**ECDSA 在以太坊中的技术报告**

本报告主要探究了以太坊在使用ECDSA来进行数字签名的时候是如何通过签名值来获得验证签名所需要的公钥。

1. 以太坊中的椭圆曲线算法

secp256k1 是区块链项目中应用最多的椭圆曲线算法，源于比特币中的应用，后来的大多数区块链项目如以太坊等都在用。

这个曲线的参数定义如下：

|  |
| --- |
| p = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F  a=0000000000000000000000000000000000000000000000000000000000000000  b=0000000000000000000000000000000000000000000000000000000000000007  G=79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798  483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8  n=FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141  h = 01 |

以上参数是十六进制的表示形式。其中secp256k1曲线方程如下所示：

y^2≡x^3+ax+b (mod p)

其中p为2^256−2^32−2^9−2^8−2^7−2^6−2^4−1

以上给出的值是非压缩格式

1. 公钥恢复原理

这里是指仅通过签名结果来和消息摘要，来计算出签名私钥对应的公钥。

在椭圆曲线的签名&验证过程我们得到了如下的公式：

R=(HG−rQ)s^(−1)

由以上公式我们可以推出来

Q=(sR−HG)r^(−1)

其中Q是公钥。

到这一步，我们知道是有办法计算出公钥的。再看一下对于验证者已知变量s,r,G,H, 唯独R点不知道，即R点（x，y）坐标未知。

但是有一层关系：r是R点的x坐标。

我们想到能不能通过这种关系求出R点完整信息即解出y值？

答案是可行的。

利用曲线方程式:

y^2≡x^3+ax+b （mod p）

我们可以解出两个符合条件的, 几何上的解释是该曲线关于x轴对称。

与此同时，根据secp256k1曲线根据签名过程，计算点𝑘𝐺，其𝑋坐标（整数模𝑝，因此在0到𝑝−1范围内）被约化模𝑛，从而得到一个介于0和𝑛−1之间的值。结果是𝑟。由于𝑛略低于𝑝（由secp256k1曲线参数可得），因此可以有两个值𝑋与𝑟匹配。一般来说只有一个，但是如果 𝑟 < 𝑝-𝑛 ，那么就会有两个。（但是产生两个X的可能性非常的低，在实践中很难出现）

因此综上所述我们总共最多可能得到四个公钥。

由于已经知道总共可能得到四个公钥，在原始签名的过程中，加上额外处理：找到公钥恢复的过程对应的索引，记为recid，附加到签名中。

1. 具体实现

在此处我们查找到一个go实现的以太坊的签名过程，来进行说明以太坊中实现，以及调用了那些工具。

3.1 准备工作

|  |
| --- |
| package main  import (  "fmt"  "github.com/ethereum/go-ethereum/common/hexutil"  "github.com/ethereum/go-ethereum/core/types"  "github.com/ethereum/go-ethereum/rlp"  "math/big"  )  func main() {  // 还原交易对象  encodedTxStr := "0xf889188504a817c800832dc6c09405e56888360ae54acf2a389bab39bd41e3934d2b80a4ee919d50000000000000000000000000000000000000000000000000000000000000007b25a041c4a2eb073e6df89c3f467b3516e9c313590d8d57f7c217fe7e72a7b4a6b8eda05f20a758396a5e681ce1ab4cec749f8560e28c9eb91072ec7a8acc002a11bb1d"  encodedTx, err := hexutil.Decode(encodedTxStr)  if err != nil {  fmt.Println("hexutil.Decode failed: ", err.Error())  return  }  // rlp解码  tx := new(types.Transaction)  if err := rlp.DecodeBytes(encodedTx, tx); err != nil {  fmt.Println("rlp.DecodeBytes failed: ", err.Error())  return  }  // chainId为1的EIP155签名器  signer := types.NewEIP155Signer(big.NewInt(1))  // 使用签名器从已签名的交易中还原账户公钥  from, err := types.Sender(signer, tx)  if err != nil {  fmt.Println("types.Sender: ", err.Error())  return  }  fmt.Println("from: ", from.Hex())  jsonTx, \_ := tx.MarshalJSON()  fmt.Println("tx: ", string(jsonTx))  } |

encodedTxStr是上篇文章得到的具有签名的交易对象的rlp编码

最终还原得到的from值为0xA2088F51Ea1f9BA308F5014150961e5a6E0A4E13，正是签名私钥对应的账号地址（私钥单向生成公钥，公钥单向生成地址）

