МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____ С.В. Казмірчук
«_____»_____ 20__ р.

На правах рукопису
УДК 004.056.5:510.22(043.3)

ДИПЛОМНА РОБОТА

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «БАКАЛАВР»

Тема: Криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA

Виконавець: П.О. Городецька
Науковий керівник: к.т.н., доц. С.С. Ільєнко
Нормоконтролер: к.т.н., доц. С.С. Ільєнко

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет: Кібербезпеки, комп'ютерної та програмної інженерії

Кафедра: Комп'ютеризованих систем захисту інформації

Освітній ступінь: Бакалавр

Спеціальність: 125 «Кібербезпека»

Освітньо-професійна програма: «Безпека інформаційних і комунікаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ С.В. Казмірчук

«__» _____ 20__ р.

ЗАВДАННЯ

на виконання дипломної роботи

здобувача вищої освіти Городецької Поліни Олександрівни

1. Тема: *Криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA*

затверджена наказом ректора від 26.04.2021 №652/ст.

2. Термін виконання з 10.05.2021 р. по 20.06.2021 р.

3. Вихідні дані: провезти аналіз теоретичних аспектів використання ЕЦП в сучасних інформаційних мережах; розробити програмний модуль забезпечення цілісності інформації на базі алгоритму ECDSA; проведення опису та тестування розробленого програмного модулю.

4. Зміст пояснювальної записки: аналіз теоретичних аспектів використання ЕЦП в сучасних інформаційних мережах; розробка програмного модулю забезпечення цілісності інформації на базі алгоритму ECDSA.

КАЛЕНДАРНИЙ ПЛАН

виконання роботи

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1	Уточнення постановки задачі	10.03.2021	Виконано
2	Аналіз літературних джерел	16.03.2021-20.03.2021	Виконано
3	Обґрунтування рішення	21.03.2021	Виконано
4	Дослідження сучасних практичних алгоритмів формування та верифікації електронно-цифрового підпису	25.03.2021-10.04.2021	Виконано
5	Розробка авторського криптографічного модулю забезпечення цілісності інформації на базі алгоритму ECDSA	14.04.2021-15.05.2021	Виконано
6	Дослідження працездатності власного програмного модуля	15.05.2021	Виконано
7	Оформлення і друк пояснювальної записки	25.05.2021	Виконано
8	Оформлення презентації	30.05.2021	Виконано
9	Отримання рецензій від рецензентів	10.06.2021	Виконано
10	Підготовка до захисту	10.06.2021-14.06.2021	Виконано

Здобувач вищої освіти П.О. Городецька

(підпис, дата)

Дипломний керівник С.С. Ільєнко

(підпис, дата)

РЕФЕРАТ

Дипломна робота на тему: «Криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA» складається зі вступу, основної частини, що містить 2 розділи, 2 висновки до кожного розділу і спільного висновку й списку джерел. Загальний обсяг роботи – 145 сторінок. Робота містить 52 рисунків. Список використаних джерел включає 22 джерела.

Метою роботи є реалізація криптографічний модулю забезпечення цілісності інформації на базі алгоритму ECDSA.

У дипломній роботі розглянуто теоретичні основи використання ЕЦП в сучасних інформаційних мережах.

Розроблено програмний модуль забезпечення цілісності інформації на базі алгоритму ECDSA.

Розроблений програмний модуль може бути використаний у сучасних комп`ютерних системах та мережах.

Ключові слова: ЕЦП, ECDSA, DSA, RSA, ГЕНЕРАЦІЯ ПІДПИСУ, ГЕНЕРАЦІЯ ПАРИ КЛЮЧІВ, ВЕРИФІКАЦІЯ ПІДПИСУ, ВЕРИФІКАЦІЯ КЛЮЧІВ, ЕЛІПТИЧНА КРИВА.

ЗМІСТ

ВСТУП

Заходи для забезпечення цілісності захищають інформацію від несанкціонованих змін. Ці заходи забезпечують впевненість у точності та повноті даних. Необхідність захисту інформації включає як дані, що зберігаються в системах, так і дані, що передаються між такими системами, як електронна пошта. Підтримуючи цілісність, потрібно не тільки контролювати доступ на системному рівні, але й надалі забезпечувати можливість того, щоб користувачі системи могли змінювати лише інформацію, на яку вони мають законні повноваження.

Як і у випадку із захистом конфіденційності, захист цілісності даних виходить за межі навмисних порушень. Ефективні засоби протидії цілісності також повинні захищати від ненавмисних змін, таких як помилки користувача або втрата даних, що є наслідком несправності системи.

Хоча всі власники систем вимагають впевненості у цілісності своїх даних, фінансова галузь має особливо гостру потребу забезпечити захист транзакцій в її системах від фальсифікацій. Одне з найвідоміших порушень цілісності фінансових даних за останній час сталося в лютому 2016 року, коли кібер-злодії заробили 1 мільярд доларів шляхом шахрайського зняття коштів з рахунку центрального банку Бангладеш у Федеральному резервному банку Нью-Йорка. Хакери виконали розроблену схему, яка включала отримання необхідних облікових даних для ініціації виведення коштів, а також зараження банківської системи шкідливим програмним забезпеченням, яке видаляло записи баз даних про перекази, а потім припиняло повідомлення про підтвердження, які мали б попередити банківські органи про шахрайство.

Є багато контрзаходів, які можна запровадити для захисту цілісності. Контроль доступу та сувора автентифікація можуть допомогти уникнути вповноважених користувачів несанкціонованих змін, також для захисту цілісності даних є адміністративний контроль, такий як розподіл

обов'язків та навчання. Перевірка хешу та цифрові підписи є найбезпечнішим способом, так як можуть допомогти гарантувати, що транзакції є автентичними та що файли не були змінені чи пошкоджені. Цифровий підпис підтверджує цілісність повідомлення.

Схеми цифрового підпису призначені для надання цифрового аналога власноручним підписам (і багатьом іншим). Цифровий підпис - це число, яке залежить від деякого секрету, який відомий тільки підписувачу (закритий ключ підписувача), і, крім того, на вміст повідомлення, яке підписуються. Підписи повинні верифікуватися - в разі спору виникає питання про те, чи суб`єкт підписав документ, неупереджена третя сторона повинна бути в змозі вирішити питання за принципом справедливості, не вимагаючи доступу до закритого ключа. Спори можуть виникнути, коли підписувач намагається відмовитися від створеного ним підпису, або коли фальсифікатор робить помилкову заяву.

Ця дипломна робота присвячена схемам асиметричного цифрового підпису. «Асиметрична» означає, що кожен суб'єкт обирає пару ключів, що складаються із закритого ключа і пов'язаного з ним відкритого ключа. Суб'єкт підтримує секретність закритого ключа, який використовує для підпису повідомлень і робить справжні копії свого відкритого ключа, які є доступними для інших організацій, які вони, в свою чергу, використовують його для перевірки підписів.

В ідеалі, схема цифрового підпису не повинна піддаватися підробці при атаці за обраним повідомленням. Це поняття безпеки було введено Голдвассер, Мікалі і Ривест. Неформально стверджує, що противник, здатний отримати підписи суб'єкта для будь-яких повідомлень за своїм вибором - не можуть успішно підробити підпис на жодному іншому повідомленні.

Схеми цифрового підпису можуть використовуватися для надання наступних базових криптографічних послуг: цілісність даних (гарантія того, що дані не були змінені несанкціонованими або невідомими засобами), аутентифікація джерела даних (гарантія того, що джерело даних відповідає заявленому) і відсутність невідмовності (впевненість в тому , що організація не

може заперечувати попередні дії або зобов'язання). Схеми цифрового підпису зазвичай використовуються як примітиви в криптографічних протоколах, які надають інші послуги, включаючи аутентифікацію суб'єкта, аутентифікована передача ключів і автентифікована ключова угода.

Схеми цифрового підпису, що використовуються сьогодні, можуть бути класифіковані відповідно до складної математичної проблеми, що лежить в основі їх безпеки.

Мета роботи - розробка авторського криптографічного модуля забезпечення цілісності інформації на базі алгоритму ECDSA.

Виходячи з мети, завданням даної дипломної роботи є:

- провести дослідження сучасних практичних алгоритмів формування та верифікації електронно-цифрового підпису;
- розробити авторський криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA;
- провести дослідження працездатності власного програмного модуля.

Об'єкт дослідження - процедура формування та верифікації електронно-цифрового підпису.

Предмет дослідження - методи функціонування захищених систем та мереж на основі використання елдектронно-цифрового підпису за умови забезпечення цілісності інформації.

Методи дослідження - проведені дослідження базуються на сучасних методах побудови захищених інформаційних мереж, методах формування та верифікації електронно-цифрового підпису.

Практична цінність роботи полягає у створенні власного криптографічного модуля на основі алгоритму ECDSA мовою програмування Python 3.6+ в поєднанні з фреймворком PyTest, що дозволяє проводити процедуру генерації пари ключів, підписання та перевірку електронно-цифрового підпису на базі використання популярних криптостійких еліптичних кривих зі стандарту NIST та SEC.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ВИКОРИСТАННЯ ЕЦП В СУЧАСНИХ ІНФОРМАЦІЙНИХ МЕРЕЖАХ

Одна з найважливіших проблем, що вирішуються з використанням асиметричних методів шифрування - проблема підтвердження авторства. Механізм, який підтверджує дійсність даних і забезпечення невідмовності об’єкта від ознайомлення/підписання даних є механізм електронного підпису. Закон України «Про електронні довірчі послуги» є законодавчою основою застосування електронного підпису й формування електронного документообігу.

Електронний підпис - інформація в електронній формі, яка приєднана до іншої інформації в електронній формі або іншим чином пов'язана з такою інформацією і яка використовується для визначення особи, яка підписує інформацію [1].

Властивості електронного підпису:

1) підписати документ може тільки законний власник ЕП;

2) автор ЕП не може від неї відмовитися;

3) у разі виникнення спору можлива участь третіх осіб (наприклад, суду) для встановлення автентичності ЕП.

Функції цифрового підпису повинні відповідати стандартам для належної операції підпису. Для того, щоб створити послідовний, законний та безпечний підпис потрібні три ключові моменти:

1) рішення, яке надає платформу для виконання процедури підпису та управління документами;

2) технологія, яка відповідальна за аутентифікацію, мобільні додатки та апаратні модулі безпеки;

3) провайдери послуг, які підтверджують підпис та діють як регестраційні органи.

Криптографічною функцією є перетворення, яке використовується для вироблення цифрового підпису. Підпис виробляється таким чином, щоб якщо

зловмисник все ж перехопив документ, то без секретного ключа він не зміг підробити підпис, а також, це дає будь-якій особі, при наявності загальнодоступного ключа, документа та ЕЦП, упевнитися, що документ дійсний.

1.1. Сучасні правові основи формування та верифікації ЕЦП

Взагалі, електронний цифровий підпис призначений для використання фізичними та юридичними суб`єктами електронного документообігу [2].

Накладання електронного цифрового підпису завершує утворення електронного документа, надаючи йому юридичної сили [2].

Згідно закону України «Про електронний цифровий підпис», закон визначає правовий статус електронного цифрового підпису та регулює відносини, що виникають при використанні електронного цифрового підпису [3].

З 2018 року закон України «Про електронний цифровий підпис» втрачає свою силу та закон України «Про електронні довірчі послуги» набуває чинності [3]. Одним з найважливіших положень Закону № 2155 є взаємне визнання українських та іноземних сертифікатів відкритих ключів та електронних підписів [4]. Законом запроваджуються такі механізми, як електронна ідентифікація, електронний підпис, електронна печатка, електронна позначка часу, реєстрована електронна доставка, інтероперабельність тощо.

У новому законі електронний цифровий підпис (ЕЦП) змінюється на кваліфікований електорнний підпис (КЕП), КЕП – це такий підпис, що відповідає всім критеріям для УЕП [5], а також таким:

- Програмне забезпеченя або обладнання, якими вони здійснюються, підлягають додатковим умовам.
- Базується на сертифікаті відкритого ключа.

Також, наказом державного комітету України з вопросів технічного врегулювання та споживчої політики був введений ДСТУ 4145-2002 («ДСТУ 4145-2002. Інформаційні технології. Криптографічний захист інформації. Цифровий підпис, оснований на еліптичних кривих. Формування та перевірка»). Це український стандарт, що описує алгоритми формування та перевірки електронного підпису, основані на властивостях груп точок еліптичних кривих над полями $GF(2^m)$ та правилах застосування цих правил до повідомлень, які обробляються в системах [6].

1.2. Дослідження сучасних методів та алгоритмів реалізації ЕЦП

Існує дві схеми, які використовують для побудови цифрового підпису:

- Симетричне шифрування. Цю схему краще використовувати маючи третє лице – арбитру, в якого є довіра двох сторін. Факт шифрування секретним ключем та передача йому арбитру є авторизацією документу.
- Асиметричне шифрування. В сфері безпеки, асиметричне шифрування більш поширине та знаходить більше застосування.

В даній роботі ми разглянемо асиметричне шифрування більш детально, а саме алгоритми RSA, DSA та ECDSA. А також, розглянемо математичну сторону еліптичних кривих, на яких заснований алгоритм ECDSA.

- DSA (Digital Signature Algorithm)

DSA був запроваджений Національним Інститутом Стандарту та Технологій (U.S. NIST) в Серпні 1991 року. DSA можна розглядати як різновид схеми-підпису Ель-Гамаля. Його безпека заснована на нерозв'язності проблеми дискретного логарифмування в простому порядку підгрупи Z_p^* [7].

ГЕНЕРАЦІЯ ПАРАМЕТРУ ДОМЕНА DSA. Параметри домена генеруються для кожного суб`єкта в певній галузі безпеки.

1. Обирає 1024-бітове просте число p та 160-бітове просте число q з властивістю $q | p - 1$.
2. Обирає елемент $h \epsilon Z_p^*$ та обчислює $g = h^{(p-1)/q} \, mod \, p$, поки $g \neq 1$.
3. Параметри домена – p, q та g .

ГЕНЕРАЦІЯ ПАРИ КЛЮЧІВ DSA. Кожна особа А в домені з параметрами домену (p, q та g) робить наступне:

1. Обирає довільне або псевдо довільне ціле x, щоб $1 \leq x \leq q - 1$.
2. Обчислює $y = g^x mod \, p$.
3. Відкритий ключ А – y; Закритий ключ А – x.

ГЕНЕРАЦІЯ ПІДПИСУ DSA. Щоб підписати повідомлення m, А робить наступне:

1. Обирає довільне або псевдо довільне ціле k, щоб $1 \leq k \leq q - 1$.
2. Обчислює $X = q^k \, mod \, p$ та $r = X \, mod \, q$. Якщо $r = 0$, повернутися до пункту 1.
3. Обчислює $k^{-1} \, mod \, q$.
4. Обчислює $e = \text{SHA} - 1(m)$.
5. Обчислює $s = k^{-1}\{e + xr\} mod \, q$. Якщо $s = 0$, повернутися до пункту 1.
6. Підпис *A* для повідомлення *m – (r, s).*

ПЕРЕВІРКА ПІДПИСУ DSA. Для того, щоб підтвердити підпис *(r,s)* повідомлення *m* від особи А, особа В отримує копії параметрів *(p, q, g)* домену *A* та відкритий ключ y, робить наступне [8]:

1. Верифікує те, що r та s цілі та в інтервалі $[1, \text{q} - 1]$.
2. Обчислює $e = \text{SHA} - 1(m)$.
3. Обчислює $w = s^{-1} \, mod \, q$.
4. Обчислює $u_1 = ew \, mod \, q$ та $u_2 = rw \, mod \, q$.
5. Обчислює $X = g^{u_1} y^{u_2} \, mod \, p$ та $v = X \, mod \, q$.

6. Підтверджує підпис тільки якщо $v = r$.

Знизу представлено блок-схему алгоритму DSA для перевірки підпису (рис 1.1), як можна побачити, для отриманого повідомленния рахується хеш (безпечний алгоритм шифрування), далі перевіряємо підпис завдяки відкритому ключу та хешу, та у разі якщо $v = r / v \neq r$ - отримуємо підтвердження/не підтвердження, відповідно:

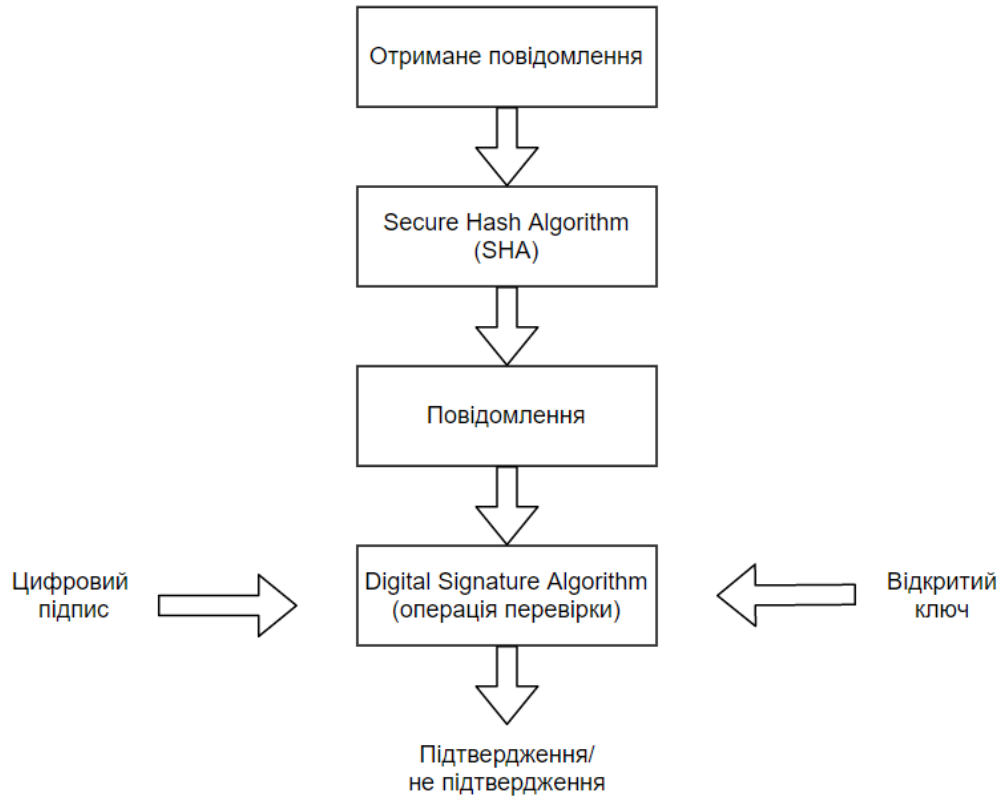

Рис 1.1. Процес перевірки підпису DSA

- Еліптична крива

Неформально, еліптична крива – тип кубичної кривої, рішення якої обмежено областю простору, тору. Еліптична функція Вейерштрасса описує як отримати із тора алгебраїчну форму еліптичної кривої.

Формально, еліптична крива визначена на полі K – це невироджена кубічна крива двох змінних, $f(X, Y) = 0$, з K-раціональною точкою.

Нехай, p $>$ 3 буде просте непарне число. Еліптична крива E, яка належить полю F_p визначається рівнянням:

$$y^2 = x^3 + ax + b, \quad (3)$$

Де $a, b \in F_p$, та $4a^3 + 27b^2 \not\equiv 0 \ (mod\ p)$. Множина $E(F_p)$ складається з всіх точок, які задовольняють визначене рівняння (3), разом зі спеціальною точкою □, яка називається точкою нескінченності.

Приклад 1. (псевдодовільна крива на полі F_{23}) Нехай, $p = 23$ та розглянемо еліптичну криву E: $y^2 = x^3 + x + 4$, визначену на полі F_{23}. (В позначеннях рівняння (3), ми маємо $a = 1$ та $b = 4$). Зауважимо, що $4a^3 + 27b^2 = 4 + 432 = 436 = 22\ (mod\ 23)$, тому E точно є еліптичною кривою. $E(F_{23})$ містить нескінченну точку □ та наступні:

(0,2) (0,21) (1,11) (1,12) (4,7) (4,16) (7,3)
(7,20) (8,8) (8,15) (9,11) (9,12) (10,5) (10,18)
(11,9) (11,14) (13,11) (13,12) (14,5) (14,18) (15,6)
(15,17) (17,9) (17,14) (18,9) (18,14) (22,5) (22,19)

ФОРМУЛА ДОДАВАННЯ. Є правило, яке називається правилом хорди та дотичної, для додавання двох точок на еліптичній кривій $E(F_{23})$, щоб задати третю точку на еліптичній кривій. Разом з цією операцією додавання набір точок кривої $E(F_{23})$ формує групу з нескінченною точкою □. Ця група використовується для формування криптосистеми еліптичної кривої. Геометрично це правило пояснюється так: нехай, $P = (x_1, y_1)$ та $Q = (x_2, y_2)$ дві окремі точки на еліптичній кривій E. Потім сума P та Q, позначена як $R = (x_3, y_3)$, визначається наступним чином. Спочатку накреслимо лінію через P та Q; ця лінія перетинає еліптичну криву в третій точці. Потім R є відображенням цієї точки на вісі x. Це відображено у Рисунку 1.2. Еліптична крива в фігурі складається з двох частин: еліпсоподібна фігура та нескінченна крива [9].

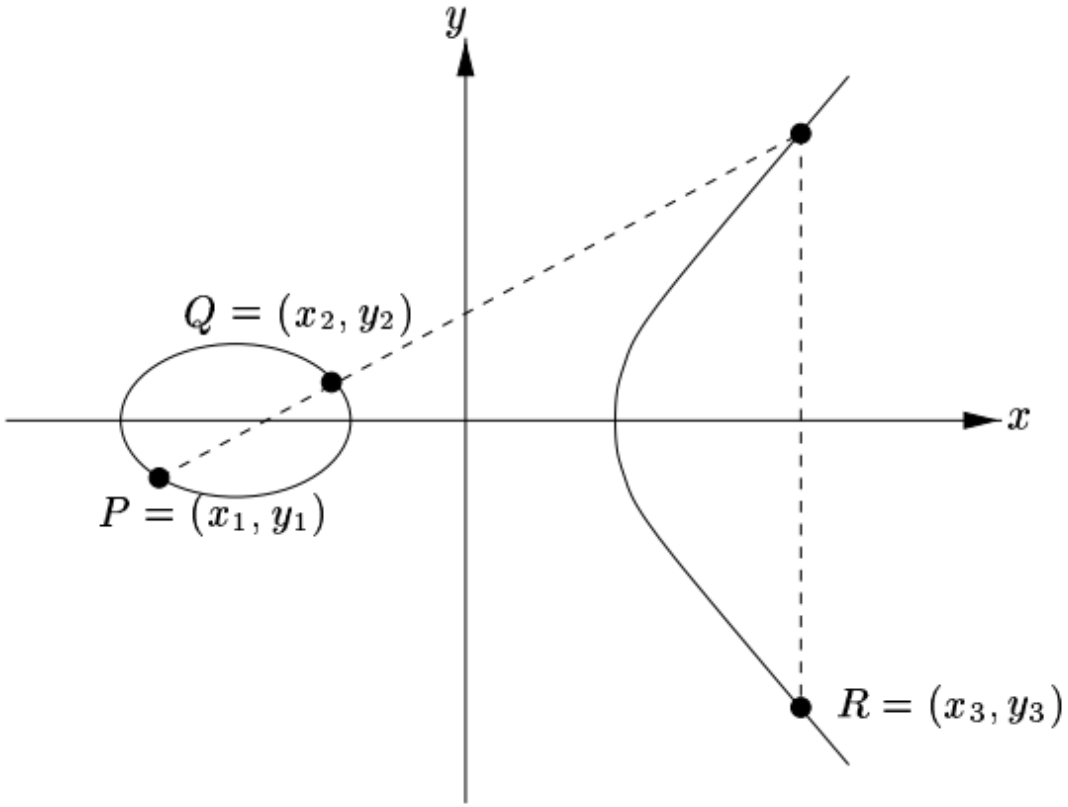

Рис 1.2. Геометричний опис додавання двох окремих точок на еліптичній кривій: $P + Q = R$

Якщо $P = (x_1, y_1)$, потім, подвоєння P, позначена, як $R = (x_3, y_3)$, визначається наступним чином. Спочатку накреслимо дотичну до еліптичної кривої та до точки P. Ця лінія перетинає еліптичну криву в другій точці. Потім R є відображеням цієї точки на вісі x. Це відображено на Рисунку 1.3.

