**Федеральное Государственное Бюджетное**

**Образовательное Учреждение**

**Высшего Профессионального Образования**

**Национальный Исследовательский Университет «МЭИ»**

**Институт информационных и вычислительных технологий**

Кафедра Прикладной Математики

**«Программная реализация алгоритма шифрования на эллиптических кривых»**

**Курсовой проект**

по учебной дисциплине

«Защита данных»

Выполнил:

Крылов К.С.

Преподаватель:

Хорев П.Б.

**Москва 2021**

ОГЛАВЛЕНИЕ

1). ВВЕДЕНИЕ…………………………………………..…………………………....3

2). ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ………………………………………………....3

2.1). Эллиптические кривые над вещественными числами и групповой закон……..3

2.2). Эллиптические кривые над конечными полями и задача дискретного логарифмирования ……………………………………………………………………6

2.3). ECDH…………………………………………………………………….……….8

2.4). Среда разработки и язык………………………………………………………….9

3). ПРОГРАММНАЯ РЕАЛИЗАЦИЯ……………………………………………….10

3.2). Графический интерфейс…………………………………………………..…….10

3.3). Реализация алгоритмов ECDH………………………………………………….11

4). ТЕСТИРОВАНИЕ РАЗРАБОТАННОЙ ПРОГРАММЫ…………..…………..…12

5). ЗАКЛЮЧЕНИЕ…………………………………………………………………...14

СПИСОК ЛИТЕРАТУРЫ…………………………………………………..…….......15

Приложение 1………………………………..………………………………….….....16

**ВВЕДЕНИЕ**

До того, как криптография на эллиптических кривых стала популярной, почти все алгоритмы с открытым ключом основывались на RSA, DSA и DH, альтернативных криптосистемах на основе модулярной арифметики. Они всё еще популярны, и часто используются вместе с эллиптическими кривыми. Однако несмотря на то, что математика, лежащая в фундаменте RSA и подобных ей алгоритмов легко объяснима и понятна многим, а [грубые реализации пишутся довольно просто](http://code.activestate.com/recipes/578838-rsa-a-simple-and-easy-to-read-implementation/), они обладают рядом недостатков по сравнению с эллиптической криптографией, о которых будет изложено ниже.

Сегодня криптосистемы на эллиптических кривых используются в [TLS](https://tools.ietf.org/html/rfc4492), [PGP](https://tools.ietf.org/html/rfc6637) и [SSH](https://tools.ietf.org/html/rfc5656), важнейших технологиях, на которых базируются современный мир информационных технологий, а также криптовалютах, например Bitcoin.

**Цель** работы – разработка приложения для шифрования текстов по алгоритму ECDH под систему android[1] на языке kotlin[2].

Были поставлены и решены следующие **задачи**:

* проектирование и реализация графического интерфейса;
* проектирование логики взаимодействия компонентов программы;
* реализация основного алгоритма шифрования;
* тестирование и отладка программы.

В **первой** главе будет приведена краткая справка о эллиптических кривых, их основные преимущества и недостатки.

Во **второй** главе будут представлены результаты проектирования программы и графического интерфейса, описание возможностей программы, описание созданных при реализации стандарта методов.

В **третьей** главе будут приведены результаты тестирования программы на предмет корректной работы алгоритмов шифрования, интерфейса и логики программы.

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**Эллиптические кривые над вещественными числами и групповой закон**

**Эллиптической кривой**[3] будем считать множество точек, описываемых уравнением y2=x3+ax+b, при этом введем дополнительное условие, что 4a2+27b2 ≠ 0, которое необходимо для того, чтобы исключить сингулярные кривые – кривые, имеющие точки, в которых они не определены.

Т.к. необходимо чтобы частью кривой являлась бесконечно удаленная точка (которую обозначим символом 0), поэтому полное определение эллиптической кривой:

{ (x, y) ∈ ℝ2 | y2 = x3 + ax + b, 4a2 + 27b2 ≠ 0 } ⋃ { 0 }

**Абелевой группой**[4]будем называть множество G, для элементов которого обозначена операция сложения, имеющая свойства:

1. **замыкание:** если a ∈ G и b ∈ G, то a + b ∈ G;
2. **ассоциативность:** (a + b) + c = a + (b + c);
3. **коммутативность:** a + b = b + a;
4. существует **нейтральный элемент** 0, такой, что a + 0 = 0 + a = 0;
5. у каждого элемента a есть **обратная величина – -**a, такая что: a + -a = 0

Определим Абелеву группу для эллиптических кривых:

1. **Элементы группы –** точки аi на эллиптической кривой.
2. **Операция сложения** задается правилом а1 + а2 + а3 = 0, где точки лежат на одной прямой и их порядок не важен, из чего немедленно следует коммутативность и ассоциативность операции сложения.
3. **Нейтральный элемент** – это бесконечно удаленная точка 0.
4. **Обратная величина точки** – это точка, симметричная данной относительно оси Х.

