成都链安
BEOSIN

WanChain

内置合约安全审计报告

审计编号：201907221404

审计合约名称：

序号	合约文件名
1	go-wanchain/core/vm/pos_control_contracts.go
2	go-wanchain/core/vm/pos_staking_contracts.go
3	go-wanchain/core/vm/random_beacon_contract.go
4	go-wanchain/core/vm/slot_leader_select_contracts.go

审计合约链接地址：https://github.com/wanchain/go-wanchain/

commit id：c088c6626e34b86b699263b11615612972111aa2

审计开始日期：2019.07.12

审计完成日期：2019.07.22

审计结果：通过（优）

审计团队：成都链安科技有限公司

审计类型及结果：

序号	审计项	审计结果
1	溢出审计	通过
2	随机数安全审计	通过
3	编程语言特性审计	通过
4	全局变量一致性审计	通过
5	异常处理审计	通过
6	冗余代码审计	通过
7	输入参数检查	通过
8	合约函数调用权限检查	通过
9	业务逻辑审计	通过

备注：审计意见及建议请见代码注释。

审计结果说明：

本公司采用静态分析、动态分析、典型案例测试和人工审核的方式对WanChain内置的四个合约：pos_control_contracts.go、pos_staking_contracts.go、slot_leader_select_contracts.go、random_beacon_contract.go的代码规范性、安全性以及业务逻辑三个方面进行多维度全面的安全审计。**经审计，pos_control_contracts.go、pos_staking_contracts.go、slot_leader_select_contracts.go、random_beacon_contract.go合约通过所有检测项，审计结果为通过(优)。**

审计结果分析

1、溢出审计

整型溢出是很多语言都存在的安全问题，它们在WanChain的内置智能合约中尤其危险。推荐使用math/big下的bigInt以及bigFloat，并对传入算术运算的参数进行必要的检查，以防止运算过程中出现溢出而产生非预期的执行逻辑。

- **安全建议：无**
- **审计结果：通过**

2、随机数安全审计

本次审计的内置合约中，random_beacon_contract.go这个合约是用来生成共识协议选举中关键的随机数因子。所以，该随机数生成是否满足去中心化、不可预测、无偏性、均匀分布和公开可验证性等特点就显得十分重要。

该随机数算法通过DKG1、DKG2和SIGN这三个阶段来产生随机数。在DKG1阶段，所有参与者对在DKG2要提交的数据做出承诺；在DKG2阶段，所有参与者协同计算得到组公钥（group public key）和组私钥份额（group secret key share）；在SIGN阶段，所有参与者使用其组私钥份额计算组签名份额（group signature share）；最终，通过组签名份额合成组签名，并计算得到random beacon的输出。整个流程如图 1：

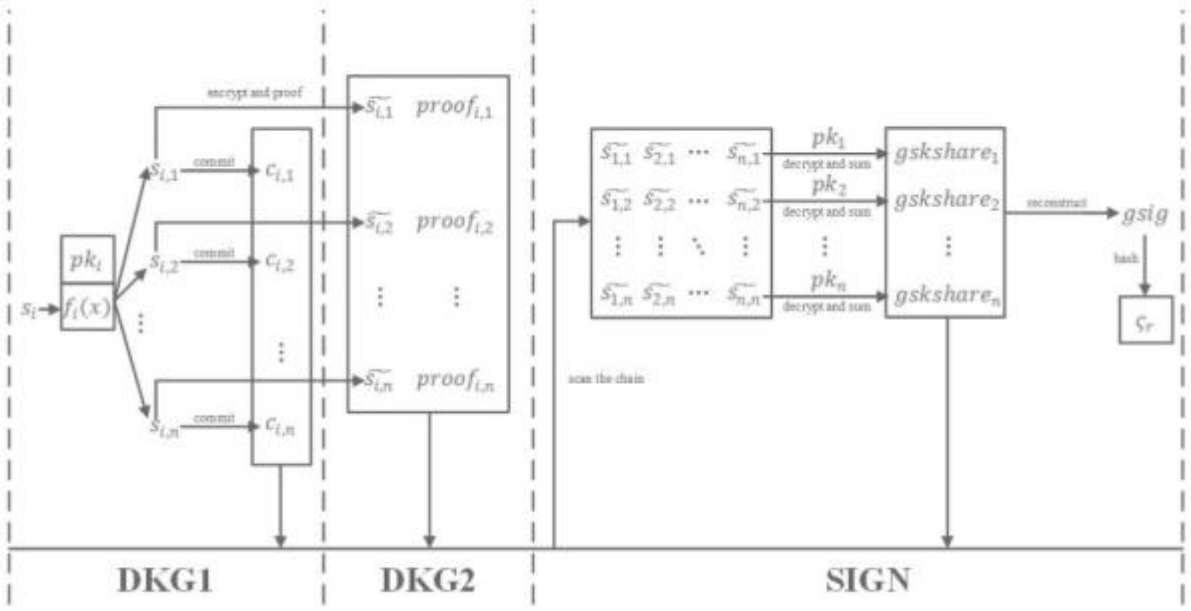

图 1 随机数生成步骤

- **安全建议：无**
- **审计结果：通过**

3、编程语言特性审计

Wangchain内置合约使用go语言进行开发，编码时需要考虑编程语言本身的特性对合约执行产生影响。比如多线程、map随机遍历等操作导致结果的不确定性。

- **安全建议：无**
- **审计结果：通过**

4、全局变量一致性审计

全局变量不会保存在数据库中，因此仅对单个节点是全局变量。如果此类节点发生故障或正在重新初始化，则全局状态变量可能不再在所有对等体上保持一致。因此，从数据库读取、写入不应依赖于全局状态变量。

- **安全建议：无**
- **审计结果：通过**

5、异常处理审计

忽略的错误可能导致错误执行或停止执行。因此，应该注意对Error进行的处理。在本次审计中，利用静态扫描分析工具发现以下9处未进行异常处理，如表 1：

文件	相关代码
pos_staking_contracts.go	p.stakeUpdateLog(contract, evm, stakerInfo)
	p.stakeAppendLog(contract, evm, stakerInfo.Address)
	p.stakeInLog(contract, evm, stakerInfo)
	p.delegateInLog(contract, evm, stakerInfo.Address)
	p.delegatOutLog(contract, evm, stakerInfo.Address)
	p.delegatOutLog(contract, evm, stakerInfo.Address)
	StoreInfo(stateDb,StakingCommonAddr,StakersInfoStakeOutKeyHash, b.Bytes())
slot_leader_select_contracts.go	addSlotScCallTimes(convert.BytesToUint64(epochIDBuf))
	posdb.GetDb().Put(epochID,scCallTimes,convert.Uint64ToBytes(times))

表 1

- **安全建议：规范写法**
- **审计结果：通过**

6、冗余代码审计

智能合约中应删除永远不会执行的代码，避免增加代码复杂度。

- **安全建议：无**
- **审计结果：通过**

7、输入参数检查

因为用户的所有输入都是不可信的，所以在使用用户输入的参数之前，应当对用户输入数据的类型、大小、有效性等做足够的检查，以避免程序出现不可预料的错误。

- **安全建议：无**
- **审计结果：通过**

8、合约函数调用权限检查

合约的调用应当执行相应的权限检查，防止越权调用造成不必要的损失。

- **安全建议：无**
- **审计结果：通过**

9、业务逻辑审计

（1）pos_control_contracts.go

- **基本信息：**

ABI 接口函数	upgradeWhiteEpochLeader(EpochId,wlIndex,wlCount)
合约地址	0x0000000000000000000000000000000000000264
调用权限	contract.Caller()==posconfig.PosOwnerAddr
合约功能	更新 EpochLeader

表 2

- **功能描述**：该合约通过暴露upgradeWhiteEpochLeader()函数接口，提供更新EpochLeader的功能。在执行业务功能之前首先对调用权限进行验证，如果不在授权范围之内，就直接退出函数。接着进行用户输入函数参数的有效性检查。最后再执行业务逻辑修改数据库状态。
- **安全建议：无**
- **审计结果：通过**

（2）pos_staking_contracts.go

- **基本信息：**

ABI 接口函数	stakeIn(secPk,bn256Pk,lockEpochs,feerRate)
	stakeAppend(addr)
	stakeUpdate(addr,lockEpochs)
	stakeUpdateFeeRate(addr,feeRate)
	partnerIn(addr,renewal)
	delegateIn(delegateAddress)
	delegateOut(delegateAddress)
ABI 接口事件	stakeIn
	stakeAppend
	stakeUpdate
	stakeUpdateFeeRate
	partnerIn
	delegateIn

	delegateOut
合约地址	0x00000000000000000000000000000000000000DA
调用权限	无
合约功能	执行抵押和代理抵押相关操作

表 3

- **功能描述**：该合约对外提供7个函数接口，主要用于执行抵押成为验证节点、代理抵押获取收益等相关操作。合约以Run()函数为入口，首先进行参数长度判断。然后根据参数选择调用不同的函数，进行不同的操作。进入特定的函数之后首先会对输入参数有效性、抵押条件等进行检查。当满足所有条件后才会执行数据库的更改。
- **安全建议：无**
- **审计结果：通过**

（3）random_beacon_contract.go

- **基本信息：**

ABI 接口函数	dkg1(info)
	dkg2(info)
	sigShare(info)
合约地址	0x0000000000000000000000000000000000000262
调用权限	无
合约功能	生成随机数

表 4

- **功能描述：**该合约对外提供三个函数，主要是用于产生一种随机数，该随机数用于选举Epoch Leader和Solt Leader。而这三个函数分别对应生成该随机数的三个阶段。在DKG1阶段，所有参与者对在DKG2要提交的数据做出承诺；在DKG2阶段，所有参与者协同计算得到组公钥（group public key）和组私钥份额（group secret key share）；在SIGN阶段，所有参与者使用其组私钥份额计算组签名份额（group signature share）；最终，通过组签名份额合成组签名，并计算得到random beacon的输出。
- **安全建议：无**
- **审计结果：通过**

（4）slot_leader_select_contracts.go

- **合约基本信息：**

ABI 接口函数	slotLeaderStage1MiSave(data)
	slotLeaderStage2InfoSave(data)
合约地址	0x0000000000000000000000000000000000000258
调用权限	无
合约功能	Slot leader 的选举

