Учреждение образования

«Белорусский государственный технологический университет»

**Кафедра информационных систем и технологий**

**«Отчёт по лабораторной работе №13**

“Исследование криптографических алгоритмов на основе эллиптических кривых”

**Выполнил:** студент 3 курса

4 группы специальности ПОИТ

Мозолевский Александр Дмитриевич

**Проверил:** преподаватель

Сазонова Дарья Владимировна

Минск 2023

**Найти точки ЭК для значений х, указанных в таблице.**

Реализация приложения:

|  |
| --- |
| package main  import (  "fmt"  "math/big"  )  func main() {  a := big.NewInt(-1)  b := big.NewInt(1)  p := big.NewInt(751)  for x := big.NewInt(516); x.Cmp(big.NewInt(550)) <= 0; x = new(big.Int).Add(x, big.NewInt(1)) {  y := new(big.Int).Sub(new(big.Int).Exp(x, big.NewInt(3), p), new(big.Int).Mul(x, a))  y.Add(y, b)  y.Mod(y, p)  if isOnCurve(x, y, a, b, p) {  fmt.Printf("(%v, %v)\n", x, y)  }  }  }  func isOnCurve(x, y, a, b, p \*big.Int) bool {  left := new(big.Int).Exp(y, big.NewInt(2), p)  right := new(big.Int).Exp(x, big.NewInt(3), p)  right.Mul(right, x)  right.Add(right, new(big.Int).Mul(x, a))  right.Add(right, b)  right.Mod(right, p)  return left.Cmp(right) == 0  } |

Результат выполнения:



**Разработать приложение для выполнения операций над точками кривой: а) kР; б) Р + Q; в) kР + lQ – R; г) Р – Q + R.**

**Создать оконное приложение для зашифрования/расшифрования собственной фамилии (или имени – по выбору) на основе ЭК, указанной в задании 1, для генерирующей точки G = (0, 1).**

**Создать оконное приложение для генерации/верификации ЭЦП на основе алгоритма ЕСDSA: ЭК Е751(–1, 1) c генерирующей точкой G = (416, 55); порядок точки q = 13. Дополнительные параметры – в соответствии с вариантом из табл. 11.8 и 11.10.**

Реализация приложения:

|  |
| --- |
| package main  import (  "crypto/elliptic"  "fmt"  "math/big"  "gitlab.com/elktree/ecc"  )  type Point struct {  x, y \*big.Int  }  func NewPoint(x, y int64) \*Point {  return &Point{  x: big.NewInt(x),  y: big.NewInt(y),  }  }  func (p \*Point) String() string {  return fmt.Sprintf("(%v, %v)", p.x, p.y)  }  func (p \*Point) Add(q \*Point, a, b, prime \*big.Int) \*Point {  if p.IsInfinity() {  return q  }  if q.IsInfinity() {  return p  }  if p.Equals(q) || p.Equals(q.Negate()) {  return p.Double(a, b, prime)  }  lambda := new(big.Int)  lambda.Sub(q.y, p.y)  lambda.Mod(lambda, prime)  if lambda.Cmp(big.NewInt(0)) == 0 {  return NewPoint(0, 0)  }  lambda.Mul(lambda.ModInverse(lambda, prime), new(big.Int).Sub(q.x, p.x))  lambda.Mod(lambda, prime)  x3 := new(big.Int)  x3.Exp(lambda, big.NewInt(2), prime)  x3.Sub(x3, p.x)  x3.Sub(x3, q.x)  x3.Mod(x3, prime)  y3 := new(big.Int)  y3.Sub(p.x, x3)  y3.Mul(lambda, y3)  y3.Sub(y3, p.y)  y3.Mod(y3, prime)  return &Point{x3, y3}  }  func (p \*Point) Subtract(q \*Point, a, b, prime \*big.Int) \*Point {  qNeg := q.Negate()  return p.Add(qNeg, a, b, prime)  }  func (p \*Point) Multiply(k \*big.Int, a, b, prime \*big.Int) \*Point {  result := NewPoint(0, 0)  accumulator := NewPoint(p.x.Int64(), p.y.Int64())  for i := k.BitLen() - 1; i >= 0; i-- {  result = result.Add(result, a, b, prime)  if k.Bit(i) == 1 {  result = result.Add(accumulator, a, b, prime)  }  accumulator = accumulator.Add(accumulator, a, b, prime)  }  return result  }  func (p \*Point) Double(a, b, prime \*big.Int) \*Point {  if p.IsInfinity() {  return p  }  if p.y.Cmp(big.NewInt(0)) == 0 {  return NewPoint(0, 0)  }  lambda := new(big.Int)  lambda.Exp(p.x, big.NewInt(2), prime)  lambda.Mul(lambda, big.NewInt(3))  lambda.Add(lambda, a)  lambda.Mod(lambda, prime)  if lambda.Cmp(big.NewInt(0)) == 0 {  return NewPoint(0, 0)  }  lambda.Mul(lambda.ModInverse(lambda, prime), big.NewInt(2).Mul(p.y, prime))  lambda.Mod(lambda, prime)  x3 := new(big.Int)  x3.Exp(lambda, big.NewInt(2), prime)  x3.Sub(x3, big.NewInt(2).Mul(p.x, prime))  x3.Mod(x3, prime)  y3 := new(big.Int)  y3.Sub(p.x, x3)  y3.Mul(lambda, y3)  y3.Sub(y3, p.y)  y3.Mod(y3, prime)  return &Point{x3, y3}  }  func (p \*Point) Negate() \*Point {  if p.IsInfinity() {  return p  }  return &Point{p.x, new(big.Int).Neg(p.y)}  }  func (p \*Point) Equals(q \*Point) bool {  if p.IsInfinity() && q.IsInfinity() {  return true  }  return p.x.Cmp(q.x) == 0 && p.y.Cmp(q.y) == 0  }  func (p \*Point) IsInfinity() bool {  return p.x.Cmp(big.NewInt(0)) == 0 && p.y.Cmp(big.NewInt(0)) == 0  }  func main() {  a := big.NewInt(-1)  b := big.NewInt(1)  prime := big.NewInt(751)  pointP := NewPoint(3, 6)  pointQ := NewPoint(5, 1)  k := big.NewInt(8)  l := big.NewInt(5)  resultA := pointP.Multiply(k, a, b, prime)  resultB := pointP.Add(pointQ, a, b, prime)  resultC := pointP.Multiply(k, a, b, prime).Add(pointQ.Multiply(l, a, b, prime), a, b, prime).Subtract(pointQ, a, b, prime)  resultD := pointP.Subtract(pointQ, a, b, prime).Add(pointP, a, b, prime)  fmt.Println("kP:", resultA)  fmt.Println("P + Q:", resultB)  fmt.Println("kP + lQ - R:", resultC)  fmt.Println("P - Q + R:", resultD)  // Encrypt and decrypt message  fmt.Println()  pub, priv, \_ := ecc.GenerateKeys(elliptic.P521())  plaintext := "Mozolevskiy"  fmt.Println("Plaintext:", plaintext)  encrypted, \_ := pub.Encrypt([]byte(plaintext))  fmt.Println("Encrypted:", encrypted)  decrypted, \_ := priv.Decrypt(encrypted)  fmt.Println("Decrypted:", string(decrypted))  // Sign and verify message  fmt.Println()  pub, priv, \_ = ecc.GenerateKeys(elliptic.P384())  plaintext = "secret secrets are no fun, secret secrets hurt someone"  sig, \_ := priv.SignMessage([]byte(plaintext))  fmt.Println("Signature:", sig)  verified, \_ := pub.VerifyMessage([]byte(plaintext), sig)  fmt.Println(verified)  } |

Результат выполнения:

