Правительство Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

(НИУ ВШЭ)

Московский институт электроники и математики им. А.Н. Тихонова

ОТЧЕТ

О ПРАКТИЧЕСКОЙ РАБОТЕ № 8

по дисциплине «Криптографические методы защиты информации»

Схемы электронной подписи

Студент гр. МКБ 241

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. С. Новиков

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

Руководитель

Заведующий кафедрой информационной безопасности киберфизических систем

канд. техн. наук, доцент

\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.О. Евсютин

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

Москва 2025

Оглавление

[1. Задание на практическую работу 3](#_Toc198628924)

[2. Теоретическая часть 3](#_Toc198628925)

[2.1 Общие сведения об электронной подписи 3](#_Toc198628926)

[2.2 Стандарты электронной подписи 4](#_Toc198628927)

[2.3 Сложение точек эллиптической кривой 7](#_Toc198628928)

[2.4 Теоретико-числовые алгоритмы для реализации криптографических преобразований 8](#_Toc198628929)

[3. Программный код и описание варианта схемы электронной цифровой подписи 10](#_Toc198628930)

[3.1 Описание 10](#_Toc198628931)

[3.2 Код 11](#_Toc198628932)

[3.3 Запуск 22](#_Toc198628933)

[3.4 Пример работы 22](#_Toc198628934)

[4. ПРИЛОЖЕНИЕ А. 24](#_Toc198628935)

# Задание на практическую работу

Целью данной работы является приобретение навыков программной реализации схем электронной подписи. Написать программную реализацию схемы электронной цифровой подписи, представленной в ГОСТ Р 34.10-2012. Программная реализация должна быть выполнена студентом самостоятельно без использования готовых библиотечных решений (допускается использование готовой реализации хэш-функции ГОСТ Р 34.11-2012

В рамках практической работы необходимо выполнить следующее: реализация программы схемы электронной цифровой подписи на языке Python.

# Теоретическая часть

## Общие сведения об электронной подписи

Электронная цифровая подпись (электронная подпись, цифровая подпись) сообщения – это строка бит, присоединяемая к сообщению и зависящая от сообщения и секретного элемента данных – ключа подписи, известного только его владельцу. Электронная цифровая подпись предназначена для аутентификации лица, подписавшего электронное сообщение. Кроме того, использование электронной цифровой подписи предоставляет возможность обеспечить следующие свойства при передаче в системе подписанного сообщения:

* осуществить контроль целостности передаваемого подписанного сообщения;
* доказательно подтвердить авторство лица, подписавшего сообщение;
* защитить сообщение от возможной подделки.

Существуют два основных подхода к построению схем электронной цифровой подписи: на основе симметричных шифров и на основе алгоритмов с открытым ключом.

Первый подход предусматривает наличие посредника или арбитра, обладающего доверием всех участников информационного обмена. Каждый пользователь имеет с посредником общий ключ шифрования, отличный от аналогичных ключей других пользователей. Эти ключи выдаются пользователям посредником и могут использоваться многократно. Такая схема достаточно хорошо работает, но проблемой является нахождение непогрешимого посредника.

Схемы электронной цифровой подписи на основе алгоритмов с открытым ключом являются наиболее распространенными на сегодняшний день, и именно такие схемы положены в основу действующих государственных стандартов.

В данном случае каждый участник электронного взаимодействия обладает двумя ключами: ключом подписи и ключом проверки. Ключ подписи представляет собой закрытый ключ, используемый для формирования электронной цифровой подписи и известный только его владельцу. Ключ проверки представляет собой открытый ключ, математически связанный с ключом подписи, известный всем участникам электронного взаимодействия и используемый ими для проверки электронной цифровой подписи данного участника.

## Стандарты электронной подписи

**2.2.1 ГОСТ Р 34.10-94**

Первый отечественный стандарт электронной цифровой подписи описан в нормативном документе «ГОСТ Р 34.10-94. Информационная технология. Криптографическая защита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма». Данный стандарт был введен 1 января 1994 года и действовал до середины 2002 года.

Важной особенностью стандартов электронной цифровой подписи является использование хэш-функций. Подписываемое сообщение предварительно сворачивается в строку фиксированной длины с помощью хэш-функции, и процедура выработки электронной цифровой подписи применяется к полученному хэш-коду вместо самого сообщения. С аналогичного действия начинается процедура проверки электронной цифровой подписи для данного сообщения.