Алгебраїчні формули для суми двох точок та подвоєння точок можна тепер вивести з геометричного опису.

1. $P + \square = \square + P = P,\ P \in E(F_p)$.
2. Якщо $P = (x, y) \in E(F_p)$, тоді $(x, y) + (x, -y) = \square$. (Точка позначена – P та називається негативним P, також, $-P$ – точка на кривій.)

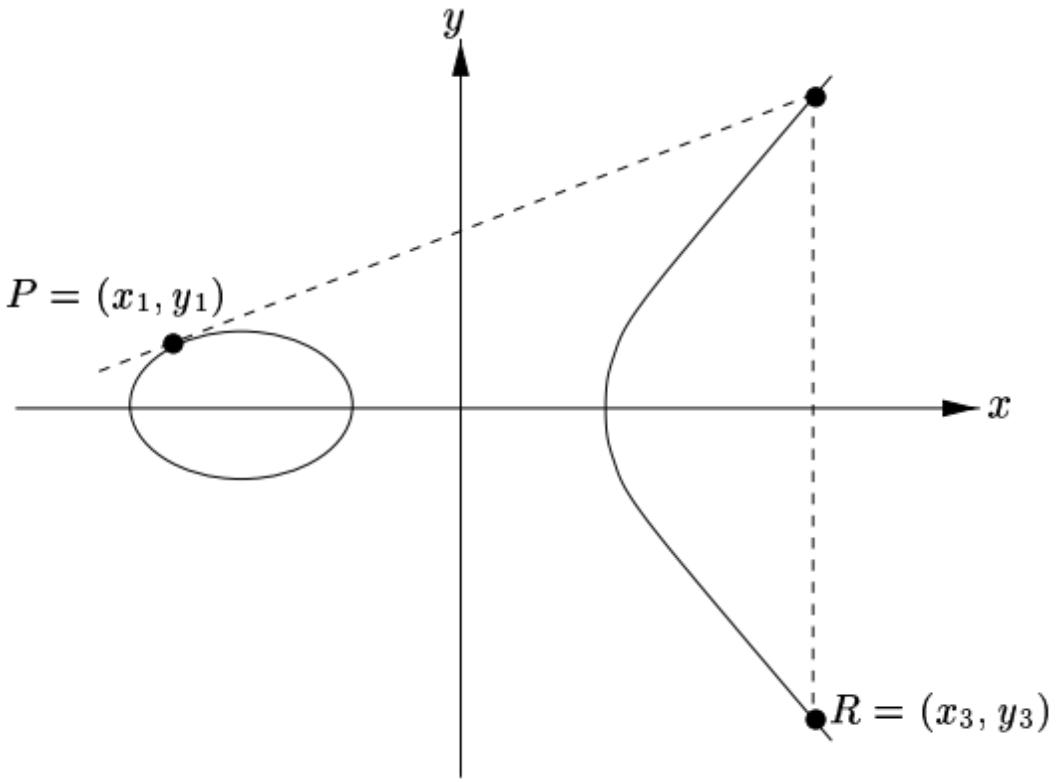

Рис 1.3. Геометричний опис подвоєння двох окремих точок на еліптичній кривій: $P + P = R$

1. Нехай, $P = (x_1, y_1) \in E(F_p)$ та $Q = (x_2, y_2) \in E(F_p)$, де $P \neq \mp Q$. $P + Q = (x_3, y_3)$, де

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ та } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

2. Нехай, $P = (x_1, y_1) \in E(F_p)$, де $P \neq -P$. $2P = (x_3, y_3)$, де

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ та } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right) - (x_1 - x_3) - y_1.$$

- ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA (Elliptic Curve Digital Signature Algorithm) – це аналог DSA з еліптичними кривими, який був запроваджений у 1999 році. Як і DSA, ECDSA використовує хеш-функцію. В даний час, єдина хеш-функція, яка призначена для використання з ECDSA – SHA-1 (алгоритм криптографічного хешування). ECDSA використовує ключі ECC (Elliptic Curve Cryptography), щоб бути впевненим, що кожен користувач унікальний та кожна транзакція безпечна. Хоча такий вид алгоритму ЕЦП пропонує функціонально нерозрізнений

результат, як і інші алгоритми ЕЦП, він використовує менші ключі, таким чином, цей алгоритм більш практичний.

ГЕНЕРАЦІЯ ПАРАМЕТРІВ ДОМЕНА ECDSA. Параметри домена для ECDSA включають в себе еліптичну криву E, яка визначена на полі Галоа (скінченне поле) F_q характеристик p, та початкову точку $h \in E(F_q)$. Це єдиний спосіб, щоб згенерувати криптографічно захищені параметри домену:

1. Обирає коефіцієнти a та b з F_q. Нехай E буде кривою $y^2 = x^2 + ax + b$, якщо $q = p$, та $y^2 + xy = x^3 + ax^2 + b$ якщо $q = 2^m$.
2. Обчислює $N = \#E(F_q)$.
3. Підтверджує, що N можна поділити на велике число $n (n > 2^{160}$ та $n > 4\sqrt{q})$. Якщо ні, то повернутися до пункту 1.
4. Підтверджує, що n не можна поділити на $q^k - 1$ для кожного k, $1 \leq k \leq 20$. Якщо ні, то повернутися до пункту 1.
5. Підтверджує, що n $\neq$ q. Якщо ні, то повернутися до пункту 1.
6. Вибирає довільну точку $G' \in E(F_q)$ та $G = (N/n)G'$. Повторювати доти, поки $G \neq$ □.

МЕТОДИ ДЛЯ ПЕРЕВІРКИ ПАРАМЕТРІВ ДОМЕНУ. Впевненість, що множина параметрів еліптичної кривої $D = (q, FR, a, b, G, n, h)$ вірні, можуть бути надані особі використовуючи один із методів:

1. A робить точну перевірку параметру домена за допомогою методу (описаного нижче)
2. A генерує D, використовуючи довірчу систему.
3. A отримує запевнення від довіреної сторони T, що та у свою чергу виповнила точну перевірку параметру домена за допомогою методу (описаного нижче).
4. A отримує запевнення від довіреної сторони T, що та у свою чергу сгенерувала D, використовуючи довірчу систему.

Метод: Точна Перевірка параметру домена ECDSA:

1. Перевірити, що q є непарним простим числом або $2(q = 2^m)$.
2. Перевірити, що це вірне уявлення для F_q.
3. Перевіряє, що Q $\neq$ □.
4. Перевіряє, що та елементи F_q (цілі в інтервалі $[0, p-1]$, у випадку, що $q = p$, та двійкові рядки довжиною m бітів, у випадку, що $q = 2^m$).
5. Перевірити, що a та b визачають еліптичну криву F_q ($4a^3 + 27b^2 \not\equiv 0\ (mod\ p)$, якщо $q = p; b \neq 0$, якщо $q = 2^m$).
6. Перевірити, що G лежить на еліптичній кривій, яка визначена параметрами a та b ($y_G^2 = x_G^3 + ax_G + b$ у випадку, що $q = p$, та $y_G^2 + x_G y_G = x_G^3 + ax_G^2 + b$,у випадку, що $q = 2^m$).
7. Перевірити, що n просте.
8. Перевірити, що $n > 2^{160}$ і те, що $n > 4\sqrt{q}$.
9. Перевірити, що $nG =$ □.
10. Обчислити $h' = \lfloor(\sqrt{q} + 1)^2/n\rfloor$ та перевірити, що $h = h'$.
11. Перевірити, що n не поділяє $q^k - 1$ для кожного k, $1 \leq k \leq 20$.
12. Перевірити, що $n \neq q$.
13. Після всіх успішних перевірок, D – дійсне, якщо хоча б одна перевірка не виконана, то D – не дійсне.

ГЕНЕРАЦІЯ ПАРИ КЛЮЧІВ ECDSA. Пара ключів особи A пов`язані з певною множиною параметрів домену еліптичної кривої $D = (q, FR, a, b, G, n, h)$. Кожна особа A робить наступне [10]:

1. Обирає довільне або псевдодовільне ціле d в інтервалі $[1, n-1]$.
2. Обчислює $Q = dG$.
3. Відкритий ключ A – Q; закритий ключ A – d.

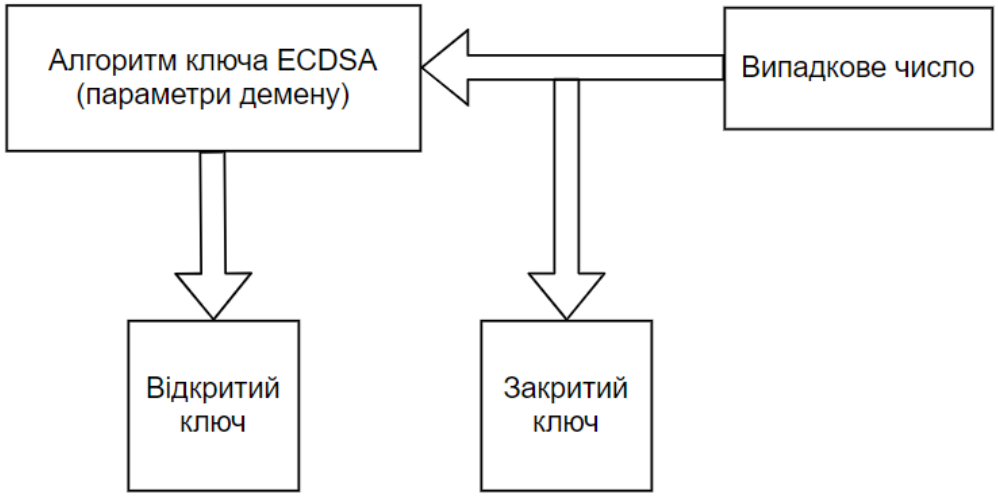

Рис1.4. Процес генерації пари ключів

МЕТОДИ ПЕРЕВІРКИ ВІДКРИТИХ КЛЮЧІВ. Перевірку відкритого ключа Q можна здійснити одним із методів [11]:

1. А робить точну перевірку ключа за допомогою методу (описаного нижче)
2. А генерує Q, використовуючи довірчу систему.
3. А отримує запевнення від довіреної сторони T, що та, у свою чергу, виповнила точну перевірку ключа за допомогою методу (описаного нижче).
4. А отримує запевнення від довіреної сторони T, що та, у свою чергу, сгенерувала Q, використовуючи довірчу систему.

Метод: Точна Перевірка відкритого ключа EDCSA:

1. Перевіряє, що Q $\neq$ □.
2. Перевіряє, що x_Q та y_Q елементи F_Q (цілі в інтервалі $[0, p-1]$, у випадку, що $q = p$, та двійкові рядки довжиною m бітів, у випадку, що $q = 2^m$).
3. Перевіряє, що Q належить еліптичній кривій визначенній на проміжку від a до b.
4. Перевіряє, що nQ = □.
5. Після всіх успішних перевірок, Q – дійсне, якщо хоча б одна перевірка не виконана, то Q – не дійсне.

ДОКАЗ ВОЛОДІННЯ ЗАКРИТОГО КЛЮЧА. Якщо особа C в змозі сертифікувати відкритий ключ Q особи A, як свій відкритий ключ, то C може стверджувати, що підписані повідомлення особою A були від особи C. Щоб уникнути цього, CA повинен вимагати всіх осіб A довести, що вони володіють закритими ключами, що відповідають їх відкритим ключам. Щоб це довести, наприклад, можна запросити A підписати повідомлення, яке було вибране CA. Також потрібно пам`ятати, що володіння закритого ключа надає різні гарантії від перевірки відкритого ключа. Це демонструє володіння закритого ключа навіть якщо він може співпасти з не вірним відкритим ключем, у той час як останній демонструє дійсність відкритого ключа, але не володіння певним закритим ключем. Виконуючи обидві умови, можна підняти високий рівень впевненості в ключах.

ГЕНЕРАЦІЯ ПІДПИСУ ECDSA. Для підписання повідомлення m, особа A з параметрами домену $D = (q, FR, a, b, G, n, h)$ та парою ключів (d, Q) робить наступне:

1. Обирає довільне або псевдодовільне ціле k, щоб $1 \leq k \leq n - 1$.
2. Обчислює $kG = (x_1, y_1)$, переводить x_1 в ціле $\overline{x_1}$.
3. Обчислює $r = x_1 mod\ n$. Якщо $r = 0$, то повернутися до пункту 1.
4. Обчислює $k^{-1}\ mod\ n$.
5. Обчислює $\mathrm{SHA} - 1(m)$ та переводить двійкові рядки в ціле e.
6. Обчислює $s = k^{-1}(e + dr) mod\ n$. Якщо $s = 0$, то повернутися до пункту 1.
7. Підпис A для повідомлення m – (r, s).

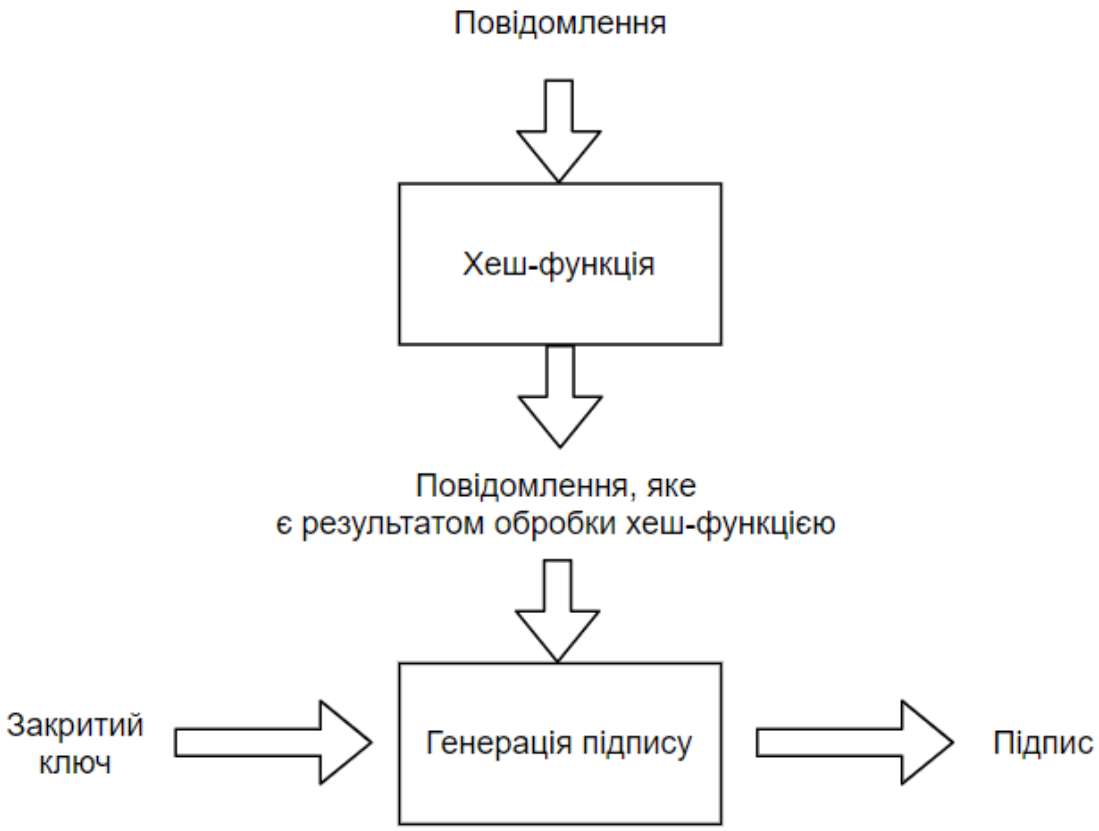

Рис 1.5. Процес генерації підпису ECDSA

ПЕРЕВІРКА ПІДПИСУ ESDCA. Для того, щоб підтвердити підпис (r, s) повідомлення m від особи A, особа B отримує копії параметрів $D = (q, FR, a, b, G, n, h)$ та відкритий ключ Q. В робить наступне:

1. Підтверджує, що r та s цілі в інтервалі $[1, n - 1]$.
2. Обчислює $\mathrm{SHA} - 1(m)$ та переводить двійкові рядки в ціле e.
3. Обчислює $w = s^{-1} mod\ n$.
4. Обчислює $u_1 = ew\ mod\ n$ та $u_2 = rw\ mod\ n$.
5. Обчислює $X = u_1 G + u_2 Q$.
6. Якщо $X =$ □, тоді відхиляє підпис. В іншому випадку, переводить x-координату X в ціле $\overline{x_1}$ та обчислює $v = \overline{x_1} mod\ n$.
7. Підтверджує підпис тільки в тому випадку, коли $v = r$.

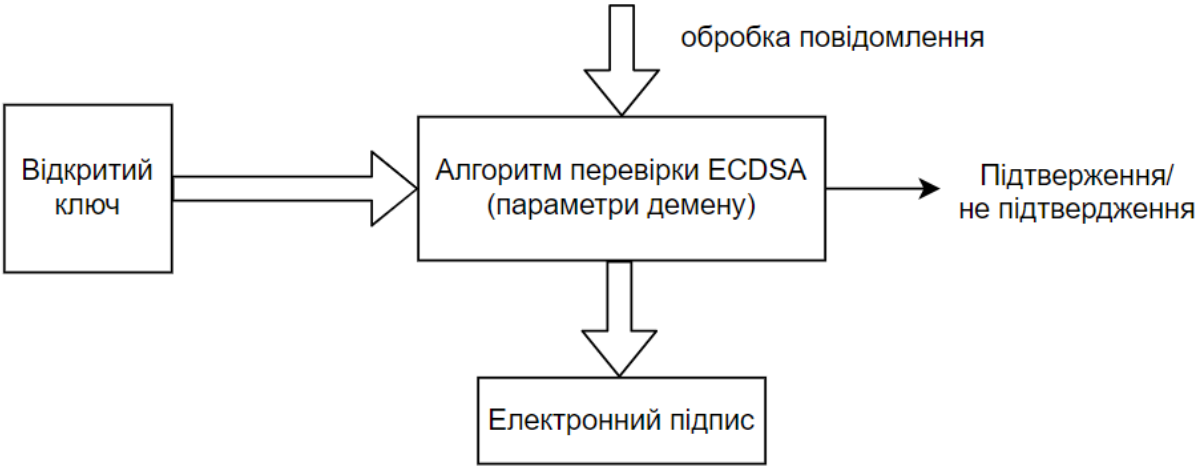

Рис 1.6. Процес перевірки підпису ECDSA

ПІДТВЕРДЖЕННЯ, ЩО ПЕРЕВІРКА ПІДПИСУ ESDCA ПРАЦЮЄ. Якщо підпис (r, s) на повідомлені m був все ж таки згенерован особою А, тоді $s = k^{-1}(e + dr) \bmod n$. Перестановка дає:

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}rd \equiv we + wrd \equiv u_1 + u_2 d \pmod n$$

З цього випливає $u_1 G + u_2 Q = (u_1 + u_2 d)G = kG$, та , як вимагалось – $v = r$.

- RSA

RSA (Rivest, Shamir, and Adleman) була названа на честь вчених із MIT, які вперше описали її в 1977 році. Це асиметричний алгоритм, який використовує відкритий ключ для шифрування, але вимагає інший ключ, відомий тільки отримувачу, для дешифрування. В цій системі, так званій – криптографія відкритого ключа, відкритий ключ – результат множення двох простих чисел. Тільки той результат, довжиною 1024, 2048 або 4096 біт, стає відкритим [12]. Але для дешифрування RSA потрібно знати два простих множники того результату. Тому що немає ще методу обчислення простих множників таких великих чисел, тільки той, хто створив відкритий ключ, також може сгенерувати закритий ключ, який потрібен для дешифрування [14].

ПРОЦЕДУРА ШИФРУВАННЯ ТА ДЕШИФРУВАННЯ RSA. Особа А хоче передати особі В зашифроване повідомлення, використовуючи алгоритм RSA. Криптосистема RSA повинна бути сформована отримувачем, тобто:

1. Особа В повинна обрати два довільних великих простих числа P та Q.
2. Обчислює значення модулю $\mathrm{N} = \mathrm{PQ}$.
3. Обчислює функцію Ейлера $\varphi(N) = (P-1)(Q-1)$ та обирає випадковим чином значення відкритого ключа при умові, що $1 < K_B \leq \varphi(N)$, найбільший спільний дільник $\left(K_B, \varphi(N)\right) = 1$.
4. Обчислює значення секретного ключа k_B, використовуючи розширений алгоритм Євкліда $k_B \equiv K_B^{-1}(mod\varphi(N))$.
5. Через незахищений канал особа В відправляє особі А пару чисел $(\mathrm{N}, \mathrm{KB})$.

Що у свою чергу робить особа А:

1. Особа А розбиває повідомлення m на блоки, кожен іх яких може бути представленим у вигляди числа $M_i = 0,1,2, \dots, N-1$.
2. Шифрує текст у виді послідовності чисел M_i за формулою $C_i = M_i^{K_B}(mod\ N)$ та відправляє криптограму $C_1, C_2, C_3, \dots, C_i$.

Коли особа В отримає прийняту криптограму $C_1, C_2, C_3, \dots, C_i.$, то розшифровує її використовуючи секретний ключ k_B за формулою $M = D_{K_B}(C) = C^{K_B}(mod\ N)$.

У результаті буде отримана послідовність чисел, які і є першопочатковим повідомленням.

СХЕМА ЕЛЕКТРОННОГО ПІДПИСУ RSA. $E_A(x) = x^e\ mod\ n$ – відкрита функція, $D_A(x) = x^d\ mod\ n$ – секретна функція дешифрування [13]:

1. Особа А обчислює хеш-образ $\mathrm{h(m)}$ повідомлення m.

2. Особа А шифрує отримане значення h(m) на своєму секретному ключі d, обчислюючи підпис $s = (h(m))^d \ mod \ n$ та відправляє особі В пару документ-підпис (m, s).

3. Особа В дешифрує за допомогою відкритого ключа відправника, тобто, обчислює $s^e mod \ n$.

4. Особа В обчислює хеш-образ h(m) отриманого повідомлення та перевіряє рівність $h(m) = s^e mod \ n$.

5. Якщо результат перевірки успішний, то підпис приймається, в іншому випадку – не приймається.

У якості хеш-функції у схемі підпису RSA використовуються функції MD.

1.3. Порівняльний аналіз алгоритмів реалізації ЕЦП

Порівнюючи DSA та ECDSA, концептуально, ECDSA – отримана з DSA заміною підгрупи порядку q з Z_p^*, згенерованою g з підгрупою точок на еліптичній кривій, які були згенеровані G. Едина значна відмінність між DSA та ECDSA – генерація r. DSA робить це за допомогою випадкового елементу $X = g^k mod \ p$ та зменшуючи його за модулем p, таким чином, отримуючи ціле в інтервалі $[1, q-1]$. В той час як ECDSA генерує в інтервалі $[1, n-1]$, беручі x-координату довільної точки kG та зменшуючи його за модулем n.

Переваг ECDSA над іншими алгоритмами значно більше, наприклад, підписи ECDSA менші, ніж підписи RSA, хоча мають схожу криптографічну силу. Відкриті ключі ECDSA менші, ніж ключі RSA, які мають схожу силу та приводять до поліпшеня ефективності комунікації [15]. На багатьох платформах операції з ECDSA обчисляються швидше, ніж операції з RSA або DSA, які мають схожу силу. Ці переваги розміру підпису, пропускна здібність та ефективність обчслювання роблять ECDSA кращим вибором для реалізації.