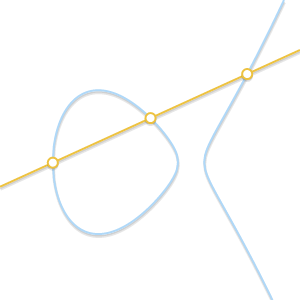


Рис.1.1 – Иллюстрация операции сложения на группе точек эллиптической кривой.

Любая Абелева группа имеет естественную структуру [модуля над кольцом](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_%D0%BD%D0%B0%D0%B4_%D0%BA%D0%BE%D0%BB%D1%8C%D1%86%D0%BE%D0%BC) [целых чисел](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BB%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE), поэтому а1 + а2 + а3 = 0 можно записать кака1 + а2 = - а3. Таким

образом, геометрически, сумма двух точек а1 и а2 – есть точка, обратная точке а3, через которая проходит прямая, проходящая через а1 и а2. [5]

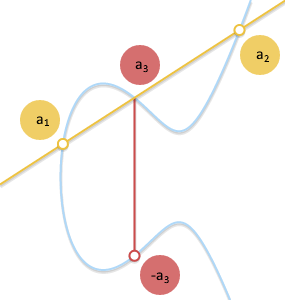


Рис.1.2 – Иллюстрация операции сложения на группе точек эллиптической кривой.

В случае, если а1 или а2 равны нулю мы не можем провести прямую через бесконечно удаленную точку, воспользуемся свойством нейтрального элемента: а1 + 0 = а1.

Если а2 = -a1 и прямая, проходящие через эти точки вертикальна, и не существует третьей точки пересечения кривой и прямой, можно воспользоваться свойством обратного элемента: a1 + -a1 = 0, т.е. бесконечно удаленная точка.

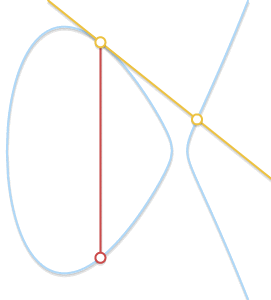
Самый сложный случай сложения - a1 + a1, т.к. через одну точку можно провести бесконечно большое количество прямых. В этом случае будем считать, что одна из точек не равна a1, а лишь стремится к ней – в таком случае прямая, проходящая через эти точки будет касательной к эллиптической кривой, тогда a1 + a1 = -a2, где a2 – точка, лежащая на пересечении прямой и кривой.

Рис.1.3 – Иллюстрация операции сложения в случае отсутствие 3й точки пересечения кривой и прямой.

Перейдем к алгебраическому методу сложения (нужно отметить, что большая часть ранее рассмотренных исключительные случаи аналогична геометрическому способу и повторно для алгебраического способа рассматриваться не будут).

Итак, есть две не нулевые точки P(xP, yP) и Q(xQ, yQ). Если они не совпадают (т.е. xP ≠ xQ), то прямая, проходящая через них имеет наклон, **.** Пересечение этой прямой и кривой – точка R(m2 – xP - xQ, yP + m(xR - xP)), т.е. (xP, yP) + (xQ, yQ) = (xR, -yR)

В случае, когда P = Q для наклона необходимо использовать уравнение

Добавим еще одну операцию – **скалярное умножение**. Скалярное умножение точки на число n есть сложение n таких чисел. Наивная реализация скалярного умножения требует n сложений, который имеет сложность О(2к), где к – количество десятичных разрядов, однако существуют алгоритмы, например алгоритм удвоения сложения[2], имеющий сложность О(k), где k – битовая длина.

**Эллиптические кривые над конечными полями и задача дискретного логарифмирования**

**Конечным полем**[6]будем называть конечное множество, для которого определены операции сложения, взятия противоположного значения, умножения и деления, при этом умножение и сложение обладают свойствами замкнутости, коммутативности и ассоциативности. Конечным полем, например, является множество чисел по модулю p, где

p – простое число. Множество целых чисел с непростым модулем не является полем, так как не имеет операции обратной умножению.

