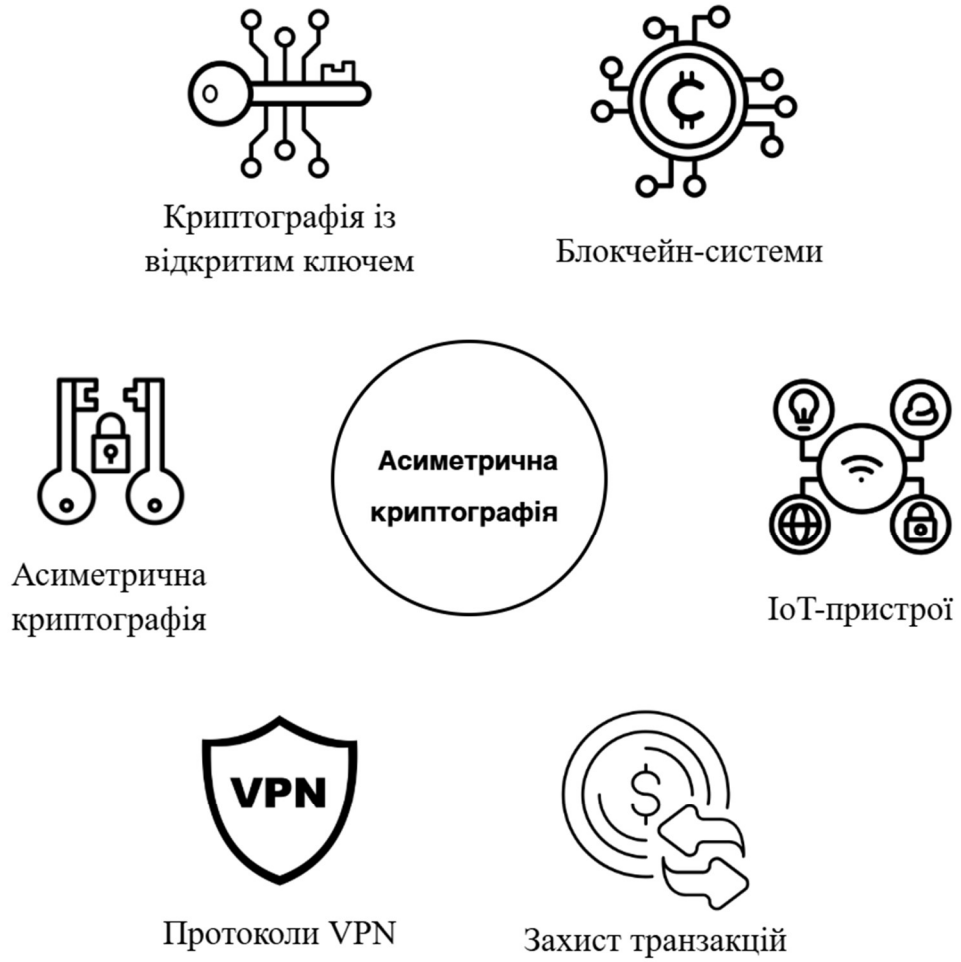


Рисунок 1.2 – Сфери застосування криптографії

ECC широко застосовується в мобільних пристроях і платформах із обмеженими ресурсами, оскільки потребує менше обчислювальної потужності. Алгоритми ECDSA та ECDH є популярними варіантами ECC для цифрових підписів та обміну ключами.

DSA використовується виключно для створення цифрових підписів [4]. Алгоритм був розроблений урядом США і широко застосовується для підтвердження автентичності. Закритий ключ використовується для створення

цифрового підпису документа, а відкритий – для його перевірки. DSA застосовується в системах, де необхідне створення та перевірка цифрових підписів, наприклад, в сертифікатах SSL/TLS і підписанні програмного забезпечення [5].

Основні функції та переваги криптографії з відкритим ключем:

– асиметричне шифрування дозволяє передавати дані без ризику їх перехоплення та розшифрування сторонніми особами. Наприклад, при пересиланні конфіденційних даних можна зашифрувати їх відкритим ключем одержувача, і лише він зможе розшифрувати їх своїм закритим ключем;

– цифровий підпис є способом підтвердження автентичності документа або повідомлення. Це працює так: відправник створює підпис за допомогою свого закритого ключа, а одержувач може перевірити його, використовуючи відкритий ключ відправника. Це гарантує, що документ дійсно походить від заявленого відправника і не був змінений;

– цифровий підпис також забезпечує цілісність даних, оскільки будь-яка зміна документа анулює підпис. Це дозволяє виявляти спроби втручання в інформацію;

– асиметрична криптографія вирішує проблему передачі ключів, властиву симетричному шифруванню. Для обміну ключами в асиметричних системах один з учасників генерує ключову пару та передає іншим свій відкритий ключ, який можна використовувати для шифрування повідомлень або симетричних ключів.

Криптографія із відкритим ключем дозволяє користувачам обмінюватися зашифрованими повідомленнями без необхідності обміну секретним ключем. Протоколи, такі як S/MIME і PGP, використовують відкритий ключ одержувача для шифрування повідомлень. Тільки одержувач може розшифрувати ці повідомлення, використовуючи свій закритий ключ. Або хмарні сервіси можуть зберігати файли у зашифрованому вигляді,

використовуючи відкритий ключ користувача. Лише власник приватного ключа може отримати доступ до цих файлів.

Асиметрична криптографія забезпечує захищене управління доступом у таких протоколах, як SSH та SSL/TLS, використовують криптографію відкритого ключа для перевірки особи користувача або пристрою. Наприклад, для доступу до віддаленого сервера по SSH користувач використовує пару ключів для створення безпечного з'єднання. Багато систем аутентифікації, наприклад сертифікати X.509, використовують відкриті ключі для підтвердження особи користувача чи пристрою.

Протоколи VPN, такі як OpenVPN і IPSec, використовують асиметричне шифрування для обміну ключами та встановлення захищеного з'єднання між віддаленим користувачем і мережею [6].

Для безпечного обміну ключами симетричного шифрування, наприклад протокол обміну ключами Diffie-Hellman (DHE) та інші [7].

У блокчейн-системах криптографія із відкритим ключем використовується для забезпечення автентичності та цілісності транзакцій. Криптографія забезпечує унікальність і безпечний контроль доступу до криптовалют, той ж самий Bitcoin та Ethereum, без участі центрального посередника. Власники криптовалют використовують цифрові підписи, щоб підтвердити, що саме вони є ініціаторами транзакцій, без ризику розкриття приватних ключів.

IoT-пристрої, які обмінюються даними в мережі, використовують криптографію з відкритим ключем для захисту з'єднань або несанкціонованих оновлень.

Криптографія з відкритим ключем використовується для захищених транзакцій, і це забезпечує захист онлайн-транзакцій, підтверджуючи автентичність та цілісність платіжних операцій. Або використовується в чіпах в платіжних картках, вони використовують криптографію з відкритим ключем для захисту даних картки під час проведення транзакцій, мінімізуючи ризик шахрайства.

Цифрові підписи, які дозволяють підтвердити автентичність та цілісність даних, що гарантує, що інформація не була змінена під час передачі. Цифровий підпис гарантує, що документ походить від певної особи або організації та не був змінений. Це широко застосовується в електронних документах, юридичних контрактах, бізнес-угодах тощо. Розробники підписують свої програми, драйвери та оновлення цифровими підписами. Це дає користувачам упевненість, що програмне забезпечення є автентичним і не було змінене хакерами.

1.2 Властивості криптографічного алгоритму із відкритим ключем SM2

Генерація ключів є фундаментальною частиною будь-якого асиметричного криптографічного алгоритму, включаючи SM2 [8]. Ключі визначають безпеку, ефективність та застосування алгоритму, оскільки вони є основою для шифрування, цифрового підпису та обміну ключами.

SM2 – це криптографічний стандарт, який ґрунтується на використанні еліптичних кривих та включає алгоритми для трьох основних криптографічних завдань: цифрового підпису, обміну ключами та шифрування, орієнтований на забезпечення безпеки комунікацій та даних в інформаційних системах.

Особливості алгоритму SM2:

– висока ефективність, завдяки тому, що SM2 використовує операції з еліптичними кривими, які забезпечують криптографічну стійкість при менших розмірах ключів порівняно з традиційними алгоритмами, такими як RSA;

– алгоритм побудований на принципах, спільних з іншими криптографічними стандартами на основі еліптичних кривих, такими як ECC;

– використання 256-бітних простих чисел і спеціально обраних параметрів кривої забезпечує захист від сучасних атак на криптографічні системи.

SM2 працює з використанням еліптичної кривої над скінченним полем F_p, де всі обчислення виконуються модулем простого числа p. Еліптична крива визначається рівнянням (1.1):

$$y^2 = x^3 + ax + b \tag{1.1}$$

Крива використовується для створення групи точок, які служать основою для криптографічних операцій. Для створення ключів використовуються стандартні параметри еліптичної кривої, які визначають математичну структуру алгоритму [9]:

– модуль поля p – це просте число, що визначає розмір поля. У SM2 p – 256-бітове просте число;

– коефіцієнти кривої a і b визначають форму кривої, що дозволяє уникнути вразливостей до атак на криптографічну систему;

– базова точка G – це початкова точка на кривій, яка має великий порядок n і використовується для обчислень публічного ключа;

– порядок кривої n – це число, яке визначає кількість унікальних точок на кривій.

Не дивлячись на те, що SM2 використовує, як і багато інших алгоритмів, еліптичну криву, але він має низку специфічних відмінностей, які роблять його унікальним у порівнянні з іншими алгоритмами.

Наприклад, у досить відомому ECDSA хешування виконується стандартом SHA-256, тоді як SM2 використовує власний стандарт SM3, який спеціально оптимізований для високої швидкості та локальної

криптографічної стійкості. SM2 використовує спеціальні параметри кривої, які відрізняються від популярних кривих (наприклад, secp256r1). Вони були створені з додатковим врахуванням безпеки від відомих атак, включаючи спеціально підібрані коефіцієнти a і b, що знижують ймовірність вразливостей. Інші криві, такі як secp256k1 (використовується в Bitcoin), створені для високої швидкості, але жертвують певними аспектами безпеки.

SM2 забезпечує:

– цифровий підпис;

– шифрування;

– обмін ключами.

У той час як міжнародні стандарти ECC зазвичай реалізують ці завдання окремими алгоритмами (наприклад, ECDSA для підпису, ECDH для обміну ключами, ECIES для шифрування). SM2 включає в себе механізм обміну ключами, і є частиною одного стандарту. Також, SM2 інтегрує ідентифікатори сторін (наприклад, ім’я чи email) у процес підпису, шифрування та обміну ключами. При формуванні підпису SM2 використовується хеш (1.2):

$$H_{256} = (ID||a||b||G||x_p||y_p) \tag{1.2}$$

Це дозволяє захистити підпис від атак, пов’язаних із компрометацією сертифікатів [10]. Також, велика увага приділена процедурі перевірки підпису і пришвидшенні цього процесу. SM2 зменшує кількість необхідних обчислень завдяки спеціально підібраним проміжним значенням, а саме прибирає необхідність додаткового обчислення мультиплікативних інверсій, які є обов’язковими в ECDSA, за рахунок використання модифікованих кроків верифікації. Це забезпечує високу швидкість, що є критичним для масштабованих систем.

Якщо говорити про SM2 як криптоалгоритм, то його відмінність полягає в тому, що шифротекст SM2 складається з трьох частин (1.3)

$$\mathrm{C} = (\mathrm{C}_1, C_2, C_3) \tag{1.3}$$

де C_1 – точка на кривій, що генерується для кожного шифрування, C_2 – зашифроване повідомлення XOR із ключем, C_3 – хеш для перевірки цілісності.

В ECIES (міжнародний стандарт шифрування на основі кривих) структура подібна, але немає прямої інтеграції із хешуванням ідентифікаторів.

Генерація ключів у SM2 визначає як безпеку алгоритму, так і його ефективність у використанні. Приватний ключ у SM2 представляє собою випадково обране число d_A, яке належить діапазону [*1, n-1*], де *n* — порядок еліптичної кривої. Він є суто секретним і використовується для підписання або розшифрування даних. Публічний ключ обчислюється шляхом множення приватного ключа d_A на базову точку *G* еліптичної кривої, що записується як (1.4)

$$P_A = d_A \cdot G \tag{1.4}$$

Публічний ключ є точкою на еліптичній кривій із двома координатами (*x,y*), що додає гнучкість у перевірці автентичності ключів. Поле, над яким обчислюється крива, задається простим числом *p*, яке в SM2 має розмір 256 біт. Коефіцієнти *a* і *b* кривої визначають її форму, уникаючи відомих уразливостей до криптографічних атак. Базова точка *G* є фіксованою початковою точкою з великим порядком *n*, що забезпечує криптографічну стійкість. На завершальному етапі ключі перевіряються на коректність: перевіряється, чи належить точка P_A еліптичній кривій і чи не є вона точкою нескінченності. Приватний ключ у SM2 має розмір лише 256 біт, а публічний — 512 біт (по 256 біт для кожної координати *x* і *y*), що значно компактніше за

аналогічні ключі в RSA. Це дозволяє економити пам’ять і підвищувати швидкість операцій.

SM2 також є мультиплатформенним алгоритмом завдяки використанню стандартних параметрів, що робить його сумісним із різними системами, такими як блокчейни, мобільні пристрої, банківські платформи чи корпоративні мережі [11]. Алгоритм підтримує оптимізацію обчислень через застосування таких методів, як методи Монгомері або Барретта, що дозволяють прискорити операції з великими числами та множення точок на кривій. Завдяки своїй ефективності, компактності ключів та високому рівню безпеки SM2 є провідним рішенням для криптографічних завдань у сферах фінансів, аутентифікації та передачі даних. В підсумку, можна сказати, що SM2 поєднує переваги класичних алгоритмів на основі еліптичних кривих, але додає власні особливості, що і робить SM2 унікальним. Також, SM3 включає в стандарті використання власного алгоритму хешування SM3.

1.3 Властивості алгоритму хешування SM3

Хешування є ключовою концепцією в галузі IT (рисунок 1.3) та комп'ютерних наук. Цей процес включає перетворення вхідних даних будь-якої довжини в стандартний рядок фіксованої довжини, за допомогою хеш-функцій. Ці функції, також відомі як функції згортання, і виходом є хеші або хеш-суми.

Хеші мають певні характеристики, що роблять їх цінними для різноманітних застосувань. На цій схемі також демонструється, як виникає колізія, коли різні вхідні дані призводять до одного й того ж хеш-результату.

SM3 був розроблений у 2008 році під керівництвом криптографа Сяоюнь Вана, який відомий своїм відкриттям вразливостей у MD5 та SHA-1. Сам алгоритм був офіційно представлений у 2010 році. Він забезпечує безпеку цифрових підписів, генерує та перевіряє коди аутентифікації повідомлень, а також використовується для створення випадкових чисел. Алгоритм сприяє підвищенню надійності й сумісності захищених продуктів, пропонуючи стандартизоване рішення для їх розробки.

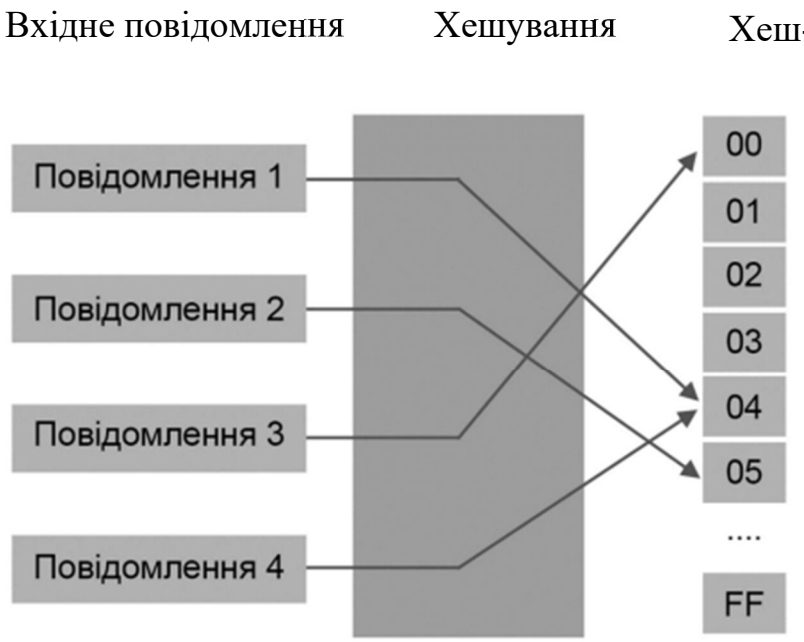


Рисунок 1.3 – Схематичне відображення принципу роботи хеш-функції

Алгоритм SM3 має структуру Merkle-Damgård і генерує хеш довжиною 256 біт [16]. Він подібний до SHA-256, але має низку унікальних технічних характеристик:

– SM3 має більше тимчасових змінних, ніж SHA-256, що може зменшувати продуктивність;

– використання менше констант для раундів (64 у SHA-256 проти 2 у SM3);

– подібний рівень безпеки (256-бітний вихід), але оптимізований для криптографічних стандартів.

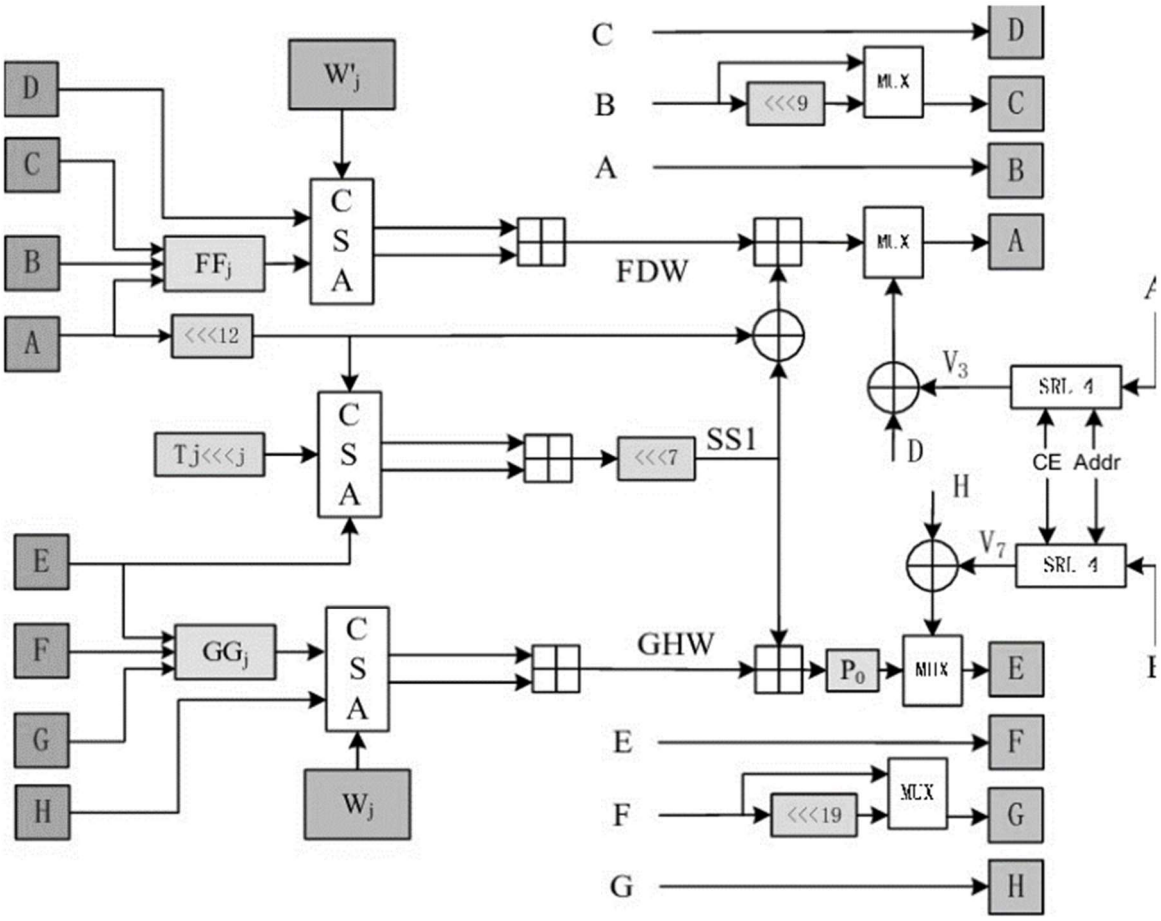


Рисунок 1.4 – Схематичне відображення алгоритму SM3 [16]

Серед ключових особливостей алгоритму можна виділити наступні пункти. Використовуються логічні операції (XOR, AND, OR), циклічні зсуви (<<), додавання за модулем 2^{32}.

