Transaction effectiveness function and code description

The first version of usechain mainly implements the function of authentication: All addresses on the chain must be authenticated and audited by the third party organizations, and only the audited address can trade on the chain.

The work is divided into four parts:

Certificate server: Responsible for the verification of the third party certification signature and the issuance of user certificates.

Wallet client: Responsible for achieving user authentication, sending transactions, and inquiring the balance.

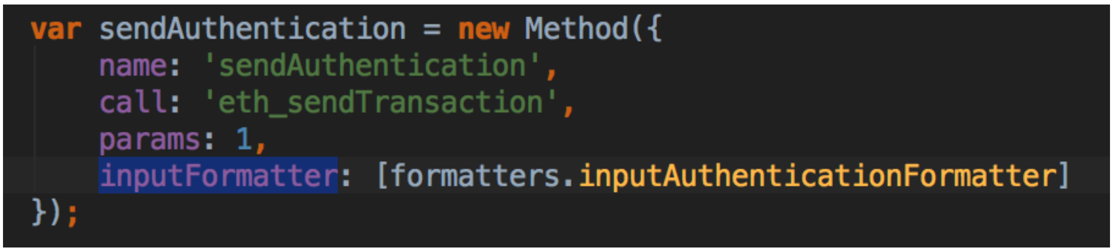
Certification contract: Responsible for recording the lawful address through the audit on the chain.

Transaction data layer: Responsible for the verification of the validity of certification transactions and the legality examination of ordinary transactions.

Catalog

1. Packaging of certified transactions

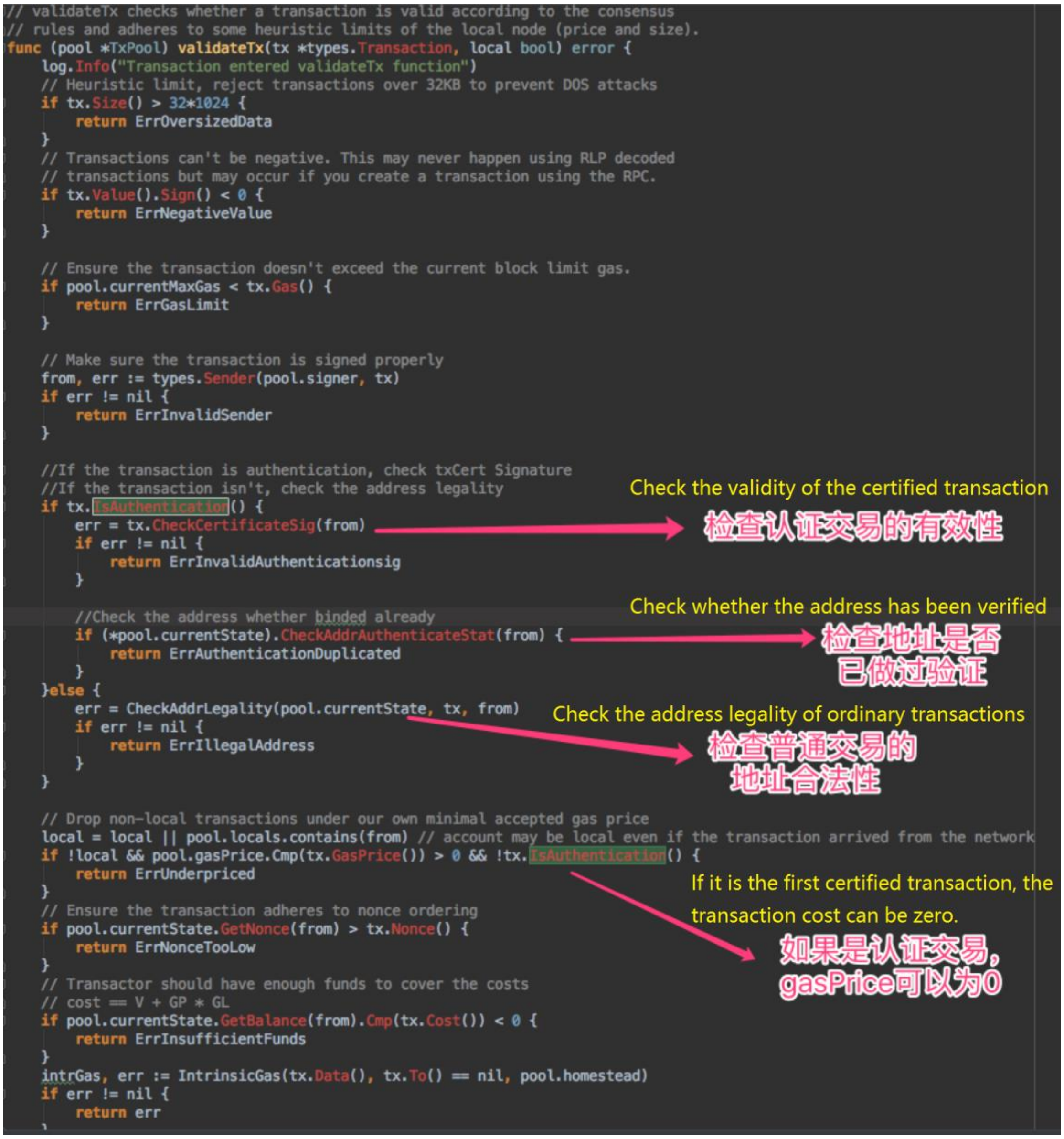
Add the new console command sendAuthentication () to authenticate the transaction.



```
var sendAuthentication = new Method({
    name: 'sendAuthentication',
    call: 'eth_sendTransaction',
    params: 1,
    inputFormatter: [formatters.inputAuthenticationFormatter]
});
```