Теперь ограничим эллиптические кривые конечным полем:

{ (x, y) ∈ F2 | y2 = x3 + ax + b (mod p), 4a2 + 27b2 ≠ 0 (mod p)} ⋃ { 0 },

Где a и b – целые числа. Все операции будут работать как и в предыдущем параграфе, но теперь используются только целые числа по модулю p. Уточним лишь, что две точки лежат на прямой в конечном поле, если существует прямая, соединяющие эти точки

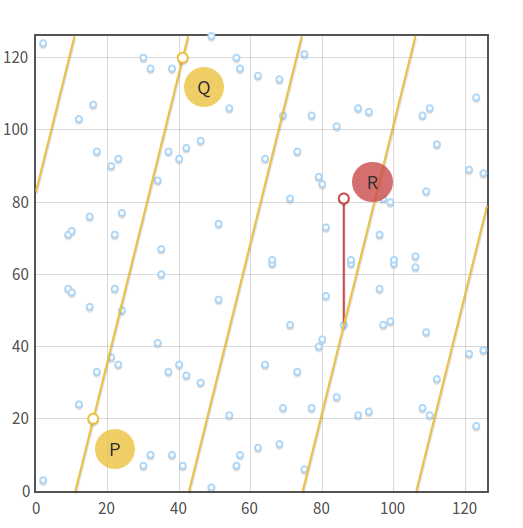
****

Рис 1.4 Иллюстрация суммы двух точек для эллиптической кривой над конечным полем.

Определим уравнения для суммы P+Q=-R для конечного поля. Если P(xP, yP) ≠ Q(xQ, yQ), то R = (m2 – xP - xQ (mod p), yP + m(xR - xP) (mod p)), где m=( yP - yQ)( xP- xQ)-1 (mod p) и при P = Q m = 3x2P+a) (2yP)-1 (mod p)

В конечном поле над эллиптической кривой, очевидно, находится конечное количество точек. Количество этих точек называется **порядком группы**. Вычисление количества точек перебором требует экспоненциального времени работы, но в 1985 году Рене Шуф изобрел алгоритм, работающий за полиномиальное время.

Скалярное умножение для целых чисел определено ток же, но с модулем как и для вещественных: nP = , но обладает одним важным свойством. Мало того, что количество точек, которые можно получить складывая точки последовательно не может превосходить p2, что следует из p – число вариантов каждой координаты в двумерном пространстве, на самом деле число таких точек может быть существенно меньше p2. Число таких точек будем называть **порядком подгруппы, порожденной точкой.** Согласно теореме Лагранжа, **порядок подгруппы —** это делитель порядка исходной группы, то есть если эллиптическая кривая содержит Т точек, а одна из подгрупп – n, то n является делителем N. Таким образом, чтобы найти порядок P группы - минимальное положительное целое n, такое, что nP = 0 – необходимо найти порядок кривой с помощью алгоритма Шуфа, найти все делители этого порядка и выбрать минимальный.

Для алгоритмов на эллиптических кривых требуются подгруппы с высоким порядком, поэтому обычно выбирается эллиптическая кривая, вычисляется её порядок (N), в качестве порядка группы (n) выбирается большой делитель, а потом находится подходящая базовая точка, а не наоборот. Число h = N/n по теореме Лагранжа всегда целое и имеет свое название – это **кофактор** группы. Теперь мы можем написать алгоритм поиска базовой точки на прямой:

1. Вычисляем порядок N эллиптической кривой.
2. Выбираем порядок n подгруппы. Чтобы алгоритм сработал, число должно быть простым и быть делителем N.
3. Вычисляем кофактор h =N/n.
4. Выбираем на кривой случайную точку P.
5. Вычисляем G = hP.
6. Если G равно 0, то возвращаемся к шагу 4. В противном случае мы нашли генератор подгруппы с порядком n и кофактором h.

Теперь мы должны обсудить вопрос: **е**сли мы знаем P и Q, то каким будет k, такое, что Q = kP? Эта задача, известная как **задача дискретного логарифмирования** для эллиптических кривых, считается «сложной», для которой не обнаружено алгоритма полиномиального времени, выполняемого на классическом компьютере. Однако у этой точки зрения нет математических доказательств.

Эта задача аналогична задаче дискретного логарифмирования, используемой в других криптосистемах, таких как DSA, протокол Диффи-Хеллмана и схема Эль-Гамаля. Названия задач совпадают неслучайно. Их разница в том, что в этих алгоритмах используется не скалярное умножение, а возведение в степень по модулю. Их задачу дискретного логарифмирования можно сформулировать так: если известны a и b, то каким будет k , такое, что ak mod(p) = b?  
Обе эти задачи дискретны, потому что в них используются конечные множества (а конкретнее — циклические подгруппы). И они являются логарифмическими.

ECC интересна тем, что на сегодняшний момент задача дискретного логарифмирования для эллиптических кривых считается сложнее по сравнению с другими схожими задачами, используемыми в криптографии. Это подразумевает, что нам потребуется меньше бит для целого k, чтобы получить тот же уровень защиты, что и в других криптосистемах.[7]

**ECDH**

Алгоритмы эллиптических кривых будут работать в циклической подгруппе эллиптической кривой над конечным полем. Поэтому алгоритмам потребуются следующие параметры:

* Простое p, задающее размер конечного поля.
* Коэффициенты a и b уравнения эллиптической кривой.
* Базовая точка G, генерирующая подгруппу.
* Порядок n подгруппы.
* Кофактор h подгруппы.

Имея всю необходимую информацию, мы можем приступить к формированию ключей. **Закрытый ключ** – это случайное целое число d от 1 до n – 1 (n – порядок подгруппы). **Открытым ключом** будет точка H = dG. В случае, когда пользователи имеют возможность выбирать кривую открытым ключем можно считать совокупность p, a, b, G, n, h и H.

ECDH (англ. Elliptic curve Diffie–Hellman) – протокол согласования ключей. Его алгоритм выглядит следующим образом:

1). Сначала Алиса и Боб генерируют собственные закрытые и открытые ключи. У Алисы есть закрытый ключ dA и открытый ключ HA = dAG, у Боба есть ключи dB и HB. Алиса, и Боб используют одинаковые параметры области определения: одну базовую точку G на одной эллиптической кривой в одинаковом конечном поле.

2). Алиса и Боб обмениваются открытыми ключами HA и HB по незащищённому каналу. Посредник (Man In the Middle) перехватывает HA и HB, но не может определить ни dA, ни dB, не решив задачу дискретного логарифмирования.

Алиса вычисляет S = dAHB (с помощью собственного закрытого ключа и открытого ключа Боба), а Боб вычисляет S = dBHA (с помощью собственного закрытого ключа и открытого ключа Алисы). S одинаков и для Алисы, и для Боба:

S = dAHB = dA(dBG) = dB(dAG) = dBHA

Получив общий секретный ключ, Алиса и Боб могут обмениваться данными с симметричным шифрованием. Например, они могут использовать координату x ключа как ключ для шифрования сообщений.[8]

В большинстве случаев нет необходимости придумывать свою кривую, вычислять ее порядок и точку на ней, безопасные кривые давно опубликованы в стандартах и библиотеках, например OpenSSL. В данной работе будет использована кривая secp256k1.

**Среда разработки и язык**

Т.к. цель работы – написание приложения для системы андроид в качестве среды разработки была выбрана android studio[1] - официальное средство разработки Android приложений от компании Google, основанное на intellijIDEA от компании JetBrains. Android studio позволяет разрабатывать android приложения на трех языках – java, c++ и kotlin[2], из которых был выбран последний, как самый новый, лаконичный, простой и типобезопасный.

**ПРОГРАММНАЯ РЕАЛИЗАЦИЯ**

**Графический интерфейс.**

Программа при запуске открывает фрагмент настроек – в нем хранится информация о параметрах эллиптической кривой. Прокрутив фрагмент в самый низ и нажав кнопку

«сгенерировать» можно получить ключи шифрования. Параметры кривой для экономии места считываются в 16ричном формате.

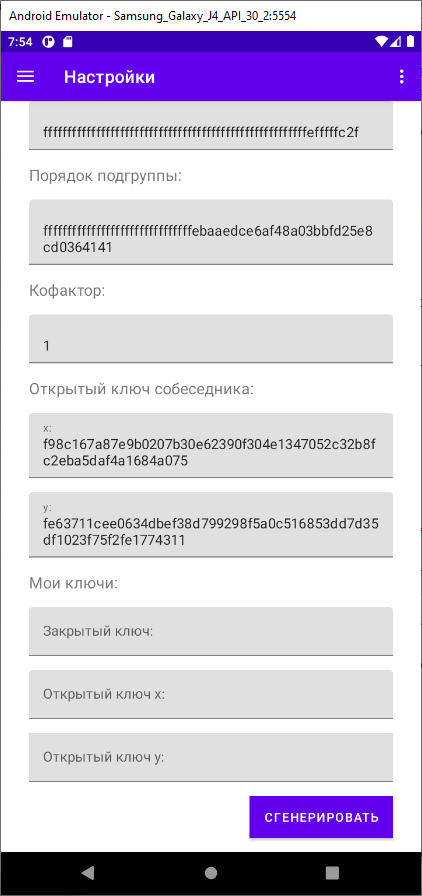
** **

Рис 2.1 Интерфейс фрагмента «настройки»

Рис 2.2 Интерфейс фрагмента «настройки»

При нажатии на левый верхний угол открывается список всех фрагментов.

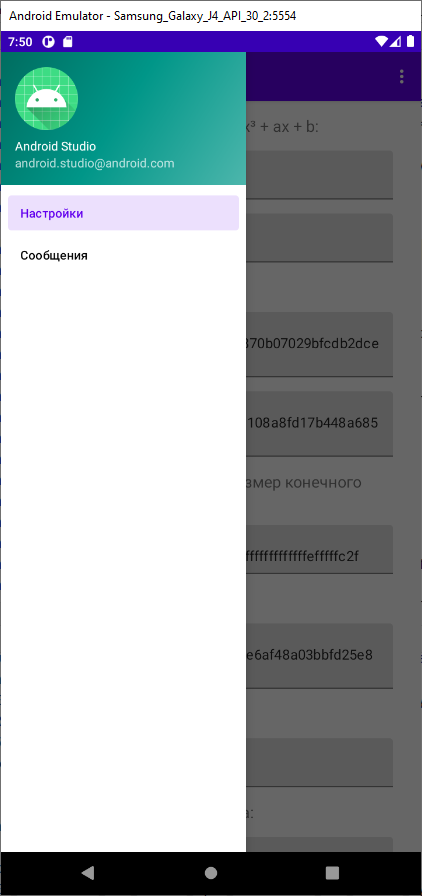
****

Рис 2.3 Интерфейс меню приложения.

В фрагменте «сообщения» можно ввести текст в строку снизу. При нажатии кнопки «отправить» сообщение зашифруется добавится в список.

**Реализация алгоритмов ECDH.**

Модулярная арифметика реализована в классе ModularArithmetic

Методы:

**Modulo** – Основная функция класса, получает на вход целые числа n и p.

Возвращает значение n по модулю p.

**inverseMod** – На вход получает целое число n и простое p. Возвращает обратную величину n по модулю p.

Операции с эллиптическими кривыми находятся в классе EllipticCurve

Методы:

**isOnCurve** – На вход получает кривую и точку, проверяет, принадлежит ли точка заданной кривой.

**negativePoint -** Отрицательное значение заданной точки. (точка, находящаяся на пересечении эллиптической кривой и прямой, параллельной оси y и пересекающей заданную точку.)

**addPoint –** функция сложения двух точек, принадлежащих эллиптической кривой.

**scalarMult** – функция, принимающая точку и число. Реализует скалярное произведение числа и точки.

Класс Cryptoprovider содержит в себе методы, относящиеся непосредственно к шифрованию.

Методы:

**randomBigInteger –** функция**,** принимающая на вход предел и возвращающая случайное число от 1 до предела.

**privateKeyGen** – метод для генерации закрытого ключа.

**publicKeyGen** – метод для генерации открытого ключа.

**encryptMessage** – метод, принимающий на вход строку и шифрующий её.

**decryptMessage** – метод, принимающий на вход зашифрованную строку и расшифровывающий её.

**ТЕСТИРОВАНИЕ РАЗРАБОТАННОЙ ПРОГРАММЫ.**

Тесты находятся в директории com.example.ellipticalcryptography:

Таблица 3.1 модульное тестирование

|  |  |  |
| --- | --- | --- |
| класс ModularArithmeticUnitTest отвечает за тестирование модулярной арифметики. | | |
| inverseMod\_Test | inverseMod(-13), 23) | 7 |
| mod\_Test | -22.modulo(23)  1931.modulo(23)  84874.modulo(23") | 1  22  4 |
| Класс EllipticCurveUnitTest отвечает за тестирование операций над эллиптической кривой. | | |
| isOnCurve\_Test | ec = EllipticCurve(  prime = 23,  a = -7,  b = 10 )  ec.isOnCurve(3, 4) | true |
| addPoint\_Test | ec = EllipticCurve(  prime = 23,  a = -7,  b = 10 )  ec.addPoint((2,2), (3, 4)) | (22, 4) |
| addPointSecp256k1\_Test | ec = Secp256k1  point =(55066263022277343669578718895168534326250603453777594175500187360389116729240, 32670510020758816978083085130507043184471273380659243275938904335757337482424)   ec.addPoint(point, point) | (89565891926547004231252920425935692360644145829622209833684329913297188986597, 12158399299693830322967808612713398636155367887041628176798871954788371653930) |
| negativePoint\_Test | ec = EllipticCurve(  prime = BigInteger("23"),  a = BigInteger("-7"),  b = BigInteger("10"), )  ec.negativePoint(8,12) | (8, 12) |
| Класс CryptoproviderUnitTest отвечает за проверку правильности создания ключей и шифрования. | | |
| keyGen\_Test | cp = Cryptoprovider(Secp256k1) cp.privateKey = BigInteger("100") cp.publicKeyGen() | (107303582290733097924842193972465022053148211775194373671539518313500194639752,103795966108782717446806684023742168462365449272639790795591544606836007446638) |
| keyEquality\_Test | val sharedKey1 = cp1.curve.scalarMult(cp1.privateKey!!, cp2.publicKey) val sharedKey2 = cp2.curve.scalarMult(cp2.privateKey!!, cp1.publicKey) | true |

**ЗАКЛЮЧЕНИЕ**

По итогам этой работы было разработано программное обеспечение, реализующее протокол ECDР и шифрование текстовых сообщений, использования ключа с регулируемой минимальной длиной и сложностью.