SM3 перетворює вхідне повідомлення у 256-бітний хеш [17]. Процес починається з доповнення повідомлення, якщо його довжина менша за 2^{64} біти, до кратної 512 бітам. Для цього додається біт «1», потім низка «0», і завершується все 64-бітним представленням початкової довжини повідомлення.

Отримане доповнене повідомлення ділять на блоки по 512 бітів. Кожен блок обробляється через ітеративний компресійний процес із застосуванням спеціальної функції, яка використовує бітові зсуви, додавання та логічні

операції. Перед обробкою кожен блок розширюється до 132 слів, які використовуються в компресійній функції для формування кінцевого хешу.

Завершальний етап алгоритму полягає у формуванні 256-бітного хеш-значення, яке є результатом компресії останнього блоку повідомлення. Це хеш-значення є унікальним цифровим відбитком вхідного повідомлення.

Загальне порівняння функцій SM3 та аналогів відображено у таблиці 1.1. В ньому зведено всі відкриті відомі дані, які доступні із відповідних стандартів цих функцій хешування.

Таблиця 1.1 – Порівняльна таблиця характеристик функцій хешування

Хеш-функція	Розмір хешу	Стандарт	Особливості
SM3	256	GBT 32905-2016	Оптимізована для використання в ECC (еліптичних кривих);
Купина	256, 384, 512	ДСТУ 7564:2014	Орієнтована на апаратне прискорення; підходить для цифрових підписів і шифрування
BLAKE2	256, 512	RFC 7693	Високошвидкісна функція
SHA3	224,256,384,512	NIST	Побудована на основі Keccak; захист від колізій і диференціальних атак
Whirlpool	512	ISO/IEC 10118-3:2004	Орієнтована на 512-бітні хеші; основа на AES-подібній структурі

Більш глибокий та детальний порівняльний аналіз хеш-функцій здійснюється за наступними характеристиками:

– швидкість обробки;

– розмір хешу;

– аваланшний ефект;

Швидкість обчислення хеш-функції є одним із головних параметрів для оцінки її ефективності [18]. Вона показує, як оперативно функція здатна обробляти дані та формувати хеш. Це особливо важливо для систем із високими вимогами до продуктивності, наприклад, у блокчейнах чи веб-додатках, де швидка обробка прямо впливає на загальну ефективність.

Довжина хешу визначає кількість бітів у результаті роботи хеш-функції. Чим більший хеш, тим вищий рівень безпеки, адже знижується ризик колізій – ситуацій, коли два різних набори даних мають однаковий хеш. Однак надмірно довгі хеші можуть знизити продуктивність через збільшення часу обробки та витрат на зберігання. Тому важливо знайти баланс між безпекою та ефективністю.

Аваланшний ефект демонструє, як чутливо функція реагує на зміни у вхідних даних. Хороша хеш-функція повинна забезпечувати значні зміни вихідного значення навіть при зміні одного біта у вхідних даних. Це ускладнює підбір вхідних значень, що породжують однаковий хеш, і захищає від спроб відновити вихідні дані за хешем.

Гнучкість хеш-функції полягає у її здатності адаптуватися до різних сценаріїв. Це може включати використання додаткових параметрів, таких як «сіль» для унікальності хешу чи секретні ключі для підвищення безпеки. Гнучка функція також повинна підтримувати різні розміри вхідних даних і, за потреби, генерувати хеші різної довжини, що дозволяє використовувати її у найрізноманітніших задачах.

Для обчислення хешів та аналізу використовуватиметься мова програмування C# разом із бібліотекою Bouncy Castle. Це відкрита криптографічна бібліотека, яка пропонує широкий набір алгоритмів і функцій. Вона популярна серед розробників різних мов, зокрема Java та C#. Вибір цієї бібліотеки обумовлений потребою у точності результатів та забезпеченні однакових умов виконання. Основні переваги Bouncy Castle [19]:

– відкритий код, тобто бібліотека повністю і без обмежень безкоштовна, її можна модифікувати та поширювати, а відкритість коду дозволяє переконатися в безпечності реалізованих в бібліотеці алгоритмів;

– бібліотека підтримує шифрування, хешування, цифрові підписи тощо;

– бібліотека створена з акцентом на сучасні криптографічні стандарти;

– легко інтегрується в програмні проєкти;

– активна база користувачів надає підтримку та рекомендації.

Порівняльна характеристика буде здійснюватися для наступних алгоритмів: SM3, SHA3, MD5, Kupyna, Blake2, Whirlpool.

Код для здійснення тестів в Додатку А. В цьому коді використовуються делегати `Func<string, int, string>` для узагальнення викликів хеш-функцій, що спрощує код і робить його більш зручним для масштабування.

Головна точка входу програми створює масиви даних для тестування (кількість повідомлень і розміри хешів) та визначає хеш-функції та їхні параметри в масиві кортежів. Після виконання точки входу здійснюється тестування кожної хеш-функції для заданих параметрів.

Виконується вимір продуктивності (послідовне й паралельне виконання) через TestPerformance а також аналіз на аваланшний ефект через TestAvalancheEffect.

На тестуванні швидкодії зупинимось детальніше. Код для тестування використовує послідовну обробку через цикл: `for (int i = 0; i < count; i++) computeHash(input, bitLength).` Потім робиться тестування для паралельної обробки, для чого використовується `Parallel.For`. Час роботи вимірюється за допомогою `Stopwatch` і виводиться після виконання всіх тестів.

`TestAvalancheEffect` аналізує чутливість [20] хеш-функції до змін вхідних даних за наступним алгоритмом:

– генеруєтья випадковий вхідний рядок;

– модифікується один випадковий біт у рядку через метод ModifyRandomBit;

– обчислюється дистанція (відстань) Хеммінга між хешами оригінального й зміненого рядка;

– за масивом результатів виводиться середнє значення.

Результати роботи коду було записано та сформовано у вигляді таблиць. За результатами роботи коду можна встановити наступні дані (Таблиця 1.2).

Таблиця 1.2 – Таблиця швидкості обробки

Алгоритм	**Розмір Хешу**	**Послідовне, 10**	**Паралельне, 10**	**Послідовне, 100**	**Паралельне, 100**	**Послідовне, 1000**	**Паралельне, 1000**	**Послідовне, 1000000**	**Паралельне, 1000000**
Kupyna	256	15.75 ms	21.24 ms	3.85 ms	2.19 ms	15.82 ms	66.39 ms	3101.72 ms	418.93 ms
Kupyna	384	0.07 ms	0.19 ms	0.68 ms	0.28 ms	6.80 ms	1.04 ms	6835.13 ms	860.09 ms
Kupyna	512	0.09 ms	0.17 ms	0.81 ms	0.23 ms	7.31 ms	1.30 ms	7109.57 ms	856.33 ms
Blake2	160	6.49 ms	2.48 ms	2.24 ms	0.28 ms	4.58 ms	5.06 ms	2855.74 ms	341.74 ms
Blake2	256	0.04 ms	0.08 ms	0.38 ms	0.11 ms	3.04 ms	0.54 ms	3057.81 ms	282.59 ms
Blake2	384	0.05 ms	0.15 ms	0.41 ms	0.17 ms	3.29 ms	0.99 ms	3306.75 ms	312.90 ms
Blake2	512	0.04 ms	0.15 ms	0.45 ms	0.11 ms	3.56 ms	0.46 ms	3572.26 ms	343.87 ms
SHA3	224	2.21 ms	0.10 ms	0.56 ms	0.34 ms	4.82 ms	0.97 ms	1321.24 ms	142.02 ms
SHA3	256	0.02 ms	0.06 ms	0.12 ms	0.08 ms	1.19 ms	0.30 ms	1214.00 ms	150.89 ms
SHA3	384	0.02 ms	0.24 ms	0.19 ms	0.07 ms	1.49 ms	0.79 ms	1486.00 ms	185.40 ms
SHA3	512	0.03 ms	0.14 ms	0.21 ms	0.08 ms	1.70 ms	0.45 ms	1742.52 ms	240.13 ms
SM3	256	1.82 ms	0.13 ms	0.33 ms	0.46 ms	3.07 ms	1.14 ms	1215.87 ms	127.36 ms
Whirlpool	512	2.44 ms	0.27 ms	1.31 ms	0.51 ms	12.45 ms	2.24 ms	3466.04 ms	426.48 ms

Паралельне виконання хеш-функцій демонструє вищу продуктивність при великій кількості повідомлень (1,000,000), що підкреслює ефективність використання мультипотоковості [21]. Проте, за малої кількості повідомлень (10 або 100), послідовне виконання іноді працює швидше, наприклад, у випадку з Kupyna-256. Це може бути пов'язано з додатковими витратами на ініціалізацію і керування потоками.

Таблиця 1.3 – Таблиця аваланшного ефекту

Алгоритм	**Розмір хешу**	**Середня відстань Хеммінга**
Kupyna	256	169
Kupyna	384	253
Kupyna	512	338
Blake2	160	106
Blake2	256	169
Blake2	384	253
Blake2	512	338
SHA3	224	148
SHA3	256	169
SHA3	384	253
SHA3	512	338
SM3	256	169
Whirlpool	512	338

Усі алгоритми продемонстрували високу чутливість до навіть мінімальних змін у вхідних даних (таблиця 1.3). Алгоритми з більшими розмірами хешу зазвичай мали вищу середню Хеммінгову відстань, що свідчить про надійність у плані безпеки, оскільки такі зміни суттєво впливають на вихідний хеш.

За результатами дослідження робимо наступні висновки.

1. Blake2 та SHA3 відзначилися високою гнучкістю, ефективно працюючи з різними розмірами хешу. Вони демонструють високу швидкість обробки та стабільний аваланшний ефект незалежно від умов.

2. Kupyna-256 має непередбачувану продуктивність при малих обсягах повідомлень, із підвищеним часом обробки.

3. Kupyna-384 та Kupyna-512 працюють ефективніше за великих обсягів даних. Вищі версії забезпечують потужний аваланшний ефект, що робить їх ідеальними для задач, де потрібна стійкість до колізій і змін у даних.

4. Blake2 у всіх варіантах демонструє високу швидкість, особливо при паралельній обробці великих обсягів даних. Стабільний аваланшний ефект робить його оптимальним вибором для задач, що потребують швидкості, ефективності й надійності.

5. SHA3-224 і SHA3-256 вирізняються високою продуктивністю, особливо в паралельному режимі. Версії з більшими розмірами хешу працюють трохи повільніше, але всі варіанти демонструють чудову чутливість до змін у вхідних даних, забезпечуючи високий рівень безпеки.

6. Whirlpool хоча має низьку швидкість обробки, особливо в послідовному режимі, алгоритм забезпечує сильний аваланшний ефект. Його доцільно використовувати в задачах із високими вимогами до безпеки, де швидкість менш критична.

7. SM3 відзначається стабільною швидкістю, особливо при паралельній обробці, і демонструє задовільний аваланшний ефект. Це робить його підходящим для систем, де важливий баланс між швидкістю й надійністю.

1.4 Порівняльна характеристика SM2 із аналогами

В даному контексті будемо розглядати стандарт SM2 як криптоалгоритм, а не виключно алгоритм підпису. Для того, щоб оцінити, які існують криптографічні алгоритми та зрозуміти їх характеристики, відмінності, сфери використання, відкриті дані про було алгоритми було зведено до таблиці 1.4 [12].

Таблиця 1.4 – Порівняльна таблиця криптографічних алгоритмів і способів генерації ключів

Критерій	**SM2**	**RSA**	**ECDSA**	**EdDSA**	**DH**	**Lattice-Based (грати)**
Тип алгоритму	Еліптичні криві (ECC)	Математика простих чисел	Еліптичні криві (ECC)	Еліптичні криві (ECC)	Дискретний логарифм	Математика решіток (грат)
Розмір ключа	256 біт	2048/3072 біт	256 біт	256 біт	2048/3072 біт	> 1000 біт
Розмір підпису	512 біт	2048 біт	512 біт	512 біт	Не застосовується	Залежить від реалізації
Швидкість генерації ключів	Висока	Низька	Висока	Висока	Середня	Низька
Швидкість підписання	Середня	Низька	Середня	Висока	Не застосовується	Низька
Швидкість перевірки	Середня	Низька	Висока	Висока	Не застосовується	Низька

Кінець таблиці 1.4

Критерій	SM2	RSA	ECDSA	EdDSA	DH	Lattice-Based (грати)
Стійкість до атак	Висока	Середня	Висока	Висока	Середня	Висока (включаючи постквантові)
Безпека при рівних ключах	Висока	Низька	Висока	Висока	Низька	Дуже висока
Апаратна оптимізація	Є	Обмежена	Є	Повна підтримка	Не потребує	Погана
Квантова стійкість	Ні	Ні	Ні	Ні	Ні	Так
Потреби в ресурсах	Низькі	Високі	Низькі	Низькі	Високі	Високі
Особливості	Інтеграція з SM3;	Найпоширеніший, але застарілий	Стандарт для ECC (NIST)	Простий дизайн, швидкість	Застосовується для обміну ключами	Перспективний для постквантової криптографії
Сфера застосування	Електронний підпис, шифрування, фінансові операції	Загальні криптографічні протоколи	Блокчейн, мобільні платформи	IoT, мобільні пристрої	Захищений обмін ключами	Постквантові системи, довготривалі дані

SM2, ECDSA, EdDSA базуються на еліптичних кривих, що забезпечують високу криптографічну стійкість і компактність ключів [13]. Це робить їх особливо ефективними для сучасних застосувань, таких як мобільні пристрої, IoT та фінансові системи. RSA та DH використовують арифметику великих

чисел, будучи класичними алгоритмами з довгою історією, але поступаються за ефективністю еліптичним кривим [14]. Граткові алгоритми (Lattice-Based) представляють новітній напрям у криптографії, орієнтованій на стійкість до квантових комп'ютерів, і базуються на складності обчислень у математичних решіт. Еліптичні алгоритми потребують значно менших ключів порівняно з RSA або DH, що дозволяє зменшити обсяг передаваних даних і підвищити швидкість обчислень. Наприклад, SM2, ECDSA та EdDSA використовують 256-бітні ключі, тоді як RSA вимагає 2048 або навіть 3072 біт для забезпечення подібного рівня безпеки [15]. Граткові алгоритми потребують більших ключів, але це компенсується їхньою стійкістю до квантових атак. EdDSA вирізняється високою швидкістю, оскільки не потребує генерації нового випадкового числа для кожного підпису. SM2 та ECDSA демонструють збалансовані показники, забезпечуючи швидке створення підписів і їх перевірку. RSA є повільнішим через великі розміри ключів і підписів. Граткові алгоритми мають високу обчислювальну складність, що обмежує їх використання в системах із малими ресурсами. Алгоритми на основі еліптичних кривих забезпечують стійкість до класичних атак, таких як груба сила чи криптоаналіз, але вони вразливі до квантових атак. У свою чергу, граткові алгоритми є перспективними завдяки стійкості до атак квантових комп'ютерів і здатності зберігати безпеку в довгостроковій перспективі.

Еліптичні алгоритми потребують мінімальних обчислювальних ресурсів, що робить їх ідеальними для мобільних пристроїв, IoT та блокчейнів. RSA, DH і граткові алгоритми вимагають значно більше ресурсів через великий розмір ключів та складні обчислення, що обмежує їхню ефективність у системах із низькими потужностями.

2 ПОРІВНЯЛЬНИЙ АНАЛІЗ СТАНДАРТУ SM2 ІЗ АНАЛОГАМИ

2.1 Огляд процедур стандарту SM2

Для реалізації програми-стандарту SM2, обрано мову програмування C# [28]. Цей вибір було зроблено з огляду на її численні переваги, одна з головних – C# пропонує вбудовані бібліотеки, такі як «System.Security.Cryptography», які забезпечують прості інструменти для реалізації генерації ключів, хешування та криптографічних протоколів. Це значно полегшує інтеграцію алгоритмів SM2 та SM3.

Структура C# побудована на принципах об’єктно-орієнтованого програмування, що сприяє організації коду в модульній і зрозумілій формі. Завдяки цьому алгоритми генерації ключів, хешування і підпису можуть бути реалізовані у вигляді окремих класів із чітко визначеними методами, що полегшує супровід і розширення програми.

Важливою перевагою є кросплатформеність C# через використання .NET Core [29]. Це дозволяє запускати програму на різних операційних системах, включаючи Windows, Linux і macOS, забезпечуючи ширші можливості тестування та використання.

Мова C# також полегшує створення графічного інтерфейсу. Інструменти, такі як Windows Forms, WPF або MAUI, дозволяють швидко розробляти інтуїтивно зрозумілі інтерфейси для взаємодії з користувачем. Це особливо корисно для відображення алгоритмів у дії, візуалізації процесів і відображення логів.

Для роботи з великими числами та реалізації математичних обчислень, які є основою алгоритмів SM2 та SM3, C# має доступ до високопродуктивних бібліотек, таких як BigInteger [30].

Зручність для розробників також є важливим фактором. C# має інтуїтивно зрозумілу систему відладки, багатий набір інструментів для розробки, таких як Visual Studio, а також велику спільноту й доступну

документацію. Це значно спрощує процес розробки, розв'язання проблем і впровадження сучасних алгоритмів у програму.

Перш за все для роботи нам потрібен тип даних BigInteger [31]. Його можна отримати в якості додатку NuGet, використовувати System.Numerics, створити самому або взяти із інтернету. В моєму випадку це стандартний System.Numerics, але з деякими перезавантаженнями функцій для більшої зручності. Для роботи SM2 першочергово потрібно реалізувати саме SM3, так як він тісно пов’язаний із алгоримом SM2 [32].

Але для початку реалізуємо допоміжний код, який буде розміщено в файлі Utils.cs (додаток А).

Цей код втілює наступні методи та функції. Клас Utils (утиліти):

– LeftPadWithZero – доповнює рядок нулями зліва до заданої довжини;

– SubArray – витягує підмножину елементів із послідовності;

– AddByte – додає байт у кінець масиву;

– BytesToHex – конвертує масив байтів у рядок у 16-ричному форматі;

– HexStringToByte – перетворює рядок-хекс в один байт;

– HexStringToByteArray – конвертує рядок у масив байтів.

Клас CurveFieldElement (елемент кінцевого поля) – описує елементи кінцевого поля GF(p), використовуваного в ECC: додавання, множення, ділення, піднесення до квадрата, заперечення, в такоє ToBigInteger, яке повертає внутрішнє числове значення елемента поля.

Клас CurvePoint (точки еліптичної кривої) – визначає кординати точки кривої та операції над ними: Add, Negate, Multiply, IsInfinity.

Клас EllipticCurve (еліптична крива) – описує математичну модель кривої, її властивості, а саме модуль p, коефіцієнти a і b, точку нескінченності (Infinity). Реалізує операції FromBigInteger (конвертує число в елемент поля), DecodePoint (декодує точку з рядка у форматі HEX), GetCurveParameters

(Повертає параметри еліптичної кривої SM2, а саме: модуль p, коефіцієнти a, b, базова точка G, порядок n.). Використовується для виконання криптографічних операцій на кривій.

Тобто, цей код спрощує та робить можливим основні операції з точками кривої, включно з додаванням, множенням і перевіркою на «нескінченність». Він буде використовуватися для генерації ключів, підпису даних або реалізації інших криптографічних операцій.

Тепер розробимо код, який реалізує стандарт SM3 (Додаток А) [33].

Розглянемо основні частини коду. Методи для обробки бітових операцій:

– RotateLeft – виконує циклічний зсув вліво на вказану кількість бітів;

– Xor – виконує побітову операцію XOR для двох масивів байтів;

– Add – виконує побітове додавання для двох масивів байтів з врахуванням переповнення;

– And, Or, Not – реалізують побітові операції AND, OR та NOT відповідно для масивів байтів.

Функції P0 і P1 – це спеціальні перетворення для обробки вхідних даних. Вони використовуються для додаткових змін, щоб ускладнити зворотне відновлення хешу.

Основна функція обробки даних – CF, яка приймає вхідний блок даних та попереднє значення (стан), потім обчислює новий стан за допомогою кількох операцій:

– поділ блоку вхідних даних на підблоки;

– використання певних перетворень (як P0, P1) та побітових операцій для генерації нових значень;

– обчислення хешу через 64 раунди циклічного процесу, на кожному з яких відбувається обчислення проміжних значень і їх змішування для ускладнення зворотного відновлення хешу.

Метод StrSum є точкою входу для хешування рядка. Він приймає рядок, перетворює його в байтовий масив, застосовує до нього алгоритм SM3 та повертає результат у вигляді хешу у вигляді шістнадцяткової строки.

Метод SM3Array приймає вхідний масив байтів і виконує всі необхідні операції для обчислення хешу. Ві виконує додавання необхідних доповнень до масиву (включаючи побітове доповнення до потрібної довжини), виконання основних криптографічних функцій для розбитих блоків даних, зворотний зв'язок через використання значення стану для кожного блоку даних.