2. Verification of the validity of certification transactions and the legality examination of ordinary transactions.

After receiving the transaction or packing the transaction, the transaction will enter the transaction buffer, and the program will check all transactions in the transaction buffer pool.(go-ethereum/core/tx-pool.go line591 validateTx())
In addition to the verification of information size, signature and other information in the Ethernet framework, Usechain has added the signature verification of identity authentication transactions and the verification of the address validity of ordinary transactions.
Taking into account that the new address does not own UST and is unable to pay transaction fees for the first authentication transaction, the gasPrice for setting up the first authentication transaction is 0.



```go
// validateTx checks whether a transaction is valid according to the consensus
// rules and adheres to some heuristic limits of the local node (price and size).
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    log.Info("Transaction entered validateTx function")
    // Heuristic limit, reject transactions over 32KB to prevent DOS attacks
    if tx.Size() > 32*1024 {
        return ErrOversizedData
    }
    // Transactions can't be negative. This may never happen using RLP decoded
    // transactions but may occur if you create a transaction using the RPC.
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }

    // Ensure the transaction doesn't exceed the current block limit gas.
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }

    // Make sure the transaction is signed properly
    from, err := types.Sender(pool.signer, tx)
    if err != nil {
        return ErrInvalidSender
    }

    //If the transaction is authentication, check txCert Signature
    //If the transaction isn't, check the address legality
    if tx.IsAuthentication() {
        err = tx.CheckCertificateSig(from)
        if err != nil {
            return ErrInvalidAuthenticationsig
        }

        //Check the address whether binded already
        if (*pool.currentState).CheckAddrAuthenticateStat(from) {
            return ErrAuthenticationDuplicated
        }
    }else {
        err = CheckAddrLegality(pool.currentState, tx, from)
        if err != nil {
            return ErrIllegalAddress
        }
    }

    // Drop non-local transactions under our own minimal accepted gas price
    local = local || pool.locals.contains(from) // account may be local even if the transaction arrived from the network
    if !local && pool.gasPrice.Cmp(tx.GasPrice()) > 0 && !tx.IsAuthentication() {
        return ErrUnderpriced
    }
    // Ensure the transaction adheres to nonce ordering
    if pool.currentState.GetNonce(from) > tx.Nonce() {
        return ErrNonceTooLow
    }
    // Transactor should have enough funds to cover the costs
    // cost == V + GP * GL
    if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
        return ErrInsufficientFunds
    }
    intrGas, err := IntrinsicGas(tx.Data(), tx.To() == nil, pool.homestead)
    if err != nil {
        return err
    }
```