表 5

- **功能描述**：该合约对外提供两个函数，执行slot leader的选择过程中的相关操作。
- **安全建议：无**
- **审计结果：通过**

合约源代码审计注释

1、 pos_control_contracts.go

```
// 成都链安 // file:go-wanchain/core/vm/pos_control_contracts.go
package vm

import (
    "errors"
    "math/big"
    "sort"
    "strings"
    "time"

    "github.com/wanchain/go-wanchain/common"
    "github.com/wanchain/go-wanchain/pos/posconfig"

    "github.com/wanchain/go-wanchain/accounts/abi"
```

```
    "github.com/wanchain/go-wanchain/core/types"
    "github.com/wanchain/go-wanchain/pos/util"
    "github.com/wanchain/go-wanchain/rlp"
)

/* the contract interface described by solidity.

pragma solidity ^0.5.1;

contract posControl {
    function upgradeWhiteEpochLeader(uint256 EpochId, uint256 wlIndex, uint256
wlCount ) public {}
}
*/
// 成都链安 // ABI 接口说明
var (
    posControlDefinition = `
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "EpochId",
                "type": "uint256"
            },
            {
                "name": "wlIndex",
                "type": "uint256"
```

```
            },
            {
                "name": "wlCount",
                "type": "uint256"
            }
        ],
        "name": "upgradeWhiteEpochLeader",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
`

    posControlAbi abi.ABI
    upgradeWhiteEpochLeaderId [4]byte

    maxWlindex = len(posconfig.WhiteList)
    minWlIndex = 0
)

// 成都链安 // 参数定义
type UpgradeWhiteEpochLeaderParam struct {
    EpochId *big.Int
    WlIndex *big.Int
    WlCount *big.Int
}
```

```
// 成都链安 // 默认参数设置,默认的 EpochLeader 为 26 个
var UpgradeWhiteEpochLeaderDefault = UpgradeWhiteEpochLeaderParam{
    EpochId: big.NewInt(0),
    WlIndex: big.NewInt(0),
    WlCount: big.NewInt(26),
}

//
//  package initialize
// 成都链安 // 包初始化，所以这个合约是更新 EPoch 参数用的，更新的是 EpochId,索引,和总数
//
func init() {
    // 成都链安 // 初始化 ABi 接口
    posControlAbi, errCscInit = abi.JSON(strings.NewReader(posControlDefinition))
    if errCscInit != nil {
        panic("err in posControl abi initialize ")
    }
    copy(upgradeWhiteEpochLeaderId[:],
posControlAbi.Methods["upgradeWhiteEpochLeader"].Id())
}

type PosControl struct {
}

//
//  contract interfaces
```

```
// 成都链安 //  合约接口
//
func (p *PosControl) RequiredGas(input []byte) uint64 {
    return 0
}

// 成都链安 // 合约执行方法入口函数

func (p *PosControl) Run(input []byte, contract *Contract, evm *EVM) ([]byte, error)
{
    // 成都链安 //  判断参数个数
    if len(input) < 4 {
        return nil, errors.New("parameter is wrong")
    }

    // 成都链安 //  check only the owner could run it.
    // 成都链安 //  只有 owner 可以调用，权限判断.
    if contract.Caller() != posconfig.PosOwnerAddr {
        return nil, errParameters
    }

    var methodId [4]byte
    copy(methodId[:], input[:4])

    // 成都链安 // 取参数的前 4 字节判断 作为调用函数的 id
    if methodId == upgradeWhiteEpochLeaderId {
        // 成都链安 // 参数解释与验证
        info, err := p.upgradeWhiteEpochLeaderParseAndValid(input[4:],
```

```
evm.Time.Uint64())
        if err != nil {
            return nil, err
        }
        // 成都链安 // 把解释好的参数信息交给函数去执行
        return p.upgradeWhiteEpochLeader(info, contract, evm)
    }

    return nil, errMethodId
}

// 成都链安 // 验证交易
func (p *PosControl) ValidTx(stateDB StateDB, signer types.Signer, tx
*types.Transaction) error {
    input := tx.Data()
    if len(input) < 4 {
        return errors.New("parameter is too short")
    }

    var methodId [4]byte
    copy(methodId[:], input[:4])

    if methodId == upgradeWhiteEpochLeaderId {
        _, err := p.upgradeWhiteEpochLeaderParseAndValid(input[4:],
uint64(time.Now().Unix()))
        if err != nil {
            return errors.New("upgradeWhiteEpochLeaderParseAndValid verify failed")
        }
        return nil
```

```
    }
    return errParameters
}

// 成都链安 // 检查 epoch
func posControlCheckEpoch(epochId uint64, time uint64) bool {
    eid, _ := util.CalEpochSlotID(time)
    if eid+posconfig.PosUpgradeEpochID > epochId { //  must send tx some epochs in advance.
        return false
    }
    return true
}

// 成都链安 // 执行合约存储
// 成都链安 // 每个 Epoch 内，社区 Community 中将选出两组成员，即 Epoch Leader 组和 Random Number Proposer 组，Epoch Leader 组负责产生区块，
// 成都链安 // Random Number Proposer 组负责生成随机数以供协议进行随机选择时使用。两组成员基于其在 Community 内部持有的权益权重进行随机选择，一旦选定，在一个完整 epoch 内，成员保持不变。

// 成都链安 // 更新 epochLeader
func (p *PosControl) upgradeWhiteEpochLeader(info *UpgradeWhiteEpochLeaderParam, contract *Contract, evm *EVM) ([]byte, error) {
    // 成都链安 // 先进行编码
    infoBytes, err := rlp.EncodeToBytes(info)
    if err != nil {
        return nil, err
```

```
    }

    // 成都链安 // 数据存储

    res := StoreInfo(evm.StateDB, PosControlPrecompileAddr,
common.BigToHash(info.EpochId), infoBytes)

    if res != nil {

        return nil, res

    }


    return nil, nil

}


// 成都链安 // 参数解释和验证

func (p *PosControl) upgradeWhiteEpochLeaderParseAndValid(payload []byte, time
uint64) (*UpgradeWhiteEpochLeaderParam, error) {

    var info UpgradeWhiteEpochLeaderParam

    // 成都链安 // unpackInput 解包

    err := posControlAbi.UnpackInput(&info, "upgradeWhiteEpochLeader", payload)

    if err != nil {

        return nil, err

    }


    //  check epoch valid

    // 成都链安 // 检查 epoch 的有效性

    // 成都链安 // 转换数值，防止溢出

    wlIndex := info.WlIndex.Uint64()

    wlCount := info.WlCount.Uint64()

    if wlIndex+wlCount >= uint64(len(posconfig.WhiteList)) {

        return nil, errors.New("wlIndex out of range")
```

```
    }
    if wlCount < posconfig.MinEpHold || wlCount > posconfig.MaxEpHold {
        return nil, errors.New("wlCount out of range")
    }
    if !posControlCheckEpoch(info.EpochId.Uint64(), time) {
        return nil, errors.New("wrong epoch for upgradeWhiteEpochLeader")
    }
    return &info, nil
}

type WhiteInfos []UpgradeWhiteEpochLeaderParam

// 成都链安 // 求 info 的长度
func (s WhiteInfos) Len() int {
    return len(s)
}

// 成都链安 // 比较大小
func (s WhiteInfos) Less(i, j int) bool {
    return s[i].EpochId.Cmp(s[j].EpochId) < 0
}

// 成都链安 //交换数据
func (s WhiteInfos) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// 成都链安 //获取当前信息
```

```
func GetWlConfig(stateDb StateDB) WhiteInfos {

    infos := make(WhiteInfos, 0)

    infos = append(infos, UpgradeWhiteEpochLeaderDefault)

    stateDb.ForEachStorageByteArray(PosControlPrecompileAddr, func(key common.Hash,
value []byte) bool {

        info := UpgradeWhiteEpochLeaderParam{}

        err := rlp.DecodeBytes(value, &info)

        if err == nil {

            infos = append(infos, info)

        }

        return true

    })

    //  sort

    sort.Stable(infos)

    return infos

}

func GetEpochWLInfo(stateDb StateDB, epochId uint64) *UpgradeWhiteEpochLeaderParam {

    infos := GetWlConfig(stateDb)

    index := len(infos) - 1

    for i := 0; i < len(infos); i++ {

        if infos[i].EpochId.Cmp(big.NewInt(int64(epochId))) == 0 {

            index = i

            break

        } else if infos[i].EpochId.Cmp(big.NewInt(int64(epochId))) > 0 {

            index = i - 1

            break

        }

    }
```

```
    return &infos[index]
}
```

2、 pos_control_contracts.go

```
// 成都链安 // file:go-wanchain/core/vm/pos_control_contracts.go
package vm

import (
    "crypto/ecdsa"
    "errors" // this is not match with other
    "github.com/wanchain/go-wanchain/params"
    "math/big"
    "strings"

    "github.com/wanchain/go-wanchain/pos/posconfig"

    "github.com/wanchain/go-wanchain/accounts/abi"
    "github.com/wanchain/go-wanchain/common"
    "github.com/wanchain/go-wanchain/core/state"
    "github.com/wanchain/go-wanchain/core/types"
    "github.com/wanchain/go-wanchain/crypto"
    "github.com/wanchain/go-wanchain/crypto/bn256"
    "github.com/wanchain/go-wanchain/log"
    "github.com/wanchain/go-wanchain/pos/util"
    "github.com/wanchain/go-wanchain/rlp"
)
```

```
/* the contract interface described by solidity.
```

// 成都链安 // stake 抵押的 solidity 定义

```
contract stake {

    function stakeIn(bytes memory secPk, bytes memory bn256Pk, uint256 lockEpochs,
uint256 feeRate) public payable {}

    function stakeUpdate(address addr, uint256 lockEpochs) public {}

    function stakeUpdateFeeRate(address addr, uint256 feeRate) public {}

    function stakeAppend(address addr) public payable {}

    function partnerIn(address addr, bool renewal) public payable {}

    function delegateIn(address delegateAddress) public payable {}

    function delegateOut(address delegateAddress) public {}


    event stakeIn(address indexed sender, address indexed posAddress, uint indexed
value, uint256 feeRate, uint256 lockEpoch);

    event stakeAppend(address indexed sender, address indexed posAddress, uint
indexed value);

    event stakeUpdate(address indexed sender, address indexed posAddress, uint
indexed lockEpoch);

    event delegateIn(address indexed sender, address indexed posAddress, uint
indexed value);

    event delegateOut(address indexed sender, address indexed posAddress);

    event stakeUpdateFeeRate(address indexed sender, address indexed posAddress,
uint indexed feeRate);

    event partnerIn(address indexed sender, address indexed posAddress, uint indexed
value, bool renewal);

}


*/
```