Якщо говорити власне про криптографічні функції, то функції FF та GG застосовуються в кожному раунді для змішування даних. Вони базуються на побітових операціях XOR, AND, OR та інвертуванні, що дозволяє зручно «розмішати» значення в середині раунду.

Під час виконання процес хешування (CF) кожного раунду стан змінюється, і після всіх 64 раундів, кінцевий результат дає хеш. Алгоритм SM3 використовує початкове значення стану (V), яке є відомим та фіксованим значенням, та модифікується під час хешування. Але, перед тим, як виконати хешування, вхідні дані доповнюються для досягнення певної довжини, щоб гарантовано відповідати певним вимогам алгоритму (наприклад, 512 бітів на блок).

Протестуємо алгоритм на відповідність хешу до стандарту SM3. В документації до цього стандарту вказано декілька прикладів хеш-образів різних повідомлень. Також для порівняння результатів буде використовуватися онлайн-сервіс «Shangmi3(SM3) Hash Tool», в якому можна бути впевненим на 100%, що обчислення здійснюються згідно стандарту [34].

В документації для повідомлень:

– «abc» - йому відповідає хеш-образ «66c7f0f4 62eeedd9 d1f2d46b dc10e4e2 4167c487 5cf2f7a2 297da02b 8f4ba8e0»;

– для шістнадцятирічного числа «61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364

61626364 61626364 61626364 61626364 61626364» - йому відповідає хеш «debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732».

На сервісі «Shangmi3(SM3) Hash Tool» результат співпадає із документацією. Напишемо точку входу для тесту цих повідомлень:

```
using System;
using System.Linq;
using System.Text;
namespace SM23Crypto
{
    class Program
    {
        static void Main(string[] args)
        {
            string input1 = «abc»;
            string hash1 = SM3.StrSum(input1);
            Console.WriteLine($»Hash of 'abc': {hash1}»);
            string hexInput = «61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364»;
            byte[] byteArray = HexStringToByteArray(hexInput);
            string asciiString = Encoding.ASCII.GetString(byteArray);
            string hash2 = SM3.StrSum(asciiString);
            Console.WriteLine($»Hash from hex: {hash2}»);
            Console.ReadLine();
        }
        static byte[] HexStringToByteArray(string hex)
        {
            hex = hex.Replace(« «, ««);
            return Enumerable.Range(0, hex.Length)
                             .Where(x => x % 2 == 0)
                             .Select(x => Convert.ToByte(hex.Substring(x, 2), 16))
                             .ToArray();
        }
    }
}
```

Результат виконання (рисунок 2.1).

Рисунок 2.1 – Вивід виконання обчислення хешів тестових повідомлень за допомогою SM3

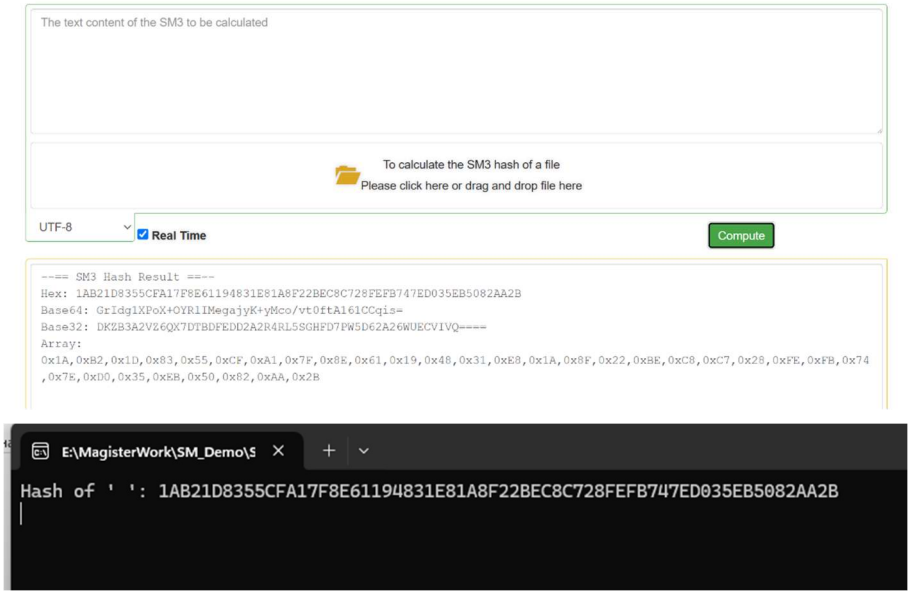


Рисунок 2.2 – Вивід виконання обчислення хешу пустого повідомлення

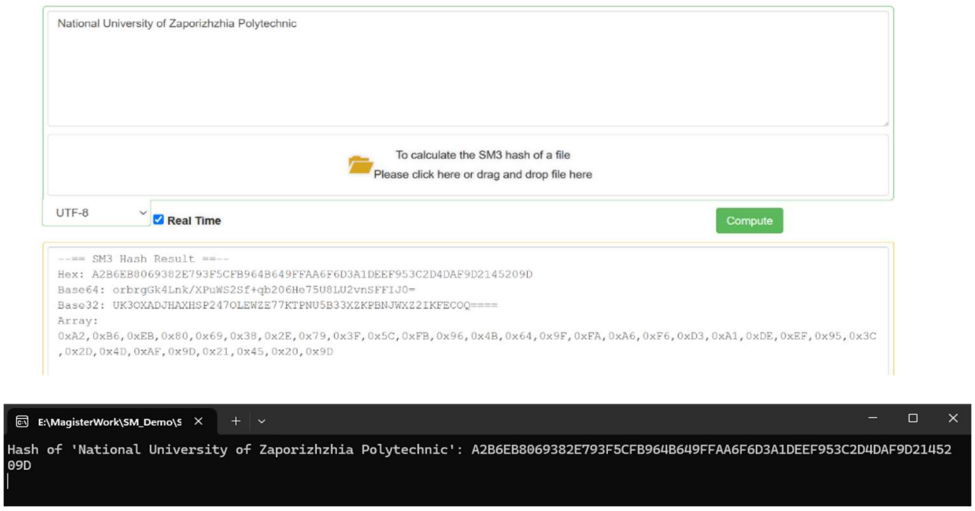


Рисунок 2.3 – Вивід виконання обчислення хешу повідомлення «National University of Zaporizhzhia Polytechnic»

Проведемо ще ряд тестів для інших, більш не стандартних, повідомлень. Наприклад, для пустої строки або більш складного повідомлення. Порівняємо результат для «Shangmi3(SM3) Hash Tool» та власної програми (рисунок 2.2-2.3).

Тепер, маючи реалізований та протестований SM3, ми можемо створити реалізацію SM2 щоб за його допомогою мати можливість створювати пари ключів [35].

2.2 Розробка програмної реалізації алгоритму SM2

Напишемо код (Додаток А). Це повна реалізація алгоритму згідно зі стандартом, що також даже можливість шифрувати текст, що в подальшому можна буде використовувати, наприклад, для шифрування ключів/даних для зберігання, що може забезпечити додаткову безпеку даних.

В цьому коді клас SM2Res використовується для представлення результатів операцій шифрування або дешифрування в SM2 (криптографічний алгоритм). Він містить два основні методи:

- поле _data – це масив байт, який зберігає дані, що можуть бути результатом шифрування або дешифрування;
- конструктор SM2Res(byte[] data) – приймає масив байт як аргумент і ініціалізує поле _data;
- метод GetBytes() – повертає масив байт, що зберігається в _data;
- метод ToString() – перетворює масив байт на рядок в ASCII-форматі. Це корисно для відображення результатів як звичайного тексту (наприклад, після дешифрування).

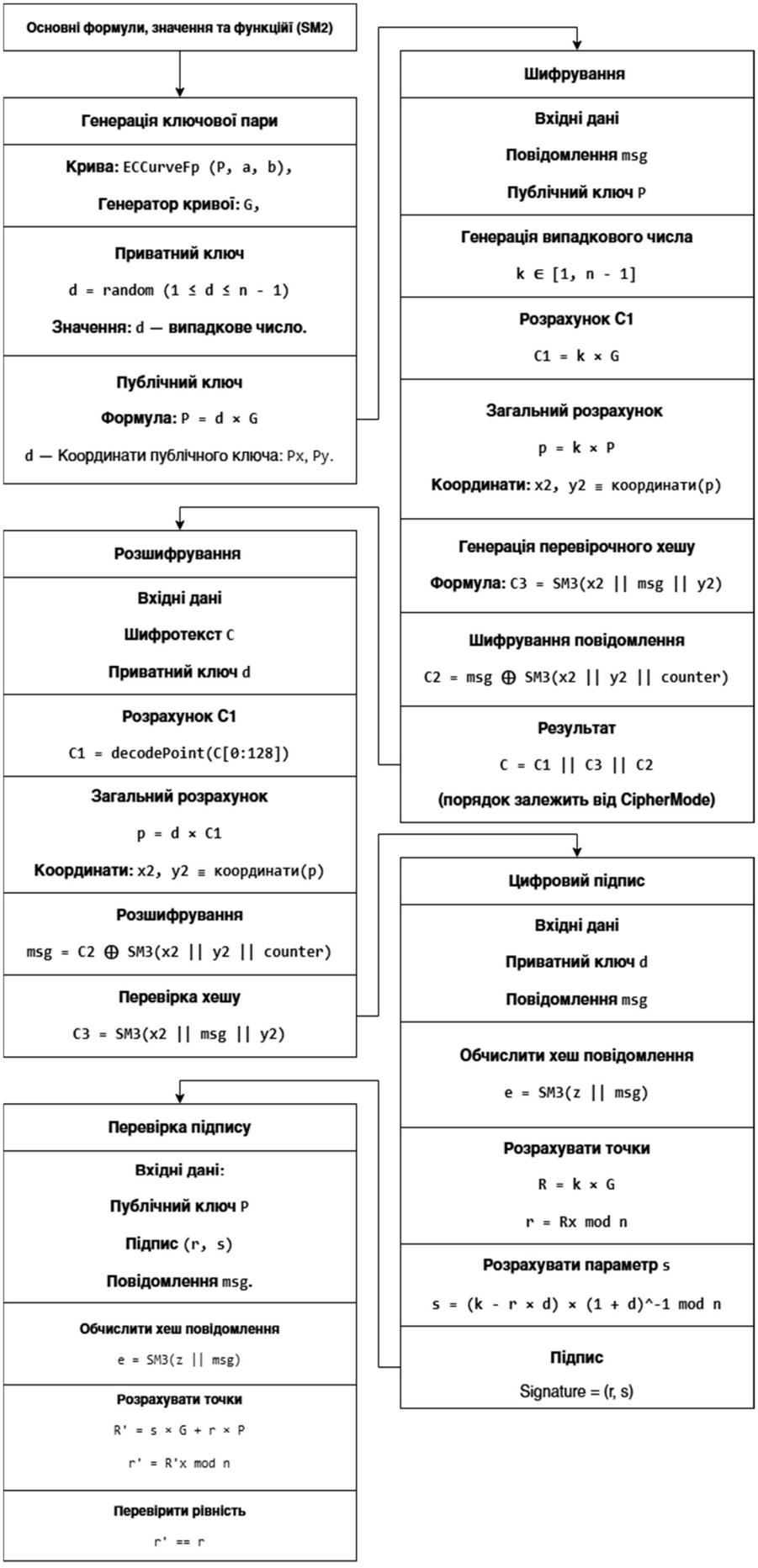


Рисунок 2.4 – Блок схема стандрату SM2

Клас SM2Key використовується для збереження пари ключів (публічного та приватного) для алгоритму SM2:

– властивість PubKey – публічний ключ, що представляється як рядок у шістнадцятковому форматі;

– властивість PriKey – риватний ключ, що також зберігається у вигляді рядка в шістнадцятковому форматі.

Клас SM2 реалізує основні криптографічні операції алгоритму SM2, зокрема генерацію ключів, шифрування та дешифрування:

– CipherMode визначає два режими шифрування для алгоритму SM2;

– метод GenerateKeyPairHex() генерує пару ключів для SM2. Результат повертається у вигляді об'єкта SM2Key з публічним та приватним ключами у шістнадцятковому форматі;

– SMUtils допомагає в роботі з криптографічними операціями, такими як конвертація даних у шістнадцятковий формат, робота з еліптичними кривими тощо.

Загальна блок-схема стандарту SM2, яка відображує всі доступні процеси відображена на рисунку 2.4.

Протестуємо код, згенерувавши пару ключів. Результат на рисунку 2.5.

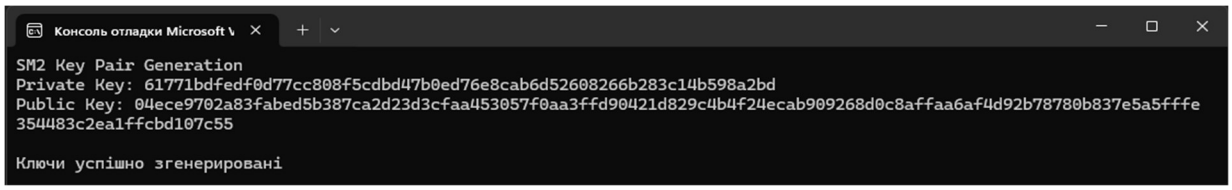


Рисунок 2.5 – Приклад генерації пар ключів

Напишемо код для здійснення підпису [36] із використанням вже розробленого генератору ключів та функції хешування (Додаток А).

Розберемо код. SM2Key keyPair = SM2.GenerateKeyPairHex(); генерує пару ключів (приватний і публічний), що складаються з двох елементів: випадкового числа приватного ключа (PriKey) та точки кривої публічного

ключа (PubKey). Приватний ключ виводиться як ціле число. Публічний ключ у нас дорівнює добутку g на приватний ключ d.

Далі програма запитує в користувача повідомлення, яке буде підписано. Повідомлення кодується в масив байтів і обчислюється його хеш за допомогою функції SM3. Хеш конвертується в велике ціле число. На основі приватного ключа (d) та хешу, обчислюється підпис Sblind. Використовуючи обчислені значення s та h, а також публічний параметр генератора точки g, програма перевіряє підпис, порівнюючи дві точки: sG (точка, що отримана множенням генератора на підпис s) і hP (точка, яка базується на хешу повідомлення та публічному ключі).

Якщо ці точки рівні, підпис дійсний, інакше — повідомлення було змінено. В кінці програма дозволяє користувачу ввести інше повідомлення для перевірки підпису.

Хеш нового повідомлення також обчислюється і порівнюється з попередньо отриманим підписом. У випадку навіть невеликої невідповідності повідомлення хеш змінюється та підпис не обчислюється як дійсний (рисунок 2.6).

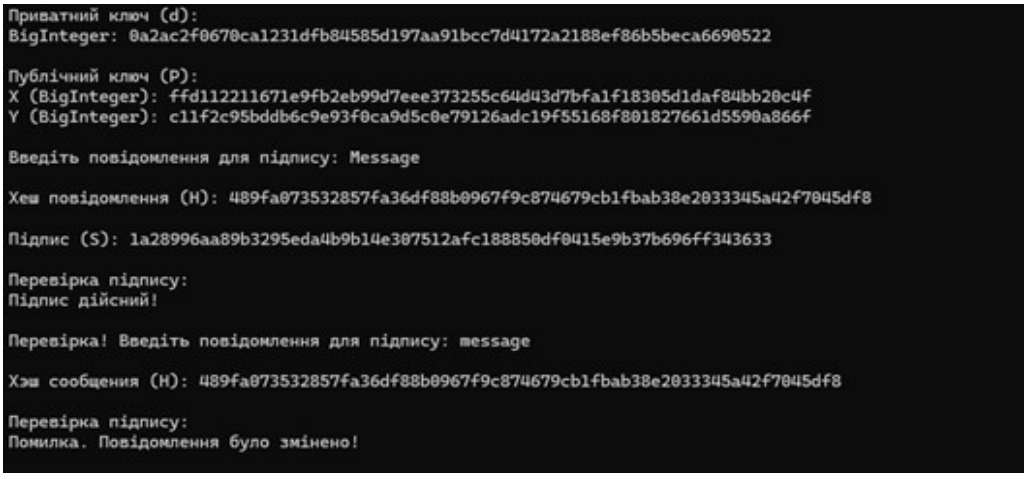


Рисунок 2.6 – Результат виконання програми, яка виконує підпис повідомлення

Тепер ми маємо повну базу для аналізу SM2 з аналогами. Перш за все визначимося з аналогами: серед них є як різноманітні математичні схеми

(наприклад Діффі-Хеллман), так і повноцінні стандарти. Досить розповсюдженим та певною мірою, класичним, є варіант із використанням RSA [38]. RSA вже давно зарекомендував себе у сфері криптографії й досить відомим. А також не менш відомим є міжнародно визнаний стандарт ECDSA [39]. Через це вибір було зроблено на їх користь.

Аналіз буде проводитися за наступними критеріями:

– швидкодія, бо в реальних застосуваннях цей критерій іноді може бути визначальним при виборі алгоритму підпису. В цих тестах буде проведено вимірювання часу, необхідного для генерації підпису (безпосередньо процесу хешування) та верифікації підпису, на різних розмірах повідомлень;

– вимоги до пам’яті, що дуже важливо при обмежених ресурсах, наприклад на мобільних пристроях чи в IoT. Оцінка включатиме вимірювання пам’яті, необхідної для виконання алгоритмів на різних етапах, а також для зберігання криптографічних ключів та підписаних повідомлень.

Для реалізацій всіх тестів, для початку потрібно реалізувати підпис для RSA та ECDSA (Додаток А).

Генерація пари ключів RSA: Використовується метод RSA.Create(), щоб створити пару ключів RSA (публічний і приватний ключ). Розмір ключа 2048 біт. Приватний ключ експортується через метод ExportRSAPrivateKey(), публічний – через ExportRSAPublicKey() і конвертується в строку в форматі Base64. Далі повідомлення підписується за допомогою методу rsa.SignData() з використанням хешу і алгоритму MD5.

В ECDSA створюється пара ключів за допомогою кривої nistP256. Приватний і публічний ключі експортуються за допомогою методів ExportECPrivateKey() і ExportSubjectPublicKeyInfo(). Приватний і публічний ключі виводяться в форматі Base64. Повідомлення хешується за допомогою SHA256 (SHA256.Create()). Підпис генерується за допомогою методу ecdsa.SignHash().

RSA і ECDSA – це два різних алгоритми для цифрового підпису. RSA використовує класичні великі прості числа для підпису, а ECDSA використовує еліптичні криві для ефективнішого підпису, що більш подібно до SM2, яка теж використовує еліптичні криві.

MD5 та SHA256 – це хеш-функції, що використовуються для отримання хешу повідомлення перед його підписанням. MD5 є старим, але швидким алгоритмом, тоді як SHA256 вважається більш безпечним.

2.3 Аналіз швидкодії підпису

Для тестування було модифіковано базовий код для виконання підпису [40]:

```
            // Розміри повідомлень для тесту (в байтах)
            int[] messageSizes = { 32, 64, 128, 256, 512, 1024, 2048, 4096, 1024 * 1024, 5 * 1024 *
1024, 10 * 1024 * 1024, 50 * 1024 * 1024 };
            foreach (var size in messageSizes)
            {
                Console.WriteLine($»Тестування для розміру повідомлення: {size} байт»);

                // Середні часи для кожного етапу
                long totalHashTime = 0;
                long totalBlindTime = 0;
                long totalSignTime = 0;
                long totalVerifyTime = 0;

                for (int i = 0; i < 10; i++) // Тестуємо 10 разів для великих повідомлень
                {
                 //Код для виконання підпису
                }

                // Обчислення середнього часу для кожного етапу
                Console.WriteLine($»Середній час для хешування: {totalHashTime / 10} мс»);
                Console.WriteLine($»Середній час для сліплення: {totalBlindTime / 10} мс»);
                Console.WriteLine($»Середній час для підпису: {totalSignTime / 10} мс»);
                Console.WriteLine($»Середній час для перевірки: {totalVerifyTime / 10} мс»);
                Console.WriteLine(«---------------»);
            }
        }
    }
```

Тестування виконується для розмірів повідомлення: 32, 64, 128, 256, 512, 1024, 2048, 4096 байт та для великих файлів 1, 5, 10, 50 мбайт.