Check the validity of the certified transaction

检查认证交易的有效性

Check whether the address has been verified

检查地址是否
已做过验证

Check the address legality of ordinary transactions

检查普通交易的
地址合法性

If it is the first certified transaction, the transaction cost can be zero.

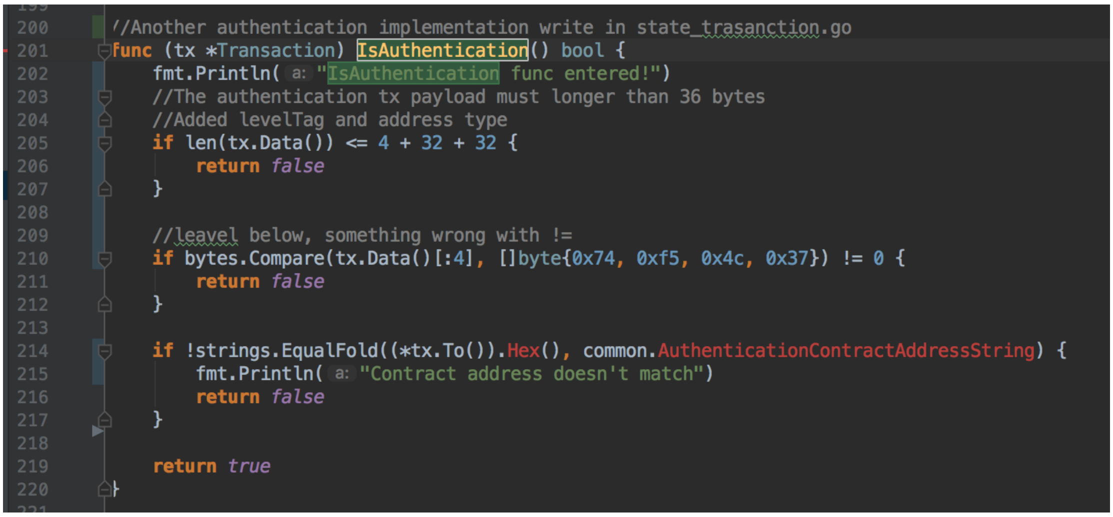
如果是认证交易，
gasPrice可以为0

(1)confirmation of the type of authentication transaction

Payload segment data structure of identity authentication:

0x74f54c37	level	addressType	Tcert
4 bytes	32 bytes	32 bytes	n*32 bytes

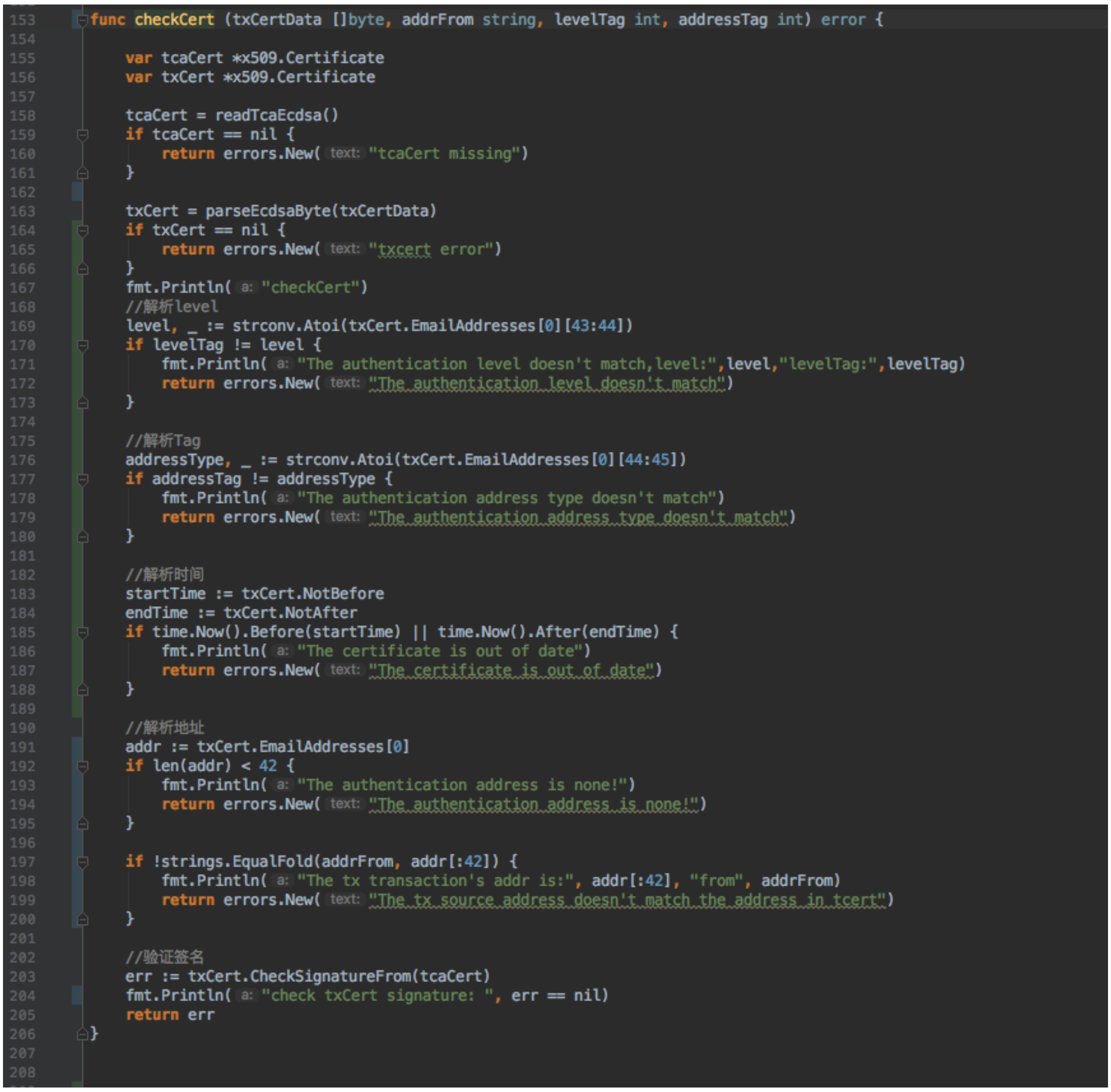
Level indicates the authentication level. AddressType indicates the address is the main one or the sub one. function hash is 0x74f54c37, if the address of <to ()> is a CA contract address, and the effective length of the transaction will be regarded as an authentication transaction, which will enter the follow-up operation such as Tcert check. If the check fails, the transaction will be abandoned; and the ordinary transaction will not carry out the authentication.



```
//Another authentication implementation write in state_trasanction.go
func (tx *Transaction) IsAuthentication() bool {
    fmt.Println( a: "IsAuthentication func entered!")
    //The authentication tx payload must longer than 36 bytes
    //Added levelTag and address type
    if len(tx.Data()) <= 4 + 32 + 32 {
        return false
    }

    //leavel below, something wrong with !=
    if bytes.Compare(tx.Data()[:4], []byte{0x74, 0xf5, 0x4c, 0x37}) != 0 {
        return false
    }

    if !strings.EqualFold((*tx.To()).Hex(), common.AuthenticationContractAddressString) {
        fmt.Println( a: "Contract address doesn't match")
        return false
    }

    return true
}
```