// 成都链安 // 抵押过程中用到的常数的定义

```
const (

    PSMinEpochNum = 7
```

```
    PSMaxEpochNum = 90


    PSMaxStake = 10500000
    PSMinStakeholderStake = 10000
    PSMinValidatorStake = 50000
    PSMinDelegatorStake = 100
    PSMinFeeRate = 0
    PSMaxFeeRate = 10000
    PSFeeRateStep        = 100
    PSNodeleFeeRate = 10000
    PSMinPartnerIn = 10000
    MaxTimeDelegate = 10
    UpdateDelay = 3
    QuitDelay = 3
    JoinDelay = 2
    PSOutKeyHash = 700
    maxPartners = 5
)

var (
    // pos staking contract abi definition
// 成都链安 // pos 抵押合约 abi 定义
    cscDefinition = `
[
    {
        "constant": false,
        "inputs": [
            {
```

```
            "name": "addr",
            "type": "address"
        }
    ],
    "name": "stakeAppend",
    "outputs": [],
    "payable": true,
    "stateMutability": "payable",
    "type": "function"
},
{
    "constant": false,
    "inputs": [
        {
            "name": "addr",
            "type": "address"
        },
        {
            "name": "lockEpochs",
            "type": "uint256"
        }
    ],
    "name": "stakeUpdate",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
},
```

```
    {
        "constant": false,
        "inputs": [
            {
                "name": "secPk",
                "type": "bytes"
            },
            {
                "name": "bn256Pk",
                "type": "bytes"
            },
            {
                "name": "lockEpochs",
                "type": "uint256"
            },
            {
                "name": "feeRate",
                "type": "uint256"
            }
        ],
        "name": "stakeIn",
        "outputs": [],
        "payable": true,
        "stateMutability": "payable",
        "type": "function"
    },
    {
        "constant": false,
```

```
        "inputs": [
            {
                "name": "addr",
                "type": "address"
            },
            {
                "name": "renewal",
                "type": "bool"
            }
        ],
        "name": "partnerIn",
        "outputs": [],
        "payable": true,
        "stateMutability": "payable",
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
                "name": "delegateAddress",
                "type": "address"
            }
        ],
        "name": "delegateIn",
        "outputs": [],
        "payable": true,
        "stateMutability": "payable",
```

```
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
                "name": "delegateAddress",
                "type": "address"
            }
        ],
        "name": "delegateOut",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
                "name": "addr",
                "type": "address"
            },
            {
                "name": "feeRate",
                "type": "uint256"
            }
        ],
```

```
        "name": "stakeUpdateFeeRate",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },

    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "v",
                "type": "uint256"
            },
            {
                "indexed": false,
                "name": "feeRate",
```

```
                "type": "uint256"
            },
            {
                "indexed": false,
                "name": "lockEpoch",
                "type": "uint256"
            }
        ],
        "name": "stakeIn",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "v",
                "type": "uint256"
```

```
            }
        ],
        "name": "stakeAppend",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "lockEpoch",
                "type": "uint256"
            }
        ],
        "name": "stakeUpdate",
        "type": "event"
    },
    {
```

```
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "v",
                "type": "uint256"
            },
            {
                "indexed": false,
                "name": "renewal",
                "type": "bool"
            }
        ],
        "name": "partnerIn",
        "type": "event"
    },
    {
        "anonymous": false,
```

```
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "v",
                "type": "uint256"
            }
        ],
        "name": "delegateIn",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
```

```
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            }
        ],
        "name": "delegateOut",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "name": "sender",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "posAddress",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "feeRate",
                "type": "uint256"
            }
        ],
```

```
        "name": "stakeUpdateFeeRate",
        "type": "event"
    }
]
`

    // pos staking contract abi object
// 成都链安 // abi 对象
    cscAbi, errCscInit = abi.JSON(strings.NewReader(cscDefinition))

    // function "stakeIn" "delegateIn" 's solidity binary id
    stakeInId [4]byte
    stakeUpdateId [4]byte
    stakeAppendId [4]byte
    partnerInId [4]byte
    delegateInId [4]byte
    delegateOutId [4]byte
    stakeUpdateFeeRateId [4]byte

    maxEpochNum = big.NewInt(PSMaxEpochNum)
    minEpochNum = big.NewInt(PSMinEpochNum)
    minStakeholderStake = new(big.Int).Mul(big.NewInt(PSMinStakeholderStake), ether)
    MinValidatorStake = new(big.Int).Mul(big.NewInt(PSMinValidatorStake), ether)
    minDelegatorStake = new(big.Int).Mul(big.NewInt(PSMinDelegatorStake), ether)
    maxTotalStake = new(big.Int).Mul(big.NewInt(PSMaxStake), ether)
    minFeeRate = big.NewInt(PSMinFeeRate)
    maxFeeRate = big.NewInt(PSMaxFeeRate)
    minPartnerIn = new(big.Int).Mul(big.NewInt(PSMinPartnerIn), ether)
    noDelegateFeeRate = big.NewInt(PSNodeleFeeRate)
```

```
    StakersInfoStakeOutKeyHash =
common.BytesToHash(big.NewInt(PSOutKeyHash).Bytes())
)

//
// param structures
// 成都链安//定义不同函数的参数结构
//
type StakeInParam struct {
    SecPk []byte //stakeholder's original public key // 成都链安 // 抵押持有者的源公钥
    Bn256Pk []byte //stakeholder's bn256 pairing public key // 成都链安 // 抵押持有者的 bn256 公钥
    LockEpochs *big.Int //lock time which is input by user // 成都链安 // 锁定时间
    FeeRate *big.Int // 成都链安 // 费率
    pub *ecdsa.PublicKey
}
type StakeUpdateParam struct {
    Addr common.Address //stakeholder's bn256 pairing public key // 成都链安 // 抵押持有者的 bn256 公钥
    LockEpochs *big.Int //lock time which is input by user
}
type PartnerInParam struct {
    Addr common.Address //stakeholder's bn256 pairing public key
    Renewal bool
}
type DelegateParam struct {
    DelegateAddress common.Address //delegation's address // 成都链安 // 代理者地址
```

```
}
type UpdateFeeRateParam struct {
    Addr common.Address
    FeeRate *big.Int
}

//
// storage structures
// 成都链安 // 存储结构
type StakerInfo struct {
    Address common.Address
    PubSec256 []byte //stakeholder's wan public key //抵押者的 wan 公钥
    PubBn256 []byte //stakeholder's bn256 public key

    Amount *big.Int //staking wan value // 成都链安 // 抵押的数量
    StakeAmount *big.Int //staking wan value
    LockEpochs uint64 //lock time which is input by user. 0 means unexpired. // 成都链安 // 抵押时间,0 意味着永不过期
    NextLockEpochs uint64 //lock time which is input by user. 0 means unexpired.
    From common.Address

    StakingEpoch uint64 //the first epoch in which stakerHolder might be selected.
    FeeRate uint64
    //NextFeeRate uint64
    Clients []ClientInfo
    Partners []PartnerInfo
}
```

```
type ValidatorInfo struct {
    TotalProbability *big.Int
    FeeRate uint64
    ValidatorAddr common.Address
    WalletAddr common.Address
    Infos []ClientProbability // the position 0 is validator and others is delegators.
}
// 成都链安 // 客户信息
type ClientInfo struct {
    Address common.Address
    Amount *big.Int
    StakeAmount *big.Int //staking wan value
    QuitEpoch uint64
}
type PartnerInfo struct {
    Address common.Address
    Amount *big.Int
    StakeAmount *big.Int //staking wan value
    Renewal bool
    LockEpochs uint64
    StakingEpoch uint64
}

type UpdateFeeRate struct {
    ValidatorAddr common.Address
    MaxFeeRate uint64
    FeeRate uint64
```

```
    ChangedEpoch uint64
}
//
// public helper structures
//
type Leader struct {
    Type uint8 `json:"type"`
    SecAddr common.Address `json:"secAddr"`
    PubSec256 []byte `json:"pubSec256"`
    PubBn256 []byte `json:"pubBn256"`
}

type ClientProbability struct {
    ValidatorAddr common.Address
    WalletAddr common.Address
    Probability *big.Int
}

type ClientIncentive struct {
    ValidatorAddr common.Address
    WalletAddr common.Address
    Incentive *big.Int
}

//
// package initialize
// 成都链安 // 初始化
func init() {
```

```
    if errCscInit != nil {
        panic("err in csc abi initialize ")
    }

    copy(stakeInId[:], cscAbi.Methods["stakeIn"].Id())
    copy(stakeAppendId[:], cscAbi.Methods["stakeAppend"].Id())
    copy(stakeUpdateId[:], cscAbi.Methods["stakeUpdate"].Id())
    copy(partnerInId[:], cscAbi.Methods["partnerIn"].Id())
    copy(delegateInId[:], cscAbi.Methods["delegateIn"].Id())
    copy(delegateOutId[:], cscAbi.Methods["delegateOut"].Id())
    copy(stakeUpdateFeeRateId[:], cscAbi.Methods["stakeUpdateFeeRate"].Id())
}

////////////////////////////
//
// pos staking contract
// 成都链安 // pos 抵押合约
type PosStaking struct {
}

//
// contract interfaces
// 成都链安 // pos 抵押合约接口
//
func (p *PosStaking) RequiredGas(input []byte) uint64 {
    return 0
}
```

```
// 成都链安 // 入口函数
func (p *PosStaking) Run(input []byte, contract *Contract, evm *EVM) ([]byte, error)
{
    if len(input) < 4 {
        return nil, errors.New("parameter is wrong")
    }

    var methodId [4]byte
    copy(methodId[:], input[:4])

    // 成都链安 // 更具方法 id 的不同分辨不同的函数
    // 成都链安 // 抵押函数
    if methodId == stakeInId {
        ret, err := p.StakeIn(input[4:], contract, evm)
        if err != nil {
            log.Info("stakein failed", "err", err)
        }
        return ret, err
    } else if methodId == stakeUpdateId {
// 成都链安 // 更新抵押时间
        return p.StakeUpdate(input[4:], contract, evm)
    } else if methodId == stakeAppendId {
// 成都链安 // 追加抵押数
        return p.StakeAppend(input[4:], contract, evm)
    } else if methodId == partnerInId {
        return p.PartnerIn(input[4:], contract, evm)
    } else if methodId == delegateInId {
```

```
// 成都链安 // 代理抵押
        return p.DelegateIn(input[4:], contract, evm)
    } else if methodId == delegateOutId {
        return p.DelegateOut(input[4:], contract, evm)
    } else if methodId == stakeUpdateFeeRateId {
// 成都链安 // 更新费率
        return p.StakeUpdateFeeRate(input[4:], contract, evm)
    }
    return nil, errMethodId
}

// 成都链安 // 交易验证
func (p *PosStaking) ValidTx(stateDB StateDB, signer types.Signer, tx
*types.Transaction) error {
    input := tx.Data()
    if len(input) < 4 {
        return errors.New("parameter is too short")
    }

    if params.IsNoStaking() {
        return errors.New("noStaking specified")
    }
    var methodId [4]byte
    copy(methodId[:], input[:4])

    // 成都链安 // 不同函数执行不同的验证
    if methodId == stakeInId {
        _, err := p.stakeInParseAndValid(input[4:])
```

```
        if err != nil {
            return errors.New("stakein verify failed")
        }
        return nil
    } else if methodId == stakeAppendId {
        _, err := p.stakeAppendParseAndValid(input[4:])
        if err != nil {
            return errors.New("stakeAppend verify failed " + err.Error())
        }
        return nil
    } else if methodId == stakeUpdateId {
        _, err := p.stakeUpdateParseAndValid(input[4:])
        if err != nil {
            return errors.New("stakeUpdate verify failed " + err.Error())
        }
        return nil
    } else if methodId == partnerInId {
        _, err := p.partnerInParseAndValid(input[4:])
        if err != nil {
            return errors.New("partnerIn verify failed " + err.Error())
        }
        return nil
    } else if methodId == delegateInId {
        _, err := p.delegateInParseAndValid(input[4:])
        if err != nil {
            return errors.New("delegateIn verify failed")
        }
        return nil
```

```
    } else if methodId == delegateOutId {
        _, err := p.delegateOutParseAndValid(input[4:])
        if err != nil {
            return errors.New("delegateOut verify failed")
        }
        return nil
    } else if methodId == stakeUpdateFeeRateId {
        _, err := p.updateFeeRateParseAndValid(input[4:])
        if err != nil {
            return errors.New("update fee rate verify failed")
        }
        return nil
    }

    return errParameters
}

// 成都链安 // 存储 stake 信息
func (p *PosStaking) saveStakeInfo(evm *EVM, stakerInfo *StakerInfo) error {
    infoBytes, err := rlp.EncodeToBytes(stakerInfo)
    if err != nil {
        return err
    }