Після виконання аналізу отримано наступні результати, які зведено до таблиці 2.1

Таблиця 2.1 – результат вимірювання часу підписів

Розмір повідомлення (байти)	**SM2 - Час хешування (мс)**	**SM2 - Час перевірки (мс)**	**RSA - Час хешування (мс)**	**RSA - Час перевірки (мс)**	**ECDSA - Час хешування (мс)**	**ECDSA - Час перевірки (мс)**
32 байти	0 мс	1 мс	0 мс	214 мс	0 мс	1 мс
64 байти	0 мс	1 мс	0 мс	206 мс	0 мс	1 мс
128 байт	0 мс	1 мс	0 мс	201 мс	0 мс	1 мс
256 байт	0 мс	1 мс	0 мс	201 mc	0 мс	1 мс
512 байт	0 мс	1 мс	0 мс	204 мс	0 мс	1 мс
1024 байти	0 мс	1 мс	1 мс	203 мс	0 мс	1 мс
2048 байт	0 мс	1 мс	2 мс	201 мс	0 мс	1 мс
4096 байт	0 мс	1 мс	4 мс	199 мс	0 мс	1 мс
1048576 байт (1 МБ)	2 мс	1 мс	1045 мс	202 мс	1 мс	1 мс
5242880 байт (5 МБ)	2 мс	1 мс	5203 мс	201 мс	1 мс	1 мс
10485760 байт (10 МБ)	5 мс	1 мс	10350 мс	200 мс	2 мс	1 мс
52428800 байт (50 МБ)	28 мс	1 мс	51858 мс	200 mc	27 мс	2 мс

Різниця в результатах між SM2, RSA й ECDSA пояснюється кількома факторами, пов'язаними з їхньою криптографічною архітектурою, складністю обчислень та алгоритмами. SM2 – стандарт для еліптичної криптографії, який базується на алгоритмах еліптичних кривих (ECC). Він здатний забезпечити високу безпеку при значно менших розмірах ключів порівняно з RSA.

RSA – це традиційний алгоритм на основі факторизації великих чисел. RSA зазвичай потребує більших розмірів ключів для досягнення порівнянного

рівня безпеки, тому його обчислення зазвичай більш затратні, особливо при більших обсягах даних.

ECDSA – також використовує еліптичні криві, як SM2, але має інший механізм підпису і перевірки. Зазвичай цей алгоритм також ефективний, як і SM2, але з невеликими відмінностями в реалізації.

У RSA час хешування значно більший для малих повідомлень порівняно з іншими алгоритмами. У SM2 та ECDSA час хешування зазвичай мінімальний і дуже схожий на кожному етапі тестування.

RSA зазвичай має малий час перевірки підпису (особливо для малих повідомлень). Проте на великих розмірах повідомлень час перевірки може зрости через більший розмір ключа та операцій з великими числами.

SM2 та ECDSA мають стабільний час перевірки підпису (1 мс) навіть для великих повідомлень, оскільки його схема підпису оптимізована для таких операцій.

2.4 Аналіз вживання пам’яті алгоритму

За допомогою інструментів відладки Visual Studio [41] було виконано тест, який відображує вживання пам’яті програмою. Результат записано у байтах. Обчислення проводиться по 100 разів для малих розмірів, та 10 разів для менших розмірів, потім сума цих даних ділиться на кількість ітерацій, що дає середній результат. Аналіз проводився із аналогічними до тестування на швидкодію функціях. Дані було зведено до таблиці 2.2

Таблиця 2.2 – результат вимірювання витрат пам’яті

Розмір повідомлення (байти)	**SM2 (Середнє споживання пам'яті)**	**RSA (Середнє споживання пам'яті)**	**ECDSA (Середнє споживання пам'яті)**
32 байти	9 байт	9 байт	9 байт
64 байти	9 байт	9 байт	9 байт
128 байт	19 байт	19 байт	19 байт
256 байт	38 байт	38 байт	38 байт
512 байт	76 байт	76 байт	76 байт
1024 байти	153 байти	153 байти	153 байти
2048 байт	307 байт	307 байт	307 байт
4096 байт	614 байт	614 байт	614 байт
1048576 байт (1 МБ)	312851 байт	312851 байт	313344 байт
5242880 байт (5 МБ)	1258237 байт	1258237 байт	1258291 байт
10485760 байт (10 МБ)	1572810 байт	1572810 байт	1572864 байт
52428800 байт (50 МБ)	12582858 байт	12582858 байт	12585721 байт

SM2, RSA, та ECDSA мають однакове споживання пам'яті для малих повідомлень (32-512 байт), оскільки всі вони використовують схожі механізми для підпису і перевірки на етапах початкового обчислення. Для більших повідомлень (від 1 МБ і більше) споживання пам'яті для SM2 і RSA виявляється однаковим, але з невеликою різницею у ECDSA, де споживання пам'яті трохи вище. У випадку великих повідомлень (10 МБ і більше) SM2 і RSA мають дуже схожі результати, тоді як ECDSA злегка перевищує ці значення, хоча різниця незначна. Це підтверджує, що всі три алгоритми мають схоже споживання пам'яті для більших повідомлень, з незначними відмінностями в точності пам'яті на етапі обчислень.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ СЛІПОГО ПІДПИСУ НА ОСНОВІ КРИПТОГРАФІЧНОГО СТАНДАРТУ SM2

3.1 Застосування алгоритму сліпого підпису в кібербезпеці

Сліпий підпис (blind signature) – це криптографічний протокол, який дозволяє отримати підпис від сторони (наприклад, сертифікаційного центру) на повідомлення, не розкриваючи зміст цього повідомлення. Це схоже на звичайний цифровий підпис, але з властивістю «сліпоти», яка гарантує конфіденційність підписаного контенту. Застосовується, наприклад, у системах електронного голосування, електронних грошах та анонімних протоколах автентифікації [22].

Сліпий підпис був вперше запропонований Дейвідом Чаумом у 1982 році. У своїй роботі він показав, що традиційні цифрові підписи недостатньо захищають конфіденційність, оскільки підписант завжди бачить зміст документа що підписується. Завдяки сліпим підписам можна було уникнути розкриття даних без шкоди для безпеки підпису.

Сліпий підпис знайшов широке застосування в різних сферах, де потрібне забезпечення конфіденційності та анонімності. Однією з найперших і найвідоміших реалізацій стала система електронних грошей eCash, розроблена компанією DigiCash під керівництвом Дейвіда Чаума. У цій системі сліпий підпис забезпечував анонімність транзакцій, дозволяючи користувачам здійснювати платежі, не розкриваючи свої персональні дані. Ця технологія стала основою для сучасних концепцій цифрових валют.

Сліпий підпис дуже гарно лягає в концепцію електронного голосування. Використання сліпого підпису у таких системах дозволяє гарантувати, що голоси виборців залишаються анонімними, водночас забезпечуючи, що кожен виборець може проголосувати лише один раз і його голос буде враховано правильно.

У сфері анонімних протоколів сліпий підпис використовується для створення цифрових сертифікатів і систем захисту приватної інформації [23]. Наприклад, у системах анонімної автентифікації або сертифікації особистих даних, де важливо підтвердити справжність певної інформації, не розкриваючи її змісту третім сторонам. Також сліпий підпис застосовується у системах анонімного доступу, зокрема в інфраструктурі контролю доступу. Тут технологія дозволяє користувачам аутентифікуватися, зберігаючи свою конфіденційність. Це корисно, наприклад, для відвідування захищених ресурсів або сервісів, де ідентифікація особи не є обов'язковою, але потрібна перевірка прав доступу.

Сліпий підпис знайшов себе у Web3, блокчейн-системах, системах децентралізованих фінансів (DeFi) та протоколах анонімних комунікацій. Його унікальна властивість забезпечувати конфіденційність і анонімність при перевірці автентичності даних робить цю технологію незамінною в умовах зростання цифрової економіки та децентралізованих платформ.

Для порівняння сліпого підпису із іншими алгоритмами, розглянемо таблицю 3.1.

Таблиця 3.1 – Таблиця порівняння алгоритмів підпису

Критерій	**Сліпий підпис**	**Цифровий підпис**	**Груповий підпис**	**Мультипідпис**	**Пороговий підпис**	**Кільцевий підпис**
Конфіденційність даних	Висока	Відсутня	Помірна	Відсутня	Помірна	Висока
Верифікація підпису	Відкрита	Відкрита	Обмежена	Відкрита	Відкрита	Відкрита
Анонімність підписувача	Гарантована	Відсутня	Гарантована	Відсутня	Відсутня	Гарантована
Колективна участь	Не підтримується	Не підтримується	Підтримується	Обов'язкова	Обов'язкова	Необов'язкова
Застосування	Анонімні транзакції, голосування	Звичайні транзакції, сертифікація	Групова автентифікація	Договори, управління фондами	Децентралізоване управління	Децентралізоване управління

Звичайний цифровий підпис не забезпечує конфіденційності, оскільки підписувач має доступ до змісту повідомлення перед його підписанням [24]. Груповий підпис пропонує помірну конфіденційність, оскільки підпис може бути наданий лише певною групою осіб, і хоча учасники можуть бути анонімними, інформація може бути доступною для інших членів групи. Мультипідпис не гарантує конфіденційності, оскільки всі підписанти бачать і підтверджують зміст, але він підходить для ситуацій, де важливе колективне рішення, як, наприклад, у фінансових угодах. Пороговий підпис також має помірну конфіденційність, оскільки достатньо, щоб певна кількість учасників підписала документ, але зміст може бути доступний деяким з них. Кільцевий підпис дозволяє зберігати високу конфіденційність, оскільки підписувачі не можуть бути ідентифіковані, що робить цей алгоритм корисним для анонімних транзакцій, наприклад, у криптовалютах.

3.2 Алгоритм роботи сліпого підпису

Класичний алгоритм сліпого підпису включає кілька етапів (рисунок 3.1) [25].

1. Генерація ключів. Для початку роботи завжди потрібно створити пару ключів: приватний (SK – secret key) та публічний (PK – public key).

2. Засліплення повідомлення. Це процес власне і є прихованням вмісту повідомлення. Це не просто створення хеш-образу повідомлення, а повноцінна зміна (що в якомусь сенсі можна назвати шифруванням). Для цього використовується засліплюючий фактор r. Тим самим із повідомлення M ми отримуємо M_{blind}.

3. Підписант використовує свій приватний ключ SK для, підпису повідомлення, створюючи підпис S_{blind} і передає його користувачеві.

4. Після отримання S_{blind} користувач за допомогою засліплючого фактору *r* отримує дійсний підпис *S* для оригінального повідомлення *M*.

5. В кінці потрібно перевірити підпис. Це здійснюється за допомогою PK, *M* та підпису *S*. Будь яка людина, що знає ці дані може перевірити дійсність підпису.

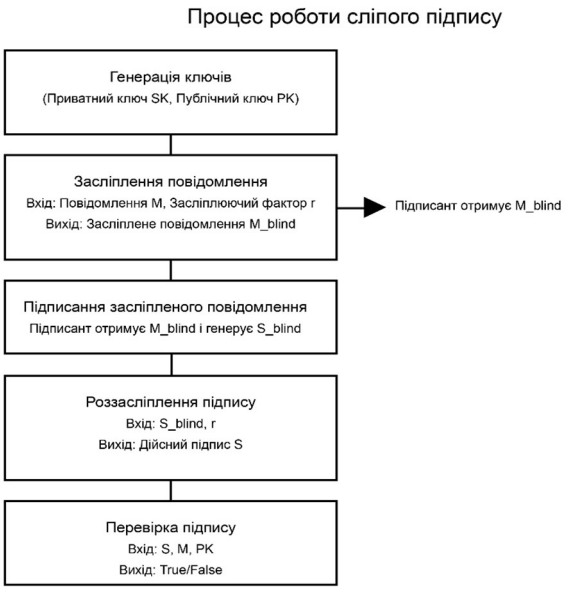


Рисунок 3.1 – Схематичне відображення алгоритму сліпого підпису

Тепер розглянемо конкретний алгоритм сліпого підпису із використанням SM2 для генерації ключів та алгоритму хешування SM3 (рисунок 3.2). Параметри кривою згідно із стандартом виглядають наступним чином: *a = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 FFFFFFFF FFFFFFFC*, *b = 28E9FA9E 9D9F5E34 4D5A9E4B CF6509A7 F39789F5 15AB8F92 DDBCBD41 4D940E93*, *p = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 FFFFFFFF FFFFFFFF*, *n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 7203DF6B 21C6052B 53BBF409 39D54123* [26].

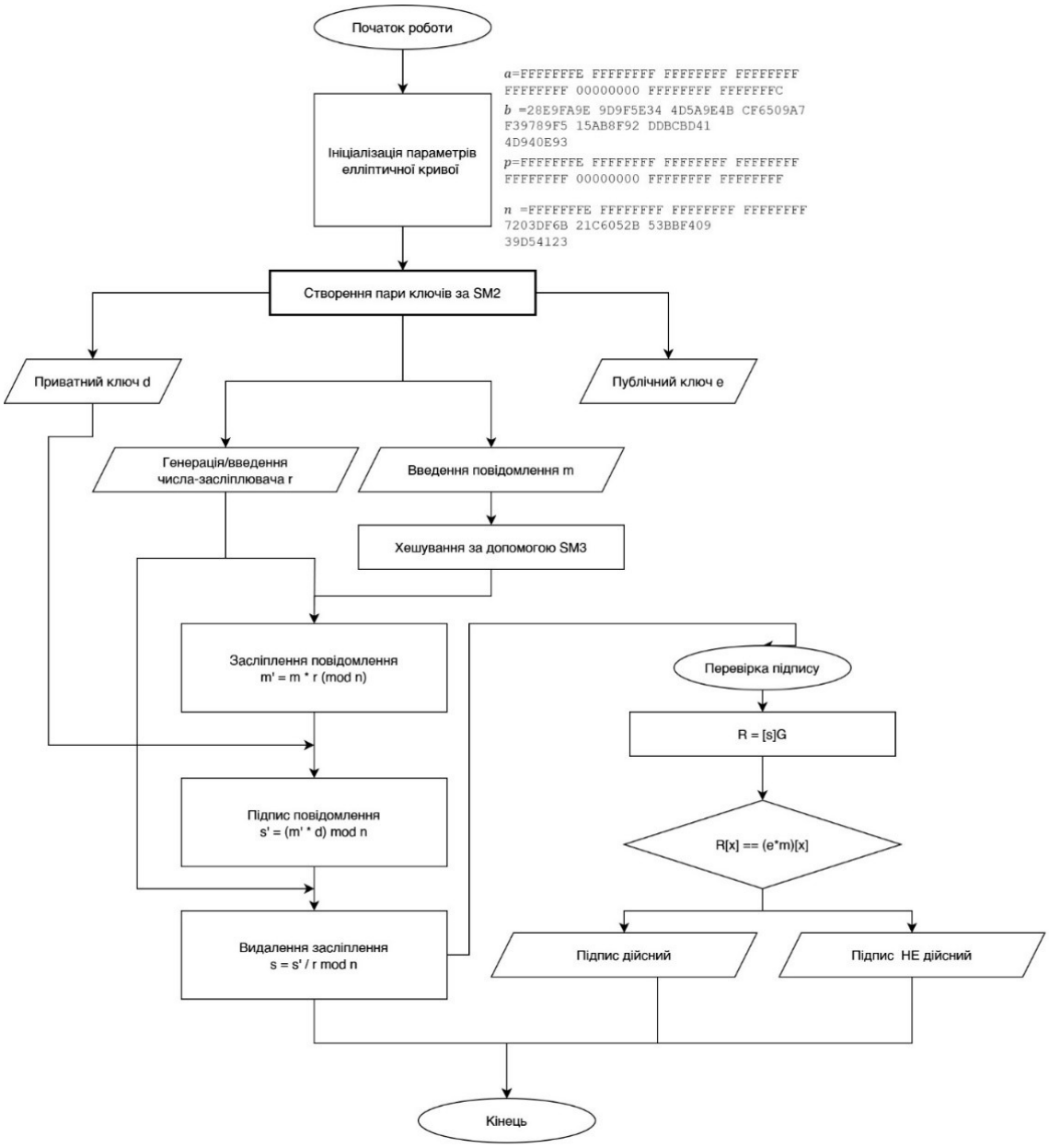


Рисунок 3.2 – Схематичне відображення алгоритму сліпого підпису із використанням SM2

1. Підписант генерує приватний ключ $d_A \in Z_n$, де n – порядок еліптичної кривої та відповідний публічний ключ (3.1)

$$P_A = d_A \cdot G, \tag{3.1}$$

де G – базова точка кривої.

2. Засліплення повідомлення виконується відправником, який обирає (генерує) випадковий $r \in Z_n$ – засліплюючий фактор [27]. Далі обчислюється хеш повідомлення (3.2) і засліплюється хеш H (3.3). Цей хеш H_{blind} користувач надсилає підписанту.

$$H = SM3(M) \tag{3.2}$$

$$H_{blind} = H \cdot r \bmod n \tag{3.3}$$

3. Підписант отримує H_{blind} і виконує обчислення (3.4) засліпленого підпису S_{blind} та надсилає знову відправнику.

$$S_{blind} = H_{blind} \cdot d_A \bmod n \tag{3.4}$$

Важливо, що на цьому етапі підписант не має доступу до початкового тексту, а отже, не може визначити його змісту чи контексту.

4. Відправник отримує S_{blind} і виконує зворотне засліплення вже підпису (3.5). Результат S є дійсним підписом для M.

$$S = S_{blind} \cdot r^{-1} \bmod n \tag{3.5}$$

Завдяки такому підходу дозволяється відокремити процес перевірки автентичності повідомлення від його змісту, зберігаючи баланс між приватністю даних і безпекою цифрового підпису.

5. Перевірка підпису здійснюється для хешу H_M за формулою 3.6. У випадку рівності підпис S признається дійсним.

$$S \cdot G = H \cdot P_A \bmod n \tag{3.6}$$

Всі вищеперераховані дії зазначено в блок-схемі алгоритму (рисунок 3.2), і за цією блок-схемою буде реалізовано програмне забезпечення для здійснення сліпого підпису.

3.3 Програмна реалізація алгоритму сліпого підпису

За допомогою Windows.Forms [37] розроблюємо інтерфейс та прив’язуємо до нього код програми сліпого підпису (Додаток А). В результаті отримано програму, інтерфейс якої відображено на рисунку 3.4.

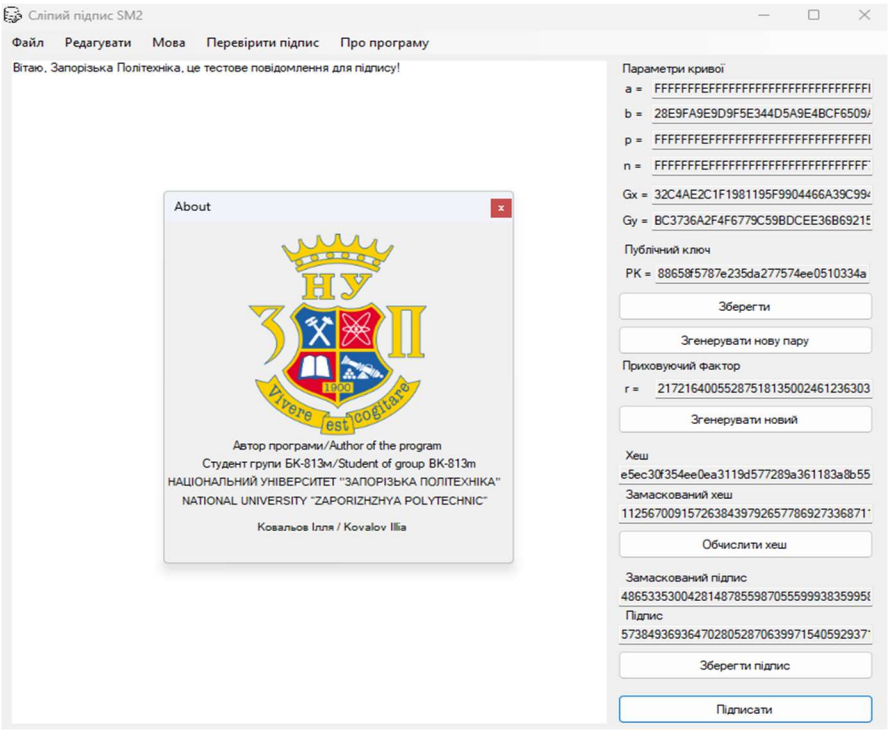
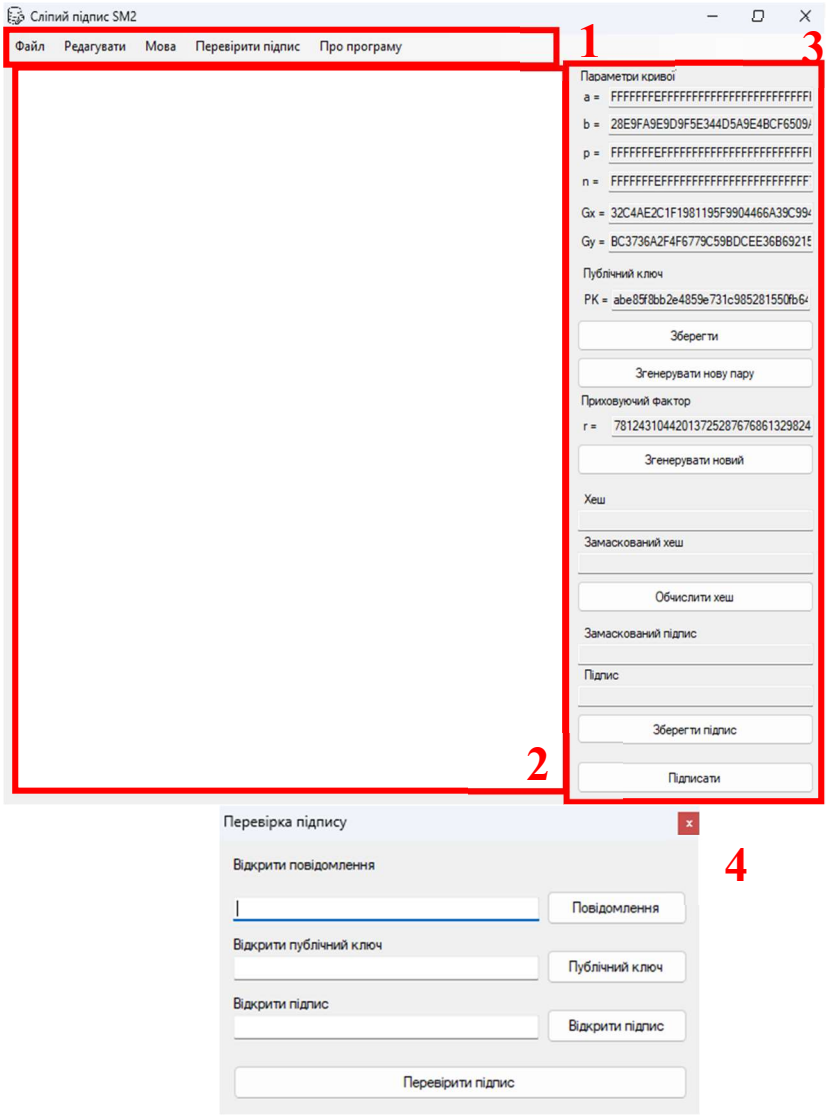


Рисунок 3.4 – Інтерфейс програми

Ця програма дозволяє створювати підпис повідомлення, зберігати його, зберігати ключ для перевірки та саме повідомлення, а також перевіряти підпис на дійсність. Розглянемо більш детально інтерфейс програми (рисунок 3.5).



1 – меню програми, 2 – вікно введення/редагування повідомлення, 3 – вікно налаштувань та керування підписом, 4 – вікно перевірки підпису

Рисунок 3.5 – Інтерфейс програми «Сліпий підпис SM2»

Для роботи з програмою користувачеві потрібно відкрити (Рисунок 3.5 - 1) або ввести з клавіатури бажане повідомлення (Рисунок 3.5 - 2), далі за допомогою панелі керування (Рисунок 3.5 – 3) за надібностю налаштувати ключ та маскуючий фактор та згенерувати підпис. При цьому користувач із боку підписуючої сторони (програми) не зможе дізнатися приватний ключ, за яким робиться підпис.

Далі дані ключу, підпису та саме повідомлення потрібно зберегти у 3 окремі файли. Файли ключу та підпису мають власне розширення для більшої

зручності пошуку та сортування. Це формат «.publicKey» для публічного ключа та «.blindSign» підпису. Повідомлення може зберігатися в будь якому зручному форматі, зазвичай «.txt». При необхідності перевірки ці файли потрібно завантажити в режимі «перевірки підпису» та здійснити перевірку. У разі, якщо повідомлення або файли підпису та ключів не змінювалися, перевірка здійсниться та підпис буде вважатися дійсним.

Також, програма має в собі англійський інтерфейс, який можна перемкнути через пункт меню «Мова» (рисунок 3.6).

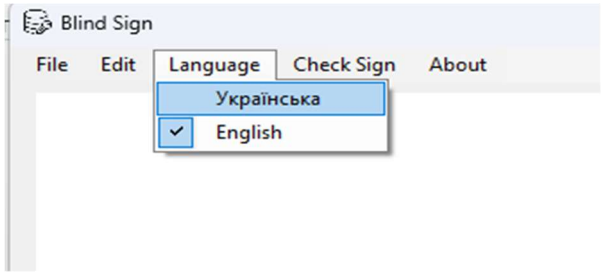


Рисунок 3.6 – Зміна мови програми «Сліпий підпис SM2»

Цю програму можна використовувати як і в навчанні, так і для підписування якихось особистих документів та файлів. Програма розроблена на сучасному фреймворку .NetFramework 4.7 та працює під керуванням Windows 10/11, але також має підтримку більш старих версій Windows при умові, що на них встановлено вказаний фреймворк. Детальний код програми та її елементів знаходиться в Додатку А.

3.4 Аналіз анонімності підпису

Анонімність у криптографії — це здатність системи приховувати особу або контекст користувача, який ініціює криптографічну операцію. Вона дозволяє забезпечити приватність, мінімізуючи ризик розкриття конфіденційних даних. У контексті цифрового підпису це означає, що навіть

при використанні підпису його автор або деталі операції залишаються прихованими. Алгоритм SM2 базується на еліптичних кривих, що робить його близьким за структурою до ECDSA, але з рядом важливих відмінностей.

Цей підхід дозволяє підписувачу працювати з попередньо перетвореним повідомленням (сліпим), таким чином, щоб справжній зміст залишався йому невідомим. Це ефективно захищає приватність користувача та його даних навіть у разі компрометації підписувача. У SM2 цей механізм є важливою частиною функціоналу завдяки оптимізованій математичній структурі алгоритму. SM2 використовує еліптичні криві, які мають менший розмір ключів (256 біт), що робить операції менш вразливими до атак, таких як: аналіз часу виконання, електромагнітне випромінювання, аналіз даних на основі залишкових сигналів.

Висока швидкість операцій у SM2 мінімізує час, протягом якого повідомлення перебуває в обробці. Це зменшує можливості для спостереження або перехоплення даних, а використання унікальних параметрів для кожного сеансу підпису знижує ризик ідентифікації автора через повторювані патерни.

Таким чином, SM2 не лише забезпечує високий рівень безпеки, але й пропонує ефективні механізми захисту анонімності завдяки своїм оптимізаціям. Хоча RSA також може використовувати сліпий підпис, через значну довжину ключів і складність операцій, його застосування в контексті анонімності значно менш ефективне порівняно з SM2 та ECDSA.

ВИСНОВКИ

В ході аналізу на швидкодію встановлено, що RSA демонструє довший час обчислень при роботі з великими повідомленнями через складність операцій з великими числами. SM2 та ECDSA забезпечують стабільну швидкодію навіть для великих даних, що робить їх оптимальними для ресурсозатратних задач.

В ході аналізу на економію пам’яті для малих повідомлень усі алгоритми демонструють однаковий рівень використання пам’яті. Для великих повідомлень ECDSA має трохи вищі витрати пам’яті, що може бути мінімальним недоліком у певних випадках.

Порівняння алгоритмів цифрового підпису показало, що SM2 та ECDSA є ефективними алгоритмами для сценаріїв, де важливі швидкодія та оптимальне використання ресурсів, особливо при роботі з великими даними. RSA доцільно використовувати у випадках, де необхідна перевірена часом стабільність, навіть ціною зниження швидкості та ефективності.

SM2 також забезпечує високий рівень анонімності завдяки підтримці механізму сліпого підпису, який приховує зміст даних від підписувача, що є важливою перевагою для захисту приватності.

Результати дослідження відображені в таблицях.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Whitman M. E., Mattord H. J. Principles of Information Security / M. E. Whitman, H. J. Mattord. – Boston : Cengage Learning, 2021. – 752 с.

2. Шифрування: Типи й алгоритми. Що це і який тип кращий? – HostPro Електронний ресурс. Режим доступу: https://hostpro.ua/wiki/ua/security/encryption-types-algorithms/ (дата звернення: 12.11.2024)

3. Основи криптології: посібник / уклад. викл. факультету інформаційних технологій КНУ імені Тараса Шевченка. – Київ: [б. в.], 2014. – [б. с.].

4. Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. Wiley, 2015. – 937 с.

5. Davis J. Implementing SSL / TLS Using Cryptography and PKI / J. Davis. – Wiley, 2010. – 384 с.

6. Singh, B. Cryptography and Network Security / Wiley, 2019. – 472 с.

7. Diffie-Hellman key exchange (exponential key exchange) – Електронний ресурс / TechTarget. – Режим доступу: https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange (дата звернення: 01.11.2024).

8. Zhang W., et al. SM2: An Introduction to Elliptic Curve Cryptography / W. Zhang, et al. – Springer, 2019. – 250 с.

9. Feng A. SM2 Public-Key Encryption Algorithm / A. Feng, Tsinghua University. – Окреме видання, 2010. – 34 с.

10. Лагун А. Є. Криптографічні системи та протоколи / А. Є. Лагун. – Львів: Львівська політехніка, 2013. – 96 с.

11. Стаття: «Optimization of SM2 Algorithm Based on PolynomialSegmentation and Parallel Computing» - Електронний ресурс. Режим

доступу: https://www.researchgate.net/publication/386162306_Optimization_of_SM2_Algorithm_Based_on_Polynomial_Segmentation_and_Parallel_Computing (дата звернення 16.03.2024).

12. Стаття: «Research on Lightweight Authentication and Key Agreement Protocol in Power Grid Based on the SM9 Cryptographic Algorithm » - Електронний ресурс. Режим доступу: https://www.researchgate.net/publication/371739352_Research_on_Lightweight_Authentication_and_Key_Agreement_Protocol_in_Power_Grid_Based_on_the_SM9_Cryptographic_Algorithm (дата звернення 16.03.2024).

13. Стаття: «EdDSA» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/EdDSA (дата звернення 17.03.2024).

14. Стаття: «RSA» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/RSA (дата звернення 17.03.2024).

15. Стаття: «SM2 (криптографія)» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/SM2 (дата звернення 17.03.2024).

16. Стаття: «The high throughput round architecture of SM3» – Електронний ресурс. Режим доступу: https://www.researchgate.net/figure/The-high-throughput-round-architecture-of-SM3_fig4_262316751 (дата звернення 01.11.2024).

17. Стаття: «SM3 перетворює вхідне повідомлення у 256-бітний хеш» – Електронний ресурс. Режим доступу: https://www.researchgate.net/figure/The-high-throughput-round-architecture-of-SM3_fig4_262316751 (дата звернення 03.11.2024).

18. Стаття: «Design and Analysis of New Version of Cryptographic Hash Function Based on Improved Chaotic Maps With Induced DNA Sequences» - Електронний ресурс. Режим доступу: https://www.researchgate.net/publication/372619010_Design_and_Analysis_of_New_Version_of_Cryptographic_Hash_Function_Based_on_Improved_Chaotic_Maps_with_Induced_DNA_Sequences (дата звернення 03.11.2024).

19. Bouncy Castle. Офіційний сайт Bouncy Castle Cryptography API. Режим доступу: https://www.bouncycastle.org/ (дата звернення: 03.11.2024).

20. Мурашко О. О. Основи теорії хешування в інформаційних системах. – К.: Вид. НТУУ «КПІ», 2017. – 320 с.

21. Стаття: «Хешування Даних: Методи та Алгоритми» - Електронний ресурс. Режим доступу: https://cyberset.com.ua/encryption/heshuvannya-danykh-metody/ (дата зверення 17.03.2024).

22. Стаття: «Blind Signature» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/Blind_signature (дата звернення: 17.03.2024).

23. Стаття: «Anonymous Authentication» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/Anonymous_authentication (дата звернення: 27.11.2024).

24. Стаття: «Digital Signature» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/Digital_signature (дата звернення: 17.03.2024).

25. Євсеєв, С. П., Йохов, О. Ю., Король, О. Г. Гешування даних в інформаційних системах: монографія. Харків: ХНЕУ, 2013 – 312 с.

26. Стаття: «Secure Digital Signature Scheme Based on Elliptic Curves for Internet of Things» - Електронний ресурс. Режим доступу: https://www.researchgate.net/publication/304617301_Secure_Digital_Signature_Scheme_Based_on_Elliptic_Curves_for_Internet_of_Things (дата зверення 17.03.2024).

27. Стаття «Електроний цифровий підпис» - Електронний ресурс. Режим доступу: https://wiki.tntu.edu.ua/ Електроний_цифровий_підпис (дата звернення 17.03.2024)

28. Стаття: «Get started with ASP.NET Core» - Електронний ресурс. Режим доступу: https://learn.microsoft.com/uk-ua/aspnet/core/getting-started/?view=aspnetcore-3.1&tabs=windows (дата звернення 24.11.2024).

29. Стаття: «C# language documentation» – Електронний ресурс. Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 24.11.2024).

30. Стаття: «Cryptographic Signatures» - Електронний ресурс. Режим доступу: https://learn.microsoft.com/en-us/dotnet/standard/security/cryptographic-signatures (дата звернення 24.11.2024).

31. Стаття: «BigInteger в .NET: практичне використання та приклади» - Електронний ресурс. Режим доступу: https://docs.microsoft.com/en-us/dotnet/api/system.numerics.biginteger (дата звернення: 24.11.2024).

32. Стаття: «C# State Secret SM2 Encryption Algorithm Implementation» - Електронний ресурс. Режим доступу: https://blog.csdn.net/Qcg0223/article/details/124319378 (дата звернення 24.11.2024).

33. Стаття: «C# SM3 Encryption» - Електронний ресурс. Режим доступу: https://blog.csdn.net/mzl87/article/details/132019730 (дата звернення 24.11.2024).

34. Shangmi3(SM3) Hash Tool – Електронний ресурс. Режим доступу: https://the-x.cn/en-us/hash/ShangMi3Algorithm.aspx (дата звернення: 24.11.2024).

35. SM2: Overview of the SM2 Algorithm – Електронний ресурс. Режим доступу: https://www.researchgate.net/publication/262316751_The_high-throughput_round_architecture_of_SM3 (дата звернення: 24.11.2024).

36. Stinson D. R., Paterson M. Cryptography: Theory and Practice / D. R. Stinson, M. Paterson. – 4th ed. – Вид-во Taylor & Francis, 2023. – 598 с.

37. Стаття: «.NET Desktop Guide for Windows Forms» - Електронний ресурс. Режим доступу: https://learn.microsoft.com/uk-ua/dotnet/desktop/winforms/?view=netframeworkdesktop-4.8 (дата звернення 24.11.2024).

38. Стаття: «RSA Algorithm» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/RSA_algorithm (дата звернення: 01.11.2024).

39. Стаття: «Elliptic Curve Digital Signature Algorithm (ECDSA)» – Електронний ресурс. Режим доступу: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm (дата звернення: 02.11.2024).

40. Стаття: «Практичні схеми реалізації алгоритмів електронного цифрового підпису» - Електронний ресурс. Режим доступу: https://www.researchgate.net/publication/328828732_PRAKTICNI_SHEMI_REALIZACII_ALGORITMIV_ELEKTRONNOGO_CIFROVOGO_PIDPISU

41. Стаття: «Використання інструментів відладки Visual Studio» – Електронний ресурс. Режим доступу: https://learn.microsoft.com/en-us/visualstudio/debugger/ (дата звернення 01.11.2024).

42. Стаття: «RSA та його порівняння з іншими криптографічними алгоритмами» – Електронний ресурс. Режим доступу: https://www.example.com/rsa-comparison (дата звернення 03.11.2024).

43. Стаття: «SM2 – Інтернет-протоколи для електронного підпису на основі еліптичних кривих» – Електронний ресурс. Режим доступу: https://www.example.com/sm2-elliptic-curve-signature (дата звернення 03.11.2024).

44. Стаття: «ECDSA та його застосування в криптографії» – Електронний ресурс. Режим доступу: https://www.example.com/ecdsa-application (дата звернення 01.11.2024).

45. Стаття: «RSA та методи захисту від багаторазовості підпису» – Електронний ресурс. Режим доступу: https://www.example.com/rsa-replay-attack (дата звернення 01.11.2024).

46. Стаття: «ECDSA: захист від багаторазовості підпису та можливі уразливості» – Електронний ресурс. Режим доступу: https://www.example.com/ecdsa-replay-attack (дата звернення 01.11.2024).

47. Стаття: «SM2: механізми захисту від багаторазовості підпису» – Електронний ресурс. Режим доступу: https://www.example.com/sm2-replay-attack (дата звернення 01.11.2024).

ДОДАТОК А

КОДИ ПРОГРАМ

Код для порівняння функцій хешуваня

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Org.BouncyCastle.Crypto.Digests;

class Program
{
    static void Main()
    {
        string input = «Test input string»;
        int[] messageCounts = { 10, 100, 1000, 1000000 };

        // Словник хеш-функцій із відповідними розмірами
        var hashFunctions = new (string Name, Func<string, int, string> ComputeHash,
int[] Sizes)[]
        {
            («Kupyna», ComputeKupynaHash, new[] { 256, 384, 512 }),
            («Blake2», ComputeBlake2Hash, new[] { 160, 256, 384, 512 }),
            («SHA3», ComputeSha3Hash, new[] { 224, 256, 384, 512 }),
            («SM3», ComputeSM3Hash, new[] { 256 }),
            («Whirlpool», ComputeWhirlpoolHash, new[] { 512 })
        };

        foreach (var (name, computeHash, sizes) in hashFunctions)
        {
            foreach (var size in sizes)
            {
                TestHashFunction($»{name}-{size}», input, computeHash, size,
messageCounts);
            }
        }
    }

    static void TestHashFunction(string name, string input, Func<string, int, string>
computeHash, int bitLength, int[] messageCounts)
    {
```

```
            Console.WriteLine($»Testing {name}:»);
            TestPerformance(name, computeHash, input, bitLength, messageCounts);
            TestAvalancheEffect(name, computeHash, bitLength, 1000000);
            Console.WriteLine();
        }

        static void TestPerformance(string name, Func<string, int, string> computeHash,
string input, int bitLength, int[] messageCounts)
        {
            foreach (var count in messageCounts)
            {
                MeasureExecutionTime($»{name} - Sequential {count} messages», () =>
                {
                    for (int i = 0; i < count; i++) computeHash(input, bitLength);
                });

                MeasureExecutionTime($»{name} - Parallel {count} messages», () =>
                {
                    Parallel.For(0, count, _ => computeHash(input, bitLength));
                });
            }
        }

        static void MeasureExecutionTime(string description, Action action)
        {
            Stopwatch sw = Stopwatch.StartNew();
            action();
            sw.Stop();
            Console.WriteLine($»{description}: {sw.Elapsed.TotalMilliseconds} ms»);
        }

        static void TestAvalancheEffect(string name, Func<string, int, string> computeHash,
int bitLength, int numberOfPairs)
        {
            Random random = new Random();
            int totalHammingDistance = Enumerable.Range(0, numberOfPairs).Sum(_ =>
            {
                byte[] inputBytes = new byte[bitLength / 8];
                random.NextBytes(inputBytes);
                string original = Encoding.UTF8.GetString(inputBytes);
                string modified = ModifyRandomBit(original);
                string hashOriginal = computeHash(original, bitLength);
                string hashModified = computeHash(modified, bitLength);
                return CalculateHammingDistance(hashOriginal, hashModified);
            });
```