(2)validation of the validity of an authentication transaction

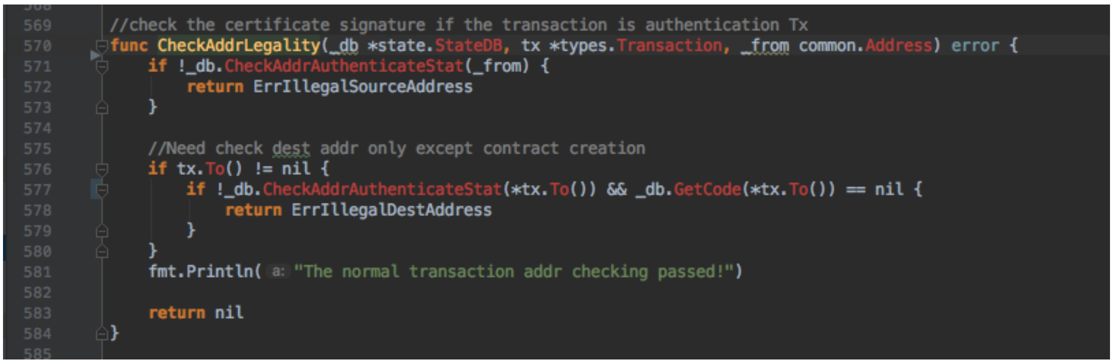
Once identified as an authentication transaction, the validity of the Tcert is verified, including whether level, Tag, address are consistent, the certificate is expired, and whether the certificate is issued by TCA.



```
func checkCert (txCertData []byte, addrFrom string, levelTag int, addressTag int) error {

    var tcaCert *x509.Certificate
    var txCert *x509.Certificate

    tcaCert = readTcaEcdsa()
    if tcaCert == nil {
        return errors.New( text: "tcaCert missing")
    }

    txCert = parseEcdsaByte(txCertData)
    if txCert == nil {
        return errors.New( text: "txcert error")
    }
    fmt.Println( a: "checkCert")
    //解析level
    level, _ := strconv.Atoi(txCert.EmailAddresses[0][43:44])
    if levelTag != level {
        fmt.Println( a: "The authentication level doesn't match,level:",level,"levelTag:",levelTag)
        return errors.New( text: "The authentication level doesn't match")
    }

    //解析Tag
    addressType, _ := strconv.Atoi(txCert.EmailAddresses[0][44:45])
    if addressTag != addressType {
        fmt.Println( a: "The authentication address type doesn't match")
        return errors.New( text: "The authentication address type doesn't match")
    }

    //解析时间
    startTime := txCert.NotBefore
    endTime := txCert.NotAfter
    if time.Now().Before(startTime) || time.Now().After(endTime) {
        fmt.Println( a: "The certificate is out of date")
        return errors.New( text: "The certificate is out of date")
    }

    //解析地址
    addr := txCert.EmailAddresses[0]
    if len(addr) < 42 {
        fmt.Println( a: "The authentication address is none!")
        return errors.New( text: "The authentication address is none!")
    }

    if !strings.EqualFold(addrFrom, addr[:42]) {
        fmt.Println( a: "The tx transaction's addr is:", addr[:42], "from", addrFrom)
        return errors.New( text: "The tx source address doesn't match the address in tcert")
    }

    //验证签名
    err := txCert.CheckSignatureFrom(tcaCert)
    fmt.Println( a: "check txCert signature: ", err == nil)
    return err
}
```