Наприклад, довжина ключа алгоритму DSA становить 2048 біт, у той час як у ECDSA ключ усього 244 біти, хоча їх потужність безпеки не відрізняється [16].

Якщо розглядати окремо алгоритм цифрового підпису RSA, то його недоліком є те, що під час обчислення модуля n, ключів e та d потрібно перевіряти велику кількість додаткових умов, якщо не виконано хоч один з них, є можливість фальсифікації цифрового підпису від того, хто побачить таке невиконання. Якщо таке може статися при підписанні важливих документів, то можуть бути невиправні наслідки. Також, цифровий підпис RSA уразливий до так званої мультипликативній атаці, це означає, що алгоритм цифрового підпису RSA дозволяє криптоаналітику без знання секретного ключа сформувати підписи під тими документами, в котрих результат хешування можна обчислити як множення результатів хешування всіх підписаних документів.

Криптографія відкритого ключа працює використовуючи алгоритми, які простіше опрацювати в одному напрямі та виникають проблеми при опрацюванні в протилежному напрямі. RSA спирається на факт, що множити прості числа для того, щоб отримати більші числа – легко, в той час як повернути великі числа в початкові прості чила – набагато складніше.

Хоча, щоб залишатися безпечним, RSA повинен мати ключі, які 2048 біт або довші. Через це процес стає повільнішим, також, це означає, що розмір ключа – важливий.

В криптографії розмір еліптичної кривої має величезну перевагу, тому що він переходить у більшу потужність для менших, мобільних пристроїв. Це більш легко та потребує менше енергії для рішення дискретного логарифму еліптичної кривої , ніж на розклад на множники, тому, для двох ключів однакового розміру, шифрування RSA має більше недоліків.

Використовуючи ECC, можна досягти такого ж рівню безпеки, використовуючи менші ключі. У світі, в якому мобільні пристрої повинні робити все більше і більше криптографії з меншою очислювальною

потужністю, ECC пропонує безпеку високого рівня з швидкішими, меншими ключами, у порівнянні з RSA [17].

Одною с переваг ECDSA є те, що сторона, яка аутентифікує переферійний пристрій, звільняється від обмежень для зберегання секрету належним чином.

Для кращого розуміння недоліків та переваг алгоритмів реалізації ЕЦП я створила Таблицю 1.1:

Таблиця 1.1. Переваги, недоліки та характеристика алгоритмів реалізації ЕЦП

Назва алгоритму	Переваги	Недоліки	Розмір ключа	Хеш-функція	Час формування ЕЦП	Час верифікації ЕЦП
RSA	Користувач може з легкістю сам змінитичисла, відкритий та закритий ключі	Потрібно перевіряти велику кількість додаткових умов, якщо не виконано хоч один з них, є можливість фальсифікації цифрового підпису від того, хто побачить таке невиконання	512, 1024, 2048 біт	SHA-1	0.029185 s	0.000799 s
	Дозволяє кільком особам обмінюватися інформацією	Працює використовуючи алгоритми, які простіше опрацювати в одному напрямі та виникають проблеми при опрацюванні в				

		протилежному напрямі				
DSA	Можливість обробки за допомогою послідовного алгоритму	Довші підписи	Закритий: 160-256 біт; відкритий: 1024-3072 біт	SHA-1 або SHA-2	0.007979 s	0.009523 s
	При перевірці підпису операції з числами виконуються за модулем довжиною 160 біт, що скорочує об`ЄМ пам`яті	Сильно обмежений в мовах				
ECDSA	Сторона, яка арутентифікує переферійний пристрій, звільняється від обмежень для зберегання секрету належним чином	Існує проблема зі створенням однакового ЦП для двох різних повідомлень	Закритий: 80-521 біт; відкритий : 112-320 біт	SHA-1 або SHA-2	0.004784 s	0.009242 s
	Переваги розміру підпису, пропускна здібність та ефективність обчслювання					
	На багатьох платформах операції з ECDSA обчисляються швидше					

1.4. Висновки до першого розділу

Сьогодні світ обертається навколо комп'ютерів. Стало необхідним мати можливість підписувати цифрове повідомлення. Для цього використовується спеціальне програмне забезпечення для створення цифрового підпису. Конфіденційність стає важливою проблемою, коли особиста інформація передається через Інтернет. Оскільки є люди, що володіють комп'ютерними навичками, які можуть порушити ваші правила, використовуючи або змінюючи ваші особисті файли, було розроблено програмне забезпечення для створення цифрових підписів.

У цьому розділі були розглянуті та проведені порівняльний аналіз таких алгоритмів електронного цифрового підпису як RSA, DSA, ECDSA. Розглянуті теоритичні поняття, які лежать в основі алгоритму ECDSA, а саме: визначення еліптичної кривої, операції над нею, поняття поля Галуа. Розглянуті генерація пареметрів домена, методи генерації ключів та методи їх перевірки, методи генерації и перевірки електроного цифрового підпису.

Хоча ECDSA має недоліки, такі як проблема зі створенням однакового ЦП для двох різних повідомлень, він є потужним алгоритмом ЕЦП і має високу криптографічну стійкість при меншому розмірі ключа у порівнянні з конкурентами. На сьогодняшний день ECDSA є стандартом ANSI, IEEE, NIST та ISO і стандартизується декількома іншими організаціями, що займаються стандартизацією.

У наступному розділі я розглянула алгоритм цифрового підпису на основі алгоритму ECDSA, так як вважаю, що в ньому багато переваг та він один із кращих алгоритмів нашого часу.

РОЗДІЛ 2. ПРОГРАМНИЙ МОДУЛЬ ЗАБЕЗПЕЧЕННЯ ЦІЛІСНОСТІ ІНФОРМАЦІЇ НА БАЗІ АЛГОРИТМУ ECDSA

Цілісність даних - це загальна точність, повнота та послідовність даних. В Рекомендаціях по стандартизації Р 50.1.053-2005 («Технічний захист інформації. Основні терміни і визначення») дається наступне визначення: Цілісність інформації (ресурсів автоматизованої інформаційної системи) — стан інформації (ресурсів автоматизованої інформаційної системи), при якому її (їх) зміна здійснюється тільки навмисно суб'єктами, що мають на нього право [18]. Цілісність даних також відноситься до безпеки даних щодо дотримання нормативних вимог та безпеки [18]. Це підтримується набором процесів, правил та стандартів, впроваджених на етапі проектування. Коли цілісність даних захищена, інформація, що зберігається в базі даних, залишатиметься повною, точною та надійною незалежно від того, як довго вона зберігається або як часто до неї здійснюється доступ. Цілісність даних також гарантує, що ваші дані захищені від будь-яких зовнішніх сил.

Цілісність даних забезпечує захист від широкого кола проблем, які передбачають мутацію даних проти цілей системи. Деякі потенційні проблеми включають [19]:

1. Фізична аварія - біти даних, надіслані через недосконалий носій, можуть бути пошкоджені. Наприклад, бездротовий сигнал може тимчасово загубитися, або провід може отримати шумний електричний сигнал.
2. Цифрова аварія - програмне забезпечення, відповідальне за передачу повідомлення, може мати помилки, які ненавмисно мутують підмножину повідомлень.
3. Зловмисник - людина посеред може змінити повідомлення, щоб заплутати кореспондентів або дізнатись цінну інформацію.

Алгоритм ECDSA використовується для гарантії цілісності даних для запобігання їх фальсифікації. Цілісність даних повідомлення важлива в мережі,

тому що зловмисник може змінити повідомлення, коли воно передається з джерела в пункт призначення.

2.1. Опис середовища розробки

Для реалізації програми на основі алгоритму ECDSA я використала мову програмування Python з версією 3.8, а також PyTest.

Python є інтерпретованою, об`єктно-орінтовною мовою програмування високого рівню з динамічною семантикою. Її вбудовані структури даних виского рівню, що поєднані з динамічною типізацією та динамічною прив`язкою [20]. Python простий, його синтаксис дуже легко вивчити, що підкреслює його легкочитність та, через це знижує вартість обслуговування програми. Python підтримує модулі та пакети, що сприяє модульності програми та повторного використання коду. Python і велика стандартна бібліотека доступні у вихідній формі або двійковій формі безкоштовно для всіх основних платформ, та можуть вільно поширюватись.

Найчастіше, програмісти обирають мову програмування Python через те, що вона надає підвищену продуктивність. Оскільки етап компіляції відсутній, цикл редагування-тестування-налагодження неймовірно швидкий. Налагодження програм Python дуже просте: помилка або невірне введення ніколи не викличе помилку сегментації [20]. Натомість, коли інтерпретатор знаходить помилку, виникає виняток. Коли програма не може зловити виняток, інтерпретатор друкує трасу стеку. Відладчик вихідного рівня дозволяє перевіряти локальні і глобальні змінні, оцінювати довільні вирази, встановлючати точки зупинки, виконувати послідовне виконання коду по рядку за раз і т.д. Відладчик написаний на самій мові Python, що свідчить про інтерспективній силі Python. З іншого боку, часто найшвидший спосіб налагодження програми – це додати декілька операторів друку до джерела:

швидкий цикл редагування-тестування-налагодження робить цей простий підхід дуже ефективним.

Python простий у використанні, але це справжня мова програмування, що пропонує набагато більше структури та підтримки для великих програм, ніж можуть запропонувати сценарії командної оболонки або пакетні файли. З іншого боку, Python також пропонує набагато більше перевірки помилок, ніж C, і, будучи мовою дуже високого рівня, він має вбудовані типи даних високого рівня, такі як гнучкі масиви та словники.

Python дозволяє писати програми компактно і зрозуміло. Програми, написані на Python, зазвичай набагато коротші за еквівалентні програми на C, C ++ або Java з кількох причин [21]:

- типи даних високого рівня дозволяють висловити складні операції в одному висловлюванні;
- групування операторів здійснюється за допомогою відступу замість початкової та кінцевої дужок;
- не потрібні декларації змінних чи аргументів.

Фреймворк PyTest дозволяє легко писати невеликі тексти, але при цьому масштабується для підтримки складного функціонального тестування додатків і бібліотек [22].

Функції PyTest:

- Детальна інформація про невдалі твердження assert (немає потреби запом`ятовувати імена self.asset*);
- Автоматичне виявлення текстових модулів і функцій;
- Модульні пристосування для управління невеликими або параметризованими довго живучими текстовими ресурсами;
- Можна використовувати з Python 3.6+;
- Багата архітектура плагінів.

2.2. Опис програмного забезпечення

З цим розробленим програмним модулем можна швидко створювати пари ключів (підпис ключа та перевірка ключа), підпис повідомлень та верифікації підпису. Також, можна погодитися на спільний секретний ключ, який базується на основі відкритих ключей, що були обміняними. Ключі та підписи дуже короткі, що робить їх обробку простішою та включення в інші протоколи легшим.

Для початку треба створити ключ для підпису (SigningKey). Його можна використовувати для підпису даних шляхом передачі даних у вигляді байтового рядка та повернення підпису (також у вигляді байтового рядка). Також можна запросити у ключа для підпису (SigningKey) відповідний ключ верифікації (VerifyingKey). Ключ верифікації (VerifyingKey) можна використовувати для перевірки підпису шляхом передачі як рядок даних, так і рядок байта підпису: він або повертає True, або помилку BadSignatureError.

```
from ecdsa import SigningKey
sk = SigningKey.generate() # uses NIST192p
vk = sk.verifying_key
signature = sk.sign(b"message")
assert vk.verify(signature, b"message")
```

Рис 2.1. Генерація ключа, формування підпису та його верифікація.

Кожен ключ для підпису (SigningKey) та ключ для верифікації (VerifyingKey) пов`язаний з конкретною кривою, наприклад з NIST192p (за замовчуванням). Довші криві більш захищені, але займають більше часу для використання, та їх результат – довші ключі та підписи.

```
from ecdsa import SigningKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
vk = sk.verifying_key
signature = sk.sign(b"message")
assert vk.verify(signature, b"message")
```

Рис 2.2. Генерація ключа, формування підпису та його верифікація (використовуючи криву NIST384p)

Ключ для підпису (SigningKey) може бути серіалізований в купу різних форматів: найшидшим є виклик $s = sk.to\_string()$, тоді відтворити його можна викликом $SigningKey.from\_string(s, curve)$. Коротка форма не записує криву, тому треба передати ту саму криву, що використовувалась для початкового ключа, до from_string(). Коротка форма ключа для підпису заснованого на NIST192p – довжиною 24 біта. Якщо точка кодування недійсна або вона не належить конкретній кривій, то функція from_string() поверне помилку MalformedPointError.

```
from ecdsa import SigningKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
sk_string = sk.to_string()
sk2 = SigningKey.from_string(sk_string, curve=NIST384p)
print(sk_string.hex())
print(sk2.to_string().hex())
```

Рис 2.3. Генерація ключів (використовуючи криву NIST384p)

$sk.to\_pem()$ та $sk.to\_der()$ будуть серіалізувати ключ для підпису в той самий формат, що використовує OpenSSL.

$SigningKey.from\_pem()/.from\_der()$ скасує цю серіалізацію. Ці формати включають в себе назву кривої, щоб не передавати ідентифікатор кривої до десеріалізації. У випадку якщо файл неправильно сформований from_der() та from_pem() поверне помилку UnexpectedDER або MalformedPointError.

```
from ecdsa import SigningKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
sk_pem = sk.to_pem()
sk2 = SigningKey.from_pem(sk_pem)
# sk and sk2 are the same key
```

Рис 2.4. Конвертація з *pem* та в *pem* (використовуючи криву NIST384p)

Так само, ключ підписання (VerifyingKey) може бути серіалізованим таким самим способом: vk.to_string()/VerifyingKey.from_string(), to_pem()/from_pem(), та to_der()/from_der(). Такий самий curve = аргумент потрібен для VerifyingKey.from_string().

```
from ecdsa import SigningKey, VerifyingKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
vk = sk.verifying_key
vk_string = vk.to_string()
vk2 = VerifyingKey.from_string(vk_string, curve=NIST384p)
# vk and vk2 are the same key

from ecdsa import SigningKey, VerifyingKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
vk = sk.verifying_key
vk_pem = vk.to_pem()
vk2 = VerifyingKey.from_pem(vk_pem)
# vk and vk2 are the same key
```

Рис 2.5. Конвертація згенерованого ключа у *строку* та зі *строки*; конвертація з *pem* та в *pem* (використовуючи криву NIST384p)

Є декілька різних шляхів для формування підпису. По суті, ECDSA приймає число, яке представляє дані, що підписуються, і повертає пару чисел, що представляють підпис. hashfunc = аргумент для sk.sign() та vk.verify() використовується для перетворення довільного рядка в дайджест фіксованої довжини, який потім перетворюється на число, яке ECDSA може підписати, і як

підпис, так і перевірка повинні використовувати однаковий підхід. Значення за замовчуванням - hashlib.sha1, але якщо використовувати NIST256p або довшу криву, то можна використати hashlib.sha256.

Також є кілька шляхів представити підпис. За замовчуванням sk. sign() та vk. verify() методи представляють це як короткий рядок, для простоти та мінімальних накладних витрат. Для використання іншої схеми, можна використати аргументи sk. sign(sigencode =) та vk. verify(sigdecode =). Вони допоміжні функції в модулі ecdsa.util, що можуть бути корисні тут.

У випадку, якщо програма перевірить велику кількість підписів, зроблених одним ключем, можна попередньо обчислити деякі внутрішні значення, щоб зробити перевірку підпису значно швидшою. Точка беззбитковості виникає приблизно при 100 перевірених підписах.

Для виконання попереднього обчислення потрібно викликати метод precompute () на екземплярі ключу для верифікації (VerifyingKey):

```
from ecdsa import SigningKey, NIST384p
sk = SigningKey.generate(curve=NIST384p)
vk = sk.verifying_key
vk.precompute()
signature = sk.sign(b"message")
assert vk.verify(signature, b"message")
```

Рис 2.6. Виклик *precompute ()* для швидкої перевірки підписів

Після виклику precompute () усі перевірки підписів за допомогою цього ключа будуть виконуватися швидше.

Приклад роботи програмного модуля зсередини:

Створюється пара ключів для кривої NIST192p та одразу зберігається на диск:

```
from ecdsa import SigningKey
sk = SigningKey.generate()
vk = sk.verifying_key
with open("private.pem", "wb") as f:
    f.write(sk.to_pem())
with open("public.pem", "wb") as f:
    f.write(vk.to_pem())
```

Рис 2.7. Створення пари ключів та збереження її на диск (використовуючи криву NIST192p)

Завантажує ключ для підпису з диску, використовує його для підпису повідомлення (використовуючи SHA-1) та записує сформований підпис на диск:

```
from ecdsa import SigningKey
with open("private.pem") as f:
    sk = SigningKey.from_pem(f.read())
with open("message", "rb") as f:
    message = f.read()
sig = sk.sign(message)
with open("signature", "wb") as f:
    f.write(sig)
```

Рис 2.8. Завантаження ключа для підпису з диску; підпис повідомлення; запис сформованого підпису на диск

Завантажує ключ для верифікації, повідомлення та підпис з диску, перевіряє підпис (припустимо, хеш SHA-1):

```
from ecdsa import VerifyingKey, BadSignatureError
vk = VerifyingKey.from_pem(open("public.pem").read())
with open("message", "rb") as f:
    message = f.read()
with open("signature", "rb") as f:
    sig = f.read()
try:
    vk.verify(sig, message)
    print "good signature"
except BadSignatureError:
    print "BAD SIGNATURE"
```

Рис 2.9. Завантаження ключа для верифікації, підпису повідомлення, та підпису з диску; перевірка підпису

2.2.1. Блок-схема програмного модуля

Дана блок-схема (рис 2.10) демонструє роботу програмного модуля на основі алгоритму ECDSA. Програма складається з трьох основних гілок: генерація ключів, підпис та верифікація підпису.

Для генерації ключів спочатку необхідно задати параметри кривої, далі генерується випадкове число, яке є закритим ключем, і для нього та заданої кривої рахується відкритий ключ.

Для підпису також необхідно задати криву, для якої було згенеровано пару ключів, задати файл для підпису та закритий ключ. На основі закритого ключа рахується відкритий ключ, хеш файлу, генеруємо випадкове число k, для нього рахуємо точку kG та на основі цього обчислюється пара (r,s), яка є підписом.

Для верифікації підпису вводимо криву, файл для верифікації, відкритий ключ та підпис. Програма перевіряє чи належить відкритий ключ кривій, рахує хеш файлу, рахує u1 та u2, T. У кінці, перевіряється, чи r==T_x, якщо так – верифікує, якщо ні – не верифікує.

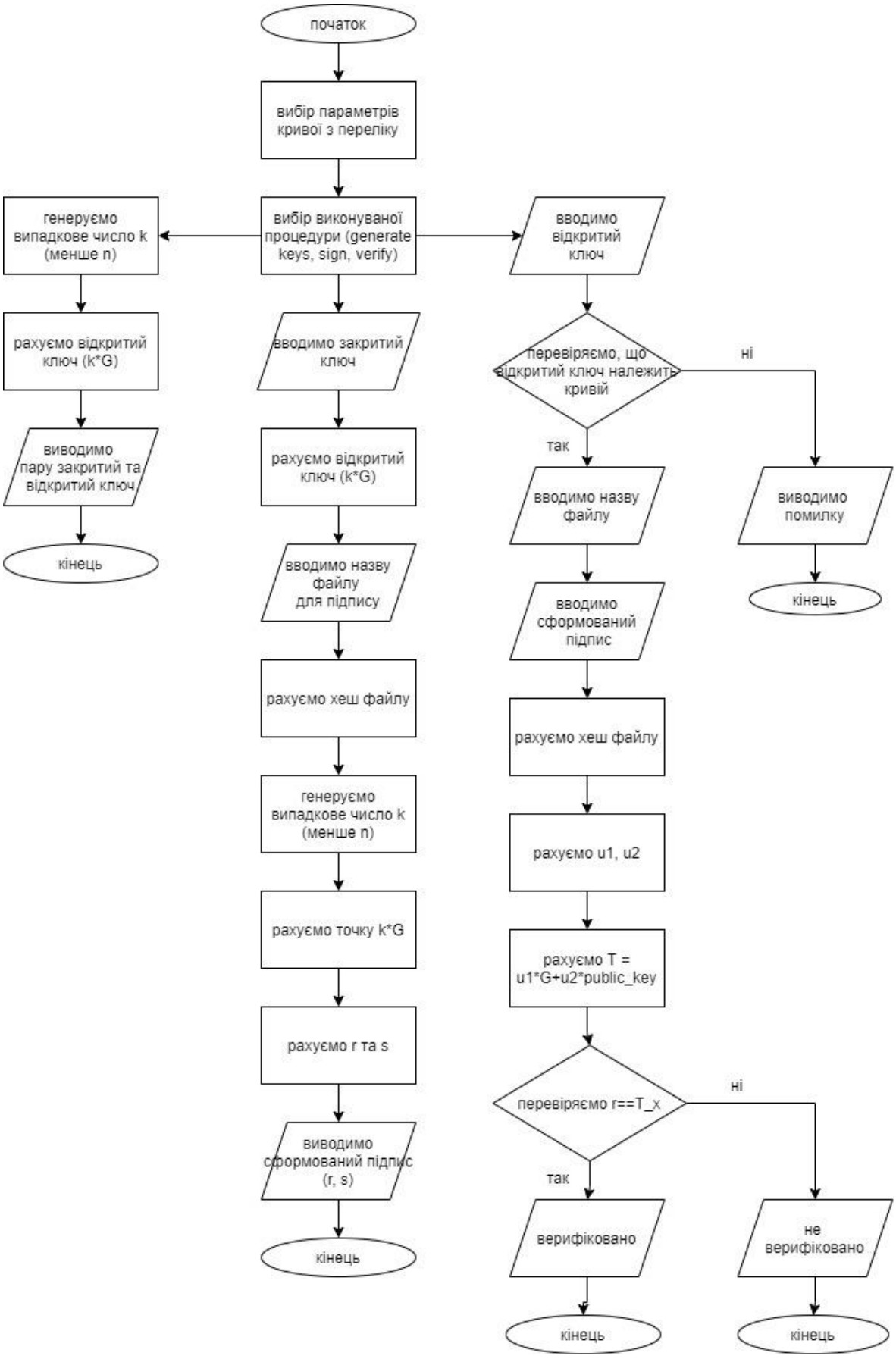

Рис 2.10. Блок-схема програмного модуля

2.3. Дослідження розробленого програмного модуля

2.3.1. Приклад роботи програмного модуля

Використання функціоналу програмного модуля дуже просте. При його запуску з`являється вибір між кривими для генерування ключів, тож ми можемо використати одну з них. Для цього ми вводимо її назву, для прикладу я обрала «NIST192p»:

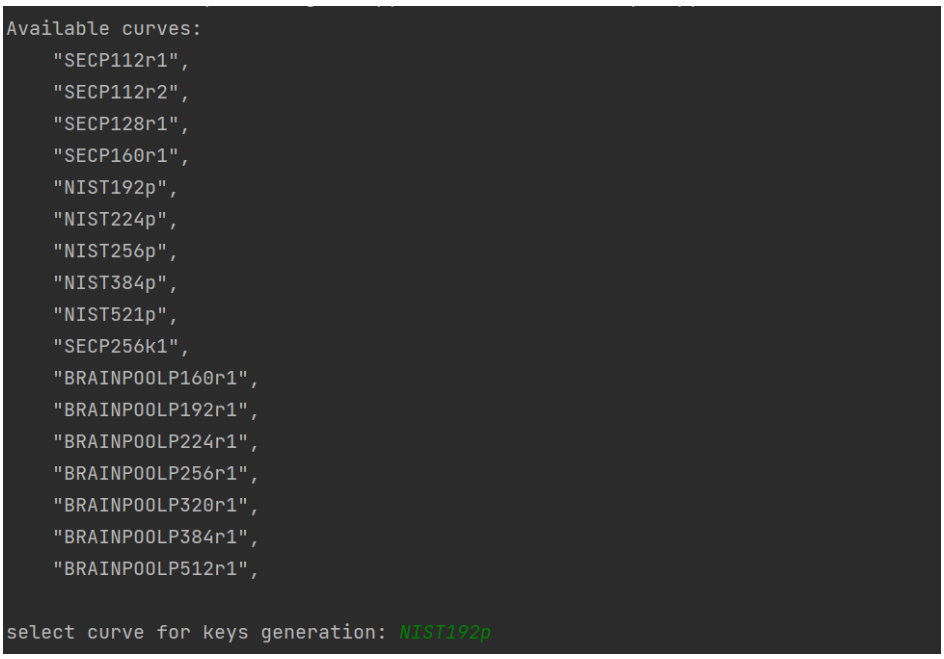

Рис 2.11. Перелік доступних кривих та повідомлення про вибір кривої для генерації ключів

Після вводу обраної кривої програма формує ключ для підпису (Signing Key) та ключ для верифікації (Verify Key), які знадобляться потім:

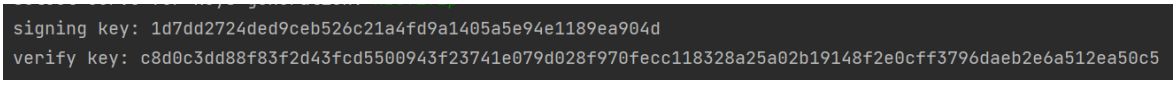

Рис 2.12. Сформовані ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для зручності можна скопіювати назву кривої (NIST192p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key) у блокнот для подільшого використання:

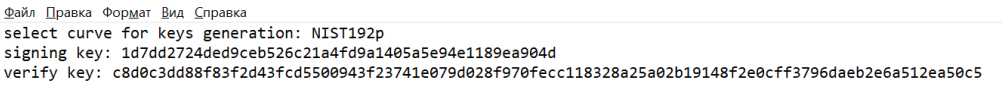

Рис 2.13. Назва кривої (NIST192p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для формування підпису, для цього вводимо назву кривої за допомогою якої було створено ключі (NIST192p):

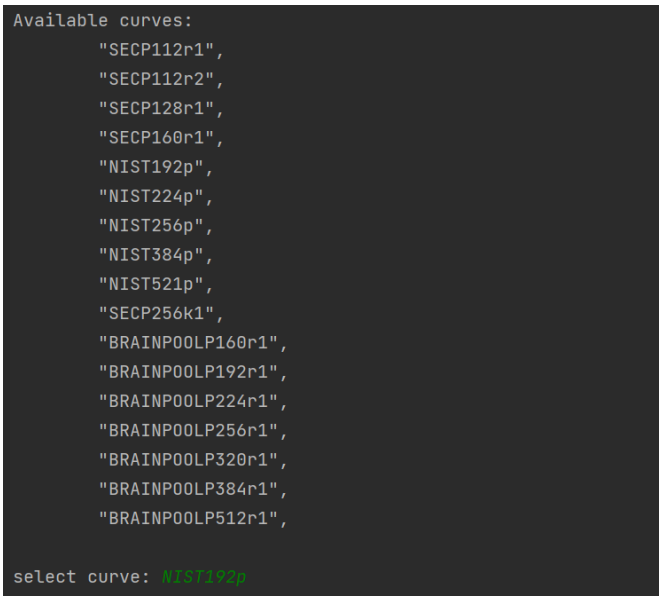

Рис 2.14. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа для підпису (Input Signing Key), який ми зберегли в блокнот, далі вводимо назву файлу, в якому зберігається повідомлення. Після цього з`являється сформований підпис для файлу (Signature), за допомогою якого ми зможемо потім перевіряти його цілісність.