    key := GetStakeInKeyHash(stakerInfo.Address)
    res := StoreInfo(evm.StateDB, StakersInfoAddr, key, infoBytes)
    if res != nil {
        return res
    }
```

```
    return nil
}
// 成都链安 // 获取 stake 信息
func (p *PosStaking) getStakeInfo(evm *EVM, addr common.Address) (*StakerInfo,
error) {
    key := GetStakeInKeyHash(addr)
    stakerBytes, err := GetInfo(evm.StateDB, StakersInfoAddr, key)
    if stakerBytes == nil {
        return nil, errors.New("item doesn't exist")
    }
    var stakerInfo StakerInfo
    err = rlp.DecodeBytes(stakerBytes, &stakerInfo)
    if err != nil {
        return nil, errors.New("parse staker info error")
    }
    return &stakerInfo, nil
}

// 成都链安 // 获取抵押费率
func (p *PosStaking) getStakeFeeRate(evm *EVM, address common.Address)
(*UpdateFeeRate, error) {
    key := GetStakeInKeyHash(address)
    // 成都链安 // 从数据库取信息
    feeBytes, err := GetInfo(evm.StateDB, StakersFeeAddr, key)
    if err != nil {
        return nil, err
    }
    if feeBytes == nil {
```

```
        return nil, nil
    }
    var feeRate UpdateFeeRate
    err = rlp.DecodeBytes(feeBytes, &feeRate)
    if err != nil {
        return nil, err
    }
    return &feeRate, nil
}

// 成都链安 // 存储费率信息
func (p *PosStaking) saveStakeFeeRate(evm *EVM, feeRate *UpdateFeeRate, address
common.Address) error {
    feeBytes, err := rlp.EncodeToBytes(feeRate)
    if err != nil {
        return err
    }
    // 成都链安 // 根据地址获取 key
    key := GetStakeInKeyHash(address)
    // 成都链安 // 根据 key 存如费率信息
    err = StoreInfo(evm.StateDB, StakersFeeAddr, key, feeBytes)
    if err != nil {
        return err
    }
    return nil
}

// 成都链安 // 抵押更新
```

```
func (p *PosStaking) StakeUpdate(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
    info, err := p.stakeUpdateParseAndValid(payload)
    if err != nil {
        return nil, err
    }

    // 成都链安 // 先获取信息
    stakerInfo, err := p.getStakeInfo(evm, info.Addr)
    if err != nil {
        return nil, err
    }
    // 成都链安 // 判断权限
    if contract.CallerAddress != stakerInfo.From {
        return nil, errors.New("Cannot update from another account")
    }

    eidNow, _ := util.CalEpochSlotID(evm.Time.Uint64())
    // 成都链安 // 判断锁定时间是否到达，
    if eidNow > stakerInfo.StakingEpoch+stakerInfo.LockEpochs-UpdateDelay {
        return nil, errors.New("cannot change at the last 3 epoch.")
    }

    // 成都链安 // 设置新的锁定时间,这里新的锁定时间来自用户参数
    stakerInfo.NextLockEpochs = info.LockEpochs.Uint64()
    // 成都链安 // 存储
    err = p.saveStakeInfo(evm, stakerInfo)
    if err != nil {
```

```
        return nil, err
    }
    // 成都链安 // 记录日志
    p.stakeUpdateLog(contract, evm, stakerInfo)
    return nil, nil
}

func (p *PosStaking) PartnerIn(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
    info, err := p.partnerInParseAndValid(payload)
    if err != nil {
        return nil, err
    }

    stakerInfo, err := p.getStakeInfo(evm, info.Addr)
    if err != nil {
        return nil, err
    }

    eidNow, _ := util.CalEpochSlotID(evm.Time.Uint64())
    if eidNow >= posconfig.ApploEpochID {
        if contract.Value().Cmp(minPartnerIn) < 0 {
            return nil, errors.New("min wan amount should >= 10000")
        }
    }
    realLockEpoch := int64(stakerInfo.LockEpochs - (eidNow + JoinDelay -
stakerInfo.StakingEpoch))
    if stakerInfo.StakingEpoch == 0 {
        realLockEpoch = int64(stakerInfo.LockEpochs)
```

```
    }
    if realLockEpoch < 0 || realLockEpoch > PSMaxEpochNum {
        return nil, errors.New("Wrong lock Epochs")
    }
    weight := CalLocktimeWeight(uint64(realLockEpoch))

    total := big.NewInt(0).Set(stakerInfo.Amount)
    for i := 0; i < len(stakerInfo.Clients); i++ {
        total.Add(total, stakerInfo.Clients[i].Amount)
    }
    total.Add(total, contract.Value())
    length := len(stakerInfo.Partners)
    found := false
    var partner *PartnerInfo = nil
    for i := 0; i < length; i++ {
        total.Add(total, stakerInfo.Partners[i].Amount)
        if stakerInfo.Partners[i].Address == contract.CallerAddress {
            partner = &stakerInfo.Partners[i]
            partner.Amount.Add(partner.Amount, contract.Value())
            partner.StakeAmount.Add(partner.StakeAmount,
big.NewInt(0).Mul(contract.Value(), big.NewInt(int64(weight))))
            partner.Renewal = info.Renewal
            found = true
        }
    }
    // check stake + partner + delegate <= 10,500,000
    // 成都链安 // 检查最大抵押数量
    if total.Cmp(maxTotalStake) > 0 {
```

```
        return nil, errors.New("partner in failed, too much stake")
    }
    if found == false {
        if length >= maxPartners {
            return nil, errors.New("Too many partners")
        }
        partner = &PartnerInfo{
            Address: contract.CallerAddress,
            Amount: contract.Value(),
            Renewal: info.Renewal,
            StakingEpoch: eidNow + JoinDelay,
            LockEpochs: uint64(realLockEpoch),
        }
        if posconfig.FirstEpochId == 0 {
            partner.StakingEpoch = 0
        }
        partner.StakeAmount = big.NewInt(0).Mul(partner.Amount,
big.NewInt(int64(weight)))
        stakerInfo.Partners = append(stakerInfo.Partners, *partner)
    }

    err = p.saveStakeInfo(evm, stakerInfo)
    if err != nil {
        return nil, err
    }
    if partner != nil {
        err = p.partnerInLog(contract, evm, &info.Addr, info.Renewal)
        if err != nil {
            return nil, err
```

```
        }
    }
    return nil, nil
}

// 成都链安 // 追加抵押数量
func (p *PosStaking) StakeAppend(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
    addr, err := p.stakeAppendParseAndValid(payload)
    if err != nil {
        return nil, err
    }

    stakerInfo, err := p.getStakeInfo(evm, addr)
    if err != nil {
        return nil, err
    }
    // 成都链安 // 权限判断
    if contract.CallerAddress != stakerInfo.From {
        return nil, errors.New("Cannot append from another account")
    }

    // add origen Amount
    // 成都链安 // 添加数量
    stakerInfo.Amount.Add(stakerInfo.Amount, contract.Value())
    // 成都链安 // 获取现在的 Epochid
    eidNow, _ := util.CalEpochSlotID(evm.Time.Uint64())
    // 成都链安 // 计算锁定时间
```

```
    realLockEpoch := int64(stakerInfo.LockEpochs - (eidNow + JoinDelay -
stakerInfo.StakingEpoch))
    //锁定时间判断
    if stakerInfo.StakingEpoch == 0 {
        realLockEpoch = int64(stakerInfo.LockEpochs)
    }
    if realLockEpoch < 0 || realLockEpoch > PSMaxEpochNum {
        return nil, errors.New("Wrong lock Epochs")
    }

    total := big.NewInt(0).Set(stakerInfo.Amount)
    for i := 0; i < len(stakerInfo.Clients); i++ {
        total.Add(total, stakerInfo.Clients[i].Amount)
    }
    for i := 0; i < len(stakerInfo.Partners); i++ {
        total.Add(total, stakerInfo.Partners[i].Amount)
    }
    // check stake + partner + delegate <= 10,500,000
    if total.Cmp(maxTotalStake) > 0 {
        return nil, errors.New("StakeAppend in failed, too much stake")
    }

    weight := CalLocktimeWeight(uint64(realLockEpoch))
    stakerInfo.StakeAmount.Mul(stakerInfo.Amount, big.NewInt(int64(weight)))
    err = p.saveStakeInfo(evm, stakerInfo)
    if err != nil {
        return nil, err
    }
    p.stakeAppendLog(contract, evm, stakerInfo.Address)
```

```
    return nil, nil
}

// 成都链安 // 执行抵押函数
func (p *PosStaking) StakeIn(payload []byte, contract *Contract, evm *EVM) ([]byte, error) {
    // 成都链安 // 执行参数检查
    info, err := p.stakeInParseAndValid(payload)
    if err != nil {
        return nil, err
    }

    // no max limit
    // amount >= PSMinStakeholderStake,
    // 成都链安 // 最小抵押数的判断
    if contract.value.Cmp(minStakeholderStake) < 0 {
        return nil, errors.New("need more Wan to be a stake holder")
    }
    // TODO: or return value - 10,500,000 to the sender?
    // 成都链安 // 最大抵押数
    if contract.value.Cmp(maxTotalStake) > 0 {
        return nil, errors.New("max stake is 10,500,000")
    }

    // NOTE: if a validator has no MinValidatorStake, but want delegate, he can partnerIn or stakeAppend later.
    // 成都链安 // 如果不满足最小抵押数,当时想通过代理来抵押,就在后面使用 partnerIn 或者 statkeAppend 函数
```

```
    // SO, don't need all in the first stakeIn.
    //if info.FeeRate.Cmp(noDelegateFeeRate) != 0 &&
contract.value.Cmp(MinValidatorStake) < 0 {
    //  return nil, errors.New("need more Wan to be a validator")
    //}
    secAddr := crypto.PubkeyToAddress(*info.pub)

    // 6. secAddr has not join the pos or has finished
    key := GetStakeInKeyHash(secAddr)
    oldInfo, err := GetInfo(evm.StateDB, StakersInfoAddr, key)
    // a. is secAddr joined?
    if oldInfo != nil {
        return nil, errors.New("public Sec address has exist")
    }

    // create stakeholder's information
    // 成都链安 // 创建抵押者的信息
    eidNow, _ := util.CalEpochSlotID(evm.Time.Uint64())
    weight := CalLocktimeWeight(info.LockEpochs.Uint64())
    stakerInfo := &StakerInfo{
        Address: secAddr,
        PubSec256: info.SecPk,
        PubBn256: info.Bn256Pk,
        Amount: contract.value,
        LockEpochs: info.LockEpochs.Uint64(),
        FeeRate: info.FeeRate.Uint64(),
        NextLockEpochs: info.LockEpochs.Uint64(),
        //NextFeeRate: info.FeeRate.Uint64(),
```

```
        From: contract.CallerAddress,
        StakingEpoch: eidNow + JoinDelay,
    }
    if posconfig.FirstEpochId == 0 {
        stakerInfo.StakingEpoch = 0
    }
    // 成都链安 // 抵押数量
    stakerInfo.StakeAmount = big.NewInt(0).Mul(stakerInfo.Amount,
big.NewInt(int64(weight)))

    // 成都链安 // 存储抵押信息
    err = p.saveStakeInfo(evm, stakerInfo)
    if err != nil {
        return nil, err
    }
    p.stakeInLog(contract, evm, stakerInfo)
    return nil, nil
}

// one wants to choose a delegation to join the pos
// 成都链安 // 如果想要选着一个代理加入这个 pos 共识,就调用该方法;代理抵押
func (p *PosStaking) DelegateIn(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
    // 成都链安 // 参数检查
    addr, err := p.delegateInParseAndValid(payload)
    if err != nil {
        return nil, err
    }
```

```
    // 成都链安 // 获取抵押信息
    stakerInfo, err := p.getStakeInfo(evm, addr)
    if err != nil {
        return nil, err
    }
    // check if the validator's feeRate is 100, can't delegatein
    // 成都链安 // 检查抵押费率来判断是否支持抵押
    if stakerInfo.FeeRate == noDelegateFeeRate.Uint64() {
        return nil, errors.New("Validator don't accept delegation.")
    }
    // check if the validator's amount(include partner) is not enough, can't
delegatein
    // 成都链安 // 这里检查 amount 总数是否达到抵押的最小数
    total := big.NewInt(0).Set(stakerInfo.Amount)
    for i := 0; i < len(stakerInfo.Partners); i++ {
        total.Add(total, stakerInfo.Partners[i].Amount)
    }
    if total.Cmp(MinValidatorStake) < 0 {
        return nil, errors.New("Validator don't have enough amount.")
    }
    // sender has not delegated by this
    var info *ClientInfo
    length := len(stakerInfo.Clients)

    totalDelegated := big.NewInt(0)
    totalDelegated.Add(totalDelegated, contract.Value())
    for i := 0; i < length; i++ {
```

```
        totalDelegated.Add(totalDelegated, stakerInfo.Clients[i].Amount)
        if stakerInfo.Clients[i].Address == contract.CallerAddress {
            if stakerInfo.Clients[i].QuitEpoch != 0 {
                return nil, errors.New("dalegater is quiting.")
            }
            weight := CalLocktimeWeight(PSMinEpochNum)
            info = &stakerInfo.Clients[i]
            // 成都链安 // 这里添加抵押的操作
            info.Amount.Add(info.Amount, contract.Value())
            info.StakeAmount.Add(info.StakeAmount,
big.NewInt(0).Mul(contract.Value(), big.NewInt(int64(weight))))
        }
    }
    // check self + partner + delegate <= 10,500,000
    // 成都链安 // 判断总数是否大于最大抵押
    if new(big.Int).Add(totalDelegated, total).Cmp(maxTotalStake) > 0 {
        return nil, errors.New("delegate over total stake limitation")
    }
    // check the totalDelegated <= 10*stakerInfo.Amount
    // 成都链安 // 最大代理数限制
    if totalDelegated.Cmp(big.NewInt(0).Mul(total, big.NewInt(MaxTimeDelegate))) > 0
{
        return nil, errors.New("over delegate limitation")
    }
    if info == nil {
        // only first delegatein check amount is valid.
        if contract.value.Cmp(minDelegatorStake) < 0 {
            return nil, errors.New("low amount")
```

```
        }
        // save
        //存储，也就是锁定 stakeAmount psMinEpochNum 默认等于 7
        weight := CalLocktimeWeight(PSMinEpochNum)
        info := &ClientInfo{
            Address: contract.CallerAddress,
            Amount: contract.value,
            StakeAmount: big.NewInt(0).Mul(contract.value,
big.NewInt(int64(weight))),
            QuitEpoch: 0,
        }
        stakerInfo.Clients = append(stakerInfo.Clients, *info)
    }

    err = p.saveStakeInfo(evm, stakerInfo)
    if err != nil {
        return nil, err
    }
    p.delegateInLog(contract, evm, stakerInfo.Address)
    return nil, nil
}
```