签名解析核心使用的是Sender方法

3.2 签名解析

types.Sender方法中核心调用了EIP155签名器的Sender方法，其源码如下。

|  |
| --- |
| // go-ethereum/core/types/transaction\_signing.go  func (s EIP155Signer) Sender(tx \*Transaction) (common.Address, error) {  if !tx.Protected() {//①  return HomesteadSigner{}.Sender(tx)  }  if tx.ChainId().Cmp(s.chainId) != 0 {//②  return common.Address{}, ErrInvalidChainId  }  //③  V := new(big.Int).Sub(tx.data.V, s.chainIdMul)  V.Sub(V, big8)  return recoverPlain(s.Hash(tx), tx.data.R, tx.data.S, V, true)  } |

Sender 方法中：

首先判断了交易是否是受保护的（是否是EIP155签名器进行的签名），如果不是，则使用HomesteadSigner签名器校验

接着判断了交易中的链ID与签名器的链ID是否一致，如果不一致则返回空地址

根据V的计算方法还原recid为27（37-1\*2-8），在recoverPlain方法会按照homestead签名方式继续解析签名。

recoverPlain源码如下所示。

|  |
| --- |
| // go-ethereum/core/types/transaction\_signing.go  func recoverPlain(sighash common.Hash, R, S, Vb \*big.Int, homestead bool) (common.Address, error) {  if Vb.BitLen() > 8 {  return common.Address{}, ErrInvalidSig  }  V := byte(Vb.Uint64() - 27)  if !crypto.ValidateSignatureValues(V, R, S, homestead) {  return common.Address{}, ErrInvalidSig  }  // encode the signature in uncompressed format  r, s := R.Bytes(), S.Bytes()  sig := make([]byte, crypto.SignatureLength)  copy(sig[32-len(r):32], r)  copy(sig[64-len(s):64], s)  sig[64] = V //①  fmt.Println("sig: ", common.Bytes2Hex(sig))  // recover the public key from the signature  pub, err := crypto.Ecrecover(sighash[:], sig) //②  if err != nil {  return common.Address{}, err  }  if len(pub) == 0 || pub[0] != 4 {  return common.Address{}, errors.New("invalid public key")  }  fmt.Println("pub: ", common.Bytes2Hex(pub))  var addr common.Address  copy(addr[:], crypto.Keccak256(pub[1:])[12:])//③  return addr, nil  } |

其中recoverPlain方法的参数分别为：

sighash是交易对象tx的rlp编码，hex值为：

0x9ef7f101dae55081553998d52d0ce57c4cf37271f800b70c0863c4a749977ef1，与我们上文中需要签名的交易hash是一致的。

R，hex值为：

41c4a2eb073e6df89c3f467b3516e9c313590d8d57f7c217fe7e72a7b4a6b8ed

S hex值为：

5f20a758396a5e681ce1ab4cec749f8560e28c9eb91072ec7a8acc002a11bb1d

Vb，十进制值为27

bool类型的homestead，值为tru

在recoverPlain方法中：

根据R、S、V拼接得到的sign，hex值为：41c4a2eb073e6df89c3f467b3516e9c313590d8d57f7c217fe7e72a7b4a6b8ed5f20a758396a5e681ce1ab4cec749f8560e28c9eb91072ec7a8acc002a11bb1d00

调用加密包中的Ecrecover方法根据签名还原公钥，该方法会调用secp256k1包中的RecoverPubkey方法。还原得到的公钥hex值为045762d11bad6617b5eef31fefd6aff1391dab0a2380817eaf882874b1d50823b13e4934f923f4b7e6a3d19219e92a04678a8fb7029c2ecf7256672b57a6cb77b0 。

根据公钥计算账号地址，取公钥pub第一位之后的值计算Keccak256，然后在取后12位以后，得到的账号地址为：0xA2088F51Ea1f9BA308F5014150961e5a6E0A4E13

至此，我们已经从签名中还原出了账号地址（公钥）。如果需要校验签名是否正确，可以通过调用secp256k1包中的VerifySignature方法，传入公钥、交易hash和签名，通过比对R值是否一致进行验证。

参考文献：

1. 以太坊交易签名解析源码解读 <https://learnblockchain.cn/article/1250>
2. 区块链中的数学-secp256k1公钥恢复实现

https://learnblockchain.cn/article/1517