lab1 报告

实验目的

- 理解非对称加密算法
- 理解椭圆曲线算法ECC
- 实现比特币上的椭圆曲线secp256k1算法

实验内容

完成签名和验签对应的函数部分

实现部分：

```
type ECC interface {
    Sign(msg []byte, secKey *big.Int) (*Signature, error)
    VerifySignature(msg []byte, signature *Signature, pubkey *Point) bool
}
```

签名流程

1. 我们已知z和满足eG=P的e。
2. 随机选取k。
3. 计算R=kG,及其x轴坐标r。
4. 计算 s=(z+re)/k。
5. (r,s) 即为签名结果。

验证流程

1. 接收签名者提供的(r,s)作为签名，z是被签名的内容的哈希值。P是签名者的公钥（或者公开的点）。
2. 计算 u=z/s 和 v=r/s。
3. 计算 uG + vP = R。
4. 如果R的x轴坐标等于r，则签名是有效的

个人实现

签名部分：

1. 调用 `crypto` 中的 `crypto.Keccak256` 函数计算msg的双sha256值，即z，并且将它存到一个int中，调用 `big.ing.setbyte()`
2. 直接用实现好的 `newRand()` 取得k，不过看wiki好像说k也不能随便取？
3. 调用 `Multi` 求出R = kG，直接取R.x即为x坐标r
4. 先求k的逆元，调用Inv函数
5. 最后利用big.int的api即可计算出s，同时还要注意模k

验证过程：

1. 同签名部分，调用 `crypto` 中的 `crypto.Keccak256` 函数计算msg的双sha256值，即z，并且将它存到一个int中，调用 `big.ing.setbyte()`
2. 把签名的s求逆元，得到$1/s$

3. 用big.int的mul接口，求出u,v
4. 最后利用提供的ADD和Multi函数，求R
5. 把R和r比较，一致返回true，否则返回false

签名部分：

```
    //计算msg的哈希，即z
    msgHash := crypto.Keccak256(msg)
    var msgVal big.Int
    msgVal.SetBytes(msgHash)

    //随机生成一个k
    randPoint, _ := newRand()

    //计算R值
    R := Multi(G, randPoint)

    //一步步计算(re+z)/k mod N
    invK := Inv(randPoint, N)
    //get r*e
    re1 := new(big.Int)
    re1.Mul(R.X, secKey)
    //calc re+z
    zpre := new(big.Int)
    zpre.Add(re1, &msgVal)
    //calc (re+z)/k
    s1 := new(big.Int)
    s1.Mul(zpre, invK)
    //calc (re+z)/k mod N,which is s
    s := new(big.Int)
    s.Mod(s1, N)

    signature := Signature{s, R.X}
    return &signature, nil
```

验签部分：

```
    msgHash := crypto.Keccak256(msg)
    hashInt := new(big.Int)
    hashInt.SetBytes(msgHash)

    //calc 1/s
    inv_s := Inv(signature.s, N)

    //calc u = z/s
    u1 := new(big.Int)
    u1.Mul(inv_s, hashInt)
    u := new(big.Int)
    u.Mod(u1, N)
    //calc v =r/s
    v1 := new(big.Int)
    v1.Mul(signature.r, inv_s)
    v := new(big.Int)
    v.Mod(v1, N)

    //calc R = uG+vP
    R := Add(Multi(G, u), Multi(pubkey, v))

    //注意这里一定要用cmp方法，不然比较失败的
    if R.X.Cmp(signature.r) == 0 {
```

```
            return true
        }
        return false
```

sha256部分：

1. 参考了SHA-2
2. 首先是有8个初始哈希值

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

   这是前八个质数的平方的小数部分

3. 还有64个初始常数

```
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x59f111f1, 0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354,
0x766a0abb, 0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

4. 首先是对消息进行预处理，把消息先加一个1bit，后面加上若干个0，使得消息位长l满足 $l \equiv 448 \mod 512$，后面补上64位原消息长度。

```
    //补1
    padding := append(msg, 0x80)
    emptyByte := new(byte)
    if len(padding)%64 < 56 {
        //mod 512 < 448
        for len(padding)%64 < 56 {
            padding = append(padding, *emptyByte)
        }
    } else {
        numOfByteToBeAdd := 64 + 56 - len(padding)
        for i := 0; i < numOfByteToBeAdd; i++ {
            padding = append(padding, *emptyByte)
        }
    }
    msgLen := make([]byte, 8)
    binary.BigEndian.PutUint64(msgLen, uint64(len(msg))*8)
```

```
    padding = append(padding, msgLen...)
```

5. 然后把消息进行切分，分割成512bit的块，注意是大端系统

```
message_blocks := [][]byte{}
    for i := 0; i < len(padding)/64; i++ {
        each_block := make([]byte, 64)
        copy(each_block, padding[i*64:i*64+64])
        message_blocks = append(message_blocks, each_block)
    }
```

6. 对每一个512位的块，开始计算扩展长，计算过程为

```
for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18)
xor(w[i-15] rightshift 3)
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19)
xor(w[i-2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1
```

具体实现为，可以用bits的rotateleft实现rotateright

```
w := [64]uint32{}
        //分成16 个 32 位整数
        for i := 0; i < 16; i++ {
            w[i] = binary.BigEndian.Uint32(block[i*4 : i*4+4])
        }

        for i := 16; i < 64; i++ {
            s0 := bits.RotateLeft32(w[i-15], -7) ^ bits.RotateLeft32(w[i-
15], -18) ^ (w[i-15] >> 3)
            s1 := bits.RotateLeft32(w[i-2], -17) ^ bits.RotateLeft32(w[i-
2], -19) ^ (w[i-2] >> 10)
            w[i] = s0 + s1 + w[i-16] + w[i-7]
        }
```

7. 初始化这次循环的哈希值

```
        a := h0
        b := h1
        c := h2
        d := h3
        e := h4
        f := h5
        g := h6
        h := h7
```

8. 开始进行计算的主循环

```
for i := 0; i < 64; i++ {
            s0 := bits.RotateLeft32(a, -2) ^ bits.RotateLeft32(a, -13) ^
bits.RotateLeft32(a, -22)
            maj := (a & b) ^ (a & c) ^ (b & c)
            t2 := s0 + maj
            s1 := bits.RotateLeft32(e, -6) ^ bits.RotateLeft32(e, -11) ^
bits.RotateLeft32(e, -25)
```

```
            ch := (e & f) ^ (^e & g)
            t1 := h + s1 + ch + k[i] + w[i]

            h = g
            g = f
            f = e
            e = d + t1
            d = c
            c = b
            b = a
            a = t1 + t2
        }
```

9. 把这段计算的哈希值加到h0-h7

```
        h0 = h0 + a
        h1 = h1 + b
        h2 = h2 + c
        h3 = h3 + d
        h4 = h4 + e
        h5 = h5 + f
        h6 = h6 + g
        h7 = h7 + h
```

10. 最后按大端系统序存放结果即可

```
hash := []uint32{h0, h1, h2, h3, h4, h5, h6, h7}
    result := [32]byte{}
    for i, u := range hash {
        binary.BigEndian.PutUint32(result[i*4:i*4+4], u)
    }
```

遇到的困难

1. 第一次用go语言，不太懂怎么用
2. 实验文档一开始看得不明白，后面才理解是在离散数学中考虑的
3. debug起来不知道怎么做，大数计算的验证难以手动验证。

总结

这次实验总的来说难度相对来说比较正常，可以让我们熟悉go语言和区块链中的签名和验签过程。