Рис 2.15. Запит на вказання ключа для підпису (Input Signing Key), введення назви файлу; сформований підпис для файлу (Signature)

Для зручності можна скопіювати сформований підпис (Signature) у блокнот для подільшого використання:

```
signature:  dec1ab07c3789f609839bc46b55e9e479b1ec2daffaee1cf08f7d30f2ca9e316462505b5be880cd39963d311b5e6f085
```

Рис 2.16. Сформований підпис для файлу (Signature)

Для перевірки файлу, для цього вводимо назву кривої за допомогою якої було створено ключі (NIST192p):

```
Available curves:
        "SECP112r1",
        "SECP112r2",
        "SECP128r1",
        "SECP160r1",
        "NIST192p",
        "NIST224p",
        "NIST256p",
        "NIST384p",
        "NIST521p",
        "SECP256k1",
        "BRAINPOOLP160r1",
        "BRAINPOOLP192r1",
        "BRAINPOOLP224r1",
        "BRAINPOOLP256r1",
        "BRAINPOOLP320r1",
        "BRAINPOOLP384r1",
        "BRAINPOOLP512r1",

select curve: NIST192p
```

Рис 2.17. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature). Програмний модуль перевіряє чи був файл змінений, з того моменту, як був створений підпис. Як можна побачити, було виведено повідомлення позитивної верифікації (data.txt is verified successfully), тобто, змін не відбувалось:

```
input verifying key: c8d0c3dd88f83f2d43fcd5500943f23741e079d028f970fecc118328a25a02b19148f2e0cff3796daeb2e6a512ea50c5
input file name for signing: data.txt
input signature: dec1ab07c3789f609839bc46b55e9e479b1ec2daffaee1cf08f7d30f2ca9e316462505b5be880cd39963d311b5e6f085
data.txt is verified successfully.
```

Рис 2.18. Запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature); позитивна верифікація

Зобимо провальну верифікацію файлу, для цього вводимо назву кривої за допомогою якої було створено ключі (NIST192p):

```
Available curves:
        "SECP112r1",
        "SECP112r2",
        "SECP128r1",
        "SECP160r1",
        "NIST192p",
        "NIST224p",
        "NIST256p",
        "NIST384p",
        "NIST521p",
        "SECP256k1",
        "BRAINPOOLP160r1",
        "BRAINPOOLP192r1",
        "BRAINPOOLP224r1",
        "BRAINPOOLP256r1",
        "BRAINPOOLP320r1",
        "BRAINPOOLP384r1",
        "BRAINPOOLP512r1",

select curve: NIST192p
```

Рис 2.19. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature). Назву файлу я обрала інший, щоб показати провальну верифікацію. Програмний модуль перевіряє чи був файл змінений, з того моменту, як був створений підпис.

Як можна побачити, було виведено повідомлення провальної верифікації (changed-data.txt is not verified successfully), тобто, відбувались зміни:

```
input verifying key: c8d0c3dd88f83f2d43fcd5500943f23741e079d028f970fecc118328a25a02b19148f2e0cff3796daeb2e6a512ea50c5
input file name for signing: changed-data.txt
input signature: dec1ab07c3789f609839bc46b55e9e479b1ec2daffaee1cf08f7d30f2ca9e316462505b5be880cd39963d311b5e6f085
changed-data.txt is not verified successfully.
```

Рис 2.20. Запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature); провальна верифікація

Таблиця, зображена нижче, демонструє як довго цей програмний модуль генерує пари ключей (keygen), підписує повідомлення (sign), верефікує ті самі підписи (verify) та верефікує підписи без спеціальних попередніх ключових обчислень (no PC verify). Всі ці значення вказані в секундах. Для зручності також надаються їх зворотні значення: скільки ключів в секунду може бути сформовано (keygen / s), скільки підписів може бути зроблено в секунду (sign / s), скільки підписів можна верефікувати в секунду (verify / s) і скільки підписів без конкретного попереднього обчислення ключа можна перевірити в секунду (no PC verify / s). Розмір необробленого підпису (як правило, найменший спосіб кодування підпису) також вказується в стовпці siglen.

Як можна побачити, всі ці значення формуються один за одним:

	siglen	keygen	keygen/s	sign	sign/s	verify	verify/s	no PC verify	no PC verify/s
NIST192p:	48	0.00077s	1293.86	0.00077s	1305.29	0.00142s	702.72	0.00301s	332.63
NIST224p:	56	0.00090s	1106.70	0.00093s	1076.09	0.00184s	544.81	0.00385s	259.74
NIST256p:	64	0.00116s	864.04	0.00131s	764.12	0.00230s	434.34	0.00498s	200.78
NIST384p:	96	0.00249s	401.80	0.00246s	405.71	0.00467s	214.00	0.01032s	96.94
NIST521p:	132	0.00470s	212.57	0.00478s	209.02	0.00924s	108.23	0.01922s	52.03
SECP256k1:	64	0.00117s	854.44	0.00122s	818.62	0.00244s	410.29	0.00459s	217.91

Рис 2.21. Формування таблиці.

	siglen	keygen	keygen/s	sign	sign/s	verify	verify/s	no PC verify	no PC verify/s
NIST192p:	48	0.00077s	1293.86	0.00077s	1305.29	0.00142s	702.72	0.00301s	332.63
NIST224p:	56	0.00090s	1106.70	0.00093s	1076.09	0.00184s	544.81	0.00385s	259.74
NIST256p:	64	0.00116s	864.04	0.00131s	764.12	0.00230s	434.34	0.00498s	200.78
NIST384p:	96	0.00249s	401.80	0.00246s	405.71	0.00467s	214.00	0.01032s	96.94
NIST521p:	132	0.00470s	212.57	0.00478s	209.02	0.00924s	108.23	0.01922s	52.03
SECP256k1:	64	0.00117s	854.44	0.00122s	818.62	0.00244s	410.29	0.00459s	217.91
BRAINPOOLP160r1:	40	0.00058s	1730.37	0.00056s	1786.08	0.00110s	907.31	0.00296s	337.92
BRAINPOOLP192r1:	48	0.00081s	1227.02	0.00076s	1309.82	0.00148s	675.25	0.00314s	318.21
BRAINPOOLP224r1:	56	0.00093s	1075.46	0.00096s	1038.53	0.00195s	513.57	0.00436s	229.36
BRAINPOOLP256r1:	64	0.00135s	738.30	0.00123s	816.24	0.00224s	447.23	0.00526s	190.01
BRAINPOOLP320r1:	80	0.00165s	606.51	0.00171s	584.51	0.00333s	300.64	0.00703s	142.17
BRAINPOOLP384r1:	96	0.00251s	398.09	0.00281s	355.97	0.00484s	206.44	0.01059s	94.46
BRAINPOOLP512r1:	128	0.00462s	216.57	0.00478s	209.23	0.00915s	109.32	0.02079s	48.09
SECP112r1:	28	0.00032s	3168.60	0.00039s	2576.43	0.00062s	1612.82	0.00138s	722.41
SECP112r2:	28	0.00034s	2948.58	0.00035s	2831.69	0.00058s	1712.09	0.00133s	749.60
SECP128r1:	32	0.00038s	2637.32	0.00040s	2485.55	0.00070s	1426.28	0.00168s	596.79
SECP160r1:	42	0.00055s	1832.57	0.00055s	1831.95	0.00102s	979.15	0.00240s	416.30

Рис 2.22. Кінцевий формат таблиці.

2.3.2. Провальна верифікація при внесені змін в ЕЦП

Для початку програмний модуль видає запрос на введення кривої з представленного списку. Для прикладу я обрала криву «NIST256p»:

```
Available curves:
    "SECP112r1",
    "SECP112r2",
    "SECP128r1",
    "SECP160r1",
    "NIST192p",
    "NIST224p",
    "NIST256p",
    "NIST384p",
    "NIST521p",
    "SECP256k1",
    "BRAINPOOLP160r1",
    "BRAINPOOLP192r1",
    "BRAINPOOLP224r1",
    "BRAINPOOLP256r1",
    "BRAINPOOLP320r1",
    "BRAINPOOLP384r1",
    "BRAINPOOLP512r1",

select curve for keys generation: NIST256p
```

Рис 2.23. Перелік доступних кривих та повідомлення про вибір кривої для генерації ключів

Після вводу обраної кривої програма формує ключ для підпису (Signing Key) та ключ для верифікації (Verify Key), які знадобляться потім:

Рис 2.24. Сформовані ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для зручності можна скопіювати назву кривої (NIST256p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key) у блокнот для подільшого використання:

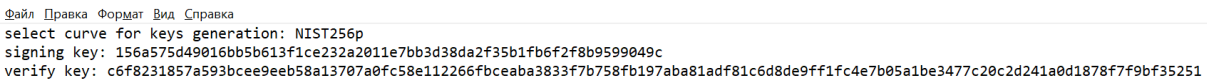

Рис 2.25. Назва кривої (NIST256p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для формування підпису, вводимо назву кривої за допомогою якої було створено ключі (NIST256p):

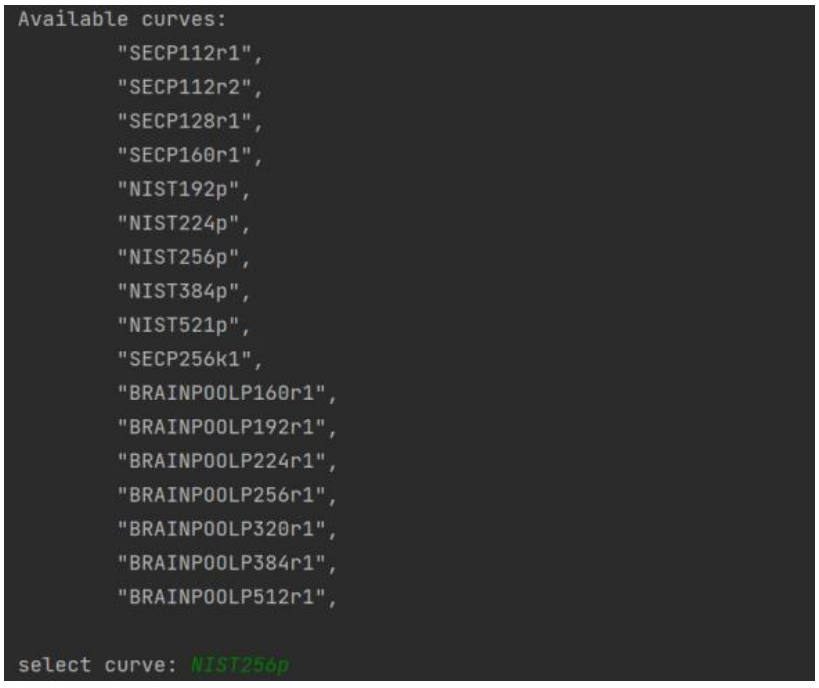

Рис 2.26. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа для підпису (Input Signing Key), який ми зберегли в блокнот, далі вводимо назву файлу, в якому

зберігається повідомлення. Після цього з`являється сформований підпис для файлу (Signature), за допомогою якого ми зможемо потім перевіряти його цілісність:

Рис 2.27. Запит на вказання ключа для підпису (Input Signing Key), введення назви файлу; сформований підпис для файлу (Signature)

Для зручності можна скопіювати сформований підпис (Signature) у блокнот для подільшого використання:

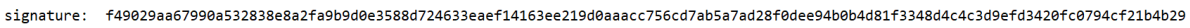

Рис 2.28. Сформований підпис (Signature)

Згенеруємо *другу* пару ключів, для цього вводимо назву кривої, яку ми використали раніше (NIST256p):

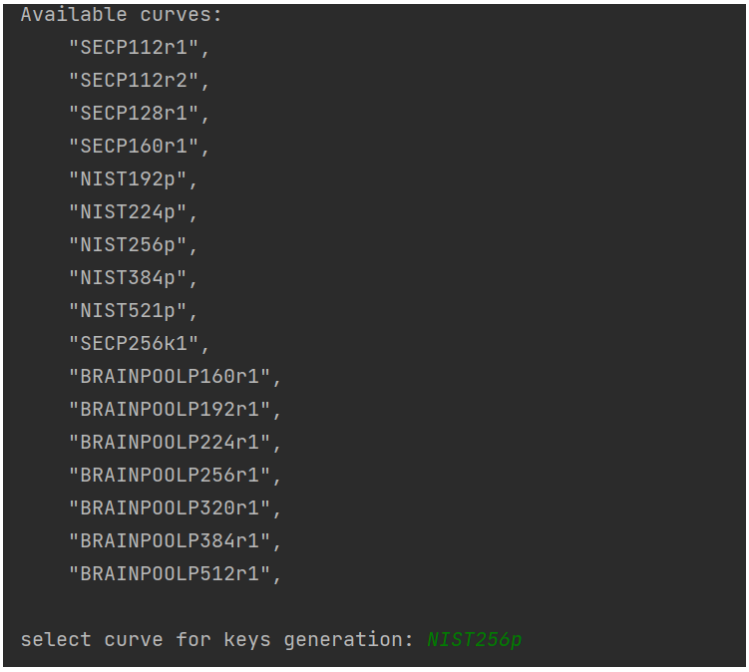

Рис 2.29. Перелік доступних кривих та повідомлення про вибір кривої для генерації другої пари ключів

Після вводу обраної кривої програма формує ключ для підпису (Signing Key) та ключ для верифікації (Verify Key), які знадобляться потім:

Рис 2.30. Сформовані ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для зручності можна скопіювати назву кривої (NIST192p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key) у блокнот для подальшого використання:

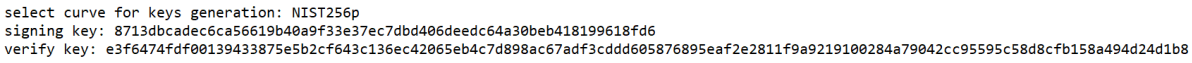

Рис 2.31. Назва кривої (NIST192p), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для формування підпису для *другої* пари ключів, вводимо назву кривої за допомогою якої було створено ключі (NIST256p):

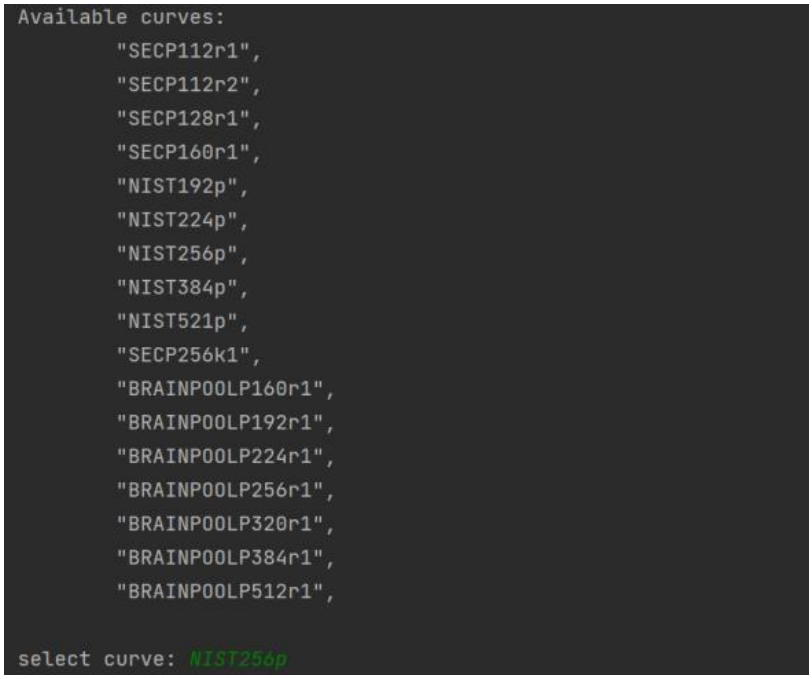

Рис 2.32. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа для підпису (Input Signing Key), який ми зберегли в блокнот, далі вводимо назву файлу, в якому зберігається повідомлення. Після цього з`являється сформований підпис для файлу (Signature), за допомогою якого ми зможемо потім перевіряти його цілісність:

Рис 2.33. Запит на вказання ключа для підпису (Input Signing Key), назву файлу, в якому зберігається повідомлення; сформований підпис для файлу (Signature)

Для зручності можна скопіювати сформований підпис (Signature) у блокнот для подальшого використання:

Рис 2.34. Сформований підпис (Signature)

Зобимо провальну верифікацію при внесені змін в ЕЦП, для цього вводимо назву кривої за допомогою якої було створено ключі (NIST256p):

Рис 2.35. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature). Ключ верифікації я вводжу з другої пари ключів, а підпис з першої пари, щоб відбулась провальна верифікація.

Як можна побачити, було виведено повідомлення провальної верифікації (data.txt is not verified successfully), тобто, підпис не пройшов верифікацію:

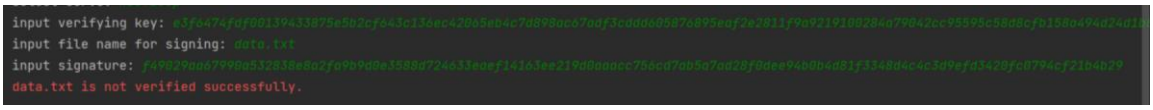

Рис 2.36. Запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature); провальна верифікація

2.3.3. Програмний модуль з додатковою інформацією про розрахунки

У цій частині ми зможемо детально розглянути як рахуються пари ключів, формується підпис та як він верифікується.

Для початку, програмний модуль видає запрос на введення кривої з представленного списку. Для прикладу я обрала криву «SECP112r1»:

Рис 2.37. Перелік доступних кривих та повідомлення про вибір кривої для генерації ключів

Спочатку виводиться повідомлення про назву кривої, яка була обрана, потім параметри кривої, а далі базова точка на цій кривій.

Після цього генерується випадкове число, яке є закритим ключем, а завдяки йому обчислюється відкритий ключ, множенням на базову точку.

У кінці виводяться сформовані ключі: ключ для підпису (Signing Key) та ключ для верифікації(Verify Key).

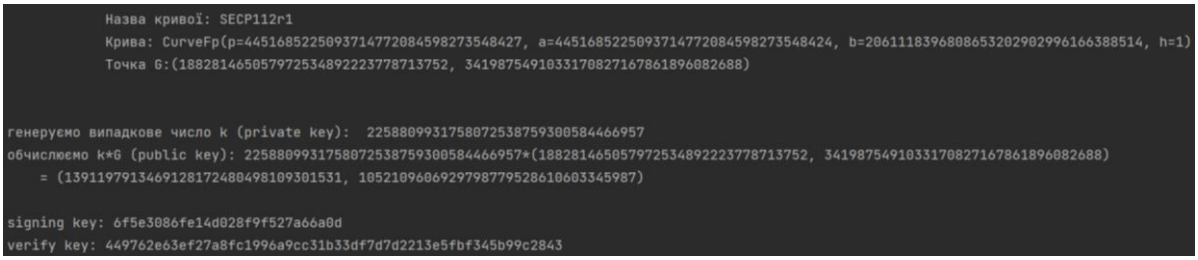

Рис 2.38. Формування ключів; сформовані ключі

Для зручності можна скопіювати назву кривої (SECP112r1), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key) у блокнот для подальшого використання:

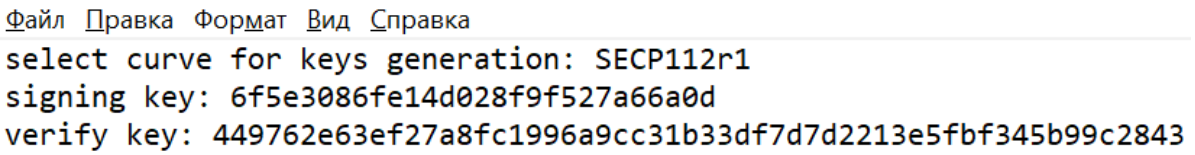

Рис 2.39. Назва кривої (SECP112r1), ключ для підпису (Signing Key) та ключ для верифікації (Verify Key)

Для формування підпису, для цього вводимо назву кривої за допомогою якої було створено ключі (SECP112r1):

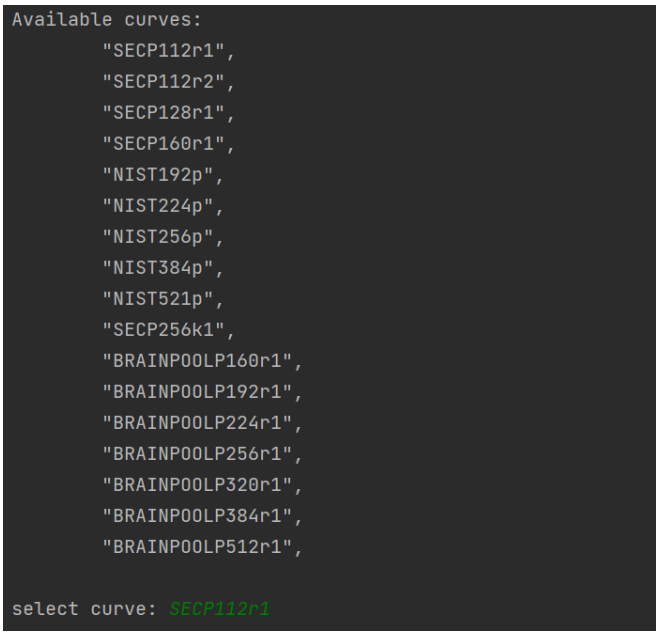

Рис 2.40. Перелік доступних кривих та повідомлення про вибір кривої

Спочатку виводиться повідомлення про назву кривої, яка була обрана, потім параметри кривої, а далі базова точка на цій кривій.

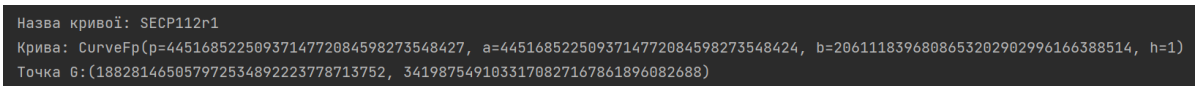

Рис 2.41. Назва кривої, параметри кривої, базова точка на цій кривій

Після цього з`явився запит на вказання ключа для підпису (Input Signing Key), який ми зберегли в блокнот, завдяки йому рахується відкритий ключ (public key), множенням на базову точку. Потім закритий ключ переводиться в десяткому систему числення.

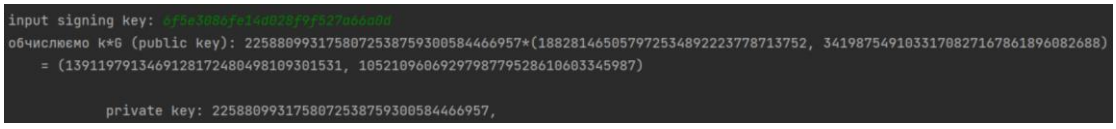

Рис 2.42. Запит на вказання ключа для підпису (Input Signing Key); формування відкритого ключа

Далі вводимо назву файлу, в якому зберігається повідомлення. Рахується хеш, потім генерується випадкове число, а завдяки йому обчислюється точка,

множенням на базову точку. Після цього рахуються прості числа *r* і *s*. Підписом для повідомлення є пара цілих чисел *r* і *s*.

Після цього з`являється сформований підпис для файлу (Signature), за допомогою якого ми зможемо потім перевіряти його цілісність:

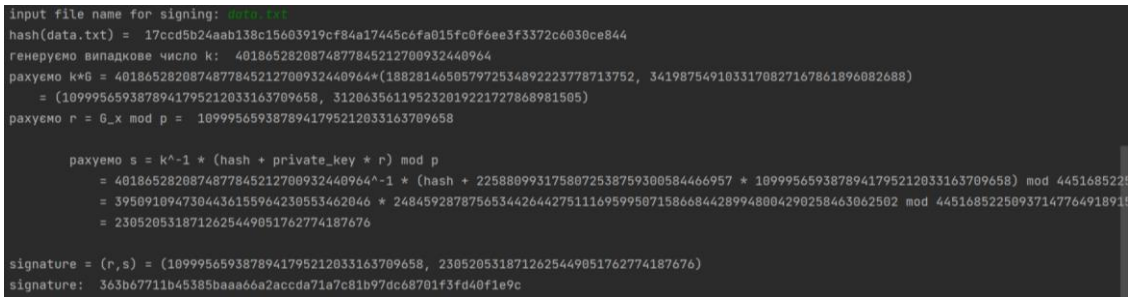

Рис 2.43. Формування підпису; сформований підпис

Для зручності можна скопіювати сформований підпис (Signature) у блокнот для подальшого використання:

```
signature:  363b67711b45385baaa66a2accda71a7c81b97dc68701f3fd40f1e9c
```

Рис 2.44. Сформований підпис (Signature)

Для перевірки файлу, вводимо назву кривої за допомогою якої було створено ключі (SECP112r1):

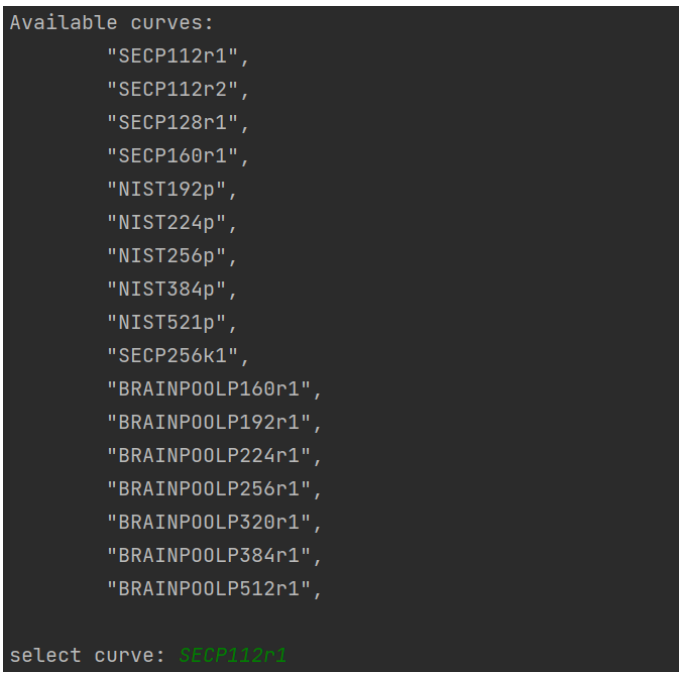

Рис 2.45. Перелік доступних кривих та повідомлення про вибір кривої

Після вводу кривої з`явився запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature).

Програмний модуль перевіряє чи був файл змінений, з того моменту, як був створений підпис. Для цього перевіряється, що числа *r* і *s* є цілими числами з інтервалу, потім рахується хеш-функція від повідомлення та якщо після всіх розрахунків v = r, то підпис приймається.

Як можна побачити, було виведено повідомлення позитивної верифікації (data.txt is verified successfully), тобто, змін не відбувалось:

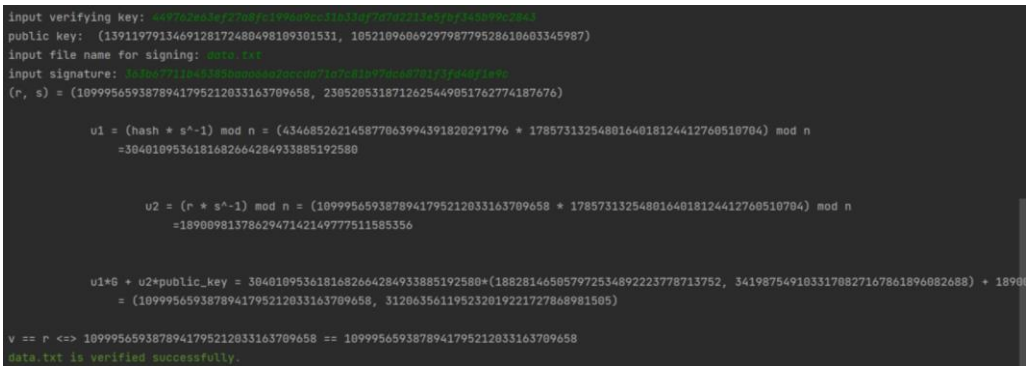

Рис 2.46. Запит на вказання ключа верифікації (Input Veryfing Key), назви файлу, в якому зберігається повідомлення та підпису (Input Signature); процес верифікації; позитивна верифікація

2.4. Висновки до другого розділу

Програмний модуль забезпечення цілісності інформації на основі алгоритму ECDSA розроблено мовою програмування Python, оскільки вона є лаконічною, простою та гнучкою. У сукупності підхід до об`єктно-орієнтовного програмування в Python спрощує програмування, робить код більш зрозумілим і одночасно додає гнучкості мови. Python - це інтерпретована мова, яка може заощадити значний час під час розробки програми, оскільки компіляція та

посилання не потрібні. Інтерпретатор можна використовувати інтерактивно, що дозволяє легко експериментувати з особливостями мови, або тестувати функції під час розробки програми знизу вгору. Це також зручний настільний калькулятор.

Цей програмний модуль забезпечує генерацію ключів, підписання, перевірку для п’яти популярних кривих NIST кривих з довжиною ключів 192, 224, 256, 384 та 521 біт. "Короткими іменами" цих кривих, як відомо інструментом OpenSSL (openssl ecparam - list_curves), є: prime192v1, secp224r1, prime256v1, secp384r1 і secp521r1. Він включає 256-бітну криву secp256k1, яку використовує Bitcoin. Існує також підтримка звичайних варіантів кривих Брейнпула від 160 до 512 біт. "Короткими назвами" цих кривих є: brainpoolP160r1, brainpoolP192r1, brainpoolP224r1, brainpoolP256r1, brainpoolP320r1, brainpoolP384r1, brainpoolP512r1. Невеликі криві зі стандарту SEC також включені (головним чином для прискореного тестування програмного модулю), це: secp112r1, secp112r2, secp128r1 та secp160r1.

Результати тестування програми підтвердили її працездатність та те, що вона відповідає всім вимогам алгоритму ECDSA.

ВИСНОВКИ

Алгоритми шифрування відкритих ключів, такі як криптографія еліптичної кривої (ECC) та алгоритм цифрового підпису еліптичної кривої (ECDSA), широко використовуються у багатьох додатках, особливо в середовищах з обмеженими ресурсами, завдяки ефективності їх використання в цих середовищах. Ці алгоритми мають відповідну ефективність та безпеку для цих середовищ. Обмежені середовища ресурсів, такі як бездротові сенсорні мережі (WSN), ідентифікатор радіочастот (RFID) та смарт-картки, вимагають високої швидкості, низького споживання та меншої пропускної здатності. ECC вважається придатним для цих середовищ з обмеженими джерелами.

ECDSA забезпечує цілісність, аутентифікацію і безвідмовність. ECDSA довела свою ефективність у виконанні, тому що використовує маленькі ключі; таким чином, вартість обчислень невелика в порівнянні з іншими алгоритмами криптографії з відкритим ключем, такими як Rivest Shamir Adleman (RSA), традиційний алгоритм цифрового підпису (DSA). Наприклад, ECDSA з 256-бітовим ключем пропонує той же рівень безпеки, що і алгоритм RSA з 3072-бітовим ключем. Збереження ефективності і безпеки в алгоритмі ECDSA важливо. З одного боку, було розроблено кілька підходів для підвищення ефективності алгоритму ECDSA з метою зниження витрат на обчислення, енергію, пам'ять, і споживання можливостей процесора.

З іншого боку, поліпшення безпеки в ECDSA не менш важливо, ніж його ефективність, тому що цей алгоритм призначений, в першу чергу, для застосування властивостей безпеки. ECDSA, як і попередні алгоритми, може бути підданий деяким уязвимостям безпеки, таким як випадковий слабкий, поганий випадковий джерело або витік бітів закритого ключа. Крім того, багато дослідників внесли поліпшення, щоб залатати дірки в безпеці алгоритму ECC / ECDSA, забезпечивши заходи протидії різним атакам. Але при виборі контрзаходів повинен дотримуватися баланс між безпекою та ефективністю.

Мною було проведено дослідження сучасних практичних алгоритмів формування та верифікації електронно-цифрового підпису, розроблено авторський криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA мовою програмування Python 3.6+ в поєднанні з фреймворком PyTest, що дозволяє проводити процедуру генерації пари ключів, підписання та перевірку електронно-цифрового підпису на базі використання популярних криптостійких еліптичних кривих зі стандарту NIST та SEC, та провести дослідження працездатності власного програмного модуля.

Розроблений програмний модуль дуже гарно справляється з поставленою задачею, містить велику кількість популярних криптостійких еліптичних кривих зі стандарту NIST та SEC, а також має підтримку звичайних варіантів кривих Брейнпула від 160 до 512 біт. Може згенерувати відкритий та закритий ключ для будь-якої кривої представленої на вибір, сформувати за допомогою відкритого ключа підпис, підписати створене повідомлення та з легкістю перевірити його за допомогою закритого ключа та підпису.

Тому, відповідно до поставленої задачі дипломної роботи було виконано наступне:

- проведено дослідження сучасних практичних алгоритмів формування та верифікації електронно-цифрового підпису;
- розроблено авторський криптографічний модуль забезпечення цілісності інформації на базі алгоритму ECDSA;
- проведено дослідження працездатності власного програмного модуля.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке ЕЦП? // Електронний ресурс // Режим доступу до ресурсу: https://www.kmu.gov.ua/usi-pitannya-po-e-poslugam/sho-tak-elektronnij-cifrovij-pidpis-ecp;

2. Електронний цифровий підпис // Електронний ресурс // Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%95%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B8%D0%B9_%D1%86%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B8%D0%B9_%D0%BF%D1%96%D0%B4%D0%BF%D0%B8%D1%81;

3. Закон України «Про електронний цифровий підпис». – К.:Відомості Верховної Ради України, 2003. - 36, ст.276;

4. ЭЦП // Електронний ресурс // Режим доступу до ресурсу: https://portfel.ua/7-listopada-2018-roku-nabuvaye-chinnosti-zakon-ukrayini-pro-elektronni-dovirchi-poslugi-ru/;

5. Закон України «Про електронні довірчі послуги». – К.:Відомості Верховної Ради України, 2017. - N 45, ст.400;

6. ДСТУ 4145-2002. Информационные технологии. Криптографическая защита информации. Цифровая подпись, основанная на эллиптических кривых. Формирование и проверка – Київ. Державний комітет України з питань технічного регулювання та споживчої політики, 2003;

7. Алгоритм DSA // Електронний ресурс // Режим доступу до ресурсу: http://solutionmes.wikidot.com/crypto-dsa;

8. Защита компьютерной информации. Эффективные методы и средства / Шаньгин В. Ф. - М. ДМК Пресс, 2010. – 155 с.;

9. J.-H. Han, Y.-J. Kim, S.-I. Jun, K.-I. Chung, and C.-H. Seo, “Implementation of ecc/ecdsa cryptography algorithms based on java card,” in Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on. IEEE, 2002, pp. 272–276;

10. ECDSA // Електронний ресурс // Режим доступу до ресурсу: https://amp.ru.autograndad.com/754869/1/ecdsa.html;

11. Digital Signature Standard (DSS) // Електронний ресурс // Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf;

12. Защита компьютерной информации. Эффективные методы и средства / Шаньгин В. Ф. - М. ДМК Пресс, 2010. – 141 с.;

13. Иванов М.А., Чугунков И.В. Криптографические методы защиты информации в компьютерных системах и сетях: Учебное пособие / Под ред. М.А. Иванова. М.: НИЯУ МИФИ, 2012. – 198 с.;

14. RSA: от простых чисел до электронной подписи // Електронний ресурс // Режим доступу до ресурсу: https://habr.com/ru/post/534014/;

15. В чем разница между DSA и RSA? // Електронний ресурс // Режим доступу до ресурсу: https://coderoad.ru/2841094/%D0%92-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-DSA-%D0%B8-RSA;

16. ECDSA // Електронний ресурс // Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/ECDSA;

17. Comparing SSH Keys - RSA, DSA, ECDSA // Електронний ресурс // Режим доступу до ресурсу: https://goteleport.com/blog/comparing-ssh-keys/;

18. Цілісність інформації // Електронний ресурс // Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A6%D1%96%D0%BB%D1%96%D1%81%D0%BD%D1%96%D1%81%D1%82%D1%8C_%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D1%96%D1%97;

19. B. Driessen, A. Poschmann, and C. Paar, “Comparison of innovative signature algorithms for wsns,” in Proceedings of the first ACM conference on Wireless network security. ACM, 2008, pp. 30–35;

20. About Python // Електронний ресурс // Режим доступу до ресурсу: https://www.python.org/about;

21. Python и C++ // Електронний ресурс // Режим доступу до ресурсу: https://pythonru.com/baza-znanij/python-ili-c;

22. Pytest — Краткое руководство // Електронний ресурс // Режим доступу до ресурсу: https://coderlessons.com/tutorials/python-technologies/uznaite-pytest/pytest-kratkoe-rukovodstvo.

ДОДАТОК А

«Generate_keys»

```
import ecdsa

def get_curve(name):
    if name == 'SECP112r1':
        return ecdsa.SECP112r1
    if name == 'SECP112r2':
        return ecdsa.SECP112r2
    if name == 'SECP128r1':
        return ecdsa.SECP128r1
    if name == 'SECP160r1':
        return ecdsa.SECP160r1
    if name == 'NIST192p':
        return ecdsa.NIST192p
    if name == 'NIST224p':
        return ecdsa.NIST224p
    if name == 'NIST256p':
        return ecdsa.NIST256p
    if name == 'NIST384p':
        return ecdsa.NIST384p
    if name == 'NIST521p':
        return ecdsa.NIST521p
    if name == 'SECP256k1':
        return ecdsa.SECP256k1
    if name == 'BRAINPOOLP160r1':
        return ecdsa.BRAINPOOLP160r1
    if name == 'BRAINPOOLP192r1':
        return ecdsa.BRAINPOOLP192r1
    if name == 'BRAINPOOLP224r1':
        return ecdsa.BRAINPOOLP224r1
    if name == 'BRAINPOOLP256r1':
        return ecdsa.BRAINPOOLP256r1
    if name == 'BRAINPOOLP320r1':
        return ecdsa.BRAINPOOLP320r1
    if name == 'BRAINPOOLP384r1':
        return ecdsa.BRAINPOOLP384r1
```

Продовження додатку А

```
  if name == 'BRAINPOOLP512r1':

return ecdsa.BRAINPOOLP512r1


def main():
    print("""Available curves:
    "SECP112r1",
    "SECP112r2",
    "SECP128r1",
    "SECP160r1",
    "NIST192p",
    "NIST224p",
    "NIST256p",
    "NIST384p",
    "NIST521p",
    "SECP256k1",
    "BRAINPOOLP160r1",
    "BRAINPOOLP192r1",
    "BRAINPOOLP224r1",
    "BRAINPOOLP256r1",
    "BRAINPOOLP320r1",
    "BRAINPOOLP384r1",
    "BRAINPOOLP512r1",
    """)

    name = input('select curve for keys generation: ')

    curve = get_curve(name)

    print(curve)

    signingKey = ecdsa.SigningKey.generate(curve=curve)
    print("\nsigning key: " + signingKey.to_string().hex())
    print("verify key: " + signingKey.verifying_key.to_string().hex())

if __name__ == '__main__':
    main()
```

Продовження додатку А

«Sign»

```
import ecdsa
import hashlib


def get_curve(name):
    if name == 'SECP112r1':
        return ecdsa.SECP112r1
    if name == 'SECP112r2':
        return ecdsa.SECP112r2
    if name == 'SECP128r1':
        return ecdsa.SECP128r1
    if name == 'SECP160r1':
        return ecdsa.SECP160r1
    if name == 'NIST192p':
        return ecdsa.NIST192p
    if name == 'NIST224p':
        return ecdsa.NIST224p
    if name == 'NIST256p':
        return ecdsa.NIST256p
    if name == 'NIST384p':
        return ecdsa.NIST384p
    if name == 'NIST521p':
        return ecdsa.NIST521p
    if name == 'SECP256k1':
        return ecdsa.SECP256k1
    if name == 'BRAINPOOLP160r1':
        return ecdsa.BRAINPOOLP160r1
    if name == 'BRAINPOOLP192r1':
        return ecdsa.BRAINPOOLP192r1
    if name == 'BRAINPOOLP224r1':
        return ecdsa.BRAINPOOLP224r1
    if name == 'BRAINPOOLP256r1':
        return ecdsa.BRAINPOOLP256r1
    if name == 'BRAINPOOLP320r1':
        return ecdsa.BRAINPOOLP320r1
    if name == 'BRAINPOOLP384r1':
```

Продовження додатку А

```
return ecdsa.BRAINPOOLP384r1

 if name == 'BRAINPOOLP512r1':
      return ecdsa.BRAINPOOLP512r1

def sha256sum(filename):
   h  = hashlib.sha256()
   b  = bytearray(128*1024)
   mv = memoryview(b)
   with open(filename, 'rb', buffering=0) as f:
      for n in iter(lambda : f.readinto(mv), 0):
         h.update(mv[:n])
   return h.hexdigest()

def main():
   print("""Available curves:
      "SECP112r1",
      "SECP112r2",
      "SECP128r1",
      "SECP160r1",
      "NIST192p",
      "NIST224p",
      "NIST256p",
      "NIST384p",
      "NIST521p",
      "SECP256k1",
      "BRAINPOOLP160r1",
      "BRAINPOOLP192r1",
      "BRAINPOOLP224r1",
      "BRAINPOOLP256r1",
      "BRAINPOOLP320r1",
      "BRAINPOOLP384r1",
      "BRAINPOOLP512r1",
      """)

   curveName = input('select curve: ')
   curve = get_curve(curveName)
```

Продовження додатку А

```
print(curve)

    key = input('input signing key: ')
    signingKey = ecdsa.SigningKey.from_string(bytearray.fromhex(key),
curve=curve)

    print(signingKey)

    fileName = input('input file name for signing: ')

    hash = sha256sum(fileName)

    print('hash({}) = '.format(fileName), hash)

    signature = signingKey.sign(bytearray.fromhex(hash))

    print('signature: ', signature.hex())


if __name__ == '__main__':
    main()
```

«Verify»

```
import ecdsa
import hashlib
from termcolor import colored

def get_curve(name):
    if name == 'SECP112r1':
        return ecdsa.SECP112r1
    if name == 'SECP112r2':
        return ecdsa.SECP112r2
    if name == 'SECP128r1':
        return ecdsa.SECP128r1
    if name == 'SECP160r1':
        return ecdsa.SECP160r1
```

Продовження додатку А

```
 if name == 'NIST192p':

  return ecdsa.NIST192p
    if name == 'NIST224p':
       return ecdsa.NIST224p
    if name == 'NIST256p':
       return ecdsa.NIST256p
    if name == 'NIST384p':
       return ecdsa.NIST384p
    if name == 'NIST521p':
       return ecdsa.NIST521p
    if name == 'SECP256k1':
       return ecdsa.SECP256k1
    if name == 'BRAINPOOLP160r1':
       return ecdsa.BRAINPOOLP160r1
    if name == 'BRAINPOOLP192r1':
       return ecdsa.BRAINPOOLP192r1
    if name == 'BRAINPOOLP224r1':
       return ecdsa.BRAINPOOLP224r1
    if name == 'BRAINPOOLP256r1':
       return ecdsa.BRAINPOOLP256r1
    if name == 'BRAINPOOLP320r1':
       return ecdsa.BRAINPOOLP320r1
    if name == 'BRAINPOOLP384r1':
       return ecdsa.BRAINPOOLP384r1
    if name == 'BRAINPOOLP512r1':
       return ecdsa.BRAINPOOLP512r1

def sha256sum(filename):
    h  = hashlib.sha256()
    b  = bytearray(128*1024)
    mv = memoryview(b)
    with open(filename, 'rb', buffering=0) as f:
        for n in iter(lambda : f.readinto(mv), 0):
            h.update(mv[:n])
    return h.hexdigest()
```