Были получены навыки по реализации криптографических стандартов, составлению пояснительной записки, оптимизации алгоритмов.

**СПИСОК ИСТОЧНИКОВ**

[1] Документация андроид

<https://developer.android.google.cn/work/guide> (Здесь и далее дата обращения 20.01.2022)

[2] Документация языка kotlin

<https://kotlinlang.org/docs/home.html>

[3] <https://ru.wikipedia.org/wiki/Эллиптическая_кривая>

[4] Винберг Э. Б. Курс алгебры. — 3-е изд. — М.: Факториал Пресс, 2002. — 544 с. — 3000 экз. — [ISBN 5-88688-060-7](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5886880607)

[5] <https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/>

[6] Ю. И. Журавлев, Ю. А. Флеров, М. Н. Вялый. Дискретный анализ. Основы высшей алгебры. — М.: МЗ Пресс, 2007. — С. 151. — 224 с.

[7] https://andrea.corbellini.name/2015/05/23/elliptic-curve-cryptography-finite-fields-and-discrete-logarithms/

[8] https://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

**Приложение 1.**

ЛИСТНИНГ ПРОГРАММЫ

Файл ModularArithmetic:

package com.example.ellipticalcryptography.calculation  
  
import java.math.BigInteger  
  
abstract class ModularArithmetic {  
 companion object{  
 //Обратная величина number по модуля prime.  
 fun inverseMod(number: BigInteger, prime: BigInteger): BigInteger {  
 if (number == BigInteger("0")) error("Zero division error")  
 if (number < BigInteger("0")) {  
 return prime - inverseMod(-number, prime)  
 }  
  
 return when{  
 number == BigInteger("0") -> error("Zero division error")  
 number < BigInteger("0") -> prime - inverseMod(-number, prime)  
 else ->{  
 //Расширенный алгоритм Евклида  
 var s = BigInteger("0")  
 var sOld = BigInteger("1")  
 var t = BigInteger("1")  
 var tOld = BigInteger("0")  
 var r = prime  
 var rOld = number  
 while(r != BigInteger("0")){  
 val quotient = rOld / r  
  
 var buff = r  
 r = rOld - quotient \* r  
 rOld = buff  
  
 buff = t  
 t = tOld - quotient \* t  
 tOld = buff  
  
 buff = s  
 s = sOld - quotient \* s  
 sOld = buff  
 }  
  
 assert((number \* sOld).modulo(prime) == BigInteger("1"))  
  
 sOld.modulo(prime)  
  
 }  
 }  
 }  
 }  
}  
  
fun BigInteger.modulo(number: BigInteger): BigInteger{  
 var result = this % number  
 if(result < BigInteger("0")) result = (result + number) % number  
 return result  
}

Файл EllipticCurve:

package com.example.ellipticalcryptography.calculation  
  
import java.math.BigInteger  
  
data class EllipticCurve(  
 val prime: BigInteger = BigInteger("1"),  
 //Коэффициенты уравнения эллиптической кривой y^2 = x^3 + ax + b.  
 val a: BigInteger = BigInteger("0"),  
 val b: BigInteger = BigInteger("0"),  
 //Точка, генерирующая подгруппу.  
 val point: Pair<BigInteger, BigInteger> =  
 BigInteger("0")  
 to BigInteger("0"),  
 val cofactor: BigInteger = BigInteger("1"),  
 //Порядок.  
 val size: BigInteger = BigInteger("1")  
)  
  
val Secp256k1 = EllipticCurve(  
 prime = BigInteger("fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f", 16),  
 a = BigInteger("0"),  
 b = BigInteger("7"),  
 point = BigInteger("79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798", 16)  
 to BigInteger("483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8", 16),  
 cofactor = BigInteger("1"),  
 size = BigInteger("fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141", 16)  
)  
  
//Возвращает true если точка лежит на кривой.  
fun EllipticCurve.isOnCurve(point: Pair<BigInteger, BigInteger>?): Boolean{  
 //null - бесконечно удаленная точка  
 if (point == null) return true  
 val x = point.first  
 val y = point.second  
 return (y \* y - x \* x \* x - this.a \* x - this.b).modulo(this.prime) == BigInteger("00")  
}  
  