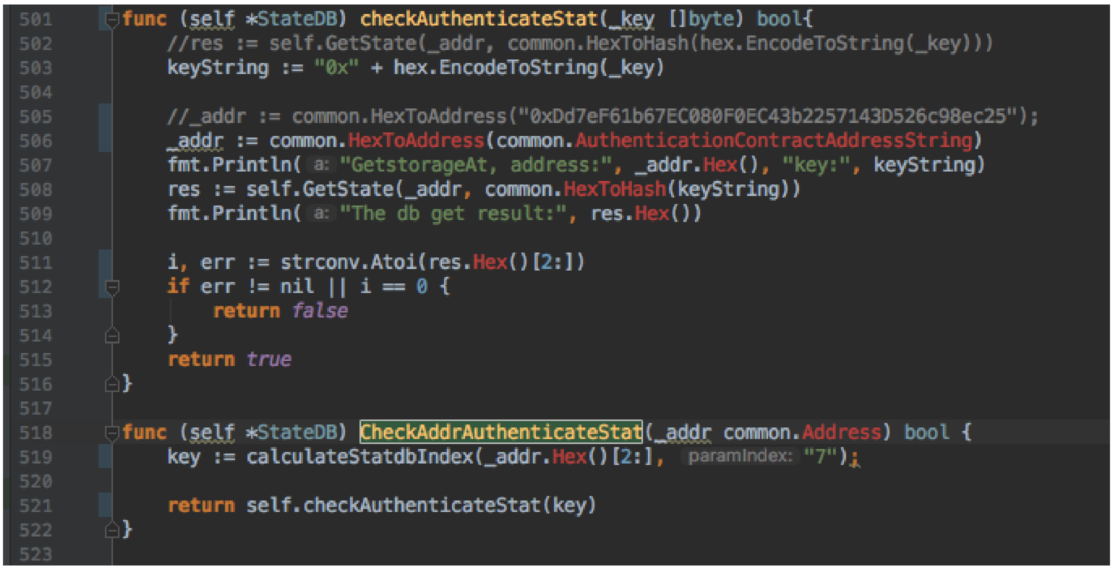
(3) audit of the legitimacy of the ordinary transaction address

Add the address validity check in validateTx (), check the from address; if <to ()> address is an ordinary address, check the validity. The address of <To ()> is empty or the contract address is not filtered.



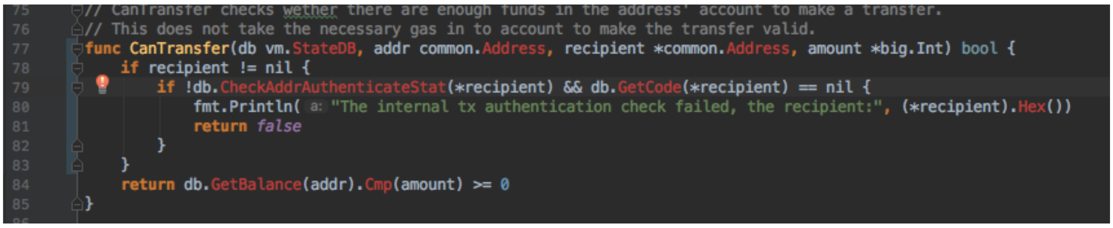
```
//check the certificate signature if the transaction is authentication Tx
func CheckAddrLegality(_db *state.StateDB, tx *types.Transaction, _from common.Address) error {
    if !_db.CheckAddrAuthenticateStat(_from) {
        return ErrIllegalSourceAddress
    }

    //Need check dest addr only except contract creation
    if tx.To() != nil {
        if !_db.CheckAddrAuthenticateStat(*tx.To()) && _db.GetCode(*tx.To()) == nil {
            return ErrIllegalDestAddress
        }
    }
    fmt.Println( a: "The normal transaction addr checking passed!")

    return nil
}
```