Продовження додатку А

```
def main():

print("""Available curves:
        "SECP112r1",
        "SECP112r2",
        "SECP128r1",
        "SECP160r1",
        "NIST192p",
        "NIST224p",
        "NIST256p",
        "NIST384p",
        "NIST521p",
        "SECP256k1",
        "BRAINPOOLP160r1",
        "BRAINPOOLP192r1",
        "BRAINPOOLP224r1",
        "BRAINPOOLP256r1",
        "BRAINPOOLP320r1",
        "BRAINPOOLP384r1",
        "BRAINPOOLP512r1",
        """)

    curveName = input('select curve: ')
    curve = get_curve(curveName)

    key = input('input verifying key: ')
    verifyingKey = ecdsa.VerifyingKey.from_string(bytearray.fromhex(key),
curve=curve)
    verifyingKey.precompute()

    print('public key: ', verifyingKey.pubkey.point)

    fileName = input('input file name for signing: ')

    hash = sha256sum(fileName)

    signature = input('input signature: ')
```

Продовження додатку А

```
 try:
        verification = verifyingKey.verify(bytearray.fromhex(signature),
bytearray.fromhex(hash))
        print(colored(fileName, 'green'), colored('is verified successfully.',
'green'))
    except ecdsa.keys.BadSignatureError:
        print(colored(fileName, 'red'), colored('is not verified successfully.', 'red'))


if __name__ == '__main__':
    main()
```

«ECDSA»

```
from six import int2byte, b
from . import ellipticcurve
from . import numbertheory
from .util import bit_length
from ._compat import remove_whitespace


class RSZeroError(RuntimeError):
    pass


class InvalidPointError(RuntimeError):
    pass


class Signature(object):

    def __init__(self, r, s):
        self.r = r
        self.s = s

    def recover_public_keys(self, hash, generator):
```

Продовження додатку А

```
curve = generator.curve()

        n = generator.order()
        r = self.r
        s = self.s
        e = hash
        x = r

        alpha = (
            pow(x, 3, curve.p()) + (curve.a() * x) + curve.b()
        ) % curve.p()
        beta = numbertheory.square_root_mod_prime(alpha, curve.p())
        y = beta if beta % 2 == 0 else curve.p() - beta

        R1 = ellipticcurve.PointJacobi(curve, x, y, 1, n)
        Q1 = numbertheory.inverse_mod(r, n) * (s * R1 + (-e % n) * generator)
        Pk1 = Public_key(generator, Q1)

        R2 = ellipticcurve.PointJacobi(curve, x, -y, 1, n)
        Q2 = numbertheory.inverse_mod(r, n) * (s * R2 + (-e % n) * generator)
        Pk2 = Public_key(generator, Q2)

        return [Pk1, Pk2]


class Public_key(object):

    def __init__(self, generator, point, verify=True):

        self.curve = generator.curve()
        self.generator = generator
        self.point = point
        n = generator.order()
        p = self.curve.p()
        if not (0 <= point.x() < p) or not (0 <= point.y() < p):
            raise InvalidPointError(
                "The public point has x or y out of range.")
```

Продовження додатку А

```
if verify and not self.curve.contains_point(point.x(), point.y()):
            raise InvalidPointError("Point does not lay on the curve")
        if not n:
            raise InvalidPointError("Generator point must have order.")

        if (
            verify
            and self.curve.cofactor() != 1
            and not n * point == ellipticcurve.INFINITY
        ):
            raise InvalidPointError("Generator point order is bad.")

    def __eq__(self, other):

        if isinstance(other, Public_key):
            return self.curve == other.curve and self.point == other.point
        return NotImplemented

    def __ne__(self, other):

        return not self == other

    def verifies(self, hash, signature):

        G = self.generator
        n = G.order()
        r = signature.r
        s = signature.s
        if r < 1 or r > n - 1:
            return False
        if s < 1 or s > n - 1:
            return False
        c = numbertheory.inverse_mod(s, n)
        u1 = (hash * c) % n

        print("""
```

Продовження додатку А

```
        u1 = (hash * s^-1) mod n = ({h} * {c}) mod n

        \t={u1}
    """.format(h=hash,c=c,u1=u1))

    u2 = (r * c) % n

    print("""
            u2 = (r * s^-1) mod n = ({r} * {c}) mod n
            \t={u2}
        """.format(r=r, c=c, u2=u2))

    if hasattr(G, "mul_add"):
        xy = G.mul_add(u1, self.point, u2)
    else:
        xy = u1 * G + u2 * self.point
    print("""
        u1*G + u2*public_key = {u1}*{G} + {u2}*{pk}
        \t= {xy}
    """.format(u1=u1,u2=u2,G=G,pk=self.point,xy=xy))

    v = xy.x() % n

    print('v == r <=> {v} == {r}'.format(v=v,r=r))
    return v == r


class Private_key(object):

    def __init__(self, public_key, secret_multiplier):


        self.public_key = public_key
        self.secret_multiplier = secret_multiplier

    def __eq__(self, other):
```

Продовження додатку А

```
    if isinstance(other, Private_key):

        return (
            self.public_key == other.public_key
            and self.secret_multiplier == other.secret_multiplier
        )
    return NotImplemented

def __ne__(self, other):

    return not self == other

def sign(self, hash, random_k):

    G = self.public_key.generator
    n = G.order()
    k = random_k % n
    print('генеруємо випадкове число k: ', k)

    ks = k + n
    kt = ks + n
    if bit_length(ks) == bit_length(n):
        p1 = kt * G
    else:
        p1 = ks * G
    print('рахуємо k*G = {k}*{G} \n\t= {kG}'.format(k=k, G=G, kG=p1))
    r = p1.x() % n
    print('рахуємо r = G_x mod p = ', r)
    if r == 0:
        raise RSZeroError("amazingly unlucky random number r")
    s = (
        numbertheory.inverse_mod(k, n)
        * (hash + (self.secret_multiplier * r) % n)
    ) % n

    print("""
    рахуємо s = k^-1 * (hash + private_key * r) mod p
```

Продовження додатку А

```
            = {k}^-1 * (hash + {private_key} * {r}) mod {p}

            = {k_inv} * {sum} mod {p}
            = {s}
        """.format(
            k=k,
            private_key=self.secret_multiplier,
            r=r,
            p=n,
            k_inv=numbertheory.inverse_mod(k, n),
            s=s,
            sum=hash + (self.secret_multiplier * r)
        ))
        if s == 0:
            raise RSZeroError("amazingly unlucky random number s")
        return Signature(r, s)


def int_to_string(x):

    assert x >= 0
    if x == 0:
        return b("\0")
    result = []
    while x:
        ordinal = x & 0xFF
        result.append(int2byte(ordinal))
        x >>= 8

    result.reverse()
    return b("").join(result)


def string_to_int(s):

    result = 0
    for c in s:
```

Продовження додатку А

```
        if not isinstance(c, int):

            c = ord(c)
        result = 256 * result + c
    return result


def digest_integer(m):

    from hashlib import sha1

    return string_to_int(sha1(int_to_string(m)).digest())


def point_is_valid(generator, x, y):

    n = generator.order()
    curve = generator.curve()
    p = curve.p()
    if not (0 <= x < p) or not (0 <= y < p):
        return False
    if not curve.contains_point(x, y):
        return False
    if (
        curve.cofactor() != 1
        and not n * ellipticcurve.PointJacobi(curve, x, y, 1)
        == ellipticcurve.INFINITY
    ):
        return False
    return True


# secp112r1 curve
_p = int(remove_whitespace("DB7C 2ABF62E3 5E668076 BEAD208B"), 16)
# s = 00F50B02 8E4D696E 67687561 51752904 72783FB1
_a = int(remove_whitespace("DB7C 2ABF62E3 5E668076 BEAD2088"), 16)
_b = int(remove_whitespace("659E F8BA0439 16EEDE89 11702B22"), 16)
```

Продовження додатку А

```
_Gx = int(remove_whitespace("09487239 995A5EE7 6B55F9C2 F098"), 16)

_Gy = int(remove_whitespace("A89C E5AF8724 C0A23E0E 0FF77500"), 16)
_r = int(remove_whitespace("DB7C 2ABF62E3 5E7628DF AC6561C5"), 16)
_h = 1
curve_112r1 = ellipticcurve.CurveFp(_p, _a, _b, _h)
generator_112r1 = ellipticcurve.PointJacobi(
    curve_112r1, _Gx, _Gy, 1, _r, generator=True
)


# secp112r2 curve
_p = int(remove_whitespace("DB7C 2ABF62E3 5E668076 BEAD208B"), 16)
# s = 022757A1 114D69E 67687561 51755316 C05E0BD4
_a = int(remove_whitespace("6127 C24C05F3 8A0AAAF6 5C0EF02C"), 16)
_b = int(remove_whitespace("51DE F1815DB5 ED74FCC3 4C85D709"), 16)
_Gx = int(remove_whitespace("4BA30AB5 E892B4E1 649DD092 8643"), 16)
_Gy = int(remove_whitespace("ADCD 46F5882E 3747DEF3 6E956E97"), 16)
_r = int(remove_whitespace("36DF 0AAFD8B8 D7597CA1 0520D04B"), 16)
_h = 4
curve_112r2 = ellipticcurve.CurveFp(_p, _a, _b, _h)
generator_112r2 = ellipticcurve.PointJacobi(
    curve_112r2, _Gx, _Gy, 1, _r, generator=True
)


# secp128r1 curve
_p = int(remove_whitespace("FFFFFFFD FFFFFFFF FFFFFFFF FFFFFFFF"),
16)
# S = 000E0D4D 69E6768 75615175 0CC03A44 73D03679
# a and b are mod p, so a is equal to p-3, or simply -3
# _a = -3
_b = int(remove_whitespace("E87579C1 1079F43D D824993C 2CEE5ED3"),
16)
_Gx = int(remove_whitespace("161FF752 8B899B2D 0C28607C
A52C5B86"), 16)
_Gy = int(remove_whitespace("CF5AC839 5BAFEB13 C02DA292
```

Продовження додатку А

```
DDED7A83"), 16)

    _r = int(remove_whitespace("FFFFFFFE 00000000 75A30D1B 9038A115"),
16)
    _h = 1
    curve_128r1 = ellipticcurve.CurveFp(_p, -3, _b, _h)
    generator_128r1 = ellipticcurve.PointJacobi(
        curve_128r1, _Gx, _Gy, 1, _r, generator=True
    )


    # secp160r1
    _p = int(remove_whitespace("FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
7FFFFFFF"), 16)
    # S = 1053CDE4 2C14D696 E6768756 1517533B F3F83345
    # a and b are mod p, so a is equal to p-3, or simply -3
    # _a = -3
_b = int(remove_whitespace("1C97BEFC 54BD7A8B 65ACF89F 81D4D4AD
C565FA45"), 16)
    _Gx = int(
        remove_whitespace("4A96B568 8EF57328 46646989 68C38BB9
13CBFC82"), 16,
    )
    _Gy = int(
        remove_whitespace("23A62855 3168947D 59DCC912 04235137
7AC5FB32"), 16,
    )
    _r = int(
        remove_whitespace("01 00000000 00000000 0001F4C8 F927AED3
CA752257"), 16,
    )
    _h = 1
    curve_160r1 = ellipticcurve.CurveFp(_p, -3, _b, _h)
    generator_160r1 = ellipticcurve.PointJacobi(
        curve_160r1, _Gx, _Gy, 1, _r, generator=True
    )
```

Продовження додатку А

```
# NIST Curve P-192:
_p = 6277101735386680763835789423207666416083908700390324961279
_r = 6277101735386680763835789423176059013767194773182842284081
# s = 0x3045ae6fc8422f64ed579528d38120eae12196d5L
# c = 0x3099d2bbbfcb2538542dcd5fb078b6ef5f3d6fe2c745de65L
_b = int(
    remove_whitespace(
        """
    64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """
    188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """
    07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811"""
    ),
    16,
)

curve_192 = ellipticcurve.CurveFp(_p, -3, _b, 1)
generator_192 = ellipticcurve.PointJacobi(
    curve_192, _Gx, _Gy, 1, _r, generator=True
)



# NIST Curve P-224:
_p = int(
    remove_whitespace(
```

Продовження додатку А

```
        """

    26959946667150639794667015087019630673557916260026308143510066298881"""
    )
)
_r = int(
    remove_whitespace(
        """
    26959946667150639794667015087019625940457807714424391721682722368061"""
    )
)
# s = 0xbd71344799d5c7fcdc45b59fa3b9ab8f6a948bc5L
# c = 0x5b056c7e11dd68f40469ee7f3c7a7d74f7d121116506d031218291fbL
_b = int(
    remove_whitespace(
        """
    B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """
    B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6 115C1D21"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """
    BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34"""
    ),
```

Продовження додатку А

```
    16,

)

curve_224 = ellipticcurve.CurveFp(_p, -3, _b, 1)
generator_224 = ellipticcurve.PointJacobi(
    curve_224, _Gx, _Gy, 1, _r, generator=True
)

# NIST Curve P-256:
_p = int(
    remove_whitespace(
        """
    115792089210356248762697446949407573530086143415290314195533631308867097853951"""
    )
)
_r = int(
    remove_whitespace(
        """
    115792089210356248762697446949407573529996955224135760342422519061068512044369"""
    )
)
# s = 0xc49d360886e704936a6678e1139d26b7819f7e90L
# c =
0x7efba1662985be9403cb055c75d4f7e0ce8d84a9c5114abcaf3177680104fa0dL
_b = int(
    remove_whitespace(
        """
    5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
```

Продовження додатку А

```
        """
    6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0
    F4A13945 D898C296"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """
    4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE
    CBB64068 37BF51F5"""
    ),
    16,
)

curve_256 = ellipticcurve.CurveFp(_p, -3, _b, 1)
generator_256 = ellipticcurve.PointJacobi(
    curve_256, _Gx, _Gy, 1, _r, generator=True
)

# NIST Curve P-384:
_p = int(
    remove_whitespace(
        """
    394020061963944792122790401001436138050797392704654466679482934042457217714968703290472660882589380018616069731123
    19"""
    )
)
_r = int(
    remove_whitespace(
        """
    394020061963944792122790401001436138050797392704654466679469052796276593991132635693989563081522949135544336539426
    43"""
    )
)

_b = int(
```

Продовження додатку А

```
    remove_whitespace(
        """
    B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112
    0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """
    AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98
    59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """
    3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C
    E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F"""
    ),
    16,
)

curve_384 = ellipticcurve.CurveFp(_p, -3, _b, 1)
generator_384 = ellipticcurve.PointJacobi(
    curve_384, _Gx, _Gy, 1, _r, generator=True
)

# NIST Curve P-521:
_p = int(
    "6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151"
)
_r = int(
    "6864797660130609714981900799081393217269435300143305409394
```

Продовження додатку А

```
    "4463459185543183397655394245057746333217197532963996371 36"
    "3321113864768612440380340372808892707005449"
)

_b = int(
    remove_whitespace(
        """
         051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B
    99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD
    3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """
          C6 858E06B7 0404E9CD 9E3ECB66 2395B442 9C648139
    053FB521 F828AF60 6B4D3DBA A14B5E77 EFE75928 FE1DC127
    A2FFA8DE 3348B3C1 856A429B F97E7E31 C2E5BD66"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """
         118 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449
    579B4468 17AFBD17 273E662C 97EE7299 5EF42640 C550B901
    3FAD0761 353C7086 A272C240 88BE9476 9FD16650"""
    ),
    16,
)

curve_521 = ellipticcurve.CurveFp(_p, -3, _b, 1)
generator_521 = ellipticcurve.PointJacobi(
    curve_521, _Gx, _Gy, 1, _r, generator=True
)
```

Продовження додатку А

```
# Certicom secp256-k1
_a =
0x0000000000000000000000000000000000000000000000000000000000000
000
_b =
0x0000000000000000000000000000000000000000000000000000000000000
007
_p =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
_Gx =
0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
_Gy =
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
_r =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

curve_secp256k1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_secp256k1 = ellipticcurve.PointJacobi(
    curve_secp256k1, _Gx, _Gy, 1, _r, generator=True
)

# Brainpool P-160-r1
_a = 0x340E7BE2A280EB74E2BE61BADA745D97E8F7C300
_b = 0x1E589A8595423412134FAA2DBDEC95C8D8675E58
_p = 0xE95E4A5F737059DC60DFC7AD95B3D8139515620F
_Gx = 0xBED5AF16EA3F6A4F62938C4631EB5AF7BDBCDBC3
_Gy = 0x1667CB477A1A8EC338F94741669C976316DA6321
_q = 0xE95E4A5F737059DC60DF5991D45029409E60FC09

curve_brainpoolp160r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp160r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp160r1, _Gx, _Gy, 1, _q, generator=True
)
```

Продовження додатку А

```
# Brainpool P-192-r1
_a = 0x6A91174076B1E0E19C39C031FE8685C1CAE040E5C69A28EF
_b = 0x469A28EF7C28CCA3DC721D044F4496BCCA7EF4146FBF25C9
_p = 0xC302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297
_Gx = 0xC0A0647EAAB6A48753B033C56CB0F0900A2F5C4853375FD6
_Gy = 0x14B690866ABD5BB88B5F4828C1490002E6773FA2FA299B8F
_q = 0xC302F41D932A36CDA7A3462F9E9E916B5BE8F1029AC4ACC1

curve_brainpoolp192r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp192r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp192r1, _Gx, _Gy, 1, _q, generator=True
)

# Brainpool P-224-r1
_a =
0x68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F
43
_b =
0x2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
_p =
0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
_Gx =
0x0D9029AD2C7E5CF4340823B2A87DC68C9E4CE3174C1E6EFDEE12C0
7D
_Gy =
0x58AA56F772C0726F24C6B89E4ECDAC24354B9E99CAA3F6D3761402
CD
_q =
0xD7C134AA264366862A18302575D0FB98D116BC4B6DDEBCA3A5A793
9F

curve_brainpoolp224r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp224r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp224r1, _Gx, _Gy, 1, _q, generator=True
)
```

Продовження додатку А

```
# Brainpool P-256-r1
_a =
0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B4
4F330B5D9
_b =
0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCD
C18FF8C07B6
_p =
0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481
D1F6E5377
_Gx =
0x8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A445
3BD9ACE3262
_Gy =
0x547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C
72F046997
_q =
0xA9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0
E82974856A7

curve_brainpoolp256r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp256r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp256r1, _Gx, _Gy, 1, _q, generator=True
)

# Brainpool P-320-r1
_a = int(
    remove_whitespace(
        """

3EE30B568FBAB0F883CCEBD46D3F3BB8A2A73513F5EB79DA66190EB
085FFA9
    F492F375A97D860EB4"""
    ),
    16,
)
_b = int(
```

Продовження додатку А

```
    remove_whitespace(
        """

520883949DFDBC42D3AD198640688A6FE13F41349554B49ACC31DCCD
884539
    816F5EB4AC8FB1F1A6"""
    ),
    16,
)
_p = int(
    remove_whitespace(
        """

D35E472036BC4FB7E13C785ED201E065F98FCFA6F6F40DEF4F92B9EC7
893EC
    28FCD412B1F1B32E27"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """

43BD7E9AFB53D8B85289BCC48EE5BFE6F20137D10A087EB6E7871E2A
10A599
    C710AF8D0D39E20611"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """

14FDD05545EC1CC8AB4093247F77275E0743FFED117182EAA9C77877A
AAC6A
    C7D35245D1692E8EE1"""
    ),
```

Продовження додатку А

```
    16,
)
_q = int(
    remove_whitespace(
        """

D35E472036BC4FB7E13C785ED201E065F98FCFA5B68F12A32D482EC7E
E8658
    E98691555B44C59311"""
    ),
    16,
)

curve_brainpoolp320r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp320r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp320r1, _Gx, _Gy, 1, _q, generator=True
)

# Brainpool P-384-r1
_a = int(
    remove_whitespace(
        """

7BC382C63D8C150C3C72080ACE05AFA0C2BEA28E4FB22787139165EF
BA91F9
    0F8AA5814A503AD4EB04A8C7DD22CE2826"""
    ),
    16,
)
_b = int(
    remove_whitespace(
        """

04A8C7DD22CE28268B39B55416F0447C2FB77DE107DCD2A62E880EA5
3EEB62
    D57CB4390295DBC9943AB78696FA504C11"""
    ),
```

Продовження додатку А

```
    16,
)
_p = int(
    remove_whitespace(
        """

8CB91E82A3386D280F5D6F7E50E641DF152F7109ED5456B412B1DA197
FB711
    23ACD3A729901D1A71874700133107EC53"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """

1D1C64F068CF45FFA2A63A81B7C13F6B8847A3E77EF14FE3DB7FCAFE
0CBD10
    E8E826E03436D646AAEF87B2E247D4AF1E"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """

8ABE1D7520F9C2A45CB1EB8E95CFD55262B70B29FEEC5864E19C054F
F991292
    80E4646217791811142820341263C5315"""
    ),
    16,
)
_q = int(
    remove_whitespace(
        """

8CB91E82A3386D280F5D6F7E50E641DF152F7109ED5456B31F166E6CA
```

Продовження додатку А

```
C0425
    A7CF3AB6AF6B7FC3103B883202E9046565"""
    ),
    16,
)

curve_brainpoolp384r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp384r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp384r1, _Gx, _Gy, 1, _q, generator=True
)

# Brainpool P-512-r1
_a = int(
    remove_whitespace(
        """

7830A3318B603B89E2327145AC234CC594CBDD8D3DF91610A83441CA
EA9863

BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1
AC4D77FC94CA"""
    ),
    16,
)
_b = int(
    remove_whitespace(
        """

3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8
B57F1117

A72BF2C7B9E7C1AC4D77FC94CADC083E67984050B75EBAE5DD2809B
D638016F723"""
    ),
    16,
)
_p = int(
```

Продовження додатку А

```
    remove_whitespace(
        """

AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639
CCA703308

717D4D9B009BC66842AECDA12AE6A380E62881FF2F2D82C68528AA60
56583A48F3"""
    ),
    16,
)
_Gx = int(
    remove_whitespace(
        """

81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D916C1B43B62EE
F4D009

8EFF3B1F78E2D0D48D50D1687B93B97D5F7C6D5047406A5E688B35220
9BCB9F822"""
    ),
    16,
)
_Gy = int(
    remove_whitespace(
        """

7DDE385D566332ECC0EABFA9CF7822FDF209F70024A57B1AA000C55B
881F81

11B2DCDE494A5F485E5BCA4BD88A2763AED1CA2B2FA8F0540678CD1
E0F3AD80892"""
    ),
    16,
)
_q = int(
    remove_whitespace(
```

Продовження додатку А

```
        """

AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639
CCA703308

70553E5C414CA92619418661197FAC10471DB1D381085DDADDB587968
29CA90069"""
    ),
    16,
)

curve_brainpoolp512r1 = ellipticcurve.CurveFp(_p, _a, _b, 1)
generator_brainpoolp512r1 = ellipticcurve.PointJacobi(
    curve_brainpoolp512r1, _Gx, _Gy, 1, _q, generator=True
)
```

«Keys»

```
class BadSignatureError(Exception):


    pass


class BadDigestError(Exception):


    pass


class MalformedPointError(AssertionError):

    pass


def _truncate_and_convert_digest(digest, curve, allow_truncate):

    if not allow_truncate:
```

Продовження додатку А

```
        if len(digest) > curve.baselen:
            raise BadDigestError(
                "this curve ({0}) is too short "
                "for the length of your digest ({1})".format(
                    curve.name, 8 * len(digest)
                )
            )
        else:
            digest = digest[: curve.baselen]
        number = string_to_number(digest)
        if allow_truncate:
            max_length = bit_length(curve.order)

            length = len(digest) * 8


            number >>= max(0, length - max_length)

        return number


    class VerifyingKey(object):

        def __str__(self):
            return self.pubkey.point

        def __init__(self, _error__please_use_generate=None):

            if not _error__please_use_generate:
                raise TypeError(
                    "Please use VerifyingKey.generate() to construct me"
                )
            self.curve = None
            self.default_hashfunc = None
            self.pubkey = None

        def __repr__(self):
```