В ГОСТ Р 34.10-94 для этой цели используется хэш-функция, определенная в стандарте ГОСТ Р 34.11-94.

Все вычисления в ГОСТ Р 34.10-94 производятся в конечном поле по модулю простого числа. Стойкость соответствующей схемы электронной цифровой подписи основывается на сложности задачи дискретного логарифмирования в мультипликативной группе такого конечного поля, а также на стойкости используемой хэш-функции. Собственно, электронная цифровая подпись представляет собой двоичный вектор длиной 512 бит.

**2.2.2 ГОСТ Р 34.10-2001**

Второй отечественный стандарт электронной цифровой подписи описан в нормативном документе «ГОСТ Р 34.10-2001. Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи». Данный стандарт был введен 1 июля 2002 года и действовал до 31 декабря 2012 года.

Отличие между ГОСТ Р 34.10-94 и ГОСТ Р 34.10-2001 заключается в том, что в стандарте 2001 года все вычисления производились в группе точек эллиптической кривой над конечным полем, в том время как в стандарте 1994 года все вычисления производились в конечном поле по модулю простого числа. Соответственно стойкость стандарта ГОСТ Р 34.10-2001 основывается на сложности задачи дискретного логарифмирования в группе точек эллиптической кривой, а также на стойкости используемой хэш-функции

ГОСТ Р 34.10-2001 определяет схему электронной цифровой подписи, процессы формирования и проверки цифровой подписи под заданным сообщением (документом), передаваемым по незащищенным телекоммуникационным каналам общего назначения в системах обработки информации различного назначения.

Для хэширования сообщения в данном стандарте используется хэш-функция ГОСТ Р 34.11-94.

Электронная цифровая подпись, вырабатываемая по ГОСТ Р 34.10-2001, представляет собой двоичный вектор длиной 512 бит.

**2.2.3 ГОСТ Р 34.10-2012**

Действующим отечественным стандартом электронной цифровой подписи является стандарт, описанный в нормативном документе «ГОСТ Р 34.10-2012. Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи».

Данный стандарт введен 1 января 2013 года на смену ГОСТ Р 34.10-2001. Замена стандарта связана с тем, что внедрение цифровой подписи на основе ГОСТ Р 34.10-2012 повышает, по сравнению с ранее действовавшей схемой электронной цифровой подписи, уровень защищенности передаваемых сообщений от подделок и искажений. Однако алгоритмы формирования и проверки электронной цифровой подписи ГОСТ Р 34.10-2012 полностью повторяют аналогичные алгоритмы ГОСТ Р 34.10-2001. Фактически между двумя данными стандартами есть только два существенных отличия:

* ГОСТ Р 34.10-2001 формирует электронную цифровую подпись длиной 512 бит, в то время как в ГОСТ Р 34.10-2012 длина электронной цифровой подписи варьируется и может составлять 512 бит и 1024 бита;
* для хэширования сообщения в ГОСТ Р 34.10-2012 используется хэш-функция ГОСТ Р 34.11-2012 вместо ГОСТ Р 34.11-94, используемой ранее в ГОСТ Р 34.10-2001.

Схема электронной цифровой подписи, представленная в ГОСТ Р 34.10-2012, оперирует следующими параметрами:

* Простое число – модуль эллиптической кривой.
* Эллиптическая кривая , задаваемая своим инвариантом или коэффициентами , .
* Целое число – порядок группы точек эллиптической кривой , такое, что  
  .
* Простое число – порядок циклической подгруппы группы точек эллиптической кривой , для которого выполнены следующие условия:
* Точка эллиптической кривой с координатами , удовлетворяющая равенству .
* Хэш-функция , .
* Ключ подписи – целое число *.*
* Ключ проверки – точка эллиптической кривой .

*Алгоритм формирования подписи*

* 1. Вычислить хэш-код сообщения : .
  2. Вычислить целое число , двоичным представлением которого является вектор , и определить . Если , то определить .
  3. Сгенерировать случайное целое число , удовлетворяющее неравенству .
  4. Вычислить точку эллиптической кривой и определить . Если , то вернуться к шагу 3
  5. Вычислить значение . Если , то вернуться к шагу 3.
  6. Вычислить двоичные векторы, соответствующие числам и , и определить цифровую подпись как конкатенацию данных двоичных векторов.