// 成都链安 // 收回代理抵押

```
func (p *PosStaking) DelegateOut(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
    addr, err := p.delegateOutParseAndValid(payload)
    if err != nil {
        return nil, err
    }
```

```
stakerInfo, err := p.getStakeInfo(evm, addr)
if err != nil {
    return nil, err
}

length := len(stakerInfo.Clients)
eidNow, _ := util.CalEpochSlotID(evm.Time.Uint64())

found := false
// 成都链安 // 循环查找,找到就退出
for i := 0; i < length; i++ {
    if stakerInfo.Clients[i].Address == contract.CallerAddress {
        // check if delegater has existed.
        if stakerInfo.Clients[i].QuitEpoch != 0 {
            return nil, errors.New("delegator has existed")
        }
        // 成都链安 // 改变状态
        stakerInfo.Clients[i].QuitEpoch = eidNow + QuitDelay
        found = true
        break
    }
}
if !found {
    return nil, errors.New("item doesn't exist")
}

//
err = p.saveStakeInfo(evm, stakerInfo)
```

```
    if err != nil {
        return nil, err
    }
    p.delegateOutLog(contract, evm, stakerInfo.Address)
    return nil, nil
}

// 成都链安 // 更新抵押费率
func (p *PosStaking) StakeUpdateFeeRate(payload []byte, contract *Contract, evm
*EVM) ([]byte, error) {
    feeRateParam, err := p.updateFeeRateParseAndValid(payload)
    if err != nil {
        return nil, err
    }
    stakeInfo, err := p.getStakeInfo(evm, feeRateParam.Addr)
    if err != nil {
        return nil, err
    }

    // if feeRate == 10000, can't change
    // 成都链安 // 等于 10000 的费率就不能改变
    if stakeInfo.FeeRate == PSMaxFeeRate || feeRateParam.FeeRate.Uint64() ==
PSMaxFeeRate {
        return nil, errors.New("feeRate equal 10000, can't change")
    }
    //更新相同的费率

    if stakeInfo.FeeRate == feeRateParam.FeeRate.Uint64() {
```

```
        return nil, errors.New("feeRate already same")
    }

    if contract.CallerAddress != stakeInfo.From {
        return nil, errors.New("cannot update fee from another account")
    }

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())
    oldFee, err := p.getStakeFeeRate(evm, stakeInfo.Address)
    if err != nil {
        return nil, err
    }

    if oldFee == nil {
        oldFee = &UpdateFeeRate{
            ValidatorAddr: stakeInfo.Address,
            MaxFeeRate: stakeInfo.FeeRate,
            FeeRate: stakeInfo.FeeRate,
            ChangedEpoch: eid,
        }
    } else if oldFee.ChangedEpoch == eid {
        return nil, errors.New("one epoch can only change one time")
    }

    feeRate := feeRateParam.FeeRate.Uint64()
    // 成都链安 // 0 <= fee <= maxFee,费率不能调高
    if feeRate > oldFee.MaxFeeRate {
        return nil, errors.New("fee rate can't bigger than old")
```

```
    }
    if feeRate > stakeInfo.FeeRate + PSFeeRateStep {
        return nil, errors.New("0 <= newFeeRate <= oldFeerate + 100")
    }

    oldFee.FeeRate = feeRate
    oldFee.ChangedEpoch = eid

    stakeInfo.FeeRate = feeRate

    err = p.saveStakeInfo(evm, stakeInfo)
    if err != nil {
        return nil, err
    }
    err = p.saveStakeFeeRate(evm, oldFee, stakeInfo.Address)
    if err != nil {
        return nil, err
    }
    err = p.stakeUpdateFeeRateLog(contract, evm, oldFee)
    if err != nil {
        return nil, err
    }
    return nil, nil
}

/*
the weight of 7 epoch: a + 7*b ~= 1000
the weight of 90 epoch: a + 90*b ~= 1500
```

```
the time of maxEpoch/minEpoch is 1.5

so the a=960, b=6.

thus, weight of 7 epoch is 960+7*6=1002

the weight of 90 epoch is 960+90*6 = 1500

the time is 1500/1002, about 1.5

*/

// 成都链安 // 计算锁定时间获得的权益

func CalLocktimeWeight(lockEpoch uint64) uint64 {

    if lockEpoch == 0 { //builtin account.

        return 960 + PSMinEpochNum*6

    }

    return 960 + lockEpoch*6

}


func GetStakeInKeyHash(address common.Address) common.Hash {

    return common.BytesToHash(address[:])

}


func GetStakersSnap(stateDb *state.StateDB) []StakerInfo {

    stakeHolders := make([]StakerInfo, 0)

    stateDb.ForEachStorageByteArray(StakersInfoAddr, func(key common.Hash, value
[]byte) bool {

        var stakerInfo StakerInfo

        err := rlp.DecodeBytes(value, &stakerInfo)

        if err != nil {

            log.Error(err.Error())

            return true

        }
```

```
        stakeHolders = append(stakeHolders, stakerInfo)
        return true
    })
    return stakeHolders
}

func StakeoutSetEpoch(stateDb *state.StateDB, epochID uint64) {
    b := big.NewInt(int64(epochID))
    StoreInfo(stateDb, StakingCommonAddr, StakersInfoStakeOutKeyHash, b.Bytes())
}

func StakeoutIsFinished(stateDb *state.StateDB, epochID uint64) bool {
    epochByte, err := GetInfo(stateDb, StakingCommonAddr,
StakersInfoStakeOutKeyHash)
    if err != nil {
        return false
    }
    finishedEpochId := big.NewInt(0).SetBytes(epochByte).Uint64()
    return finishedEpochId >= epochID
}

//
// package param check helper functions
//
// 成都链安 // 参数检验
func (p *PosStaking) stakeInParseAndValid(payload []byte) (StakeInParam, error) {
    var info StakeInParam
    err := cscAbi.UnpackInput(&info, "stakeIn", payload)
```

```
    if err != nil {
        return info, err
    }
    // 1. SecPk is valid
    if info.SecPk == nil {
        return info, errors.New("wrong secPk for stakeIn")
    }
    pub := crypto.ToECDSAPub(info.SecPk)
    if nil == pub {
        return info, errors.New("secPk is invalid")
    }
    info.pub = pub

    // 2. Bn256Pk is valid
    if info.Bn256Pk == nil {
        return info, errors.New("wrong bn256Pk for stakeIn")
    }
    var g1 bn256.G1
    _, err = g1.Unmarshal(info.Bn256Pk)
    if err != nil {
        return info, errors.New("wrong point for bn256Pk")
    }

    // 3. Lock time >= min epoch, <= max epoch
    //这里有最大 lockEpochs 的检查防止溢出。
    if info.LockEpochs.Cmp(minEpochNum) < 0 || info.LockEpochs.Cmp(maxEpochNum) > 0
{
        return info, errors.New("invalid lock time")
    }
```

```
    // 4. 0 <= FeeRate <= 10000
    if info.FeeRate.Cmp(maxFeeRate) > 0 || info.FeeRate.Cmp(minFeeRate) < 0 {
        return info, errors.New("fee rate should between 0 to 100")
    }

    return info, nil
}

func (p *PosStaking) partnerInParseAndValid(payload []byte) (PartnerInParam, error)
{
    var info PartnerInParam
    err := cscAbi.UnpackInput(&info, "partnerIn", payload)
    if err != nil {
        return info, err
    }
    return info, nil
}

func (p *PosStaking) stakeUpdateParseAndValid(payload []byte) (StakeUpdateParam,
error) {
    var info StakeUpdateParam
    err := cscAbi.UnpackInput(&info, "stakeUpdate", payload)
    if err != nil {
        return info, err
    }
    // Lock time >= min epoch, <= max epoch
    if info.LockEpochs.Uint64() != 0 && (info.LockEpochs.Cmp(minEpochNum) < 0 ||
info.LockEpochs.Cmp(maxEpochNum) > 0) {
        return info, errors.New("invalid lock time")
    }
```

```
    return info, nil
}
func (p *PosStaking) stakeAppendParseAndValid(payload []byte) (common.Address,
error) {
    var addr common.Address
    err := cscAbi.UnpackInput(&addr, "stakeAppend", payload)
    if err != nil {
        return addr, err
    }

    return addr, nil
}
func (p *PosStaking) delegateInParseAndValid(payload []byte) (common.Address, error)
{
    var addr common.Address
    err := cscAbi.UnpackInput(&addr, "delegateIn", payload)
    if err != nil {
        return addr, err
    }

    return addr, nil
}
func (p *PosStaking) delegateOutParseAndValid(payload []byte) (common.Address,
error) {
    var addr common.Address
    err := cscAbi.UnpackInput(&addr, "delegateOut", payload)
    if err != nil {
        return addr, err
```

```
    }


    return addr, nil

}
```