Продовження додатку А

```
    pub_key = self.to_string("compressed")
    return "VerifyingKey.from_string({0!r}, {1!r}, {2})".format(
        pub_key, self.curve, self.default_hashfunc().name
    )

def __eq__(self, other):

    if isinstance(other, VerifyingKey):
        return self.curve == other.curve and self.pubkey == other.pubkey
    return NotImplemented

def __ne__(self, other):

    return not self == other

@classmethod
def from_public_point(
    cls, point, curve=NIST192p, hashfunc=sha1, validate_point=True
):

    self = cls(_error__please_use_generate=True)
    if not isinstance(point, ellipticcurve.PointJacobi):
        point = ellipticcurve.PointJacobi.from_affine(point)
    self.curve = curve
    self.default_hashfunc = hashfunc
    try:
        self.pubkey = ecdsa.Public_key(
            curve.generator, point, validate_point
        )
    except ecdsa.InvalidPointError:
        raise MalformedPointError("Point does not lay on the curve")
    self.pubkey.order = curve.order
    return self

def precompute(self, lazy=False):
```

Продовження додатку А

```
      self.pubkey.point = ellipticcurve.PointJacobi.from_affine(
      self.pubkey.point, True
      )

        if not lazy:
        self.pubkey.point * 2

@staticmethod
def _from_raw_encoding(string, curve):

    order = curve.order

    assert len(string) == curve.verifying_key_length
    xs = string[: curve.verifying_key_length // 2]
    ys = string[curve.verifying_key_length // 2 :]

    assert len(xs) == curve.verifying_key_length // 2
    assert len(ys) == curve.verifying_key_length // 2
    x = string_to_number(xs)
    y = string_to_number(ys)

    return ellipticcurve.PointJacobi(curve.curve, x, y, 1, order)

@staticmethod
def _from_compressed(string, curve):

    if string[:1] not in (b("\x02"), b("\x03")):
        raise MalformedPointError("Malformed compressed point encoding")

    is_even = string[:1] == b("\x02")
    x = string_to_number(string[1:])
    order = curve.order
    p = curve.curve.p()
    alpha = (pow(x, 3, p) + (curve.curve.a() * x) + curve.curve.b()) % p
    try:
        beta = square_root_mod_prime(alpha, p)
    except SquareRootError as e:
```

Продовження додатку А

```
            raise MalformedPointError(
                "Encoding does not correspond to a point on curve", e
            )
        if is_even == bool(beta & 1):
            y = p - beta
        else:
            y = beta
        return ellipticcurve.PointJacobi(curve.curve, x, y, 1, order)

    @classmethod
    def _from_hybrid(cls, string, curve, validate_point):

        assert string[:1] in (b("\x06"), b("\x07"))

        point = cls._from_raw_encoding(string[1:], curve)

        if validate_point and (
            point.y() & 1
            and string[:1] != b("\x07")
            or (not point.y() & 1)
            and string[:1] != b("\x06")
        ):
            raise MalformedPointError("Inconsistent hybrid point encoding")

        return point

    @classmethod
    def from_string(
        cls,
        string,
        curve=NIST192p,
        hashfunc=sha1,
        validate_point=True,
        valid_encodings=None,
    ):

        if valid_encodings is None:
```

Продовження додатку А

```
        valid_encodings = set(
            ["uncompressed", "compressed", "hybrid", "raw"]
        )
    string = normalise_bytes(string)
    sig_len = len(string)
    if sig_len == curve.verifying_key_length and "raw" in valid_encodings:
        point = cls._from_raw_encoding(string, curve)
    elif sig_len == curve.verifying_key_length + 1 and (
        "hybrid" in valid_encodings or "uncompressed" in valid_encodings
    ):
        if (
            string[:1] in (b("\x06"), b("\x07"))
            and "hybrid" in valid_encodings
        ):
            point = cls._from_hybrid(string, curve, validate_point)
        elif string[:1] == b("\x04") and "uncompressed" in valid_encodings:
            point = cls._from_raw_encoding(string[1:], curve)
        else:
            raise MalformedPointError(
                "Invalid X9.62 encoding of the public point"
            )
    elif (
        sig_len == curve.verifying_key_length // 2 + 1
        and "compressed" in valid_encodings
    ):
        point = cls._from_compressed(string, curve)
    else:
        raise MalformedPointError(
            "Length of string does not match lengths of "
            "any of the enabled ({1}) encodings of {0} "
            "curve.".format(curve.name, ", ".join(valid_encodings))
        )
    return cls.from_public_point(point, curve, hashfunc, validate_point)

@classmethod
def from_pem(cls, string, hashfunc=sha1, valid_encodings=None):
```

Продовження додатку А

```
    return cls.from_der(
        der.unpem(string),
        hashfunc=hashfunc,
        valid_encodings=valid_encodings,
    )

@classmethod
def from_der(cls, string, hashfunc=sha1, valid_encodings=None):

    if valid_encodings is None:
        valid_encodings = set(["uncompressed", "compressed", "hybrid"])
    string = normalise_bytes(string)

    s1, empty = der.remove_sequence(string)
    if empty != b"":
        raise der.UnexpectedDER(
            "trailing junk after DER pubkey: %s" % binascii.hexlify(empty)
        )
    s2, point_str_bitstring = der.remove_sequence(s1)

    oid_pk, rest = der.remove_object(s2)
    oid_curve, empty = der.remove_object(rest)
    if empty != b"":
        raise der.UnexpectedDER(
            "trailing junk after DER pubkey objects: %s"
            % binascii.hexlify(empty)
        )
    if not oid_pk == oid_ecPublicKey:
        raise der.UnexpectedDER(
            "Unexpected object identifier in DER "
            "encoding: {0!r}".format(oid_pk)
        )
    curve = find_curve(oid_curve)
    point_str, empty = der.remove_bitstring(point_str_bitstring, 0)
    if empty != b"":
        raise der.UnexpectedDER(
            "trailing junk after pubkey pointstring: %s"
```

Продовження додатку А

```
            % binascii.hexlify(empty)
        )

    if len(point_str) == curve.verifying_key_length:
        raise der.UnexpectedDER("Malformed encoding of public point")
    return cls.from_string(
        point_str,
        curve,
        hashfunc=hashfunc,
        valid_encodings=valid_encodings,
    )

@classmethod
def from_public_key_recovery(
    cls,
    signature,
    data,
    curve,
    hashfunc=sha1,
    sigdecode=sigdecode_string,
    allow_truncate=True,
):

    data = normalise_bytes(data)
    digest = hashfunc(data).digest()
    return cls.from_public_key_recovery_with_digest(
        signature,
        digest,
        curve,
        hashfunc=hashfunc,
        sigdecode=sigdecode,
        allow_truncate=allow_truncate,
    )

@classmethod
def from_public_key_recovery_with_digest(
    cls,
```

Продовження додатку А

```
    signature,
    digest,
    curve,
    hashfunc=sha1,
    sigdecode=sigdecode_string,
    allow_truncate=False,
):

    generator = curve.generator
    r, s = sigdecode(signature, generator.order())
    sig = ecdsa.Signature(r, s)

    digest = normalise_bytes(digest)
    digest_as_number = _truncate_and_convert_digest(
        digest, curve, allow_truncate
    )
    pks = sig.recover_public_keys(digest_as_number, generator)


    verifying_keys = [
        cls.from_public_point(pk.point, curve, hashfunc) for pk in pks
    ]
    return verifying_keys

def _raw_encode(self):

    order = self.curve.curve.p()
    x_str = number_to_string(self.pubkey.point.x(), order)
    y_str = number_to_string(self.pubkey.point.y(), order)
    return x_str + y_str

def _compressed_encode(self):

    order = self.curve.curve.p()
    x_str = number_to_string(self.pubkey.point.x(), order)
    if self.pubkey.point.y() & 1:
        return b("\x03") + x_str
```

Продовження додатку А

```
    else:
        return b("\x02") + x_str

def _hybrid_encode(self):

    raw_enc = self._raw_encode()
    if self.pubkey.point.y() & 1:
        return b("\x07") + raw_enc
    else:
        return b("\x06") + raw_enc

def to_string(self, encoding="raw"):

    assert encoding in ("raw", "uncompressed", "compressed", "hybrid")
    if encoding == "raw":
        return self._raw_encode()
    elif encoding == "uncompressed":
        return b("\x04") + self._raw_encode()
    elif encoding == "hybrid":
        return self._hybrid_encode()
    else:
        return self._compressed_encode()

def to_pem(self, point_encoding="uncompressed"):

    return der.topem(self.to_der(point_encoding), "PUBLIC KEY")

def to_der(self, point_encoding="uncompressed"):

    if point_encoding == "raw":
        raise ValueError("raw point_encoding not allowed in DER")
    point_str = self.to_string(point_encoding)
    return der.encode_sequence(
        der.encode_sequence(
            encoded_oid_ecPublicKey, self.curve.encoded_oid
        ),
```

Продовження додатку А

```
        der.encode_bitstring(point_str, 0),
    )

def verify(
    self,
    signature,
    data,
    hashfunc=None,
    sigdecode=sigdecode_string,
    allow_truncate=True,
):

    data = normalise_bytes(data)

    hashfunc = hashfunc or self.default_hashfunc
    digest = hashfunc(data).digest()
    return self.verify_digest(signature, digest, sigdecode, allow_truncate)

def verify_digest(
    self,
    signature,
    digest,
    sigdecode=sigdecode_string,
    allow_truncate=False,
):

    digest = normalise_bytes(digest)
    number = _truncate_and_convert_digest(
        digest, self.curve, allow_truncate,
    )

    try:
        r, s = sigdecode(signature, self.pubkey.order)
        print('(r, s) = ({r}, {s})'.format(r=r,s=s))
    except (der.UnexpectedDER, MalformedSignature) as e:
        raise BadSignatureError("Malformed formatting of signature", e)
```

Продовження додатку А

```
        sig = ecdsa.Signature(r, s)
        if self.pubkey.verifies(number, sig):
            return True
        raise BadSignatureError("Signature verification failed")


class SigningKey(object):

    def __str__(self):
        return """
            private key: {},
        """.format(self.privkey.secret_multiplier)

    def __init__(self, _error__please_use_generate=None):

        if not _error__please_use_generate:
            raise TypeError("Please use SigningKey.generate() to construct me")
        self.curve = None
        self.default_hashfunc = None
        self.baselen = None
        self.verifying_key = None
        self.privkey = None

    def __eq__(self, other):

        if isinstance(other, SigningKey):
            return (
                self.curve == other.curve
                and self.verifying_key == other.verifying_key
                and self.privkey == other.privkey
            )
        return NotImplemented

    def __ne__(self, other):

        return not self == other
```

Продовження додатку А

```
    @classmethod
    def generate(cls, curve=NIST192p, entropy=None, hashfunc=sha1):

        secexp = randrange(curve.order, entropy)
        print('генеруємо випадкове число k (private key): ', secexp)
        return cls.from_secret_exponent(secexp, curve, hashfunc)

    @classmethod
    def from_secret_exponent(cls, secexp, curve=NIST192p, hashfunc=sha1):

        self = cls(_error__please_use_generate=True)
        self.curve = curve
        self.default_hashfunc = hashfunc
        self.baselen = curve.baselen
        n = curve.order
        if not 1 <= secexp < n:
            raise MalformedPointError(
                "Invalid value for secexp, expected integer "
                "between 1 and {0}".format(n)
            )
        pubkey_point = curve.generator * secexp
        print("обчислюємо k*G (public key): {k}*{G} \n\t=
{pk}".format(k=secexp,G=curve.generator,pk=pubkey_point))
        if hasattr(pubkey_point, "scale"):
            pubkey_point = pubkey_point.scale()
        self.verifying_key = VerifyingKey.from_public_point(
            pubkey_point, curve, hashfunc, False
        )
        pubkey = self.verifying_key.pubkey
        self.privkey = ecdsa.Private_key(pubkey, secexp)
        self.privkey.order = n
        return self

    @classmethod
    def from_string(cls, string, curve=NIST192p, hashfunc=sha1):

        string = normalise_bytes(string)
```

Продовження додатку А

```
    if len(string) != curve.baselen:
        raise MalformedPointError(
            "Invalid length of private key, received {0}, "
            "expected {1}".format(len(string), curve.baselen)
        )
    secexp = string_to_number(string)
    return cls.from_secret_exponent(secexp, curve, hashfunc)

@classmethod
def from_pem(cls, string, hashfunc=sha1):

    if not PY2 and isinstance(string, str):  # pragma: no branch
        string = string.encode()


    private_key_index = string.find(b"-----BEGIN EC PRIVATE KEY-----")
    if private_key_index == -1:
        private_key_index = string.index(b"-----BEGIN PRIVATE KEY-----")

    return cls.from_der(der.unpem(string[private_key_index:]), hashfunc)

@classmethod
def from_der(cls, string, hashfunc=sha1):

    s = normalise_bytes(string)
    curve = None

    s, empty = der.remove_sequence(s)
    if empty != b(""):
        raise der.UnexpectedDER(
            "trailing junk after DER privkey: %s" % binascii.hexlify(empty)
        )

    version, s = der.remove_integer(s)


    if der.is_sequence(s):
```

Продовження додатку А

```
    if version not in (0, 1):
        raise der.UnexpectedDER(
            "expected version '0' or '1' at start of privkey, got %d"
            % version
        )

    sequence, s = der.remove_sequence(s)
    algorithm_oid, algorithm_identifier = der.remove_object(sequence)
    curve_oid, empty = der.remove_object(algorithm_identifier)
    curve = find_curve(curve_oid)

    if algorithm_oid not in (oid_ecPublicKey, oid_ecDH, oid_ecMQV):
        raise der.UnexpectedDER(
            "unexpected algorithm identifier '%s'" % (algorithm_oid,)
        )
    if empty != b"":
        raise der.UnexpectedDER(
            "unexpected data after algorithm identifier: %s"
            % binascii.hexlify(empty)
        )

    s, _ = der.remove_octet_string(s)


    s, empty = der.remove_sequence(s)
    if empty != b(""):
        raise der.UnexpectedDER(
            "trailing junk after DER privkey: %s"
            % binascii.hexlify(empty)
        )

    version, s = der.remove_integer(s)


if version != 1:
    raise der.UnexpectedDER(
        "expected version '1' at start of DER privkey, got %d"
```

Продовження додатку А

```
            % version
        )

    privkey_str, s = der.remove_octet_string(s)

    if not curve:
        tag, curve_oid_str, s = der.remove_constructed(s)
        if tag != 0:
            raise der.UnexpectedDER(
                "expected tag 0 in DER privkey, got %d" % tag
            )
        curve_oid, empty = der.remove_object(curve_oid_str)
        if empty != b(""):
            raise der.UnexpectedDER(
                "trailing junk after DER privkey "
                "curve_oid: %s" % binascii.hexlify(empty)
            )
        curve = find_curve(curve_oid)

    if len(privkey_str) < curve.baselen:
        privkey_str = (
            b("\x00") * (curve.baselen - len(privkey_str)) + privkey_str
        )
    return cls.from_string(privkey_str, curve, hashfunc)

def to_string(self):

    secexp = self.privkey.secret_multiplier
    s = number_to_string(secexp, self.privkey.order)
    return s

def to_pem(self, point_encoding="uncompressed", format="ssleay"):

    assert format in ("ssleay", "pkcs8")
    header = "EC PRIVATE KEY" if format == "ssleay" else "PRIVATE
KEY"
    return der.topem(self.to_der(point_encoding, format), header)
```

Продовження додатку А

```
def to_der(self, point_encoding="uncompressed", format="ssleay"):

    if point_encoding == "raw":
        raise ValueError("raw encoding not allowed in DER")
    assert format in ("ssleay", "pkcs8")
    encoded_vk = self.get_verifying_key().to_string(point_encoding)

    ec_private_key = der.encode_sequence(
        der.encode_integer(1),
        der.encode_octet_string(self.to_string()),
        der.encode_constructed(0, self.curve.encoded_oid),
        der.encode_constructed(1, der.encode_bitstring(encoded_vk, 0)),
    )

    if format == "ssleay":
        return ec_private_key
    else:
        return der.encode_sequence(

            der.encode_integer(1),
            der.encode_sequence(
                der.encode_oid(*oid_ecPublicKey), self.curve.encoded_oid
            ),
            der.encode_octet_string(ec_private_key),
        )

def get_verifying_key(self):

    return self.verifying_key

def sign_deterministic(
    self,
    data,
    hashfunc=None,
    sigencode=sigencode_string,
    extra_entropy=b"",
):
```

Продовження додатку А

```
    hashfunc = hashfunc or self.default_hashfunc
    data = normalise_bytes(data)
    extra_entropy = normalise_bytes(extra_entropy)
    digest = hashfunc(data).digest()

    return self.sign_digest_deterministic(
        digest,
        hashfunc=hashfunc,
        sigencode=sigencode,
        extra_entropy=extra_entropy,
        allow_truncate=True,
    )

def sign_digest_deterministic(
    self,
    digest,
    hashfunc=None,
    sigencode=sigencode_string,
    extra_entropy=b"",
    allow_truncate=False,
):

    secexp = self.privkey.secret_multiplier
    hashfunc = hashfunc or self.default_hashfunc
    digest = normalise_bytes(digest)
    extra_entropy = normalise_bytes(extra_entropy)

    def simple_r_s(r, s, order):
        return r, s, order

    retry_gen = 0
    while True:
        k = rfc6979.generate_k(
            self.curve.generator.order(),
            secexp,
            hashfunc,
```

```
            digest,
            retry_gen=retry_gen,
            extra_entropy=extra_entropy,
        )
        try:
            r, s, order = self.sign_digest(
                digest,
                sigencode=simple_r_s,
                k=k,
                allow_truncate=allow_truncate,
            )
            break
        except RSZeroError:
            retry_gen += 1

    return sigencode(r, s, order)

def sign(
    self,
    data,
    entropy=None,
    hashfunc=None,
    sigencode=sigencode_string,
    k=None,
    allow_truncate=True,
):

    hashfunc = hashfunc or self.default_hashfunc
    data = normalise_bytes(data)
    h = hashfunc(data).digest()
    return self.sign_digest(h, entropy, sigencode, k, allow_truncate)

def sign_digest(
    self,
    digest,
    entropy=None,
    sigencode=sigencode_string,
    k=None,
    allow_truncate=False,
```

Продовження додатку А

```
    ):

        digest = normalise_bytes(digest)
        number = _truncate_and_convert_digest(
            digest, self.curve, allow_truncate,
        )
        r, s = self.sign_number(number, entropy, k)
        print('signature = (r,s) = ({r}, {s})'.format(r=r,s=s))
        return sigencode(r, s, self.privkey.order)

    def sign_number(self, number, entropy=None, k=None):

        order = self.privkey.order

        if k is not None:
            _k = k
        else:
            _k = randrange(order, entropy)

        assert 1 <= _k < order
        sig = self.privkey.sign(number, _k)
        return sig.r, sig.s
```

«Eleptic Curve»

```
from __future__ import division

try:
    from gmpy2 import mpz

    GMPY = True
except ImportError:
    try:
        from gmpy import mpz

        GMPY = True
    except ImportError:
        GMPY = False
```

Продовження додатку А

```
from six import python_2_unicode_compatible
from . import numbertheory


@python_2_unicode_compatible
class CurveFp(object):

    if GMPY:  # pragma: no branch

        def __init__(self, p, a, b, h=None):

            self.__p = mpz(p)
            self.__a = mpz(a)
            self.__b = mpz(b)



            self.__h = h

    else:

        def __init__(self, p, a, b, h=None):

            self.__p = p
            self.__a = a
            self.__b = b
            self.__h = h

    def __eq__(self, other):

        if isinstance(other, CurveFp):
            return (
                self.__p == other.__p
                and self.__a == other.__a
                and self.__b == other.__b
            )
        return NotImplemented
```

Продовження додатку А

```
    def __ne__(self, other):

        return not self == other

    def __hash__(self):
        return hash((self.__p, self.__a, self.__b))

    def p(self):
        return self.__p

    def a(self):
        return self.__a

    def b(self):
        return self.__b

    def cofactor(self):
        return self.__h

    def contains_point(self, x, y):

        return (y * y - ((x * x + self.__a) * x + self.__b)) % self.__p == 0

    def __str__(self):
        return "CurveFp(p=%d, a=%d, b=%d, h=%d)" % (
            self.__p,
            self.__a,
            self.__b,
            self.__h,
        )


class PointJacobi(object):

    def __str__(self):
        return """({x}, {y})""".format(x=self.x(), y=self.y())
```

Продовження додатку А

```
def __init__(self, curve, x, y, z, order=None, generator=False):

    self.__curve = curve
    if GMPY:
        self.__coords = (mpz(x), mpz(y), mpz(z))
        self.__order = order and mpz(order)
    else:
        self.__coords = (x, y, z)
        self.__order = order
    self.__generator = generator
    self.__precompute = []

def _maybe_precompute(self):
    if not self.__generator or self.__precompute:
        return

    order = self.__order
    assert order
    precompute = []
    i = 1
    order *= 2
    coord_x, coord_y, coord_z = self.__coords
    doubler = PointJacobi(self.__curve, coord_x, coord_y, coord_z, order)
    order *= 2
    precompute.append((doubler.x(), doubler.y()))

    while i < order:
        i *= 2
        doubler = doubler.double().scale()
        precompute.append((doubler.x(), doubler.y()))

    self.__precompute = precompute

def __getstate__(self):

    state = self.__dict__.copy()
```

Продовження додатку А

```
    return state

def __setstate__(self, state):
    self.__dict__.update(state)

def __eq__(self, other):

    x1, y1, z1 = self.__coords
    if other is INFINITY:
        return not y1 or not z1
    if isinstance(other, Point):
        x2, y2, z2 = other.x(), other.y(), 1
    elif isinstance(other, PointJacobi):
        x2, y2, z2 = other.__coords
    else:
        return NotImplemented
    if self.__curve != other.curve():
        return False
    p = self.__curve.p()

    zz1 = z1 * z1 % p
    zz2 = z2 * z2 % p


    return (x1 * zz2 - x2 * zz1) % p == 0 and (
        y1 * zz2 * z2 - y2 * zz1 * z1
    ) % p == 0

def __ne__(self, other):

    return not self == other

def order(self):

    return self.__curve

def x(self):
```

Продовження додатку А

```
    x, _, z = self.__coords
    if z == 1:
        return x
    p = self.__curve.p()
    z = numbertheory.inverse_mod(z, p)
    return x * z ** 2 % p

def y(self):

    _, y, z = self.__coords
    if z == 1:
        return y
    p = self.__curve.p()
    z = numbertheory.inverse_mod(z, p)
    return y * z ** 3 % p

def scale(self):
    x, y, z = self.__coords
    if z == 1:
        return self



    p = self.__curve.p()
    z_inv = numbertheory.inverse_mod(z, p)
    zz_inv = z_inv * z_inv % p
    x = x * zz_inv % p
    y = y * zz_inv * z_inv % p
    self.__coords = (x, y, 1)
    return self

def to_affine(self):

    _, y, z = self.__coords
    if not y or not z:
        return INFINITY
    self.scale()
```

Продовження додатку А

```
    x, y, z = self.__coords
    return Point(self.__curve, x, y, self.__order)

@staticmethod
def from_affine(point, generator=False):

        return PointJacobi(
        point.curve(), point.x(), point.y(), 1, point.order(), generator
    )

    def _double_with_z_1(self, X1, Y1, p, a):


    XX, YY = X1 * X1 % p, Y1 * Y1 % p
    if not YY:
        return 0, 0, 1
    YYYY = YY * YY % p
    S = 2 * ((X1 + YY) ** 2 - XX - YYYY) % p
    M = 3 * XX + a
    T = (M * M - 2 * S) % p

    Y3 = (M * (S - T) - 8 * YYYY) % p
    Z3 = 2 * Y1 % p
    return T, Y3, Z3

def _double(self, X1, Y1, Z1, p, a):

    if Z1 == 1:
        return self._double_with_z_1(X1, Y1, p, a)
    if not Y1 or not Z1:
        return 0, 0, 1

    XX, YY = X1 * X1 % p, Y1 * Y1 % p
    if not YY:
        return 0, 0, 1
    YYYY = YY * YY % p
    ZZ = Z1 * Z1 % p
```