*Алгоритм проверки подписи*

* 1. По полученной подписи вычислить целые числа и. Если выполнены неравенства , , то перейти к следующему шагу. В противном случае подпись неверна.
  2. Вычислить хэш-код сообщения .
  3. Вычислить целое число , двоичным представлением которого является вектор , и определить . Если , то определить .
  4. Вычислить значение .
  5. Вычислить значения , .
  6. Вычислить точку эллиптической кривой и определить значение .
  7. Если выполнено равенство , то подпись принимается, в противном случае, подпись неверна.

**2.2.4 DSS**

Стандарт электронной цифровой подписи, действующий в США, носит название DSS (Digital Signature Standard). Он имеет следующую историю. Исходный стандарт опубликован в 1994 году в документе FIPS PUB 186. Данный документ несколько раз пересматривался и публиковался в следующих редакциях:

* 1998 год, FIPS PUB 186-1;
* 2000 год, FIPS PUB 186-2;
* 2009 год, FIPS PUB 186-3;
* 2013 год, FIPS PUB 186-4;
* 2023 год, FIPS PUB 186-5.

Актуальный стандарт DSS включает три алгоритма, предназначенных для формирования и проверки электронной цифровой подписи:

* алгоритм цифровой подписи на основе криптосистемы RSA;
* алгоритм цифровой подписи на основе эллиптических кривых ECDSA (Elliptic Curve Digital Signature Algorithm);
* алгоритм цифровой подписи на основе кривых Эвардса EdDSA (Edwards Curve Digital Signature Algorithm).

Алгоритм ECDSA по своему устройству схож с аналогичным алгоритмом, представленным в российском стандарте электронной цифровой подписи. Алгоритм EdDSA основан на семействе эллиптических кривых Эдвардса, обладающих некоторыми преимуществами перед эллиптическими кривыми в форме Вейерштрасса.

## 2.3 Сложение точек эллиптической кривой

Эллиптической кривой над конечным полем вычетов по модулю простого числа  
 называется множество точек , удовлетворяющих уравнению  
, где *a*, *b*   и , дополненное бесконечно удаленной точкой 0, не имеющей численного выражения. Данное множество точек, обозначаемое , представляет собой абелеву группу относительно операции сложения точек.

Операция сложения точек эллиптической кривой задается следующим образом. Чтобы сложить точки и , необходимо провести через них прямую, которая в общем случае будет проходить еще через одну точку эллиптической кривой. Эту третью точку необходимо симметрично отразить относительно оси абсцисс, полученный результат и будет представлять собой сумму .

Зная координаты двух исходных точек и , достаточно легко вывести формулы для нахождения координат третьей точки . При этом необходимо учесть три случая.

Первый случай. Складываются две одинаковые точки и . При выводе координат результирующей точки необходимо воспользоваться уравнением касательной к эллиптической кривой. Формулы для нахождения координат точки  
 имеют вид

(1)

Второй случай. Складываются две разные точки, и , причем  
. При выводе координат результирующей точки необходимо воспользоваться уравнением секущей к эллиптической кривой. Формулы для нахождения координат точки имеют вид

(2)

Третий случай. Складываются две разные точки, и , причем  
, . Такие точки являются взаимно обратными элементами группы , то есть , поэтому их сумма дает нейтральный элемент группы, то есть бесконечно удаленную точку 0.

## 2.4 Теоретико-числовые алгоритмы для реализации криптографических преобразований

**2.4.1 Нахождение обратного элемента по модулю простого числа**

Формулы (1) и (2) используют операцию деления, под которой в арифметике остатков подразумевается умножение на обратное по модулю значение. Для нахождения обратного значения по модулю натурального числа применяется расширенный алгоритм Евклида.

Вход: целые числа .

Выход: и целые , , такие, что .

1. Полагаем , , , .

2. Пока , выполнять следующее:

2.1. , , , ;

2.2. , , , , , .

3. , , и возврат .

Чтобы найти , необходимо подать на вход алгоритма Евклида пару , и если , вернуть в качестве значение .