//Отрицательное значение точки. (точка, находящаяся на пересечении эллиптической кривой и прямой,  
//параллельной оси y и пересекающей заданную точку.)  
fun EllipticCurve.negativePoint(point: Pair<BigInteger, BigInteger>?): Pair<BigInteger, BigInteger>?{  
 if (point == null) return point  
 assert(this.isOnCurve(point))  
 val x = point.first  
 val y = point.second  
  
 val result = x to ((-y).modulo(this.prime))  
  
 assert(this.isOnCurve(point))  
 return result  
}  
  
//Сложение двух точек - третья точка, лежащая на кривой.  
fun EllipticCurve.addPoint(point1: Pair<BigInteger, BigInteger>?, point2: Pair<BigInteger, BigInteger>?): Pair<BigInteger, BigInteger>?{  
 assert(this.isOnCurve(point1))  
 assert(this.isOnCurve(point2))  
  
 if (point1 == null) return point2  
 if (point2 == null) return point1  
  
  
 val x1 = point1.first  
 val y1 = point1.second  
 val x2 = point2.first  
 val y2 = point2.second  
  
  
 //point1 + (-point1) = 0  
 if ((x1 == x2) and (y1 != y2)) return null  
  
 var m = BigInteger("0")  
  
 if (x1 == x2){  
 //point1 == point2.  
 m = (BigInteger("3") \* x1 \* x1 + this.a) \* ModularArithmetic.inverseMod(BigInteger("2") \* y1, this.prime)  
 } else {  
 //point1 != point2.  
 m = (y1 - y2) \* ModularArithmetic.inverseMod(x1 - x2, this.prime)  
 }  
  
 val x3 = m \* m - x1 - x2  
 val y3 = y1 + m \* (x3 - x1)  
 val result = (x3.modulo(this.prime) to (-y3).modulo(this.prime))  
  
 assert(this.isOnCurve(result))  
 return result  
}  
  
fun EllipticCurve.scalarMult(multiplier: BigInteger, point: Pair<BigInteger, BigInteger>?): Pair<BigInteger, BigInteger>?{  
 var k = multiplier  
 assert(this.isOnCurve(point))  
 if ((k.modulo(this.size) == BigInteger("0")) or (point == null))  
 return null  
  
 //k \* point = -k \* (-point)  
 if (k < BigInteger("0")) return scalarMult(-k, this.negativePoint(point)!!)  
  
 var result: Pair<BigInteger, BigInteger>? = null  
 var addend = point  
  
 while(k != BigInteger("0")){  
 if ((k and BigInteger("1")) != BigInteger("0")){  
 result = this.addPoint(result, addend)  
 }  
  
 addend = this.addPoint(addend, addend)!!  
 k = k.shiftRight(1)  
 }  
  
 assert(this.isOnCurve(point))  
 return result  
}

Файл Cryptoprovider:

package com.example.ellipticalcryptography.calculation  
  
import java.math.BigInteger  
import java.util.\*  
import com.example.ellipticalcryptography.calculation.ModularArithmetic as Mod  
import com.example.ellipticalcryptography.calculation.\*  
import com.example.ellipticalcryptography.calculation.ModularArithmetic.Companion.inverseMod  
  
  
fun randomBigInteger(n: BigInteger): BigInteger {  
 val rand = Random()  
 var result = BigInteger(n.bitLength(), rand)  
 while (result.compareTo(n) >= 0) {  
 result = BigInteger(n.bitLength(), rand)  
 }  
 return result  
}  
  
//a, b - коэффициенты эллиптической кривой.  
class Cryptoprovider(ellepticCurve: EllipticCurve) {  
 val curve = ellepticCurve  
 var privateKey: BigInteger? = null  
 var publicKey: Pair<BigInteger, BigInteger>? = null  
  
 fun privateKeyGen(){  
 privateKey = randomBigInteger(curve.size)  
 }  
  
 fun publicKeyGen(){  
 if (privateKey != null) publicKey = curve.scalarMult(privateKey!!, curve.point)  
 }  
  
 fun encryptMessage(message: String, key: Pair<BigInteger, BigInteger>): String {  
 if (privateKey == null) {  
 privateKeyGen()  
 publicKeyGen()  
 }  
  
 if (publicKey == null) {  
 publicKeyGen()  
 }  
 val sharedKey = curve.scalarMult(privateKey!!, key)?.first  
 val cypheredMessage = BigInteger(message.toByteArray())  
  
 if (sharedKey != null) {  
 return (cypheredMessage + sharedKey).modulo(curve.prime).toByteArray().toString()  
 } else return ""  
 }  
  
  
  
 fun decryptMessage(message: String, key: Pair<BigInteger, BigInteger>): String {  
 if ((privateKey == null) or (publicKey == null)) error("Generated keys expected.")  
 val sharedKey = curve.scalarMult(privateKey!!, key)?.first  
 val uncypheredMessage = BigInteger(message.toByteArray())  
  
 if (sharedKey != null) {  
 return (uncypheredMessage - sharedKey).modulo(curve.prime).toByteArray().toString()  
 } else return ""  
 }  
}