Продовження додатку А

```
    S = 2 * ((X1 + YY) ** 2 - XX - YYYY) % p
    M = (3 * XX + a * ZZ * ZZ) % p
    T = (M * M - 2 * S) % p

    Y3 = (M * (S - T) - 8 * YYYY) % p
    Z3 = ((Y1 + Z1) ** 2 - YY - ZZ) % p

    return T, Y3, Z3

def double(self):

    X1, Y1, Z1 = self.__coords

    if not Y1:
        return INFINITY

    p, a = self.__curve.p(), self.__curve.a()

    X3, Y3, Z3 = self._double(X1, Y1, Z1, p, a)

    if not Y3 or not Z3:
        return INFINITY
    return PointJacobi(self.__curve, X3, Y3, Z3, self.__order)

def _add_with_z_1(self, X1, Y1, X2, Y2, p):

    H = X2 - X1
    HH = H * H
    I = 4 * HH % p
    J = H * I
    r = 2 * (Y2 - Y1)
    if not H and not r:
        return self._double_with_z_1(X1, Y1, p, self.__curve.a())
    V = X1 * I
    X3 = (r ** 2 - J - 2 * V) % p
    Y3 = (r * (V - X3) - 2 * Y1 * J) % p
    Z3 = 2 * H % p
```

```
    return X3, Y3, Z3

def _add_with_z_eq(self, X1, Y1, Z1, X2, Y2, p):

    A = (X2 - X1) ** 2 % p
    B = X1 * A % p
    C = X2 * A
    D = (Y2 - Y1) ** 2 % p
    if not A and not D:
        return self._double(X1, Y1, Z1, p, self.__curve.a())
    X3 = (D - B - C) % p
    Y3 = ((Y2 - Y1) * (B - X3) - Y1 * (C - B)) % p
    Z3 = Z1 * (X2 - X1) % p
    return X3, Y3, Z3

def _add_with_z2_1(self, X1, Y1, Z1, X2, Y2, p):

    Z1Z1 = Z1 * Z1 % p
    U2, S2 = X2 * Z1Z1 % p, Y2 * Z1 * Z1Z1 % p
    H = (U2 - X1) % p
    HH = H * H % p
    I = 4 * HH % p
    J = H * I
    r = 2 * (S2 - Y1) % p
    if not r and not H:
        return self._double_with_z_1(X2, Y2, p, self.__curve.a())
    V = X1 * I
    X3 = (r * r - J - 2 * V) % p
    Y3 = (r * (V - X3) - 2 * Y1 * J) % p
    Z3 = ((Z1 + H) ** 2 - Z1Z1 - HH) % p
    return X3, Y3, Z3

def _add_with_z_ne(self, X1, Y1, Z1, X2, Y2, Z2, p):

    Z1Z1 = Z1 * Z1 % p
    Z2Z2 = Z2 * Z2 % p
    U1 = X1 * Z2Z2 % p
```

Продовження додатку А

```
    U2 = X2 * Z1Z1 % p
    S1 = Y1 * Z2 * Z2Z2 % p
    S2 = Y2 * Z1 * Z1Z1 % p
    H = U2 - U1
    I = 4 * H * H % p
    J = H * I % p
    r = 2 * (S2 - S1) % p
    if not H and not r:
        return self._double(X1, Y1, Z1, p, self.__curve.a())
    V = U1 * I
    X3 = (r * r - J - 2 * V) % p
    Y3 = (r * (V - X3) - 2 * S1 * J) % p
    Z3 = ((Z1 + Z2) ** 2 - Z1Z1 - Z2Z2) * H % p

    return X3, Y3, Z3

def __radd__(self, other):

    return self + other

def _add(self, X1, Y1, Z1, X2, Y2, Z2, p):

    if not Y1 or not Z1:
        return X2, Y2, Z2
    if not Y2 or not Z2:
        return X1, Y1, Z1
    if Z1 == Z2:
        if Z1 == 1:
            return self._add_with_z_1(X1, Y1, X2, Y2, p)
        return self._add_with_z_eq(X1, Y1, Z1, X2, Y2, p)
    if Z1 == 1:
        return self._add_with_z2_1(X2, Y2, Z2, X1, Y1, p)
    if Z2 == 1:
        return self._add_with_z2_1(X1, Y1, Z1, X2, Y2, p)
    return self._add_with_z_ne(X1, Y1, Z1, X2, Y2, Z2, p)

def __add__(self, other):
```

Продовження додатку А

```
    if self == INFINITY:
        return other
    if other == INFINITY:
        return self
    if isinstance(other, Point):
        other = PointJacobi.from_affine(other)
    if self.__curve != other.__curve:
        raise ValueError("The other point is on different curve")

    p = self.__curve.p()
    X1, Y1, Z1 = self.__coords
    X2, Y2, Z2 = other.__coords

    X3, Y3, Z3 = self._add(X1, Y1, Z1, X2, Y2, Z2, p)

    if not Y3 or not Z3:
        return INFINITY
    return PointJacobi(self.__curve, X3, Y3, Z3, self.__order)

def __rmul__(self, other):

    return self * other

def _mul_precompute(self, other):

    X3, Y3, Z3, p = 0, 0, 1, self.__curve.p()
    _add = self._add
    for X2, Y2 in self.__precompute:
        if other % 2:
            if other % 4 >= 2:
                other = (other + 1) // 2
                X3, Y3, Z3 = _add(X3, Y3, Z3, X2, -Y2, 1, p)
            else:
                other = (other - 1) // 2
                X3, Y3, Z3 = _add(X3, Y3, Z3, X2, Y2, 1, p)
        else:
```

Продовження додатку А

```
            other //= 2

        if not Y3 or not Z3:
            return INFINITY
        return PointJacobi(self.__curve, X3, Y3, Z3, self.__order)

    @staticmethod
    def _naf(mult):

        ret = []
        while mult:
            if mult % 2:
                nd = mult % 4
                if nd >= 2:
                    nd -= 4
                ret.append(nd)
                mult -= nd
            else:
                ret.append(0)
            mult //= 2
        return ret

    def __mul__(self, other):

        if not self.__coords[1] or not other:
            return INFINITY
        if other == 1:
            return self
        if self.__order:

            other = other % (self.__order * 2)
        self._maybe_precompute()
        if self.__precompute:
            return self._mul_precompute(other)

        self = self.scale()
        X2, Y2, _ = self.__coords
```

Продовження додатку А

```
    X3, Y3, Z3 = 0, 0, 1
    p, a = self.__curve.p(), self.__curve.a()
    _double = self._double
    _add = self._add

    for i in reversed(self._naf(other)):
        X3, Y3, Z3 = _double(X3, Y3, Z3, p, a)
        if i < 0:
            X3, Y3, Z3 = _add(X3, Y3, Z3, X2, -Y2, 1, p)
        elif i > 0:
            X3, Y3, Z3 = _add(X3, Y3, Z3, X2, Y2, 1, p)

    if not Y3 or not Z3:
        return INFINITY

    return PointJacobi(self.__curve, X3, Y3, Z3, self.__order)

def mul_add(self, self_mul, other, other_mul):

    if other == INFINITY or other_mul == 0:
        return self * self_mul
    if self_mul == 0:
        return other * other_mul
    if not isinstance(other, PointJacobi):
        other = PointJacobi.from_affine(other)

    self._maybe_precompute()
    other._maybe_precompute()
    if self.__precompute and other.__precompute:
        return self * self_mul + other * other_mul

    if self.__order:
        self_mul = self_mul % self.__order
        other_mul = other_mul % self.__order



    X3, Y3, Z3 = 0, 0, 1
```

Продовження додатку А

```
p, a = self.__curve.p(), self.__curve.a()



self.scale()
X1, Y1, Z1 = self.__coords
other.scale()
X2, Y2, Z2 = other.__coords

_double = self._double
_add = self._add


mAmB_X, mAmB_Y, mAmB_Z = _add(X1, -Y1, Z1, X2, -Y2, Z2, p)
pAmB_X, pAmB_Y, pAmB_Z = _add(X1, Y1, Z1, X2, -Y2, Z2, p)
mApB_X, mApB_Y, mApB_Z = _add(X1, -Y1, Z1, X2, Y2, Z2, p)
pApB_X, pApB_Y, pApB_Z = _add(X1, Y1, Z1, X2, Y2, Z2, p)

if not pApB_Y or not pApB_Z:
    return self * self_mul + other * other_mul


self_naf = list(reversed(self._naf(int(self_mul))))
other_naf = list(reversed(self._naf(int(other_mul))))

if len(self_naf) < len(other_naf):
    self_naf = [0] * (len(other_naf) - len(self_naf)) + self_naf
elif len(self_naf) > len(other_naf):
    other_naf = [0] * (len(self_naf) - len(other_naf)) + other_naf

for A, B in zip(self_naf, other_naf):
    X3, Y3, Z3 = _double(X3, Y3, Z3, p, a)


    if A == 0:
        if B == 0:
            pass
```

Продовження додатку А

```
            elif B < 0:
                X3, Y3, Z3 = _add(X3, Y3, Z3, X2, -Y2, Z2, p)
            else:
                assert B > 0
                X3, Y3, Z3 = _add(X3, Y3, Z3, X2, Y2, Z2, p)
        elif A < 0:
            if B == 0:
                X3, Y3, Z3 = _add(X3, Y3, Z3, X1, -Y1, Z1, p)
            elif B < 0:
                X3, Y3, Z3 = _add(X3, Y3, Z3, mAmB_X, mAmB_Y, mAmB_Z,
p)
            else:
                assert B > 0
                X3, Y3, Z3 = _add(X3, Y3, Z3, mApB_X, mApB_Y, mApB_Z, p)
        else:
            assert A > 0
            if B == 0:
                X3, Y3, Z3 = _add(X3, Y3, Z3, X1, Y1, Z1, p)
            elif B < 0:
                X3, Y3, Z3 = _add(X3, Y3, Z3, pAmB_X, pAmB_Y, pAmB_Z, p)
            else:
                assert B > 0
                X3, Y3, Z3 = _add(X3, Y3, Z3, pApB_X, pApB_Y, pApB_Z, p)

        if not Y3 or not Z3:
            return INFINITY

        return PointJacobi(self.__curve, X3, Y3, Z3, self.__order)

    def __neg__(self):

        x, y, z = self.__coords
        return PointJacobi(self.__curve, x, -y, z, self.__order)


class Point(object):
```

Продовження додатку А

```
def __init__(self, curve, x, y, order=None):

    self.__curve = curve
    if GMPY:
        self.__x = x and mpz(x)
        self.__y = y and mpz(y)
        self.__order = order and mpz(order)
    else:
        self.__x = x
        self.__y = y
        self.__order = order

    if self.__curve:
        assert self.__curve.contains_point(x, y)

    if curve and curve.cofactor() != 1 and order:
        assert self * order == INFINITY

def __eq__(self, other):

    if isinstance(other, Point):
        return (
            self.__curve == other.__curve
            and self.__x == other.__x
            and self.__y == other.__y
        )
    return NotImplemented

def __ne__(self, other):

    return not self == other

def __neg__(self):
    return Point(self.__curve, self.__x, self.__curve.p() - self.__y)

def __add__(self, other):
```

Продовження додатку А

```
    if not isinstance(other, Point):
        return NotImplemented
    if other == INFINITY:
        return self
    if self == INFINITY:
        return other
    assert self.__curve == other.__curve
    if self.__x == other.__x:
        if (self.__y + other.__y) % self.__curve.p() == 0:
            return INFINITY
        else:
            return self.double()

    p = self.__curve.p()

    l = (
        (other.__y - self.__y)
        * numbertheory.inverse_mod(other.__x - self.__x, p)
    ) % p

    x3 = (l * l - self.__x - other.__x) % p
    y3 = (l * (self.__x - x3) - self.__y) % p

    return Point(self.__curve, x3, y3)

def __mul__(self, other):


    def leftmost_bit(x):
        assert x > 0
        result = 1
        while result <= x:
            result = 2 * result
        return result // 2

    e = other
    if e == 0 or (self.__order and e % self.__order == 0):
```

Продовження додатку А

```
        return INFINITY
    if self == INFINITY:
        return INFINITY
    if e < 0:
        return (-self) * (-e)


    e3 = 3 * e
    negative_self = Point(self.__curve, self.__x, -self.__y, self.__order)
    i = leftmost_bit(e3) // 2
    result = self

    while i > 1:
        result = result.double()
        if (e3 & i) != 0 and (e & i) == 0:
            result = result + self
        if (e3 & i) == 0 and (e & i) != 0:
            result = result + negative_self

        i = i // 2

    return result

def __rmul__(self, other):

    return self * other

def __str__(self):
    if self == INFINITY:
        return "infinity"
    return "(%d,%d)" % (self.__x, self.__y)

def double(self):


    if self == INFINITY:
        return INFINITY
```

Продовження додатку А

```
        p = self.__curve.p()
        a = self.__curve.a()

        l = (
            (3 * self.__x * self.__x + a)
            * numbertheory.inverse_mod(2 * self.__y, p)
        ) % p

        x3 = (l * l - 2 * self.__x) % p
        y3 = (l * (self.__x - x3) - self.__y) % p

        return Point(self.__curve, x3, y3)

    def x(self):
        return self.__x

    def y(self):
        return self.__y

    def curve(self):
        return self.__curve

    def order(self):
        return self.__order


INFINITY = Point(None, None, None)
```

«Number Theory»

```
from __future__ import division

import sys
from six import integer_types, PY2
from six.moves import reduce

try:
    xrange
```

Продовження додатку А

```
except NameError:
    xrange = range
try:
    from gmpy2 import powmod

    GMPY2 = True
    GMPY = False
except ImportError:
    GMPY2 = False
    try:
        from gmpy import mpz

        GMPY = True
    except ImportError:
        GMPY = False

import math
import warnings


class Error(Exception):


    pass


class SquareRootError(Error):
    pass


class NegativeExponentError(Error):
    pass


def modular_exp(base, exponent, modulus):
    if exponent < 0:
        raise NegativeExponentError(
```

Продовження додатку А

```
        "Negative exponents (%d) not allowed" % exponent
      )
    return pow(base, exponent, modulus)


def polynomial_reduce_mod(poly, polymod, p):

    assert polymod[-1] == 1

    assert len(polymod) > 1

    while len(poly) >= len(polymod):
      if poly[-1] != 0:
        for i in xrange(2, len(polymod) + 1):
          poly[-i] = (poly[-i] - poly[-1] * polymod[-i]) % p
      poly = poly[0:-1]

    return poly


def polynomial_multiply_mod(m1, m2, polymod, p):

    prod = (len(m1) + len(m2) - 1) * [0]

      for i in xrange(len(m1)):
      for j in xrange(len(m2)):
        prod[i + j] = (prod[i + j] + m1[i] * m2[j]) % p

    return polynomial_reduce_mod(prod, polymod, p)


def polynomial_exp_mod(base, exponent, polymod, p):

    assert exponent < p

    if exponent == 0:
      return [1]
```

Продовження додатку А

```
    G = base
    k = exponent
    if k % 2 == 1:
        s = G
    else:
        s = [1]

    while k > 1:
        k = k // 2
        G = polynomial_multiply_mod(G, G, polymod, p)
        if k % 2 == 1:
            s = polynomial_multiply_mod(G, s, polymod, p)

    return s


def jacobi(a, n):

    assert n >= 3
    assert n % 2 == 1
    a = a % n
    if a == 0:
        return 0
    if a == 1:
        return 1
    a1, e = a, 0
    while a1 % 2 == 0:
        a1, e = a1 // 2, e + 1
    if e % 2 == 0 or n % 8 == 1 or n % 8 == 7:
        s = 1
    else:
        s = -1
    if a1 == 1:
        return s
    if n % 4 == 3 and a1 % 4 == 3:
        s = -s
    return s * jacobi(n % a1, a1)
```

Продовження додатку А

```
def square_root_mod_prime(a, p):

    assert 0 <= a < p
    assert 1 < p

    if a == 0:
        return 0
    if p == 2:
        return a

    jac = jacobi(a, p)
    if jac == -1:
        raise SquareRootError("%d has no square root modulo %d" % (a, p))

    if p % 4 == 3:
        return pow(a, (p + 1) // 4, p)

    if p % 8 == 5:
        d = pow(a, (p - 1) // 4, p)
        if d == 1:
            return pow(a, (p + 3) // 8, p)
        if d == p - 1:
            return (2 * a * pow(4 * a, (p - 5) // 8, p)) % p
        raise RuntimeError("Shouldn't get here.")

    if PY2:

        range_top = min(0x7FFFFFFF, p)
    else:
        range_top = p
    for b in xrange(2, range_top):
        if jacobi(b * b - 4 * a, p) == -1:
            f = (a, -b, 1)
            ff = polynomial_exp_mod((0, 1), (p + 1) // 2, f, p)
            assert ff[1] == 0
            return ff[0]
    raise RuntimeError("No b found.")
```

Продовження додатку А

```
if GMPY2:

    def inverse_mod(a, m):

        if a == 0:
            return 0
        return powmod(a, -1, m)


elif GMPY:

    def inverse_mod(a, m):

        if a == 0:
            return 0
        a = mpz(a)
        m = mpz(m)

        lm, hm = mpz(1), mpz(0)
        low, high = a % m, m
        while low > 1:
            r = high // low
            lm, low, hm, high = hm - lm * r, high - low * r, lm, low

        return lm % m


elif sys.version_info >= (3, 8):

    def inverse_mod(a, m):

        if a == 0:
            return 0
        return pow(a, -1, m)


else:
```

```
    def inverse_mod(a, m):


        if a == 0:
            return 0

        lm, hm = 1, 0
        low, high = a % m, m
        while low > 1:
            r = high // low
            lm, low, hm, high = hm - lm * r, high - low * r, lm, low

        return lm % m


try:
    gcd2 = math.gcd
except AttributeError:

    def gcd2(a, b):

        while a:
            a, b = b % a, a
        return b


def gcd(*a):

    if len(a) > 1:
        return reduce(gcd2, a)
    if hasattr(a[0], "__iter__"):
        return reduce(gcd2, a[0])
    return a[0]


def lcm2(a, b):
```

Продовження додатку А

```
 return (a * b) // gcd(a, b)

def lcm(*a):

    if len(a) > 1:
        return reduce(lcm2, a)
    if hasattr(a[0], "__iter__"):
        return reduce(lcm2, a[0])
    return a[0]


def factorization(n):

    assert isinstance(n, integer_types)

    if n < 2:
        return []

    result = []


    for d in smallprimes:
        if d > n:
            break
        q, r = divmod(n, d)
        if r == 0:
            count = 1
            while d <= n:
                n = q
                q, r = divmod(n, d)
                if r != 0:
                    break
                count = count + 1
            result.append((d, count))

       if n > smallprimes[-1]:
        if is_prime(n):
```

Продовження додатку А

```
            result.append((n, 1))
        else:
            d = smallprimes[-1]
            while 1:
                d = d + 2
                q, r = divmod(n, d)
                if q < d:
                    break
                if r == 0:
                    count = 1
                    n = q
                    while d <= n:
                        q, r = divmod(n, d)  # see if it does.
                        if r != 0:
                            break
                        n = q
                        count = count + 1
                    result.append((d, count))
            if n > 1:
                result.append((n, 1))

    return result


def phi(n):

    assert isinstance(n, integer_types)

    if n < 3:
        return 1

    result = 1
    ff = factorization(n)
    for f in ff:
        e = f[1]
        if e > 1:
            result = result * f[0] ** (e - 1) * (f[0] - 1)
```

```
        else:
            result = result * (f[0] - 1)
    return result


def carmichael(n):

    return carmichael_of_factorized(factorization(n))


def carmichael_of_factorized(f_list):

    if len(f_list) < 1:
        return 1

    result = carmichael_of_ppower(f_list[0])
    for i in xrange(1, len(f_list)):
        result = lcm(result, carmichael_of_ppower(f_list[i]))

    return result


def carmichael_of_ppower(pp):

    p, a = pp
    if p == 2 and a > 2:
        return 2 ** (a - 2)
    else:
        return (p - 1) * p ** (a - 1)


def order_mod(x, m):

    if m <= 1:
        return 0

    assert gcd(x, m) == 1
```

```
    z = x
    result = 1
    while z != 1:
        z = (z * x) % m
        result = result + 1
    return result


def largest_factor_relatively_prime(a, b):

      while 1:
        d = gcd(a, b)
        if d <= 1:
            break
        b = d
        while 1:
            q, r = divmod(a, d)
            if r > 0:
                break
            a = q
    return a


def kinda_order_mod(x, m):  # pragma: no cover

          warnings.warn(
        "Function is unused by library code. If you use this code, "
        "please open an issue in "
        "https://github.com/tlsfuzzer/python-ecdsa",
        DeprecationWarning,
    )

    return order_mod(x, largest_factor_relatively_prime(m, x))


def is_prime(n):
```

Продовження додатку А

```
global miller_rabin_test_count

miller_rabin_test_count = 0

if n <= smallprimes[-1]:
    if n in smallprimes:
        return True
    else:
        return False

if gcd(n, 2 * 3 * 5 * 7 * 11) != 1:
    return False

t = 40
n_bits = 1 + int(math.log(n, 2))
for k, tt in (
    (100, 27),
    (150, 18),
    (200, 15),
    (250, 12),
    (300, 9),
    (350, 8),
    (400, 7),
    (450, 6),
    (550, 5),
    (650, 4),
    (850, 3),
    (1300, 2),
):
    if n_bits < k:
        break
    t = tt

s = 0
r = n - 1
while (r % 2) == 0:
    s = s + 1
```

Продовження додатку А

```
        r = r // 2
    for i in xrange(t):
        a = smallprimes[i]
        y = pow(a, r, n)
        if y != 1 and y != n - 1:
            j = 1
            while j <= s - 1 and y != n - 1:
                y = pow(y, 2, n)
                if y == 1:
                    miller_rabin_test_count = i + 1
                    return False
                j = j + 1
            if y != n - 1:
                miller_rabin_test_count = i + 1
                return False
    return True


def next_prime(starting_value):

    if starting_value < 2:
        return 2
    result = (starting_value + 1) | 1
    while not is_prime(result):
        result = result + 2
    return result


smallprimes = [
    2,
    3,
    5,
    7,
    11,
    13,
    17,
    19,
```

Продовження додатку А

23,
29,
31,
37,
41,
43,
47,
53,
59,
61,
67,
71,
73,
79,
83,
89,
97,
101,
103,
107,
109,
113,
127,
131,
137,
139,
149,
151,
157,
163,
167,
173,
179,
181,
191,
193,
197,

Продовження додатку А

199,
211,
223,
227,
229,
233,
239,
241,
251,
257,
263,
269,
271,
277,
281,
283,
293,
307,
311,
313,
317,
331,
337,
347,
349,
353,
359,
367,
373,
379,
383,
389,
397,
401,
409,
419,
421,

Продовження додатку А

431,
433,
439,
443,
449,
457,
461,
463,
467,
479,
487,
491,
499,
503,
509,
521,
523,
541,
547,
557,
563,
569,
571,
577,
587,
593,
599,
601,
607,
613,
617,
619,
631,
641,
643,
647,
653,

Продовження додатку А

659,
661,
673,
677,
683,
691,
701,
709,
719,
727,
733,
739,
743,
751,
757,
761,
769,
773,
787,
797,
809,
811,
821,
823,
827,
829,
839,
853,
857,
859,
863,
877,
881,
883,
887,
907,
911,

Продовження додатку А

919,
929,
937,
941,
947,
953,
967,
971,
977,
983,
991,
997,
1009,
1013,
1019,
1021,
1031,
1033,
1039,
1049,
1051,
1061,
1063,
1069,
1087,
1091,
1093,
1097,
1103,
1109,
1117,
1123,
1129,
1151,
1153,
1163,
1171,

Продовження додатку А

```
    1181,
    1187,
    1193,
    1201,
    1213,
    1217,
    1223,
    1229,
]

miller_rabin_test_count = 0
```