**2.4.2 Возведение в степень по модулю**

Криптографические алгоритмы с открытым ключом при их использовании на практике оперируют числами большой битовой длины (или просто большими числами), когда речь идет о сотнях и тысячах бит. Для некоторых операцией над такими числами созданы специальные алгоритмы. В случае криптосистем RSA и Эль-Гамаля необходимо иметь алгоритм, который позволит осуществлять быстрое возведение в степень по модулю. Данный алгоритм представлен ниже.

*Алгоритм возведения в степень по модулю.*

Вход: , .

Выход: .

1. . Если , то переход к шагу 5.

2. .

3. Если , то .

4. Для выполняем следующее:

4.1. .

4.2. Если , то .

5. Возврат .

Сложение точек эллиптической кривой осуществляется по аналогичному алгоритму. Если для данной точки необходимо вычислить точку , то искомая точка представляется в виде или в зависимости от четности числа . Далее происходит удвоение точки по формулам (1), после чего процесс повторяется, пока не будет вычислена искомая точка.

**2.4.3 Тесты целых чисел на простоту**

Еще одним важным аспектом криптографии с открытым ключом является использование простых чисел.

Наиболее развитые вероятностные алгоритмы проверки чисел на простоту основаны на малой теореме Ферма.

*Малая теорема Ферма.*

Пусть — простое число, и . Тогда .

Соотношение, приведенное в теореме, используется в тесте, проверяющем, является ли заданное число составным. Этот тест называют тестом Ферма.

*Тест Ферма.*

Вход: нечетное число .

Выход: ответ на вопрос «является ли простым».

1. Для выполняем следующее:

1.1. Выбираем случайное целое число .

1.2. Вычисляем с помощью алгоритма возведения в степень по модулю.

1.3. Если , то возврат «*n* — составное».

Тест Ферма по основанию определяет простоту с вероятностью , после итераций вероятность ошибки составляет .

# Программный код и описание варианта схемы электронной цифровой подписи

## Описание

* язык: Python (версия 3.6+);
* особенности реализация без библиотек, только стандартные библиотеки (os, random, hashlib, typing), установка пакетов не требуется;
* extended\_gcd - расширенный алгоритм Евклида для нахождения НОД и коэффициентов;
* mod\_inverse - находит мультипликативное обратное a по модулю m;
* generate\_keypair - генерация ключевой пары: закрытый ключ d, открытый ключ Q;
* sign\_message - формирование подписи для сообщения;
* verify\_signature - проверка подписи для сообщения;
* process\_file - обработка файла: подпись или проверка;
* обоснование выбора библиотеки hashlib - пакет pygost на PyPI является placeholder-ом, созданным Яндексом для защиты от атак типа "dependency confusion", и не содержит функциональности ГОСТ-алгоритмов. Настоящая библиотека PyGOST (<https://github.com/ilyaTT/pygost_0_15>), реализующая ГОСТ Р 34.11-2012 ("Стрибог"), требует установки из исходного кода, что связано с дополнительными сложностями, такими как настройка окружения и возможные ошибки в Windows (например, проблемы с setup.py или зависимостями).
* интерактивный режим с действиями generate/sign/verify.

## Код

Ссылка на код:

https://github.com/vit81g/Cybersecurity\_HSE/tree/main/HomeWorks/Cryptographic/practic03

import os # Импортируем модуль os для взаимодействия с операционной системой  
import random # Импортируем модуль random для генерации случайных чисел и случайного выбора  
import hashlib # Библиотека hashlib из стандартной библиотеки Python была выбрана для реализации хэш-функции в скрипте  
from typing import Tuple, Optional # Импортируем аннотации типов Tuple и Optional  
  
  
def extended\_gcd(a: int, b: int) -> Tuple[int, int, int]:  
 *"""  
 Расширенный алгоритм Евклида для вычисления НОД и коэффициентов Безу.  
 Вычисляет gcd(a, b) и находит x, y такие, что a \* x + b \* y = gcd(a, b)  
 Аргументы: a (int): первое целое число; b (int): второе целое число.  
 Возвращает:  
 Tuple[int, int, int]:  
 gcd (int): наибольший общий делитель a и b  
 x (int): коэффициент при a в уравнении Безу.  
 y (int): коэффициент при b в уравнении Безу.  
 """* if a == 0:  
 # азовый случай: gcd(0, b) = b; 0\*x + 1\*b = b  
 return b, 0, 1  
 # рекурсивный вызов: gcd(b mod a, a)  
 gcd, x1, y1 = extended\_gcd(b % a, a)  
 # переход к исходным коэффициентам x, y  
 x = y1 - (b // a) \* x1  
 y = x1  
 return gcd, x, y  
  