The validity of checking the address is whether the address has been recorded as a legitimate address by the CA contract by accessing the statDB of the CA contract.



```
func (self *StateDB) checkAuthenticateStat(_key []byte) bool{
    //res := self.GetState(_addr, common.HexToHash(hex.EncodeToString(_key)))
    keyString := "0x" + hex.EncodeToString(_key)

    //_addr := common.HexToAddress("0xDd7eF61b67EC080F0EC43b2257143D526c98ec25");
    _addr := common.HexToAddress(common.AuthenticationContractAddressString)
    fmt.Println( a: "GetstorageAt, address:", _addr.Hex(), "key:", keyString)
    res := self.GetState(_addr, common.HexToHash(keyString))
    fmt.Println( a: "The db get result:", res.Hex())

    i, err := strconv.Atoi(res.Hex()[2:])
    if err != nil || i == 0 {
        return false
    }
    return true
}

func (self *StateDB) CheckAddrAuthenticateStat(_addr common.Address) bool {
    key := calculateStatdbIndex(_addr.Hex()[2:],  paramIndex: "7");

    return self.checkAuthenticateStat(key)
}
```

It is also necessary to pay attention to the modification of the internal transaction. When the contract initiates a internal transaction, it is necessary to ensure that the <to ()> address is not an unlawful ordinary address (which can be empty, the contract address or the legitimate address that has been verified). Where the modification is placed on the <CanTransfer ()> function of the check function of EVM executing transfer, in addition to checking sufficient conditions for the balance, add the check of the legitimacy of the address.

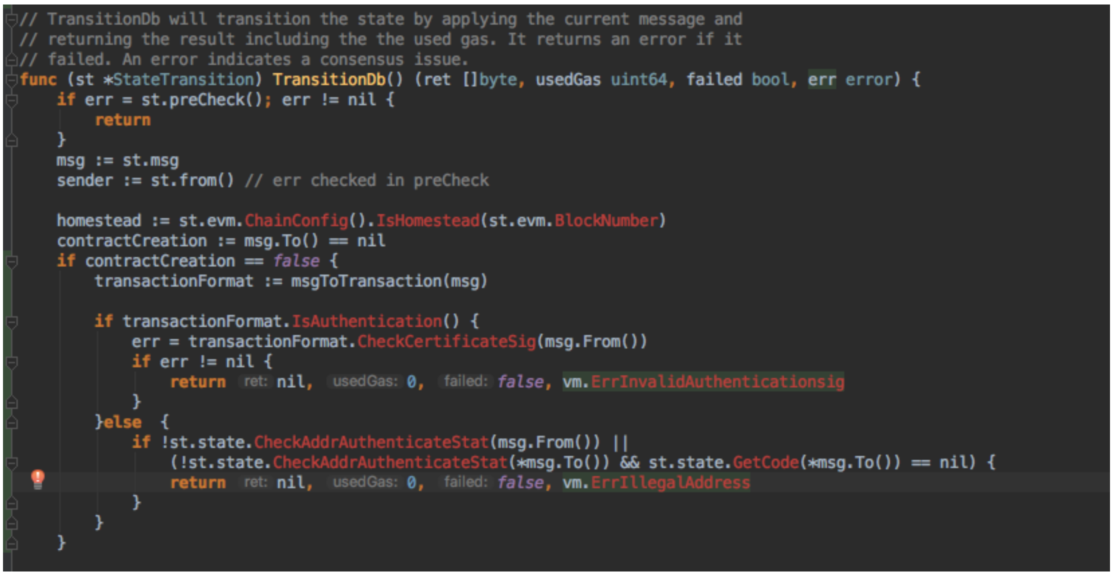


```
// CanTransfer checks wether there are enough funds in the address' account to make a transfer.
// This does not take the necessary gas in to account to make the transfer valid.
func CanTransfer(db vm.StateDB, addr common.Address, recipient *common.Address, amount *big.Int) bool {
    if recipient != nil {
        if !db.CheckAddrAuthenticateStat(*recipient) && db.GetCode(*recipient) == nil {
            fmt.Println( a: "The internal tx authentication check failed, the recipient:", (*recipient).Hex())
            return false
        }
    }
    return db.GetBalance(addr).Cmp(amount) >= 0
}
```

(4) block filtering

In addition, it should be noted that checking in the transaction buffer pool can prevent normal nodes from abandoning illegal transactions, but if malicious nodes package invalid transactions into blocks and then broadcast, the normal node should be able to recognize that the block is invalid and cannot be incorporated into the chain.