// 成都链安 // 费率更新前的参数检查

```
func (p *PosStaking) updateFeeRateParseAndValid(payload []byte)
(*UpdateFeeRateParam, error) {

    var updateFeeRateParam UpdateFeeRateParam

    err := cscAbi.UnpackInput(&updateFeeRateParam, "stakeUpdateFeeRate", payload)

    if err != nil {

        return nil, err

    }
```

// 成都链安 // 费率范围检查

```
    if updateFeeRateParam.FeeRate.Cmp(maxFeeRate) > 0 ||
updateFeeRateParam.FeeRate.Cmp(minFeeRate) < 0 {

        return nil, errors.New("fee rate should between 0 to 10000")

    }


    return &updateFeeRateParam, nil

}
```

// 成都链安 // 生成日志

```
func (p *PosStaking) stakeInLog(contract *Contract, evm *EVM, info *StakerInfo)
error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid < posconfig.ApploEpochID {

        params := make([]common.Hash, 5)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = common.BigToHash(contract.Value())
```

```
        params[2] = common.BigToHash(new(big.Int).SetUint64(info.FeeRate))

        params[3] = common.BigToHash(new(big.Int).SetUint64(info.LockEpochs))

        params[4] = info.Address.Hash()

        sig := crypto.Keccak256([]byte(cscAbi.Methods["stakeIn"].Sig()))

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    } else {

        // event stakeIn(address indexed sender, address indexed posAddress, uint
indexed v, uint feeRate, uint lockEpoch);

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = info.Address.Hash()

        params[2] = common.BigToHash(contract.Value())

        //

        data := make([]byte, 0)

        data = append(data,
common.BigToHash(new(big.Int).SetUint64(info.FeeRate)).Bytes()...)

        data = append(data,
common.BigToHash(new(big.Int).SetUint64(info.LockEpochs)).Bytes()...)

        sig := cscAbi.Events["stakeIn"].Id().Bytes()

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, data)

    }

}


func (p *PosStaking) stakeAppendLog(contract *Contract, evm *EVM, validator
common.Address) error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid < posconfig.AppIoEpochID {

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())
```

```
        params[1] = common.BigToHash(contract.Value())

        params[2] = validator.Hash()


        sig := crypto.Keccak256([]byte(cscAbi.Methods["stakeAppend"].Sig()))

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    } else {

        // event stakeAppend(address indexed sender, address indexed posAddress,
uint indexed v);

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = validator.Hash()

        params[2] = common.BigToHash(contract.Value())


        sig := cscAbi.Events["stakeAppend"].Id().Bytes()

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    }

}


func (p *PosStaking) stakeUpdateLog(contract *Contract, evm *EVM, info *StakerInfo)
error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid < posconfig.ApploEpochID {

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = common.BigToHash(new(big.Int).SetUint64(info.NextLockEpochs))

        params[2] = info.Address.Hash()


        sig := crypto.Keccak256([]byte(cscAbi.Methods["stakeUpdate"].Sig()))
```

```
        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    } else {

        // event stakeUpdate(address indexed sender, address indexed posAddress,
uint indexed lockEpoch);

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = info.Address.Hash()

        params[2] = common.BigToHash(new(big.Int).SetUint64(info.NextLockEpochs))


        sig := cscAbi.Events["stakeUpdate"].Id().Bytes()

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    }

}


func (p *PosStaking) delegateInLog(contract *Contract, evm *EVM, validator
common.Address) error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid < posconfig.ApploEpochID {

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = common.BigToHash(contract.Value())

        params[2] = validator.Hash()


        sig := crypto.Keccak256([]byte(cscAbi.Methods["delegateIn"].Sig()))

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    } else {

        // event delegateIn(address indexed sender, address indexed posAddress, uint
indexed v);
```

```
        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = validator.Hash()

        params[2] = common.BigToHash(contract.Value())


        sig := cscAbi.Events["delegateIn"].Id().Bytes()

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)


    }

}


func (p *PosStaking) delegateOutLog(contract *Contract, evm *EVM, validator
common.Address) error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid < posconfig.ApploEpochID {

        params := make([]common.Hash, 2)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = validator.Hash()


        sig := crypto.Keccak256([]byte(cscAbi.Methods["delegateOut"].Sig()))

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    } else {

        // event delegateOut(address indexed sender, address indexed posAddress);

        params := make([]common.Hash, 2)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = validator.Hash()


        sig := cscAbi.Events["delegateOut"].Id().Bytes()
```

```
        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    }

}


func (p *PosStaking) stakeUpdateFeeRateLog(contract *Contract, evm *EVM, feeInfo
*UpdateFeeRate) error {

    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())

    if eid >= posconfig.ApploEpochID {

        // event stakeUpdateFeeRate(address indexed sender, address indexed
posAddress, uint indexed feeRate);

        params := make([]common.Hash, 3)

        params[0] = common.BytesToHash(contract.Caller().Bytes())

        params[1] = common.BytesToHash(feeInfo.ValidatorAddr.Bytes())

        params[2] = common.BigToHash(new(big.Int).SetUint64(feeInfo.FeeRate))


        //data := make([]byte, 0)

        //data = append(data,
common.BigToHash(new(big.Int).SetUint64(feeInfo.MaxFeeRate)).Bytes()...)

        //data = append(data,
common.BigToHash(new(big.Int).SetUint64(feeInfo.ChangedEpoch)).Bytes()...)


        sig := cscAbi.Events["stakeUpdateFeeRate"].Id().Bytes()

        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig),
params, nil)

    }

    return nil

}


func (p *PosStaking) partnerInLog(contract *Contract, evm *EVM, addr
*common.Address, renew bool) error {
```

```
    eid, _ := util.CalEpochSlotID(evm.Time.Uint64())
    if eid >= posconfig.ApploEpochID {
        // event partnerIn(address indexed sender, address indexed posAddress, uint indexed v, bool renewal);
        params := make([]common.Hash, 3)
        params[0] = common.BytesToHash(contract.Caller().Bytes())
        params[1] = common.BytesToHash(addr.Bytes())
        params[2] = common.BigToHash(contract.Value())

        var renewal = uint64(0)
        if renew {
            renewal = uint64(1)
        }
        data := make([]byte, 0)
        data = append(data, common.BigToHash(new(big.Int).SetUint64(renewal)).Bytes()...)

        sig := cscAbi.Events["partnerIn"].Id().Bytes()
        return precompiledScAddLog(contract.Address(), evm, common.BytesToHash(sig), params, data)
    }
    return nil
}
```

3、 random_beacon_contract.go

```
// 成都链安 // file:go-wanchain/core/vm/random_beacon_contract.go
package vm
```

```
import (
    "bytes"
    "encoding/binary"
    "errors"
    "fmt"
    "math/big"
    "strconv"
    "strings"
    "time"

    "github.com/wanchain/go-wanchain/core/types"
    "github.com/wanchain/go-wanchain/pos/rbselection"
    "github.com/wanchain/go-wanchain/pos/util"

    "github.com/wanchain/go-wanchain/accounts/abi"
    "github.com/wanchain/go-wanchain/common"
    "github.com/wanchain/go-wanchain/crypto"
    bn256 "github.com/wanchain/go-wanchain/crypto/bn256/cloudflare"
    "github.com/wanchain/go-wanchain/log"
    "github.com/wanchain/go-wanchain/pos/posconfig"
    "github.com/wanchain/go-wanchain/rlp"
)

const (
    _ int = iota
    RbDkg1Stage
    RbDkg1ConfirmStage
    RbDkg2Stage
```

```
    RbDkg2ConfirmStage
    RbSignStage
    RbSignConfirmStage
)

var (
    // random beacon smart contract abi definition
    // 成都链安 // beacon 合约 abi 定义
    rbSCDefinition = `[
{
"constant": false,
"inputs": [
{
"name": "info",
"type": "string"
}
],
"name": "dkg1",
"outputs": [
],
"payable": false,
"type": "function"
},
{
"constant": false,
"inputs": [
{
"name": "info",
```

```
"type": "string"
}
],
"name": "dkg2",
"outputs": [
],
"payable": false,
"type": "function"
},
{
"constant": false,
"inputs": [
{
"name": "info",
"type": "string"
}
],
"name": "sigShare",
"outputs": [
],
"payable": false,
"type": "function"
}
]`
    // random beacon smart contract abi object
    rbSCAbi, errRbSCInit = abi.JSON(strings.NewReader(rbSCDefinition))

    // function "dkg1" "dkg2" "sigShare" 's solidity binary id
```

```
    // 成都链安 // 函数 dkg1 dkg2 sigShare 的 solidity 二进制 id
    dkg1Id [4]byte
    dkg2Id [4]byte
    sigShareId [4]byte

    // prefix for the key hash
    // 成都链安 // hash 头
    kindCij = []byte{100}
    kindEns = []byte{101}
    kindR = []byte{102}

    // bn256 curve's hBase
    hBase = new(bn256.G2).ScalarBaseMult(big.NewInt(int64(1)))

    // errors
    // 成都链安 // 错误定义
    errDkg1Parse = errors.New("dkg1 payload parse failed")
    errDkg2Parse = errors.New("dkg2 payload parse failed")
    errSigParse = errors.New("sig payload parse failed")
    errRSCode = errors.New("rs code verify failed")
    errDiscreteLogarithmsEQ = errors.New("the equality of discrete logarithms verify
failed")
    errRlpCij = errors.New("rlp encode cij failed")
    errRlpEncryptShare = errors.New("rlp encode encrypt share failed")
    errUnRlpCij = errors.New("rlp decode cij failed")
    errUnRlpEncryptShare = errors.New("rlp decode encrypt share failed")
    errInvalidCommitBytes = errors.New("invalid dkg commit bytes")
    errInvalidEncryptShareBytes = errors.New("invalid dkg encrypt share bytes")
```

```
)


// return:
//  current stage index;
//  elapsed slots number of current stage;
//  left slots number of current stage;
//成都链安/返回不同阶段的 slots
func GetRBStage(slotId uint64) (int, int, int) {
    if slotId <= posconfig.Cfg().Dkg1End {
        return RbDkg1Stage, int(slotId), int(posconfig.Cfg().Dkg1End - slotId)
    } else if slotId < posconfig.Cfg().Dkg2Begin {
        return RbDkg1ConfirmStage, int(slotId - posconfig.Cfg().Dkg1End - 1),
int(posconfig.Cfg().Dkg2Begin - slotId - 1)
    } else if slotId <= posconfig.Cfg().Dkg2End {
        return RbDkg2Stage, int(slotId - posconfig.Cfg().Dkg2Begin),
int(posconfig.Cfg().Dkg2End - slotId)
    } else if slotId < posconfig.Cfg().SignBegin {
        return RbDkg2ConfirmStage, int(slotId - posconfig.Cfg().Dkg2End - 1),
int(posconfig.Cfg().SignBegin - slotId - 1)
    } else if slotId <= posconfig.Cfg().SignEnd {
        return RbSignStage, int(slotId - posconfig.Cfg().SignBegin),
int(posconfig.Cfg().SignEnd - slotId)
    } else {
        return RbSignConfirmStage, int(slotId - posconfig.Cfg().SignEnd - 1),
int(posconfig.SlotCount - slotId - 1)
    }
}


// 成都链安 // 每一个 Epoch 有 12k 个插槽.随机 beacon 协议用来生成一个随机数 r
// One Epoch has 12k slots, random beacon protocol is used to generate a random r,
```

// 成都链安 // r 用来生成下一个 epoch 的 leader 和 slot 的 leader

// which will be used to generate next epoch leaders and slot leaders.