  
def mod\_inverse(a: int, m: int) -> Optional[int]:  
 *"""  
 Вычисление мультипликативного обратного элемента a по модулю m.  
 Ищет x такое, что a \* x ≡ 1 (mod m).  
 Аргументы: a (int): число, для которого ищем обратное; m (int): модуль.  
 Возвращает: int или None: обратное a^{-1} mod m, либо None, если gcd(a, m) ≠ 1  
 """* gcd, x, \_ = extended\_gcd(a, m)  
 if gcd != 1:  
 # обратного не существует  
 return None  
 # нормализуем в диапазон [0, m-1]  
 return x % m  
  
  
class EllipticCurve:  
 *"""  
 Класс для работы с эллиптической кривой над конечным полем F\_p.  
 Уравнение: y^2 = x^3 + a\*x + b (mod p)  
 """* def \_\_init\_\_(self, a: int, b: int, p: int):  
 *"""  
 Инициализация параметров кривой.  
 Аргументы:  
 a (int): коэффициент a в уравнении кривой.  
 b (int): коэффициент b в уравнении кривой.  
 p (int): простое основание конечного поля F\_p.  
 """* self.a = a  
 self.b = b  
 self.p = p  
  
 def is\_point\_on\_curve(self, P: Tuple[int, int]) -> bool:  
 *"""  
 Проверяет, лежит ли точка P = (x, y) на кривой.  
 Возвращает True, если y^2 mod p == (x^3 + a\*x + b) mod p  
 """* x, y = P  
 left = (y \* y) % self.p  
 right = (x \* x \* x + self.a \* x + self.b) % self.p  
 return left == right  
  
 def add\_points(self,  
 P: Optional[Tuple[int, int]],  
 Q: Optional[Tuple[int, int]]  
 ) -> Optional[Tuple[int, int]]:  
 *"""  
 Сложение двух точек P и Q на эллиптической кривой.  
 Реализованы все случаи:  
 - P = None или Q = None (точка на бесконечности)  
 - P == Q (удвоение точки)  
 - P == -Q (результат — точка на бесконечности)  
 - обычное сложение разных точек  
 Аргументы: P, Q: координаты точек или None  
 Возвращает: новую точку на кривой или None (точка на бесконечности)  
 """* if P is None:  
 return Q  
 if Q is None:  
 return P  
  
 x1, y1 = P  
 x2, y2 = Q  
  
 # P + (-P) = бесконечность  
 if x1 == x2 and (y1 + y2) % self.p == 0:  
 return None  
  
 if P == Q:  
 # удвоение точки P  
 if y1 == 0:  
 # касательная вертикальна => бесконечность  
 return None  
 # вычисляем λ = (3\*x1^2 + a) / (2\*y1) mod p  
 inv = mod\_inverse((2 \* y1) % self.p, self.p)  
 if inv is None:  
 return None  
 lam = ((3 \* x1 \* x1 + self.a) \* inv) % self.p  
 else:  
 # сложение P и Q, P != Q  
 denom = (x2 - x1) % self.p  
 inv = mod\_inverse(denom, self.p)  
 if inv is None:  
 return None  
 lam = ((y2 - y1) \* inv) % self.p  
  