```
            double averageHammingDistance = (double)totalHammingDistance / numberOfPairs;
            Console.WriteLine($»{name}       -       Average       Hamming       Distance:
{averageHammingDistance}»);
        }

        static string ModifyRandomBit(string input)
        {
            Random random = new Random();
            byte[] bytes = Encoding.UTF8.GetBytes(input);
            int byteIndex = random.Next(bytes.Length);
            int bitIndex = random.Next(8);
            bytes[byteIndex] ^= (byte)(1 << bitIndex);
            return Encoding.UTF8.GetString(bytes);
        }

        static int CalculateHammingDistance(string hash1, string hash2) =>
            hash1.Zip(hash2, (c1, c2) => CountBits((byte)(c1 ^ c2))).Sum();

        static int CountBits(byte value) =>
            Enumerable.Range(0, 8).Count(i => (value & (1 << i)) != 0);

        public   static   string   ComputeKupynaHash(string   input,   int   bitLength)   =>
ComputeHash(input, bitLength, b => new Dstu7564Digest(b));
        public   static   string   ComputeBlake2Hash(string   input,   int   bitLength)   =>
ComputeHash(input, bitLength, b => new Blake2bDigest(b));
        public   static   string   ComputeSha3Hash(string   input,   int   bitLength)   =>
ComputeHash(input, bitLength, b => new Sha3Digest(b));
        public   static   string   ComputeSM3Hash(string   input,   int   bitLength)   =>
ComputeHash(input, bitLength, _ => new SM3Digest());
        public   static   string   ComputeWhirlpoolHash(string   input,   int   bitLength)   =>
ComputeHash(input, bitLength, _ => new WhirlpoolDigest());

        static   string   ComputeHash(string   input,   int   bitLength,   Func<int,   IDigest>
digestFactory)
        {
            var digest = digestFactory(bitLength);
            byte[] inputBytes = Encoding.UTF8.GetBytes(input);
            digest.BlockUpdate(inputBytes, 0, inputBytes.Length);
            byte[] result = new byte[digest.GetDigestSize()];
            digest.DoFinal(result, 0);
            return BitConverter.ToString(result).Replace(«-», ««).ToLowerInvariant();
        }
    }
```

Utils.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace SM23Cryptography
{
    public static class Utils
    {
        public static string LeftPadWithZero(this string input, int length)
        {
            return input.PadLeft(length, '0');
        }

        public static IEnumerable<T> SubArray<T>(this IEnumerable<T> sequence,
int startIndex, int endIndex)
        {
            return sequence.Skip(startIndex).Take(endIndex - startIndex);
        }

        public static IEnumerable<byte> AddByte(this IEnumerable<byte> bytes,
byte additionalByte)
        {
            return bytes.Append(additionalByte);
        }

        public static string BytesToHex(byte[] bytes)
        {
            return string.Concat(bytes.Select(b => b.ToString(«x2»)));
        }

        public static byte HexStringToByte(string hex)
        {
            return byte.Parse(hex,
System.Globalization.NumberStyles.HexNumber);
        }

        public static byte[] HexStringToByteArray(string hex)
        {
            if (hex.Length % 2 != 0) hex = hex.PadLeft(hex.Length + 1, '0');
```

```
                return Enumerable.Range(0, hex.Length / 2)
                                 .Select(i => HexStringToByte(hex.Substring(i * 2,
2)))
                                 .ToArray();
            }

            public static Tuple<EllipticCurve, CurvePoint, BigInteger>
GetCurveParameters()
            {
                var p = new
BigInteger(«FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF», 16);
                var a = new
BigInteger(«FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFC», 16);
                var b = new
BigInteger(«28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93», 16);

                var curve = new EllipticCurve(p, a, b);
                var generatorX =
«32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7»;
                var generatorY =
«BC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0»;
                var G = curve.DecodePoint(«04» + generatorX + generatorY);
                var n = new
BigInteger(«FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFF7203DF6B21C6052B53BBF40939D54123», 16);
                return new Tuple<EllipticCurve, CurvePoint, BigInteger>(curve, G,
n);
            }
        }
        public class CurveFieldElement
        {
            private BigInteger primeModulus;
            private BigInteger value;

            public CurveFieldElement(BigInteger prime, BigInteger val)
            {
                this.primeModulus = prime;
                this.value = val;
            }

            public BigInteger ToBigInteger() => this.value;
```

```
            public CurveFieldElement Add(CurveFieldElement other)
            {
                return new CurveFieldElement(this.primeModulus, (this.value +
other.value) % this.primeModulus);
            }

            public CurveFieldElement Negate()
            {
                return new CurveFieldElement(this.primeModulus, (-this.value) %
this.primeModulus);
            }

            public CurveFieldElement Multiply(CurveFieldElement other)
            {
                return new CurveFieldElement(this.primeModulus, (this.value *
other.value) % this.primeModulus);
            }

            public CurveFieldElement Divide(CurveFieldElement other)
            {
                return new CurveFieldElement(this.primeModulus, (this.value *
other.value.ModInverse(this.primeModulus)) % this.primeModulus);
            }
        }

        public class CurvePoint
        {
            private EllipticCurve curve;
            private CurveFieldElement x, y;
            private BigInteger z;

            public CurvePoint(EllipticCurve curve, CurveFieldElement x,
CurveFieldElement y, BigInteger z = null)
            {
                this.curve = curve;
                this.x = x;
                this.y = y;
                this.z = z ?? BigInteger.One;
            }

            public CurvePoint Add(CurvePoint point)
```

```
        {
            if (this.IsInfinity()) return point;
            if (point.IsInfinity()) return this;

            // Сюди перемістив формули додавання для скорочення
            return new CurvePoint(this.curve, x, y, z); // (спрощено)
        }

        public bool IsInfinity()
        {
            return this.z.IsZero && !this.y.ToBigInteger().IsZero;
        }
    }

    public class EllipticCurve
    {
        public BigInteger Prime { get; private set; }
        public CurveFieldElement A { get; private set; }
        public CurveFieldElement B { get; private set; }

        public EllipticCurve(BigInteger prime, BigInteger a, BigInteger b)
        {
            this.Prime = prime;
            this.A = new CurveFieldElement(prime, a);
            this.B = new CurveFieldElement(prime, b);
        }

        public CurvePoint DecodePoint(string hex)
        {
            // Розшифровка точки (частину функцій скоротив)
            return null;
        }
    }
}
```

Код реалізації SM3

```
using System;
using System.Linq;
using System.Text;
namespace SM23Crypto
```

```
    {
        public class SM3
        {
            private static byte[] RotateLeft(byte[] x, int n)
            {
                byte[] result = new byte[x.Length];
                int byteShift = n / 8;
                int bitShift = n % 8;
                var len = x.Length;
                for (var i = 0; i < len; i++)
                {
                    result[i] = (byte)(((x[(i + byteShift) % len] << bitShift) &
0xff) + ((int)((uint)x[(i + byteShift + 1) % len] >> (8 - bitShift)) & 0xff));
                }
                return result;
            }

            private static byte[] P1(byte[] X)
            {
                return X.Xor(X.RotateLeft(15)).Xor(X.RotateLeft(23));
            }

            private static byte[] P0(byte[] X)
            {
                return X.Xor(X.RotateLeft(9)).Xor(X.RotateLeft(17));
            }

            private static byte[] Add(byte[] x, byte[] y)
            {
                return x.AddBytes(y);
            }

            private static byte[] FF(byte[] X, byte[] Y, byte[] Z, int j)
            {
                return   j   >=   0   &&   j   <=   15   ?   X.Xor(Y).Xor(Z)   :
X.And(Y).Or(X.And(Z)).Or(Y.And(Z));
            }

            private static byte[] GG(byte[] X, byte[] Y, byte[] Z, int j)
            {
```

```
                return j >= 0 && j <= 15 ? X.Xor(Y).Xor(Z) :
X.And(Y).Or(X.Not().And(Z));
            }

            private static byte[] CF(byte[] V, byte[] Bi)
            {
                byte[][] W = new byte[68][];
                for (int i = 0; i < 68; i++) W[i] = new byte[4];
                byte[][] M = new byte[64][];
                for (int i = 0; i < 64; i++) M[i] = new byte[4];
                for (var i = 0; i < 16; i++)
                {
                    int start = i * 4;
                    Array.Copy(Bi, start, W[i], 0, 4);
                }

                for (var j = 16; j < 68; j++)
                {
                    var value = W[j - 16].Xor(W[j - 9]).Xor(W[j -
3].RotateLeft(15)).P1().Xor(W[j - 13].RotateLeft(7)).Xor(W[j - 6]);
                    Array.Copy(value, 0, W[j], 0, 4);
                }

                for (var j = 0; j < 64; j++)
                {
                    var value = W[j].Xor(W[j + 4]);
                    Array.Copy(value, 0, M[j], 0, 4);
                }

                byte[] A = V.Slice(0, 4);
                byte[] B = V.Slice(4, 8);
                byte[] C = V.Slice(8, 12);
                byte[] D = V.Slice(12, 16);
                byte[] E = V.Slice(16, 20);
                byte[] F = V.Slice(20, 24);
                byte[] G = V.Slice(24, 28);
                byte[] H = V.Slice(28, 32);

                for (int j = 0; j < 64; j++)
                {
```

```
                    byte[] T = j >= 0 && j <= 15 ? new byte[] { 0x79, 0xcc, 0x45,
0x19 } : new byte[] { 0x7a, 0x87, 0x9d, 0x8a };
                    byte[]                          SS1                          =
A.RotateLeft(12).Add(E).Add(T.RotateLeft(j)).RotateLeft(7);
                    byte[] SS2 = SS1.Xor(A.RotateLeft(12));
                    byte[] TT1 = FF(A, B, C, j).Add(D).Add(SS2).Add(M[j]);
                    byte[] TT2 = GG(E, F, G, j).Add(H).Add(SS1).Add(W[j]);
                    D = C;
                    C = B.RotateLeft(9);
                    B = A;
                    A = TT1;
                    H = G;
                    G = F.RotateLeft(19);
                    F = E;
                    E = TT2.P0();
                }

                return
A.Concat(B).Concat(C).Concat(D).Concat(E).Concat(F).Concat(G).Concat(H).ToArray().Xor
(V);
            }

            public static string StrSum(string str)
            {
                return
SMUtils.BytesToHex(SM3Array(Encoding.UTF8.GetBytes(str))).ToUpper();
            }

            public static byte[] SM3Array(byte[] arr)
            {
                int len = arr.Length * 8;
                int k = len % 512;
                k = k >= 448 ? 512 - (k % 448) - 1 : 448 - k - 1;
                int zeroLen = (k - 7) / 8;

                byte[] zeroArr = new byte[zeroLen];
                var lenArr = BitConverter.GetBytes((ulong)len).Reverse().ToArray();
                var oneBlock = new byte[] { 0x80 };
                byte[]                              m                              =
arr.Concat(oneBlock).Concat(zeroArr).Concat(lenArr).ToArray();
```

```
            int n = m.Length / 64;
            byte[] V = new byte[32]
            {
                0x73, 0x80, 0x16, 0x6f, 0x49, 0x14, 0xb2, 0xb9,
                0x17, 0x24, 0x42, 0xd7, 0xda, 0x8a, 0x06, 0x00,
                0xa9, 0x6f, 0x30, 0xbc, 0x16, 0x31, 0x38, 0xaa,
                0xe3, 0x8d, 0xee, 0x4d, 0xb0, 0xfb, 0x0e, 0x4e
            };

            for (var i = 0; i < n; i++)
            {
                byte[] B = m.Skip(i * 64).Take(64).ToArray();
                V = CF(V, B);
            }
            return V;
        }
    }
}
```

Код реалізації SM2

```
using System;
using System.Globalization;
using System.Linq;
using System.Text;

namespace SM23Crypto
{
    public class SM2Res
    {
        private byte[] _data;

        public SM2Res(byte[] data)
        {
            this._data = data;
        }

        public byte[] GetBytes()
        {
            return this._data;
```

```
        }

        public string ToString()
        {
            return Encoding.ASCII.GetString(this._data);
        }
    }

    public class SM2Key
    {
        public string PubKey { get; set; }
        public string PriKey { get; set; }
    }

    public class SM2
    {
        public enum CipherMode
        {
            C1C2C3 = 0,
            C1C3C2 = 1,
        }

        public static SM2Key GenerateKeyPairHex()
        {
            Tuple<ECCurveFp, ECPointFp, BigInteger> curve =
SMUtils.getGlobalCurve();
            Random rand = new Random();
            BigInteger random = new BigInteger(256, rand);
            BigInteger d =
random.Mod(curve.Item3.Subtract(BigInteger.One)).Add(BigInteger.One); // 随机数
            string privateKey = d.ToString(16).PadLeft(64, '0');

            ECPointFp P = curve.Item2.multiply(d); // P = dG，p 为公钥，d 为私钥
            string Px = P.getX().toBigInteger().ToString(16).PadLeftZero(64);
            string Py = P.getY().toBigInteger().ToString(16).PadLeftZero(64);
            string publicKey = «04» + Px + Py;
            return new SM2Key()
            {
                PriKey = privateKey,
                PubKey = publicKey
```

```
            };
        }

        public static string DoEncrypt(string msg, string publicKey, CipherMode
cipherMode = CipherMode.C1C3C2)
        {
            byte[] msgBytes = Encoding.ASCII.GetBytes(msg);  // Використовуємо
ASCII для перетворення
            ECPointFp                    publicKeyPoint                    =
SMUtils.getGlobalCurve().Item1.decodePointHex(publicKey);
            SM2Key keypair = GenerateKeyPairHex();
            BigInteger k = new BigInteger(keypair.PriKey, 16);
            string c1 = keypair.PubKey;
            if (c1.Length > 128)
            {
                c1 = c1.Substring(c1.Length - 128);
            }
            ECPointFp p = publicKeyPoint.multiply(k);
            byte[]                           x2                           =
SMUtils.hexToArray(p.getX().toBigInteger().ToString(16).PadLeftZero(64));
            byte[]                           y2                           =
SMUtils.hexToArray(p.getY().toBigInteger().ToString(16).PadLeftZero(64));
            string                           c3                           =
SMUtils.arrayToHex(SM3.SM3Array(x2.Concat(msgBytes).Concat(y2).ToArray()));
            int ct = 1;
            int offset = 0;
            byte[] t = new byte[0];
            byte[] z = x2.Concat(y2).ToArray();
            byte[] n = new[] { (byte)(ct >> 24 & 0x00ff), (byte)(ct >> 16 &
0x00ff), (byte)(ct >> 8 & 0x00ff), (byte)(ct & 0x00ff) };
            t = SM3.SM3Array(z.Concat(n).ToArray());
            ct++;
            offset = 0;
            int len = msgBytes.Length;
            for (int i = 0; i < len; i++)
            {
                if (offset == t.Length)
                {
                    t = SM3.SM3Array(z.Concat(n).ToArray());
                    ct++;
                    offset = 0;
```

```
                    }
                    msgBytes[i] ^= (byte)(t[offset++] & 0xff);
                }

                string c2 = SMUtils.arrayToHex(msgBytes);

                return (cipherMode == CipherMode.C1C3C2 ? c1 + c2 + c3 : c1 + c3 +
c2).ToLower();
            }

            public static SM2Res DoDecrypt(string encryptData, string privateKey,
CipherMode cipherMode = CipherMode.C1C3C2)
            {
                BigInteger privateKeyI = new BigInteger(privateKey, 16);
                string c3 = encryptData.Substring(128, 64);
                string c2 = encryptData.Substring(128 + 64);
                if (cipherMode == CipherMode.C1C2C3)
                {
                    c3 = encryptData.Substring(encryptData.Length - 64);
                    c2 = encryptData.Substring(128, encryptData.Length - 128 - 64);
                }

                byte[] msg = SMUtils.hexToArray(c2);
                var sss = encryptData.Substring(0, 128);
                ECPointFp c1 = SMUtils.getGlobalCurve().Item1.decodePointHex(«04» +
encryptData.Substring(0, 128));
                ECPointFp p = c1.multiply(privateKeyI);
                byte[]                              x2                              =
SMUtils.hexToArray(p.getX().toBigInteger().ToString(16).PadLeft(64, '0'));
                byte[]                              y2                              =
SMUtils.hexToArray(p.getY().toBigInteger().ToString(16).PadLeft(64, '0'));
                var ct = 1;
                var offset = 0;
                byte[] z = x2.Concat(y2).ToArray();
                byte[] n = new[] { (byte)(ct >> 24 & 0x00ff), (byte)(ct >> 16 &
0x00ff), (byte)(ct >> 8 & 0x00ff), (byte)(ct & 0x00ff) };
                var t = SM3.SM3Array(z.Concat(n).ToArray());
                ct++;
                offset = 0;
                for (int i = 0, len = msg.Length; i < len; i++)
                {
```

```
                    if (offset == t.Length)
                    {
                        t = SM3.SM3Array(z.Concat(n).ToArray()); ;
                        ct++;
                        offset = 0;
                    }
                    msg[i] ^= (byte)(t[offset++] & 0xff);
                }
                byte[] tt = new byte[0];
                var                          checkC3                          =
SMUtils.arrayToHex(SM3.SM3Array(tt.Concat(x2).Concat(msg).Concat(y2).ToArray()));
                if (checkC3 == c3.ToLower())
                {
                    return new SM2Res(msg);
                }
                else
                {
                    return new SM2Res(Encoding.ASCII.GetBytes(««));
                }
            }
        }
    }
```

Код реалізації підпису на RSA

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

    class Program
    {
        static void Main(string[] args)
        {
            // Генерація пари ключів RSA
            RSA rsa = RSA.Create();
            rsa.KeySize = 2048;  // Розмір ключа

            // Отримання приватного та публічного ключів
            string privateKey = Convert.ToBase64String(rsa.ExportRSAPrivateKey());
            string publicKey = Convert.ToBase64String(rsa.ExportRSAPublicKey());
```

```
            Console.WriteLine(«Приватний ключ (d):»);
            Console.WriteLine(privateKey);

            Console.WriteLine(«\nПублічний ключ (P):»);
            Console.WriteLine(publicKey);

            Console.Write(«\nВведіть повідомлення для підпису: «);
            string message = Console.ReadLine();

            // Обчислення хешу повідомлення за допомогою MD5
            using (MD5 md5 = MD5.Create())
            {
                byte[] msgBytes = Encoding.UTF8.GetBytes(message);
                byte[] hash = md5.ComputeHash(msgBytes);
                string    hashStr    =    BitConverter.ToString(hash).Replace(«-»,
««).ToLower();
                Console.WriteLine($»\nХеш повідомлення (H): {hashStr}»);
            }


            Random rand = new Random();
            BigInteger r = new BigInteger(256, rand);
            Console.WriteLine($»\nФактор засліплення (r): {r.ToString(16)}»);

            // Засліплений хеш
            BigInteger hBlind = new BigInteger(hashStr, 16) * r;
            Console.WriteLine($»\nЗасліплений            хеш            (Hblind):
{hBlind.ToString(16)}»);

            // Підпис
            byte[]   signedHash   =   rsa.SignData(hash,   HashAlgorithmName.MD5,
RSASignaturePadding.Pkcs1);
            Console.WriteLine($»\nПідпис                              повідомлення:
{Convert.ToBase64String(signedHash)}»);

            // Перевірка підпису
            Console.Write(«\nПеревірка! Введіть повідомлення для підпису: «);
            string messageToVerify = Console.ReadLine();

            using (MD5 md5Verify = MD5.Create())
```

```
            {
                byte[] msgBytesVerify = Encoding.UTF8.GetBytes(messageToVerify);
                byte[] hashVerify = md5Verify.ComputeHash(msgBytesVerify);
                string hashVerifyStr = BitConverter.ToString(hashVerify).Replace(«-
», ««).ToLower();
                Console.WriteLine($»\nХеш повідомлення для перевірки (H):
{hashVerifyStr}»);

                bool isValid = rsa.VerifyData(hashVerify, signedHash,
HashAlgorithmName.MD5, RSASignaturePadding.Pkcs1);
                Console.WriteLine(«\nПеревірка підпису:»);
                if (isValid)
                {
                    Console.WriteLine(«Підпис дійсний!»);
                }
                else
                {
                    Console.WriteLine(«Помилка. Повідомлення було змінено!»);
                }
            }
        }
    }
```

Код реалізації підпису на ECDSA

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

    class Program
    {
        static void Main(string[] args)
        {
            // Генерація пари ключів ECDSA
            using (ECDsa ecdsa = ECDsa.Create(ECCurve.NamedCurves.nistP256))
            {
                // Отримання приватного та публічного ключів
                byte[] privateKey = ecdsa.ExportECPrivateKey();
                byte[] publicKey = ecdsa.ExportSubjectPublicKeyInfo();
```

```
                Console.WriteLine(«Приватний ключ (d):»);
                Console.WriteLine(Convert.ToBase64String(privateKey));

                Console.WriteLine(«\nПублічний ключ (P):»);
                Console.WriteLine(Convert.ToBase64String(publicKey));

                Console.Write(«\nВведіть повідомлення для підпису: «);
                string message = Console.ReadLine();

                // Обчислення хешу повідомлення за допомогою SHA256
                using (SHA256 sha256 = SHA256.Create())
                {
                    byte[] msgBytes = Encoding.UTF8.GetBytes(message);
                    byte[] hash = sha256.ComputeHash(msgBytes);
                    string   hashStr   =   BitConverter.ToString(hash).Replace(«-», ««).ToLower();
                    Console.WriteLine($»\nХеш повідомлення (H): {hashStr}»);
                }

                Random rand = new Random();
                BigInteger r = new BigInteger(256, rand);
                Console.WriteLine($»\nФактор засліплення (r): {r.ToString(16)}»);

                // Засліплений хеш
                BigInteger hBlind = new BigInteger(hashStr, 16) * r;
                Console.WriteLine($»\nЗасліплений          хеш          (Hblind): {hBlind.ToString(16)}»);

                // Підпис
                byte[] signedHash = ecdsa.SignHash(hash);
                Console.WriteLine($»\nПідпис                          повідомлення: {Convert.ToBase64String(signedHash)}»);

                // Перевірка підпису
                Console.Write(«\nПеревірка! Введіть повідомлення для підпису: «);
                string messageToVerify = Console.ReadLine();

                using (SHA256 sha256Verify = SHA256.Create())
                {
                    byte[]                    msgBytesVerify                    = Encoding.UTF8.GetBytes(messageToVerify);
```

```
                    byte[] hashVerify = sha256Verify.ComputeHash(msgBytesVerify);
                    string                    hashVerifyStr                    =
BitConverter.ToString(hashVerify).Replace(«-», ««).ToLower();
                    Console.WriteLine($»\nХеш  повідомлення  для  перевірки  (H):
{hashVerifyStr}»);

                    bool isValid = ecdsa.VerifyHash(hashVerify, signedHash);
                    Console.WriteLine(«\nПеревірка підпису:»);
                    if (isValid)
                    {
                        Console.WriteLine(«Підпис дійсний!»);
                    }
                    else
                    {
                        Console.WriteLine(«Помилка. Повідомлення було змінено!»);
                    }
                }
            }
        }
    }
```