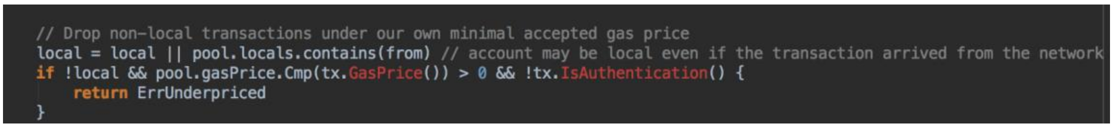
So when the new block is received and the status detection is carried out, the monitoring of the identity authentication transaction and the monitoring of the legitimacy of the ordinary transaction address are added at the TransactionDB function.



```
// TransitionDb will transition the state by applying the current message and
// returning the result including the the used gas. It returns an error if it
// failed. An error indicates a consensus issue.
func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64, failed bool, err error) {
    if err = st.preCheck(); err != nil {
        return
    }
    msg := st.msg
    sender := st.from() // err checked in preCheck

    homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
    contractCreation := msg.To() == nil
    if contractCreation == false {
        transactionFormat := msgToTransaction(msg)

        if transactionFormat.IsAuthentication() {
            err = transactionFormat.CheckCertificateSig(msg.From())
            if err != nil {
                return  ret: nil,  usedGas: 0,  failed: false, vm.ErrInvalidAuthenticationsig
            }
        }else  {
            if !st.state.CheckAddrAuthenticateStat(msg.From()) ||
                (!st.state.CheckAddrAuthenticateStat(*msg.To()) && st.state.GetCode(*msg.To()) == nil) {
                return  ret: nil,  usedGas: 0,  failed: false, vm.ErrIllegalAddress
            }
        }
    }
```

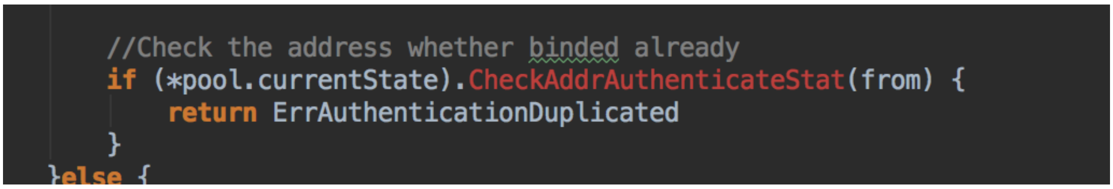
(5) gas consumption removal for certified transactions

The balance of the user's new address's UST is 0, which is unable to pay the miners' fees, so the gasPrice of the revised authentication transaction is 0.



```
    // Drop non-local transactions under our own minimal accepted gas price
    local = local || pool.locals.contains(from) // account may be local even if the transaction arrived from the network
    if !local && pool.gasPrice.Cmp(tx.GasPrice()) > 0 && !tx.IsAuthentication() {
        return ErrUnderpriced
    }
```

At the same time, in order to avoid malicious attacks on the network by sending a large number of authentication transactions, when the identity authentication transaction, if the binding address has been verified successfully, is not allowed to bind here (to be debatable and perfect, because the identity authentication is now hierarchical, and the personal authentication can also be allowed to continue to study after the authentication)



```
        //Check the address whether binded already
        if (*pool.currentState).CheckAddrAuthenticateStat(from) {
            return ErrAuthenticationDuplicated
        }
    }else {
```

In addition, the certificate server also restricts the maximum number of tcert issued by the same ecert (currently 1000).