 # координаты результирующей точки R = (x3, y3)  
 x3 = (lam \* lam - x1 - x2) % self.p  
 y3 = (lam \* (x1 - x3) - y1) % self.p  
 return x3, y3  
  
 def multiply\_point(self, P: Tuple[int, int], k: int) -> Optional[Tuple[int, int]]:  
 *"""  
 Умножение точки P на скаляр k методом двойного и сложения.  
 Алгоритм двоичного разложения k:  
 - инициализируем R = None (0·P)  
 - для каждого бита k, если бит=1, R = R + P  
 - P = 2P при каждом шаге  
 Аргументы:  
 P (Tuple[int,int]): исходная точка на кривой  
 k (int): скалярный множитель  
 Возвращает: k·P как точку на кривой или None  
 """* result: Optional[Tuple[int, int]] = None # накопитель  
 addend = P  
 while k > 0:  
 if k & 1:  
 result = self.add\_points(result, addend)  
 addend = self.add\_points(addend, addend)  
 k >>= 1  
 return result  
  
  
def hash\_message(message: bytes) -> int:  
 *"""  
 Хэширует сообщение в целое число.  
 Здесь для примера используется SHA-256 из hashlib.  
 В реальной системе рекомендуется ГОСТ Р 34.11-2012, но проблема с установкой в Windows.  
 Аргументы: message (bytes): данные для хэширования  
 Возвращает: int: целочисленное представление хэша (big-endian)  
 """* digest = hashlib.sha256(message).digest()  
 return int.from\_bytes(digest, 'big')  
  
  
def generate\_keypair(curve: EllipticCurve,  
 G: Tuple[int, int],  
 q: int  
 ) -> Tuple[int, Tuple[int, int]]:  
 *"""  
 Генерация ключевой пары (секретного и публичного ключей).  
 Процесс:  
 1. Выбираем случайный d ∈ [1, q-1]  
 2. Вычисляем Q = d·G  
 3. Проверяем, что Q лежит на кривой и Q ≠ ∞  
 4. Возвращаем (d, Q)  
 Аргументы:  
 curve (EllipticCurve): параметры кривой  
 G (Tuple[int,int]): базовая точка порядка q  
 q (int): порядок подгруппы  
 Возвращает:  
 Tuple[int, Tuple[int,int]]:  
 d (int): секретный ключ  
 Q (Tuple[int,int]): публичный ключ  
 """* while True:  
 d = random.randrange(1, q)  
 Q = curve.multiply\_point(G, d)  
 if Q is not None and curve.is\_point\_on\_curve(Q):  
 return d, Q  
  
  
def sign\_message(message: bytes,  
 curve: EllipticCurve,  
 G: Tuple[int, int],  
 q: int,  
 d: int  
 ) -> Tuple[int, int]:  
 *"""  
 Формирование ЭЦП (r, s) по ГОСТ Р 34.10-2012.  
 Алгоритм:  
 1. h = H(message), e = h mod q (если e=0, e=1)  
 2. выбираем случайный k ∈ [1, q-1]  
 3. вычисляем P = k·G  
 4. r = P.x mod q (если r=0 — повторяем с новым k)  
 5. s = (r\*d + k\*e) mod q (если s=0 — повторяем)  
 Аргументы:  
 message (bytes): данные для подписи.  
 curve (EllipticCurve): параметры кривой.  
 G (Tuple[int,int]): базовая точка.  
 q (int): порядок подгруппы.  
 d (int): секретный ключ.  
 Возвращает: Tuple[int,int]: подпись (r, s)  
 """* h = hash\_message(message)  
 e = h % q  
 if e == 0:  
 e = 1  
  
 while True:  
 k = random.randrange(1, q)  
 P = curve.multiply\_point(G, k)  
 if P is None:  
 continue  
 r = P[0] % q  
 if r == 0:  
 continue  
 s = (r \* d + k \* e) % q  
 if s == 0:  
 continue  
 return r, s  
  
  
def verify\_signature(message: bytes,  
 signature: Tuple[int, int],  
 curve: EllipticCurve,  
 G: Tuple[int, int],  
 q: int,  
 Q: Tuple[int, int]  
 ) -> bool:  
 *"""  
 Проверка ЭЦП по ГОСТ Р 34.10-2012.  
 Шаги проверки:  
 1. Проверяем диапазон: 0 < r, s < q  
 2. Проверяем, лежит ли Q (публичный ключ) на кривой  
 3. Вычисляем h = H(message), e = h mod q (если e=0, e=1)  
 4. Вычисляем обратное v = e^{-1} mod q  
 5. Вычисляем z1 = s·v mod q и z2 = (-r)·v mod q  
 6. Строим точку C = z1·G + z2·Q  
 7. Подпись считается валидной, если C ≠ ∞ и (C.x mod q) == r  
 Аргументы:  
 message (bytes): подписанные данные.  
 signature (Tuple[int,int]): полученная подпись (r, s)  
 curve (EllipticCurve): параметры кривой.  
 G (Tuple[int,int]): базовая точка.  
 q (int): порядок подгруппы.  
 Q (Tuple[int,int]): публичный ключ.  
 Возвращает: bool: True, если подпись верна, иначе False  
 """* r, s = signature  
 # 1. проверяем корректность r и s  
 if not (0 < r < q and 0 < s < q):  
 return False  
  