Файл ExampleUnitTest:

package com.example.ellipticalcryptography  
  
import com.example.ellipticalcryptography.calculation.\*  
import com.example.ellipticalcryptography.calculation.ModularArithmetic.Companion.inverseMod  
import org.junit.Test  
  
import org.junit.Assert.\*  
import java.math.BigInteger  
  
/\*\*  
 \* Example local unit test, which will execute on the development machine (host).  
 \*  
 \* See [testing documentation](http://d.android.com/tools/testing).  
 \*/  
class ModularArithmeticUnitTest {  
 @Test  
 fun inverseMod\_Test(){  
 assert(inverseMod(BigInteger("-13"), BigInteger("23")) == BigInteger("7"))  
 }  
  
 @Test  
 fun mod\_Test(){  
 assert(BigInteger("-22").modulo(BigInteger("23")) == BigInteger("1"))  
 assert(BigInteger("1931").modulo(BigInteger("23")) == BigInteger("22"))  
 assert((-BigInteger("-84874")).modulo(BigInteger("23")) == BigInteger("4"))  
 }  
  
}  
  
class EllipticCurveUnitTest {  
 @Test  
 fun isOnCurve\_Test() {  
  
 val ec = EllipticCurve(  
 prime = BigInteger("23"),  
 a = BigInteger("-7"),  
 b = BigInteger("10"),  
 )  
  
 assertEquals(ec.isOnCurve(BigInteger("3") to BigInteger("4")), true)  
  
 }  
  
 @Test  
 fun addPoint\_Test() {  
 val ec = EllipticCurve(  
 prime = BigInteger("23"),  
 a = BigInteger("-7"),  
 b = BigInteger("10"),  
 )  
  
 val calculatedPoint = ec.addPoint(  
 BigInteger("2") to BigInteger("2"),  
 BigInteger("3") to BigInteger("4"))  
 assert(calculatedPoint == BigInteger("22") to BigInteger("4"))  
 }  
  
 @Test  
 fun addPointSecp256k1\_Test() {  
 val ec = Secp256k1  
 val point =  
 BigInteger("55066263022277343669578718895168534326250603453777594175500187360389116729240") to  
 BigInteger("32670510020758816978083085130507043184471273380659243275938904335757337482424")  
  
 val calculatedPoint = ec.addPoint(point, point)  
 assert(  
 calculatedPoint ==  
 BigInteger("89565891926547004231252920425935692360644145829622209833684329913297188986597") to  
 BigInteger("12158399299693830322967808612713398636155367887041628176798871954788371653930")  
 )  
 }  
  
 @Test  
 fun negativePoint\_Test() {  
 val ec = EllipticCurve(  
 prime = BigInteger("23"),  
 a = BigInteger("-7"),  
 b = BigInteger("10"),  
 )  
  
 assert(ec.negativePoint(BigInteger("8") to BigInteger("12"))  
 == BigInteger("8") to BigInteger("11"))  
 }  
}  
  
class CryptoproviderUnitTest {  
 @Test  
 fun keyGen\_Test() {  
 val cp = Cryptoprovider(Secp256k1)  
 cp.privateKey = BigInteger("100")  
 cp.publicKeyGen()  
 assert(cp.publicKey ==  
 BigInteger("107303582290733097924842193972465022053148211775194373671539518313500194639752") to  
 BigInteger("103795966108782717446806684023742168462365449272639790795591544606836007446638"))  
 }  
  
 @Test  
 fun keyEquality\_Test(){  
 val cp1 = Cryptoprovider(Secp256k1)  
 val cp2 = Cryptoprovider(Secp256k1)  
  
 cp1.privateKeyGen()  
 cp1.publicKeyGen()  
 cp2.privateKeyGen()  
 cp2.publicKeyGen()  
  
 val sharedKey1 = cp1.curve.scalarMult(cp1.privateKey!!, cp2.publicKey)  
 val sharedKey2 = cp2.curve.scalarMult(cp2.privateKey!!, cp1.publicKey)  
  
 assert(sharedKey1 == sharedKey2)  
 }  
  
 @Test  
 fun messageEqulity\_Test(){  
 val cp1 = Cryptoprovider(Secp256k1)  
 val cp2 = Cryptoprovider(Secp256k1)  
  
 cp1.privateKeyGen()  
 cp1.publicKeyGen()  
 cp2.privateKeyGen()  
 println(cp1.publicKey?.first?.toString(16))  
 println(cp1.publicKey?.second?.toString(16))  
 cp2.publicKeyGen()  
  
 val message = "Hello"  
 val encryptedMessage = cp1.encryptMessage(message, cp2.publicKey!!)  
 assertEquals(message, cp2.decryptMessage(encryptedMessage, cp1.publicKey!!))  
 }  
}