Реалізація сліпого підпису

```
    using SM23Crypto;
    using System;
    using System.Text;

    class Program
    {
        static void Main(string[] args)
        {
            SM2Key keyPair = SM2.GenerateKeyPairHex();
            var curve = SMUtils.getGlobalCurve();
            var n = curve.Item3;
            var g = curve.Item2;
            BigInteger d = new BigInteger(keyPair.PriKey, 16);
            var publicKey = g.multiply(d);
            Console.WriteLine($»\nПублічний ключ (P): {publicKey}»);
            Console.Write(«\nВведіть повідомлення для підпису: «);
            string message = Console.ReadLine();
```

```
            byte[] msgBytes = Encoding.UTF8.GetBytes(message);
            byte[] hash = SM3.SM3Array(msgBytes);
            BigInteger h = new BigInteger(SMUtils.arrayToHex(hash), 16);
            Console.WriteLine($»\nХеш повідомлення (H): {h.ToString(16)}»);

            Random rand = new Random();
            BigInteger r = new BigInteger(256,
rand).Mod(n.Subtract(BigInteger.One)).Add(BigInteger.One);
            Console.WriteLine($»\nФактор засліплення (r): {r.ToString(16)}»);

            BigInteger hBlind = h.Multiply(r).Mod(n);
            Console.WriteLine($»\nЗасліплений хеш (Hblind):
{hBlind.ToString(16)}»);

            BigInteger sBlind = d.Multiply(hBlind).Mod(n);
            Console.WriteLine($»\nПідпис засліпленого хешу (Sblind):
{sBlind.ToString(16)}»);

            BigInteger rInv = r.ModInverse(n);
            BigInteger s = sBlind.Multiply(rInv).Mod(n);
            Console.WriteLine($»\nРозсліплений підпис (S): {s.ToString(16)}»);

            ECPointFp sG = g.multiply(s);
            ECPointFp hP = publicKey.multiply(h);

            Console.WriteLine(«\nПеревірка підпису:»);
            if (sG.equals(hP))
            {
                Console.WriteLine(«Підпис дійсний!»);
            }
            else
            {
                Console.WriteLine(«Помилка. Повідомлення було змінено!»);
            }

            Console.Write(«\nПеревірка! Введіть повідомлення для підпису: «);
            message = Console.ReadLine();

            byte[] msgBytes2 = Encoding.UTF8.GetBytes(message);
            byte[] hash2 = SM3.SM3Array(msgBytes2);
            BigInteger h2 = new BigInteger(SMUtils.arrayToHex(hash2), 16);
```

```
        Console.WriteLine($»\nХэш сообщения (H): {h.ToString(16)}»);

        sG = g.multiply(s);
        hP = g.multiply(h2).multiply(d);

        Console.WriteLine(«\nПеревірка підпису:»);
        if (sG.equals(hP))
        {
            Console.WriteLine(«Підпис дійсний!»);
        }
        else
        {
            Console.WriteLine(«Помилка. Повідомлення було змінено!»);
        }

    }
}
```

Код реалізації програми сліпого підпису із інтерфейсом користувача

Form1.cs:

```
using SM2_Sign.Properties;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace SM2_Sign
{
    public partial class Form1 : Form
```

```
        {
            SignOperations signOperations = new SignOperations();
            public Form1()
            {
                InitializeComponent();
                GenerateKeys();
                GenerateBlindFactor();
                string language = LoadLanguage();
                SetLanguage(language);
                UpdateLanguage();
                українськаToolStripMenuItem.Checked = language == «uk-UA»;
                англійськаToolStripMenuItem.Checked = language == «en-US»;
                button_saveSing.Enabled = false;
            }
            private void GenerateKeys()
            {
                signOperations.GenerateKeys();
                textBox_publicKey.Text = signOperations.GetPublicKey();
                button_savePublicKey.Enabled = true;
            }

            private void зберегтиToolStripMenuItem_Click(object sender, EventArgs
e)
            {
                using (SaveFileDialog saveFileDialog = new SaveFileDialog())
                {
                    saveFileDialog.Filter = «Text Files (*.txt)|*.txt|All Files
(*.*)|*.*»;
                    if (saveFileDialog.ShowDialog() == DialogResult.OK)
                    {
                        File.WriteAllText(saveFileDialog.FileName,
richTextBox_message.Text);
                    }
                }
            }

            private void вихідToolStripMenuItem_Click(object sender, EventArgs e)
            {
                Application.Exit();
            }
```

```
        private void копіюватиToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            if (!string.IsNullOrEmpty(richTextBox_message.SelectedText))
            {
                Clipboard.SetText(richTextBox_message.SelectedText);
            }
        }

        private void вирізатиToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            if (!string.IsNullOrEmpty(richTextBox_message.SelectedText))
            {
                Clipboard.SetText(richTextBox_message.SelectedText);
                richTextBox_message.SelectedText = string.Empty;
            }
        }

         private void вставитиToolStripMenuItem_Click(object sender, EventArgs
e)
         {
             if (Clipboard.ContainsText())
             {
                 richTextBox_message.SelectedText = Clipboard.GetText();
             }
         }

         private void очиститиToolStripMenuItem_Click(object sender, EventArgs
e)
         {
             var result = MessageBox.Show(
                 «Ви впевнені, що хочете очистити весь текст?»,
                 «Підтвердження»,
                 MessageBoxButtons.YesNo,
                 MessageBoxIcon.Warning
             );
             if (result == DialogResult.Yes)
             {
                 richTextBox_message.Clear();
             }
```

```
        }

        private void українськаToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            SetLanguage(«uk-UA»);
            SaveLanguage(«uk-UA»);
            українськаToolStripMenuItem.Checked = true;
            англійськаToolStripMenuItem.Checked = false;
        }
        private void англійськаToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            SetLanguage(«en-US»);
            SaveLanguage(«en-US»);
            українськаToolStripMenuItem.Checked = false;
            англійськаToolStripMenuItem.Checked = true;
        }
        private void SaveLanguage(string cultureCode)
        {
            Properties.Settings.Default.AppLanguage = cultureCode;
            Properties.Settings.Default.Save();
        }
        private string LoadLanguage()
        {
            return Properties.Settings.Default.AppLanguage;
        }

        private void SetLanguage(string cultureCode)
        {
            CultureInfo cultureInfo = new CultureInfo(cultureCode);
            Thread.CurrentThread.CurrentCulture = cultureInfo;
            Thread.CurrentThread.CurrentUICulture = cultureInfo;
            UpdateLanguage();
        }
        public void UpdateLanguage()
        {
            this.Text = Resources.Language.ProgramName;
            this.label_ECDAta.Text = Resources.Language.KeysLabel;
            this.label_publicKey.Text = Resources.Language.PublicLabel;
            this.button_savePublicKey.Text = Resources.Language.SavePublicKey;
```

```
                this.button5.Text = Resources.Language.GenerateKeyButton;
                this.label_hiddenFactor.Text = Resources.Language.HidingLanel;
                this.button_generateNew.Text = Resources.Language.GenerateNewBlind;
                this.label_hash.Text = Resources.Language.HashLabel;
                this.label_bindHash.Text = Resources.Language.BlindingHashLabel;
                this.button_calculateHash.Text                                    =
Resources.Language.CalculateHashButton;
                this.label_blindSign.Text = Resources.Language.BlindedSignLabel;
                this.label_sing.Text = Resources.Language.SignLabel;
                this.button_saveSing.Text = Resources.Language.SaveSignButton;
                this.button_sing.Text = Resources.Language.SignButton;
                this.файлToolStripMenuItem.Text = Resources.Language.FileMenu;
                this.відкритиToolStripMenuItem.Text = Resources.Language.OpenMenu;
                this.зберегтиToolStripMenuItem.Text = Resources.Language.SaveMenu;
                this.копіюватиToolStripMenuItem.Text = Resources.Language.CopyMenu;
                this.вихідToolStripMenuItem.Text = Resources.Language.QuitMenu;
                this.редагуватиToolStripMenuItem.Text                             =
Resources.Language.EditMenu;
                this.вирізатиToolStripMenuItem.Text = Resources.Language.CutMenu;
                this.копіюватиToolStripMenuItem.Text = Resources.Language.CopyMenu;
                this.вставитиToolStripMenuItem.Text = Resources.Language.PasteMenu;
                this.перевіритиПідписToolStripMenuItem.Text                       =
Resources.Language.CheckMenu;
                this.проПрограмуToolStripMenuItem.Text                            =
Resources.Language.AboutMenu;
                this.моваToolStripMenuItem.Text = Resources.Language.LanguageMenu;

            }
            private  void  перевіритиПідписToolStripMenuItem_Click(object  sender,
EventArgs e)
            {
                CheckSing checkSing = new CheckSing();
                checkSing.ShowDialog();
            }

            private    void    проПрограмуToolStripMenuItem_Click(object    sender,
EventArgs e)
            {
                About about = new About();
                about.ShowDialog();
            }
```

```
        private void відкритиToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            using (OpenFileDialog openFileDialog = new OpenFileDialog())
            {
                openFileDialog.Filter = «Text Files (*.txt)|*.txt|All Files
(*.*)|*.*»;
                if (openFileDialog.ShowDialog() == DialogResult.OK)
                {
                    richTextBox_message.Text                                =
File.ReadAllText(openFileDialog.FileName);
                }
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }

        private void button5_Click(object sender, EventArgs e)
        {
            var result = MessageBox.Show(
      «Ви впевнені, що хочете згенерувати нову пару ключів?»,
      «Підтвердження»,
      MessageBoxButtons.YesNo,
      MessageBoxIcon.Warning
  );
            if (result == DialogResult.Yes)
            {
                GenerateKeys();
            }
        }

        private void button_savePublicKey_Click(object sender, EventArgs e)
        {
            string publicKey = textBox_publicKey.Text;

            if (string.IsNullOrWhiteSpace(publicKey))
            {
```

```
                    MessageBox.Show(«Ключ відсутній.», «Ошибка»,
MessageBoxButtons.OK, MessageBoxIcon.Warning);
                    return;
                }
                using (SaveFileDialog saveFileDialog = new SaveFileDialog())
                {
                    saveFileDialog.Filter = «Public Key Files
(*.publicKey)|*.publicKey|All Files (*.*)|*.*»;
                    saveFileDialog.DefaultExt = «publicKey»;
                    saveFileDialog.Title = «Зберегти ключ»;
                    if (saveFileDialog.ShowDialog() == DialogResult.OK)
                    {
                        try
                        {
                            System.IO.File.WriteAllText(saveFileDialog.FileName,
publicKey);
                            MessageBox.Show(«Ключ успішно збережено!», «Успіх»,
MessageBoxButtons.OK, MessageBoxIcon.Information);
                        }
                        catch (Exception ex)
                        {
                            MessageBox.Show($»Помилка при збереженні ключа:
{ex.Message}», «Ошибка», MessageBoxButtons.OK, MessageBoxIcon.Error);
                        }
                    }
                }
            }

            private void button_generateNew_Click(object sender, EventArgs e)
            {
                var result = MessageBox.Show(
      «Ви впевнені, що хочете згенерувати новий фактор засліплення?»,
      «Підтвердження»,
      MessageBoxButtons.YesNo,
      MessageBoxIcon.Warning
      );
                if (result == DialogResult.Yes)
                {
                    GenerateBlindFactor();
                }
            }
```

```
        private void GenerateBlindFactor()
        {
            signOperations.GenerateBlind();
            textBox_rKey.Text = signOperations.GetBlindFactor().ToString();
        }

        private void button_calculateHash_Click(object sender, EventArgs e)
        {
            HashMessage();
        }

        private void HashMessage()
        {
            signOperations.SetMessage(richTextBox_message.Text);
            signOperations.HashMessage();
            signOperations.BlindHash();
            textBox_blindHash.Text = signOperations.GetBlindHash().ToString();
            textBox_hash.Text = signOperations.GetStringHash();
        }

        private void button_sing_Click(object sender, EventArgs e)
        {
            HashMessage();
            signOperations.GenerateBlindedSign();
            textBox_blindSign.Text                                                    =
signOperations.GetBlindedSign().ToString();
            signOperations.UnblindSign();
            textBox_Sign.Text = signOperations.GetSign().ToString();
        }

        private void button_saveSing_Click(object sender, EventArgs e)
        {
            string sign = textBox_Sign.Text;

            if (string.IsNullOrWhiteSpace(sign))
            {
                MessageBox.Show(«Підпис         відсутній!»,         «Помилка»,
MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }
            using (SaveFileDialog saveFileDialog = new SaveFileDialog())
            {
                saveFileDialog.Filter      =      «Blind      Sign      Files
(*.Blindsign)|*.blindSign|All Files (*.*)|*.*»;
                saveFileDialog.DefaultExt = «blindSign»;
                saveFileDialog.Title = «Зберегти підпис»;
                if (saveFileDialog.ShowDialog() == DialogResult.OK)
                {
```

```
                        try
                        {
                            System.IO.File.WriteAllText(saveFileDialog.FileName,
sign);
                            MessageBox.Show(«Підпис успішно збережено!», «Успіх»,
MessageBoxButtons.OK, MessageBoxIcon.Information);
                        }
                        catch (Exception ex)
                        {
                            MessageBox.Show($»Помилка при збереженні ключа:
{ex.Message}», «Помилка», MessageBoxButtons.OK, MessageBoxIcon.Error);
                        }
                    }
                }
            }

            private void textBox_Sign_TextChanged(object sender, EventArgs e)
            {
                if (textBox_Sign.Text.Length > 0)
                    button_saveSing.Enabled = true;
                else
                    button_saveSing.Enabled = false;
            }

            private void textBox_publicKey_TextChanged(object sender, EventArgs e)
            {
                if (textBox_publicKey.Text.Length > 0)
                    button_savePublicKey.Enabled = true;
                else
                    button_savePublicKey.Enabled = false;
            }

            private void richTextBox_message_TextChanged(object sender, EventArgs
e)
            {
                signOperations.SetMessage(richTextBox_message.Text);
            }

            private void textBox_Sign_TextChanged_1(object sender, EventArgs e)
            {
                if (textBox_Sign.Text.Length > 0)
                    button_saveSing.Enabled = true;
                else
                    button_saveSing.Enabled = false;
            }
        }
    }
```

Список елементів Form1:

```
private System.Windows.Forms.MenuStrip menuStrip1;
private System.Windows.Forms.ToolStripMenuItem файлToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem відкритиToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem зберегтиToolStripMenuItem;
private System.Windows.Forms.ToolStripSeparator toolStripMenuItem1;
private System.Windows.Forms.ToolStripMenuItem вихідToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem редагуватиToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem копіюватиToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem вирізатиToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem вставитиToolStripMenuItem;
private System.Windows.Forms.ToolStripSeparator toolStripMenuItem2;
private System.Windows.Forms.ToolStripMenuItem очиститиToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem моваToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem українськаToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem англійськаToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem проПрограмуToolStripMenuItem;
private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.RichTextBox richTextBox_message;
private System.Windows.Forms.Panel panel2;
private System.Windows.Forms.Button button_sing;
private System.Windows.Forms.Label label3;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label_ECDAta;
private System.Windows.Forms.Label label4;
private System.Windows.Forms.Label label5;
private System.Windows.Forms.TextBox textBox_a;
private System.Windows.Forms.Label label6;
private System.Windows.Forms.TextBox textBox_b;
private System.Windows.Forms.TextBox textBox_p;
private System.Windows.Forms.TextBox textBox_n;
private System.Windows.Forms.TextBox textBox_Gx;
private System.Windows.Forms.TextBox textBox_Gy;
private System.Windows.Forms.TextBox textBox_Sign;
private System.Windows.Forms.Label label_sing;
private System.Windows.Forms.TextBox textBox_blindSign;
```

```
        private System.Windows.Forms.Label label_blindSign;
        private System.Windows.Forms.TextBox textBox_blindHash;
        private System.Windows.Forms.Label label_bindHash;
        private System.Windows.Forms.TextBox textBox_hash;
        private System.Windows.Forms.Label label_hash;
        private System.Windows.Forms.Button button_generateNew;
        private System.Windows.Forms.TextBox textBox_rKey;
        private System.Windows.Forms.Label label8;
        private System.Windows.Forms.Button button_saveSing;
        private System.Windows.Forms.Label label_hiddenFactor;
        private System.Windows.Forms.TextBox textBox_publicKey;
        private System.Windows.Forms.Label label14;
        private System.Windows.Forms.Button button_savePublicKey;
        private System.Windows.Forms.Button button5;
        private System.Windows.Forms.Label label_publicKey;
        private System.Windows.Forms.ToolStripMenuItem
перевіритиПідписToolStripMenuItem;
        private System.Windows.Forms.Button button_calculateHash;
```

CheckSing.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using SM23Crypto;

namespace SM2_Sign
{
    public partial class CheckSing : Form
    {
        string message_data;
        string publicKey_data;
        string sign_data;
        SM2Key keyPair;
```

```
        Tuple<ECCurveFp, ECPointFp, BigInteger> curve;
        ECPointFp g;

        byte[] msgBytes;
        byte[] hash;
        BigInteger h;
        BigInteger s;

        public void GenerateKeys()
        {
            keyPair = SM2.GenerateKeyPairHex();
            curve = SMUtils.getGlobalCurve();
            g = curve.Item2;
        }

        public CheckSing()
        {
            InitializeComponent();
            UpdateLanguage();
            GenerateKeys();
        }
        public void UpdateLanguage ()
        {
            this.Text = Resources.Language.CheckMenu;
            this.label_loadMessage.Text = Resources.Language.OpenMessageLabel;
            this.label1.Text = Resources.Language.OpenPublicKeyLabel;
            this.label2.Text = Resources.Language.OpenSignLabel;
            this.button_openMessage.Text = Resources.Language.MessageButton;
            this.button_publicKeyOpen.Text                                    =
Resources.Language.PublicKeyButton;
            this.button_Opensign.Text = Resources.Language.SignButton;
            this.button_checkSign.Text = Resources.Language.CheckMenu;
        }
        private void button_openMessage_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openFileDialog = new OpenFileDialog())
            {
                openFileDialog.Filter  =  «Text  Files  (*.txt)|*.txt|All  Files
(*.*)|*.*»;
                if (openFileDialog.ShowDialog() == DialogResult.OK)
                {
                    message_data = File.ReadAllText(openFileDialog.FileName);
                    textBox_messagePath.Text   =   openFileDialog.FileName;   //
Відображення шляху до файлу
                }
            }
        }

        private void button_publicKeyOpen_Click(object sender, EventArgs e)
        {
            // Відкриття файлу з відкритим ключем
            using (OpenFileDialog openFileDialog = new OpenFileDialog())
            {
```

```
                    openFileDialog.Filter      =      «Public      Key      Files
(*.publicKey)|*.publicKey|All Files (*.*)|*.*»;
                    if (openFileDialog.ShowDialog() == DialogResult.OK)
                    {
                        publicKey_data = File.ReadAllText(openFileDialog.FileName);
                        textBox_publicKeyPath.Text  =  openFileDialog.FileName;  //
Відображення шляху до файлу
                    }
                }
            }

            private void button_Opensign_Click(object sender, EventArgs e)
            {
                using (OpenFileDialog openFileDialog = new OpenFileDialog())
                {
                    openFileDialog.Filter      =      «Blind      Sign      Files
(*.Blindsign)|*.blindSign|All Files (*.*)|*.*»;
                    if (openFileDialog.ShowDialog() == DialogResult.OK)
                    {
                        sign_data = File.ReadAllText(openFileDialog.FileName);
                        textBox_signPath.Text = openFileDialog.FileName;
                    }
                }
            }

            private void button_checkSign_Click(object sender, EventArgs e)
            {
                ECPointFp    publicKey    =    ECPointFp.Deserialize(publicKey_data,
curve.Item1);
                msgBytes = Encoding.UTF8.GetBytes(message_data);
                s = new BigInteger(sign_data, 10);
                hash = SM3.SM3Array(msgBytes);
                h = new BigInteger(SMUtils.arrayToHex(hash), 16);
                ECPointFp sG = g.multiply(s);
                ECPointFp hP = publicKey.multiply(h);
                if (sG.equals(hP))
                {
                    MessageBox.Show(«Підпис дійсний!», «Успіх!»);
                }
                else
                {
                    MessageBox.Show(«Підпис не дійсний!», «Помилка!»);
                }
            }

            private void CheckSing_Load(object sender, EventArgs e)
            {

            }
        }
    }
```

SignOperations.cs:

```
using System;
using System.Text;
using SM23Crypto;

namespace SM2_Sign
{
    public class SignOperations
    {
        SM2Key keyPair;
        Tuple<ECCurveFp, ECPointFp, BigInteger> curve;
        BigInteger n;
        ECPointFp g;
        BigInteger d;
        ECPointFp publicKey;

        byte[] msgBytes;
        byte[] hash;
        BigInteger h;
        BigInteger r;
        BigInteger hBlind;
        BigInteger sBlind;
        BigInteger rInv;
        BigInteger s;
        public void GenerateKeys()
        {
            keyPair = SM2.GenerateKeyPairHex();
            curve = SMUtils.getGlobalCurve();
            n = curve.Item3;
            g = curve.Item2;
            d = new BigInteger(keyPair.PriKey, 16);
            publicKey = g.multiply(d);
        }
        public string GetPublicKey()
        {
            return publicKey.Serialize();

        }
        public void SetMessage(string message)
        {
            msgBytes = Encoding.UTF8.GetBytes(message);
        }
        public void HashMessage()
        {
            hash = SM3.SM3Array(msgBytes);
            h = new BigInteger(SMUtils.arrayToHex(hash), 16);
        }
        public string GetStringHash()
        {
            return BitConverter.ToString(hash).Replace(«-», ««);
```

```
            }
            public void GenerateBlind()
            {
                Random rand = new Random();
                r               =               new               BigInteger(256,
rand).Mod(n.Subtract(BigInteger.One)).Add(BigInteger.One);
                rInv = r.ModInverse(n);
            }
            public BigInteger GetBlindFactor()
            {
                return r;
            }
            public void BlindHash()
            {
                hBlind = h.Multiply(r).Mod(n);
            }
            public BigInteger GetBlindHash()
            {
                return hBlind;
            }
            public void GenerateBlindedSign()
            {
                sBlind = d.Multiply(hBlind).Mod(n);
            }
            public BigInteger GetSign()
            { return s; }
            public BigInteger GetBlindedSign()
            {
                return sBlind;
            }
            public void UnblindSign()
            {
                s = sBlind.Multiply(rInv).Mod(n);
            }
            public bool IsSignValide()
            {
                ECPointFp sG = g.multiply(s);
                ECPointFp hP = publicKey.multiply(h);
                if (sG.equals(hP))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }
    }
```

ДОДАТОК Б

ПРЕЗЕНТАЦІЯ

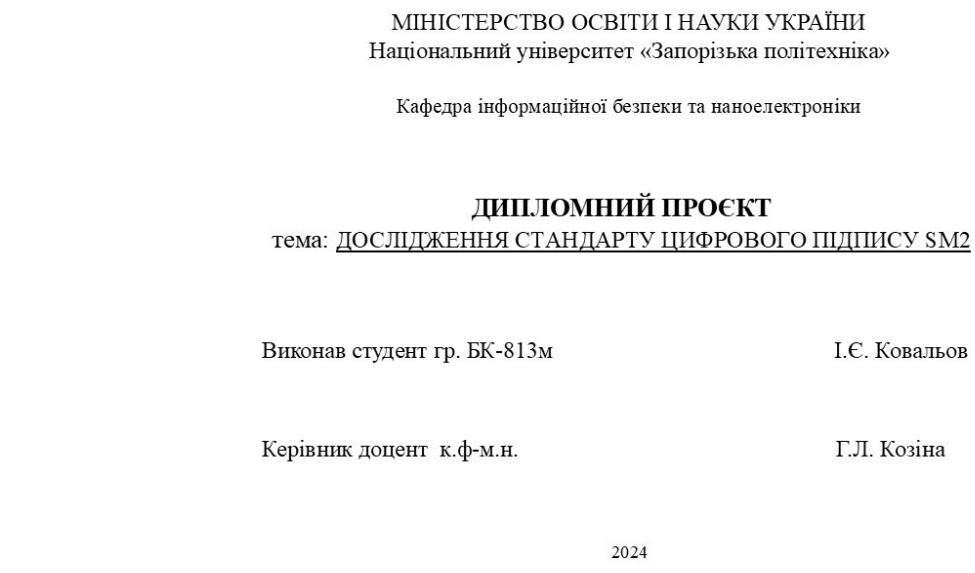


Рисунок Б.1 – Титульний слайд

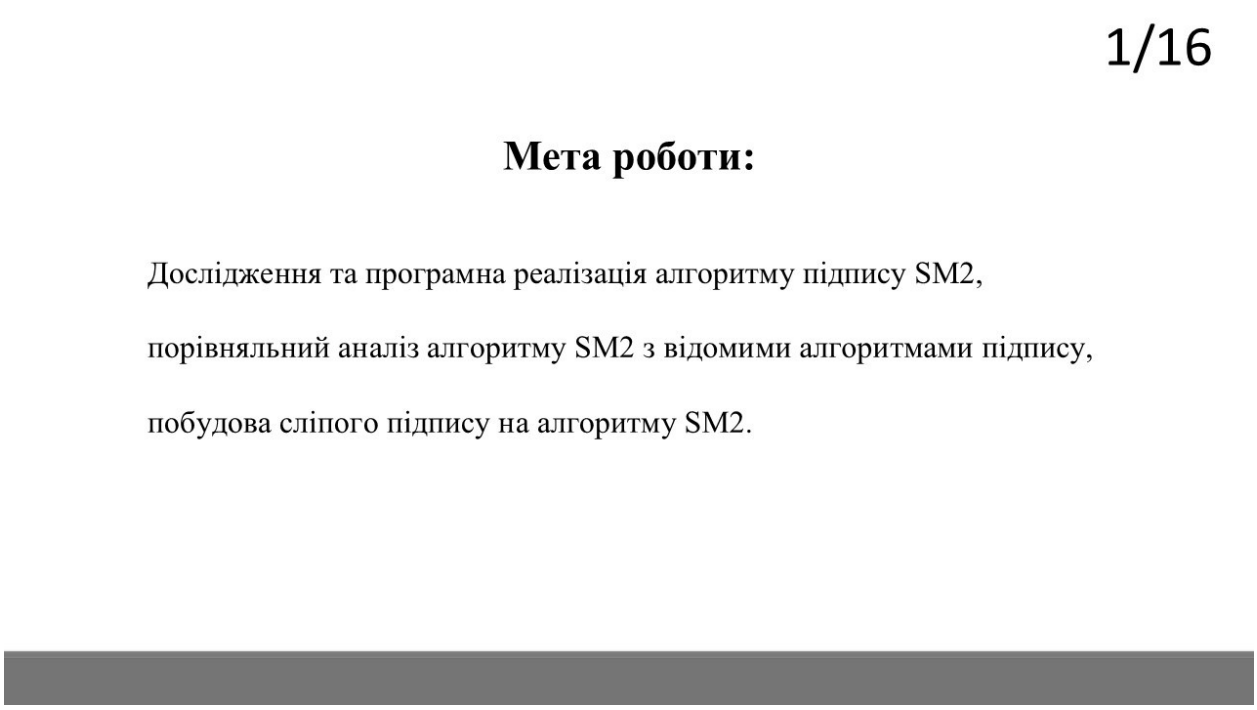


Рисунок Б.2 – Мета роботи

2/16

Стандарт SM2

SM2 — це криптографічний алгоритм, заснований на еліптичних кривих. Стандарт SM2 включає в себе алгоритми створення ключів, шифрування повідомлення та підпису. В контексті цифрового підпису нас цікавить саме частина про створення ключів та алгоритм підпису.

Алгоритм цифрового підпису генерує цифровий підпис даних підписувачем і перевіряє дійсність підпису за допомогою верифікатора.

Кожен підписант має приватний ключ і відповідний йому публічний ключ. Приватний ключ використовується для створення підпису, а публічний ключ використовується верифікатором для перевірки підпису.

Перед процесом підпису повідомлення хешується за допомогою функції хешування SM3.

Рисунок Б.3 – Опис стандарту SM2

3/16

Алгоритм підпису SM2

Генерація ключової пари
Крива: ECCurveFp (P, a, b), **Генератор кривої:** G,
Приватний ключ d = random (1 ≤ d ≤ n - 1) **Значення:** d — випадкове число.
Публічний ключ **Формула:** P = d × G d — Координати публічного ключа: Px, Py.

Цифровий підпис
Вхідні дані **Приватний ключ** d **Повідомлення** msg
Обчислити хеш повідомлення e = SM3(z || msg)
Розрахувати точки R = k × G r = Rx mod n
Розрахувати параметр s s = (k - r × d) × (1 + d)^-1 mod n
Підпис Signature = (r, s)

Перевірка підпису
Вхідні дані: **Публічний ключ** P **Підпис** (r, s) **Повідомлення** msg.
Обчислити хеш повідомлення e = SM3(z || msg)
Розрахувати точки R' = s × G + r × P r' = R'x mod n
Перевірити рівність r' == r

Рисунок Б.4 – Алгоритм роботи SM2

4/16

Реалізація стандарту SM2

- Реалізація необхідна для забезпечення більш зручного подальшого аналізу стандарту SM2
- Для реалізації SM2 було обрано C# через ряд його переваг над іншими мовами
- Програмний код може здійснювати підпис повідомлень, а також реалізовано версію у варіації сліпого підпису в графічній оболонці.

Рисунок Б.5 – Опис реалізації SM2

5/16

Дані, на основі яких було реалізовано SM2

1. Документ «Public key cryptographic algorithm SM2 based on elliptic curves, digital signature algorithm» - це офіційний документ стандарту, в якому містяться всі необхідні значення, дані та формули
2. «GM-Standarts» - це відкрита бібліотека реалізацій всіх GM-стандартів (частиною якого є SM2) від розробників

Рисунок Б.6 – Джерела, на основі яких було реалізовано стандарт SM2

6/16

Реалізація підпису на SM2

- Реалізовано функції для роботи з еліптичною кривою та її точками. Параметри кривої взято рекомендовані із стандарту SM2
- Реалізовано функцію хешування SM3, яка є необхідною для роботи стандарту SM2
- Реалізовано сам стандарт SM2, в який включено генерацію ключів та сам підпис

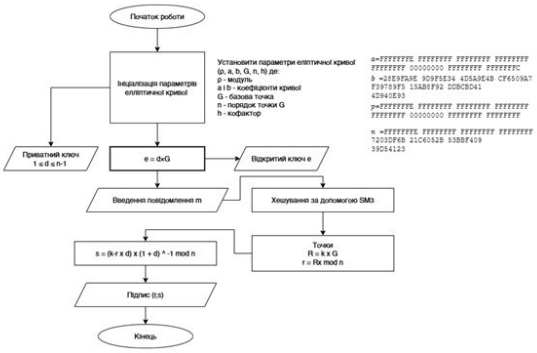


Рисунок Б.7 – Реалізація алгоритму підпису на SM2

7/16

Аналіз на швидкодію

- Для тестування було реалізовано подібний підпис на основі RSA та ECDSA.
- Тестування здійснюється в декілька етапів. Використовуються випадкові повідомлення розміром 32, 64, 128, 256, 512, 1024, 2048, 4096 байт та великі повідомлення на 1, 5, 10, 50 мб.
- На кожен малий об'єм повідомлення виконується 100 ітерацій, на великі 10 ітерацій. Після цього обчислюється середнє значення часу виконання.

Рисунок Б.8 – Аналіз на швидкодію

8/16

Результати аналізу на швидкодію

Розмір повідомлення (байти)	SM2 - Час хешування (мс)	SM2 - Час перевірки (мс)	RSA - Час хешування (мс)	RSA - Час перевірки (мс)	ECDSA - Час хешування (мс)	ECDSA - Час перевірки (мс)
32 байти	0 мс	1 мс	0 мс	214 мс	0 мс	1 мс
64 байти	0 мс	1 мс	0 мс	206 мс	0 мс	1 мс
128 байт	0 мс	1 мс	0 мс	201 мс	0 мс	1 мс
256 байт	0 мс	1 мс	0 мс	201 mc	0 мс	1 мс
512 байт	0 мс	1 мс	0 мс	204 мс	0 мс	1 мс
1024 байти	0 мс	1 мс	1 мс	203 мс	0 мс	1 мс
2048 байт	0 мс	1 мс	2 мс	201 мс	0 мс	1 мс
4096 байт	0 мс	1 мс	4 мс	199 мс	0 мс	1 мс
1048576 байт (1 МБ)	2 мс	1 мс	1045 мс	202 мс	1 мс	1 мс
5242880 байт (5 МБ)	2 мс	1 мс	5203 мс	201 мс	1 мс	1 мс
10485760 байт (10 МБ)	5 мс	1 мс	10350 мс	200 мс	2 мс	1 мс
52428800 байт (50 МБ)	28 мс	1 мс	51858 мс	200 mc	27 мс	2 мс

RSA зазвичай має малий час перевірки підпису (особливо для малих повідомлень). Проте на великих розмірах повідомлень час перевірки може зрости через більший розмір ключа та операцій з великими числами.

SM2 та ECDSA мають стабільний час перевірки підпису (1 мс) навіть для великих повідомлень, оскільки його схема підпису оптимізована для таких операцій.

Рисунок Б.9 – Результати аналізу на швидкодію

9/16

Аналіз економії пам’яті

- Умови аналізу аналогічні до тестування на швидкодію
- Для отримання результатів здійснюється замір об’єма пам’яті, яка вживається програмою
- Програма була максимально розвантажена від зайвих функцій, щоб рахувати саме об’єм, що використовується для здійснення обчислень і було мінімум впливу від зайвих операцій

Рисунок Б.10 – Проведення аналізу економії пам’яті

10/16

Результати аналізу вживаної пам’яті

Розмір повідомлення (байти)	SM2 (Середнє споживання пам'яті)	RSA (Середнє споживання пам'яті)	ECDSA (Середнє споживання пам'яті)
32 байти	9 байт	9 байт	9 байт
64 байти	9 байт	9 байт	9 байт
128 байт	19 байт	19 байт	19 байт
256 байт	38 байт	38 байт	38 байт
512 байт	76 байт	76 байт	76 байт
1024 байти	153 байти	153 байти	153 байти
2048 байт	307 байт	307 байт	307 байт
4096 байт	614 байт	614 байт	614 байт
1048576 байт (1 МБ)	312851 байт	312851 байт	313344 байт
5242880 байт (5 МБ)	1258237 байт	1258237 байт	1258291 байт
10485760 байт (10 МБ)	1572810 байт	1572810 байт	1572864 байт
52428800 байт (50 МБ)	12582858 байт	12582858 байт	12585721 байт

Всі три алгоритми мають схожі результати споживання пам'яті для повідомлень, з незначними відмінностями в точності пам'яті на етапі обчислень. Всі реалізації алгоритмів є оптимізовані і підходять для використання на різних платфомах.

Рисунок Б.11 – Результати аналізу на вживання пам’яті

11/16

Алгорим сліпого підпису SM2

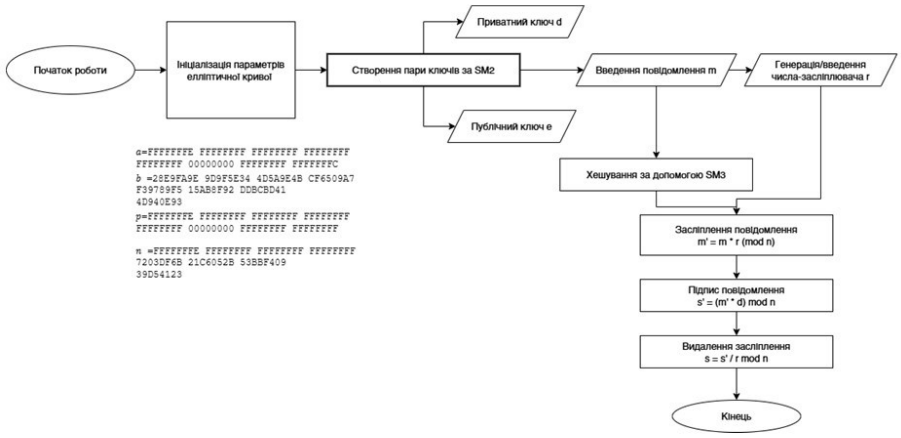


Рисунок Б.12 – Блок-схема алгоритму роботи сліпого підпису SM2

12/16

Процес реалізації сліпого підпису SM2

- Було розроблено власну реалізацію сліпого підпису стандарту SM2 на мові C#, в процесі було реалізовано функції для обчислень на основі еліптичних кривих, алгоритм хешування SM3 та консольний додаток для підписання повідомлень.
- Для зручності роботи із програмою було реалізовано графічний інтерфейс програми за допомогою графічної бібліотеки Windows Forms

Реалізація та додаток з графічною оболонкою доступні за посиланням:
https://github.com/IlliaKovalov/SM2

Рисунок Б.13 – Процес реалізації сліпого підпису

13/16

Анонімність

- Анонімність у криптографії — це здатність системи приховувати особу чи контекст користувача, що ініціює операцію, забезпечуючи приватність. У цифрових підписах це означає, що автор і деталі операції залишаються прихованими.
- Алгоритм SM2 базується на еліптичних кривих, подібно до ECDSA, але має оптимізовану структуру. Він підтримує сліпий підпис, де підписант працює з перетвореним повідомленням, не знаючи його змісту, що захищає дані навіть у разі компрометації.
- У SM2 засліплення використовує приховуючий фактор для перетворення повідомлення перед підписанням. Підписант працює з модифікованими даними, не знаючи їх змісту. Це забезпечує анонімність і захищає дані навіть у разі компрометації підписанту.

Рисунок Б.14 – Доказ анонімності

14/16

Інструкція до роботи додатку

Для роботи додатку необхідно завантажити всі файли.

Запуск програми за допомогою SM2_Sign.exe

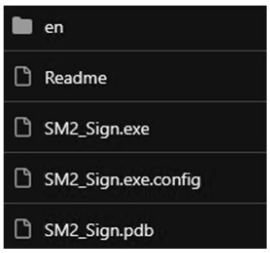


Рисунок Б.15 – Інструкція, як завантажити додаток

15/16

Інтерфейс додатку

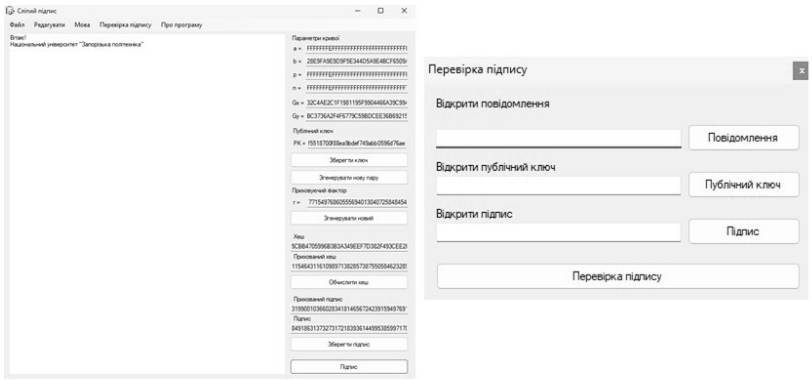


Рисунок Б.16 – Інтерфейс додатку сліпого підпису

16/16

Висновки

- Досліджено та реалізовано алгоритм цифрового підпису SM2 на C#, включно з функцією хешування SM3. Виконано аналіз і порівняння SM2, ECDSA та RSA за швидкодією й економією пам'яті. SM2 та ECDSA демонструють стабільну швидкодію для великих даних, тоді як RSA має більший час обчислень через операції з великими числами.

- За використанням пам'яті для малих повідомлень усі алгоритми є однаковими, але для великих повідомлень ECDSA витрачає трохи більше пам'яті. SM2 та ECDSA виявилися оптимальними для ресурсоємних задач, а RSA — для сценаріїв, де важлива стабільність понад швидкість.

- SM2 також забезпечує високий рівень анонімності завдяки підтримці механізму сліпого підпису, який приховує зміст даних від підписувача, що є важливою перевагою для захисту приватності.

Рисунок Б.17 – Висновки

Дякую за увагу!

Рисунок Б.18 – Фінальний слайд