 # 2. убеждаемся, что Q лежит на кривой  
 if not curve.is\_point\_on\_curve(Q):  
 return False  
  
 # 3. хэшируем сообщение  
 h = hash\_message(message)  
 e = h % q  
 if e == 0:  
 e = 1  
  
 # 4. обратное e по модулю q  
 v = mod\_inverse(e, q)  
 if v is None:  
 return False  
  
 # 5. вычисляем вспомогательные множители  
 z1 = (s \* v) % q  
 z2 = (-r \* v) % q  
  
 # 6. C = z1·G + z2·Q  
 P1 = curve.multiply\_point(G, z1)  
 P2 = curve.multiply\_point(Q, z2)  
 C = curve.add\_points(P1, P2)  
  
 # 7. проверяем соответствие координаты  
 if C is None:  
 return False  
 return (C[0] % q) == r  
  
  
def process\_file(input\_file: str,  
 output\_file: str,  
 curve: EllipticCurve,  
 G: Tuple[int, int],  
 q: int,  
 key: Tuple[int, Tuple[int, int]],  
 mode: str = "sign"  
 ) -> None:  
 *"""  
 Упрощённый интерфейс для подписи и проверки файловой подписи.  
 Аргументы:  
 input\_file (str): путь к файлу с исходным сообщением.  
 output\_file (str): путь к файлу подписи (для sign) или к выводу результатов проверки.  
 curve, G, q: параметры эллиптической кривой.  
 key: для mode="sign" — (d, \_), для mode="verify" — (\_, Q)  
 mode (str): "sign" или "verify"  
 """* if not os.path.exists(input\_file):  
 raise FileNotFoundError(f"Не найден файл сообщения: {input\_file}")  
 with open(input\_file, "rb") as f:  
 msg = f.read()  
  
 if mode == "sign":  
 d, \_ = key  
 # создаём подпись и сохраняем r и s в output\_file  
 r, s = sign\_message(msg, curve, G, q, d)  
 with open(output\_file, "w") as f:  
 f.write(f"{r} {s}")  
 else:  
 # проверяем существование файла подписи  
 if not os.path.exists(output\_file):  
 raise FileNotFoundError(f"Не найден файл подписи: {output\_file}")  
 with open(output\_file, "r") as f:  
 r\_str, s\_str = f.read().split()  
 r, s = int(r\_str), int(s\_str)  
 \_, Q = key  
 # выполняем проверку и записываем результат в verify\_result.txt  
 ok = verify\_signature(msg, (r, s), curve, G, q, Q)  
 with open("verify\_result.txt", "w", encoding="utf-8") as f:  
 f.write("Подпись верна" if ok else "Подпись неверна")  
  
  
def main() -> None:  
 *"""  
 Основная функция: интерактивный CLI для генерации ключей, подписи и проверки.  
 Пользователь по шагам выбирает операцию и вводит необходимые параметры.  
 """* # --- Пример параметров для тестов (малое поле) ---  
 p = 17 # простое  
 a, b = 2, 2  
 curve = EllipticCurve(a, b, p)  
 G = (5, 1)  
 q = 19  
 # -----------------------------------------------  
  
 print("ГОСТ Р 34.10-2012: электронная подпись на Python")  
 while True:  
 mode = input("Выберите операцию (generate/sign/verify): ").strip().lower()  
 if mode in {"generate", "sign", "verify"}:  
 break  
  
 if mode == "generate":  
 # генерация новой пары ключей  
 d, Q = generate\_keypair(curve, G, q)  
 print(f"Секретный ключ d = {d}")  
 print(f"Публичный ключ Q = {Q}")  
 return  
  
 msg\_file = input("Путь к файлу сообщения: ").strip()  
 sig\_file = input("Путь к файлу подписи: ").strip()  
  
 if mode == "sign":  
 d = int(input("Введите секретный ключ d: ").strip())  
 process\_file(msg\_file, sig\_file, curve, G, q, (d, None), "sign")  
 print("Подпись успешно сформирована.")  
 else:  
 x, y = map(int, input("Введите публичный ключ Q (x y): ").split())  
 process\_file(msg\_file, sig\_file, curve, G, q, (0, (x, y)), "verify")  
 with open("verify\_result.txt", encoding="utf-8") as f:  
 print("Результат проверки:", f.read())  
  