// 成都链安 // 这个协议有三步,dkg1 , dkg2,sigShare

// Random beacon protocol has 3 stages --- dkg1 (in 1k,2k slots), dkg2 (in 4k,5k slots), sigShare (in 8k, 9k slots)

// 成都链安 // RNP 星群中的节点通过 DKG1、DKG2、SIGN 三个阶段的工作完成随机数的更新，保证了链上随机数的安全性

// 成都链安 // 其每一轮产生的随机数将作为星群构建、出块者选择和其他随机源应用的重要种子，维持着共识的健康运转

// 成都链安 // slot 是一个区块的生成时间，即每个 slot 内产生一个新的区块，而 epoch 是由大量连续 slot 构成的时间周期，是协议完整运行的一个循环

```
type RandomBeaconContract struct {
}


//
// package init
//
func init() {
    if errRbSCInit != nil {
        panic("err in rb smart contract abi initialize")
    }

    copy(dkg1Id[:], rbSCAbi.Methods["dkg1"].Id())
    copy(dkg2Id[:], rbSCAbi.Methods["dkg2"].Id())
    copy(sigShareId[:], rbSCAbi.Methods["sigShare"].Id())
}
```

```
//
// contract interface
//
func (c *RandomBeaconContract) RequiredGas(input []byte) uint64 {
    return 0
}

// 成都链安 // 合约入口函数
func (c *RandomBeaconContract) Run(input []byte, contract *Contract, evm *EVM)
([]byte, error) {
    // check data
    if len(input) < 4 {
        return nil, errParameters
    }

    var methodId [4]byte
    copy(methodId[:], input[:4])

    // 成都链安 // 根据不同的参数选择不同的执行方法
    if methodId == dkg1Id {
        return c.dkg1(input[4:], contract, evm)
    } else if methodId == dkg2Id {
        return c.dkg2(input[4:], contract, evm)
    } else if methodId == sigShareId {
        return c.sigShare(input[4:], contract, evm)
    } else {
        log.SyslogErr("random beacon contract no match id found")
        return nil, errors.New("no function")
```

```
    }
}

// 成都链安 // 交易数据验证
func (c *RandomBeaconContract) ValidTx(stateDB StateDB, signer types.Signer, tx
*types.Transaction) error {
    if posconfig.FirstEpochId == 0 {
        return errParameters
    }
    if stateDB == nil || signer == nil || tx == nil {
        return errParameters
    }

    payload := tx.Data()
    if len(payload) < 4 {
        return errParameters
    }

    from, err := types.Sender(signer, tx)
    if err != nil {
        return err
    }

    return ValidPosRBTx(stateDB, from, payload)
}
func getRBProposerGroup(eid uint64) ([]bn256.G1, error) {
    ep := util.GetEpocherInst()
    if ep == nil {
        return nil, errors.New("GetEpocherInst() == nil")
```

```
    }
    pks := ep.GetRBProposerG1(eid)
    if len(pks) == 0 {
        return nil, errors.New("len(pks) == 0")
    }
    return pks, nil
}

//
// params or gas check functions
//
func ValidPosRBTx(stateDB StateDB, from common.Address, payload []byte) error {
    log.Debug("ValidPosRBTx")
    var methodId [4]byte
    copy(methodId[:], payload[:4])

    if methodId == dkg1Id {
        _, err := validDkg1(stateDB, uint64(time.Now().Unix()), from, payload[4:])
        return err
    } else if methodId == dkg2Id {
        _, err := validDkg2(stateDB, uint64(time.Now().Unix()), from, payload[4:])
        return err
    } else if methodId == sigShareId {
        _, _, _, err := validSigShare(stateDB, uint64(time.Now().Unix()), from,
payload[4:])
        return err
    } else {
        return errParameters
    }
```

```
}

// ValidDkg1 verify DKG1 precompiled contract transaction
// 成都链安 // 验证 DKG1 预编译合约交易
// 'time' is the time when tx is sealed into block,
//// 成都链安 // 时间是交易被写入区块的时间
// 'time' should be block timestamp when the method is called by sealing
// block and block verify.
// 成都链安 // 当方法被打包区块或者验证区块的时候调用的时间就应该是区块的时间戳
// 'time' should be current system time when the method is called by tx pool.
// 成都链安 // 当方法被交易池调用的话,时间就应该是当前系统时间
// 'caller' is the caller of DKG1. It should be set as Contract.CallerAddress
// when called by precompiled contract. And should be set as tx's sender when
// called by tx pool.
// 成都链安 // caller 是 DKG1 的调用者,当被预编译合约调用的时候它就应该被设置为
Contract.CallerAddress
// 成都链安 // 被交易池调用的时候就应该被设置为交易的发送者
// 成都链安 // 执行 DKG1 验证
func validDkg1(stateDB StateDB, time uint64, caller common.Address,
    payload []byte) (*RbDKG1FlatTxPayload, error) {

    var dkg1FlatParam RbDKG1FlatTxPayload
    err := rlp.DecodeBytes(payload, &dkg1FlatParam)
    if err != nil {
        return nil, logError(errDkg1Parse)
    }
```

```
    dkg1Param, err := Dkg1FlatToDkg1(&dkg1FlatParam)
    if err != nil {
        return nil, logError(err)
    }

    eid := dkg1Param.EpochId
    pid := dkg1Param.ProposerId

    pks, err := getRBProposerGroupVar(eid)
    if err != nil {
        return nil, err
    }
    // 1. EpochId: weather in a wrong time
    if !isValidEpochStageVar(eid, RbDkg1Stage, time) {
        return nil, logError(errors.New("invalid rb stage, expect RbDkg1Stage.
epochId " + strconv.FormatUint(eid, 10)))
    }

    // 2. ProposerId: weather in the random commit

    if !isInRandomGroupVar(pks, eid, pid, caller) {
        return nil, logError(errors.New("invalid proposer, proposerId " +
strconv.FormatUint(uint64(pid), 10)))
    }

    // 3. prevent reset
    existC, err := GetCji(stateDB, eid, pid)
    if err == nil && len(existC) != 0 {
        return nil, logError(errors.New("dkg1 commit exist already, proposerId " +
```

```
strconv.FormatUint(uint64(pid), 10)))
    }

    // 4.Commit has the same size
    // check same size
    nr := len(pks)
    if nr != len(dkg1Param.Commit) {
        return nil, logError(buildError("error in dkg1 params have invalid commits length", eid, pid))
    }

    // 5. Reed-Solomon code verification
    x := make([]big.Int, nr)
    for i := 0; i < nr; i++ {
        x[i].SetBytes(GetPolynomialX(&pks[i], uint32(i)))
        x[i].Mod(&x[i], bn256.Order)
    }

    temp := make([]bn256.G2, nr)
    for j := 0; j < nr; j++ {
        temp[j] = *dkg1Param.Commit[j]
    }

    if !rbselection.RScodeVerify(temp, x, int(posconfig.Cfg().PolymDegree)) {
        return nil, logError(errRSCode)
    }

    return &dkg1FlatParam, nil
}
```

```
//DKG2 验证
func validDkg2(stateDB StateDB, time uint64, caller common.Address,
    payload []byte) (*RbDKG2FlatTxPayload, error) {

    var dkg2FlatParam RbDKG2FlatTxPayload
    err := rlp.DecodeBytes(payload, &dkg2FlatParam)
    if err != nil {
        return nil, logError(errDkg2Parse)
    }

    dkg2Param, err := Dkg2FlatToDkg2(&dkg2FlatParam)
    if err != nil {
        return nil, logError(err)
    }

    eid := dkg2Param.EpochId
    pid := dkg2Param.ProposerId

    pks, err := getRBProposerGroupVar(eid)
    if err != nil {
        return nil, err
    }

    // 1. EpochId: weather in a wrong time
    if !isValidEpochStageVar(eid, RbDkg2Stage, time) {
        return nil, logError(errors.New("invalid rb stage, expect RbDkg2Stage. error
epochId " + strconv.FormatUint(eid, 10)))
    }
```

```
    // 2. ProposerId: weather in the random commit
    if !isInRandomGroupVar(pks, eid, pid, caller) {
        return nil, logError(errors.New("error proposerId " +
strconv.FormatUint(uint64(pid), 10)))
    }

    // prevent reset
    existE, err := GetEncryptShare(stateDB, eid, pid)
    if err == nil && len(existE) != 0 {
        return nil, logError(errors.New("dkg2 encrypt share exist already,
proposerId " + strconv.FormatUint(uint64(pid), 10)))
    }

    commit, err := GetCji(stateDB, eid, pid)
    if err != nil || len(commit) == 0 {
        return nil, logError(buildError("error in dkg2 can't get commit data", eid,
pid))
    }

    // 3. EncryptShare, Commit, Proof has the same size
    // check same size
    nr := len(pks)
    if nr != len(dkg2Param.EnShare) || nr != len(dkg2Param.Proof) || nr !=
len(commit) {
        return nil, logError(buildError("error in dkg2 params have different
length", eid, pid))
    }

    // 4. proof verification
```

```
    for j := 0; j < nr; j++ {
        // get send public Key
        if !rbselection.VerifyDLEQ(dkg2Param.Proof[j], pks[j], *hBase,
*dkg2Param.EnShare[j], *commit[j]) {
            return nil, logError(errDiscreteLogarithmsEQ)
        }
    }

    return &dkg2FlatParam, nil
}
```