# запуск  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

## Запуск

Сохранить код в файл, например <script>.py.

Запустить в терминале: python <script>.py.

Дополнительные инструкции:

* generate – генерация ключевой пары: закрытый ключ d, открытый ключ Q;
* sign - формирование подписи для сообщения
* выбрать файл с сообщением
* выбрать файл с подписью
* ввести d (секретный ключ)
* verify - проверка подписи для сообщения
* указать файл с сообщением
* указать файл с подписью
* ввести открытый ключ Q, два числа через пробел

## Пример работы

Пример вывода результата работы программы «электронной подписи»

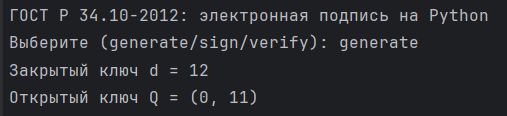


Рисунок 1 – Пример работы программы «электронной подписи» - generate

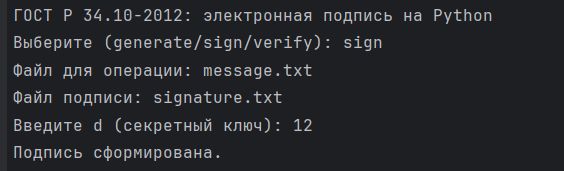


Рисунок 2 – Пример работы программы «электронной подписи» - sign

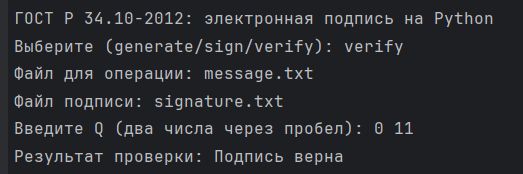


Рисунок 3 – Пример работы программы «электронной подписи» - verify

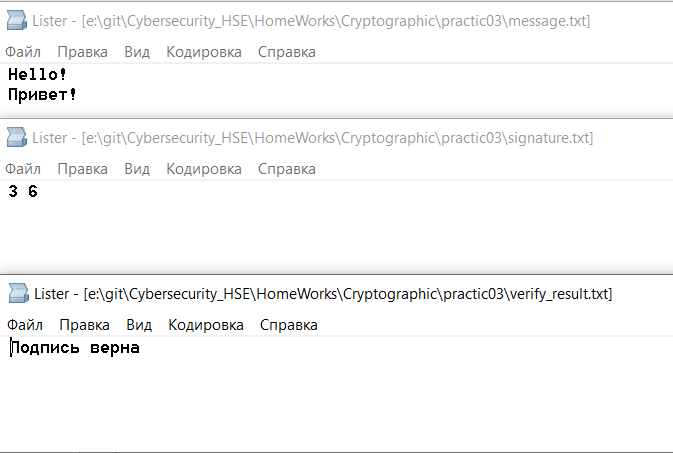


Рисунок 4 – Пример файлов: message.txt (исходное сообщение), signature.txt (подпись), verify\_result.txt (отчет)

# ПРИЛОЖЕНИЕ А.

**Cписк использованных источников**

1. Стинсон Д. Криптография. Теория и практика. -М.: Техносфера, 2006. – 608 с.
2. Фурсенко С.Ю. «Practical Cryptography in Python: реализация криптографических схем на практике». — М.: БХВ-Петербург, 2020. — 256 с.
3. Python Software Foundation. Python Documentation. – URL: <https://docs.python.org/3/>
4. Мартин Р. Чистый код: создание, анализ и рефакторинг // «Библиотека программиста (Питер)», 2024г.
5. Шоу Зед. Легкий способ выучить Python 3 // «Бомбора», 2021г.