// 成都链安 // 验证 sig 阶段

```
func validSigShare(stateDB StateDB, time uint64, caller common.Address,
    payload []byte) (*RbSIGTxPayload, []bn256.G1, []RbCijDataCollector, error) {

    var sigShareParam RbSIGTxPayload
    err := rlp.DecodeBytes(payload, &sigShareParam)
    if err != nil {
        return nil, nil, nil, logError(errors.New("error in dkg param has a wrong
struct"))
    }

    eid := sigShareParam.EpochId
    pid := sigShareParam.ProposerId

    pks, err := getRBProposerGroupVar(eid)
    if err != nil {
        return nil, nil, nil, err
    }
```

```
    // 1. EpochId: weather in a wrong time
    if !isValidEpochStageVar(eid, RbSignStage, time) {
        return nil, nil, nil, logError(errors.New("invalid rb stage, expect
RbSignStage. error epochId " + strconv.FormatUint(eid, 10)))
    }

    // 2. ProposerId: weather in the random commit
    if !isInRandomGroupVar(pks, eid, pid, caller) {
        return nil, nil, nil, logError(errors.New(" error proposerId " +
strconv.FormatUint(uint64(pid), 10)))
    }

    // 3. Verification
    M, err := getRBMVar(stateDB, eid)
    if err != nil {
        return nil, nil, nil, logError(buildError("getRBM error", eid, pid))
    }
    m := new(big.Int).SetBytes(M)

    var gPKShare bn256.G2

    dkgData := make([]RbCijDataCollector, 0)
    for id := range pks {
        if IsJoinDKG2(stateDB, eid, uint32(id)) {
            dkgDatum, err := GetCji(stateDB, eid, uint32(id))
            if err == nil && dkgDatum != nil {
                dkgData = append(dkgData, RbCijDataCollector{dkgDatum, &pks[id]})
                gPKShare.Add(&gPKShare, dkgDatum[pid])
```

```
            }
        }
    }

    if uint(len(dkgData)) < posconfig.Cfg().RBThres {
        return nil, nil, nil, logError(buildError("insufficient proposer", eid,
pid))
    }

    mG := new(bn256.G1).ScalarBaseMult(m)
    pair1 := bn256.Pair(sigShareParam.GSignShare, hBase)
    pair2 := bn256.Pair(mG, &gPKShare)
    if pair1.String() != pair2.String() {
        return nil, nil, nil, logError(buildError("unequal si gi", eid, pid))
    }

    return &sigShareParam, pks, dkgData, nil
}

//
// public get methods
//
func GetDkg1Id() []byte {
    return dkg1Id[:]
}

func GetDkg2Id() []byte {
    return dkg2Id[:]
}
```

```
func GetSigShareId() []byte {
    return sigShareId[:]
}

// get key hash
// 成都链安 // 获取 key 的 hash
func GetRBKeyHash(kind []byte, epochId uint64, proposerId uint32) *common.Hash {
    keyBytes := make([]byte, 16)
    copy(keyBytes, kind)
    copy(keyBytes[4:], UIntToByteSlice(epochId))
    copy(keyBytes[12:], UInt32ToByteSlice(proposerId))
    hash := common.BytesToHash(crypto.Keccak256(keyBytes))
    return &hash
}

// get r's key hash
// 成都链安 // 获取 r 的 key 哈希
func GetRBRKeyHash(epochId uint64) *common.Hash {
    keyBytes := make([]byte, 12)
    copy(keyBytes, kindR[:])
    copy(keyBytes[4:], UIntToByteSlice(epochId))
    hash := common.BytesToHash(crypto.Keccak256(keyBytes))
    return &hash
}

// get r of one epoch, if not exist return r in epoch 0
// 成都链安 // 获取一个 epoch 内的 r,如果不存在就返回 epoch 0 的 r
```

```
func GetR(db StateDB, epochId uint64) *big.Int {
    if epochId == posconfig.FirstEpochId {
        return GetStateR(db, posconfig.FirstEpochId)
    }
    r := GetStateR(db, epochId)
    if r == nil {
        if epochId > posconfig.FirstEpochId+2 {
            log.SyslogWarning("***Can not found random r just use the first epoch R", "epochId", epochId)
        }
        r = GetStateR(db, posconfig.FirstEpochId)
    }
    return r
}

// get r of one epoch
// 成都链安 // 获取一个 epoch 内的 r 随机数
func GetStateR(db StateDB, epochId uint64) *big.Int {
    if epochId == posconfig.FirstEpochId {
        return new(big.Int).SetBytes(crypto.Keccak256(big.NewInt(1).Bytes()))
    }
    hash := GetRBRKeyHash(epochId)
    rBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if len(rBytes) != 0 {
        r := new(big.Int).SetBytes(rBytes)
        return r
    }
    return nil
```

```
}


// get sig of one epoch, stored in "sigShare" function
func GetSig(db StateDB, epochId uint64, proposerId uint32) (*RbSIGTxPayload, error)
{
    hash := GetRBKeyHash(sigShareId[:], epochId, proposerId)
    payloadBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if len(payloadBytes) == 0 {
        return nil, nil
    }

    var sigParam RbSIGTxPayload
    err := rlp.DecodeBytes(payloadBytes, &sigParam)
    if err != nil {
        return nil, errSigParse
    }

    return &sigParam, nil
}


// get M
func GetRBM(db StateDB, epochId uint64) ([]byte, error) {
    epochIdBigInt := big.NewInt(int64(epochId + 1))
    preRandom := GetR(db, epochId)

    buf := epochIdBigInt.Bytes()
    buf = append(buf, preRandom.Bytes()...)
    return crypto.Keccak256(buf), nil
}
```

```
func GetRBAddress() common.Address {
    return randomBeaconPrecompileAddr
}

func GetPolynomialX(pk *bn256.G1, proposerId uint32) []byte {
    return crypto.Keccak256(pk.Marshal(), big.NewInt(int64(proposerId)).Bytes())
}

func GetCji(db StateDB, epochId uint64, proposerId uint32) ([]*bn256.G2, error) {
    hash := GetRBKeyHash(kindCij, epochId, proposerId)
    dkgBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if len(dkgBytes) == 0 {
        return nil, nil
    }

    cij := make([][]byte, 0)
    err := rlp.DecodeBytes(dkgBytes, &cij)
    if err != nil {
        return nil, errUnRlpCij
    }

    return BytesToCij(&cij)
}

// get encrypt share
func GetEncryptShare(db StateDB, epochId uint64, proposerId uint32) ([]*bn256.G1,
error) {
    hash := GetRBKeyHash(kindEns, epochId, proposerId)
```

```
    dkgBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if len(dkgBytes) == 0 {
        return nil, nil
    }

    enShare := make([][]byte, 0)
    err := rlp.DecodeBytes(dkgBytes, &enShare)
    if err != nil {
        return nil, errUnRlpEncryptShare
    }

    return BytesToEncryptShare(&enShare)
}

func IsJoinDKG2(db StateDB, epochId uint64, proposerId uint32) bool {
    hash := GetRBKeyHash(kindEns, epochId, proposerId)
    dkgBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if len(dkgBytes) == 0 {
        return false
    }

    return true
}

//
// more public functions
//
// active:  participate in all stages --- dkg1 dkg2 sign
```

```
func IsRBActive(db StateDB, epochId uint64, proposerId uint32) bool {
    hash := GetRBKeyHash(sigShareId[:], epochId, proposerId)
    payloadBytes := db.GetStateByteArray(randomBeaconPrecompileAddr, *hash)
    if payloadBytes == nil || len(payloadBytes) == 0 {
        return false
    }
    if IsJoinDKG2(db, epochId, proposerId) {
        return true
    }

    return false
}

//
// help function for serial
//
func UIntToByteSlice(num uint64) []byte {
    b := make([]byte, 8)
    binary.LittleEndian.PutUint64(b, num)
    return b
}
func UInt32ToByteSlice(num uint32) []byte {
    b := make([]byte, 4)
    binary.LittleEndian.PutUint32(b, num)
    return b
}

func ByteSliceToUInt(bs []byte) uint64 {
```

```
    return binary.LittleEndian.Uint64(bs)
}
func ByteSliceToUInt32(bs []byte) uint32 {
    return binary.LittleEndian.Uint32(bs)
}

//
// structures for params
//
type RbDKG1FlatTxPayload struct {
    EpochId uint64
    ProposerId uint32
    // 128 --
    Commit [][]byte
}

type RbDKG2FlatTxPayload struct {
    EpochId uint64
    ProposerId uint32
    // encrypt share
    EnShare [][]byte
    Proof []rbselection.DLEQproofFlat
}

type RbSIGTxPayload struct {
    EpochId uint64
    ProposerId uint32
    GSignShare *bn256.G1
```

```
}

//
// params bytes fields to object
//
type RbDKG1TxPayload struct {
    EpochId uint64
    ProposerId uint32
    Commit []*bn256.G2
}

type RbDKG2TxPayload struct {
    EpochId uint64
    ProposerId uint32
    // encrypt share
    EnShare []*bn256.G1
    Proof []rbselection.DLEQproof
}

//
// storage structures
//
type RbDKGTxPayloadPure struct {
    EpochId uint64
    ProposerId uint32
    EnShare []*bn256.G1
    Commit []*bn256.G2
}
```

```
type RbCijDataCollector struct {
    cij []*bn256.G2
    pk *bn256.G1
}

type RbSIGDataCollector struct {
    data *RbSIGTxPayload
    pk *bn256.G1
}

//
// storage parse functions
//
func BytesToCij(d *[][]byte) ([]*bn256.G2, error) {
    l := len(*d)
    cij := make([]*bn256.G2, l, l)
    g2s := make([]bn256.G2, l, l)
    for i := 0; i < l; i++ {
        //var g2 bn256.G2
        left, err := g2s[i].UnmarshalPure((*d)[i])
        if err != nil {
            return nil, err
        }

        if len(left) != 0 {
            return nil, errInvalidCommitBytes
        }
```

```
        cij[i] = &g2s[i]
    }

    return cij, nil
}

func BytesToEncryptShare(d *[][]byte) ([]*bn256.G1, error) {
    l := len(*d)
    ens := make([]*bn256.G1, l, l)
    g1s := make([]bn256.G1, l, l)
    for i := 0; i < l; i++ {
        left, err := g1s[i].UnmarshalPure((*d)[i])
        if err != nil {
            return nil, err
        }

        if len(left) != 0 {
            return nil, errInvalidEncryptShareBytes
        }

        ens[i] = &g1s[i]
    }

    return ens, nil
}

//
// params bytes fields convert functions
```

```
//
func Dkg1FlatToDkg1(d *RbDKG1FlatTxPayload) (*RbDKG1TxPayload, error) {
    var dkgParam RbDKG1TxPayload
    dkgParam.EpochId = d.EpochId
    dkgParam.ProposerId = d.ProposerId

    l := len(d.Commit)
    dkgParam.Commit = make([]*bn256.G2, l, l)
    g2s := make([]bn256.G2, l, l)
    for i := 0; i < l; i++ {
        //var g2 bn256.G2
        left, err := g2s[i].Unmarshal(d.Commit[i])
        if err != nil {
            return nil, err
        }

        if len(left) != 0 {
            return nil, errInvalidCommitBytes
        }

        dkgParam.Commit[i] = &g2s[i]
    }

    return &dkgParam, nil
}

func Dkg1ToDkg1Flat(d *RbDKG1TxPayload) *RbDKG1FlatTxPayload {
    var df RbDKG1FlatTxPayload
```

```
    df.EpochId = d.EpochId
    df.ProposerId = d.ProposerId

    l := len(d.Commit)

    df.Commit = make([][]byte, l)
    for i := 0; i < l; i++ {
        df.Commit[i] = d.Commit[i].Marshal()
    }

    return &df
}

func Dkg2FlatToDkg2(d *RbDKG2FlatTxPayload) (*RbDKG2TxPayload, error) {
    var dkg2Param RbDKG2TxPayload
    dkg2Param.EpochId = d.EpochId
    dkg2Param.ProposerId = d.ProposerId

    l := len(d.EnShare)
    dkg2Param.EnShare = make([]*bn256.G1, l, l)
    g1s := make([]bn256.G1, l, l)
    for i := 0; i < l; i++ {
        left, err := g1s[i].Unmarshal(d.EnShare[i])
        if err != nil {
            return nil, err
        }

        if len(left) != 0 {
            return nil, errInvalidEncryptShareBytes
```

```
        }

        dkg2Param.EnShare[i] = &g1s[i]
    }

    l = len(d.Proof)
    dkg2Param.Proof = make([]rbselection.DLEQproof, l, l)
    for i := 0; i < l; i++ {
        (&dkg2Param.Proof[i]).ProofFlatToProof(&d.Proof[i])
    }

    return &dkg2Param, nil
}

func Dkg2ToDkg2Flat(d *RbDKG2TxPayload) *RbDKG2FlatTxPayload {
    var df RbDKG2FlatTxPayload
    df.EpochId = d.EpochId
    df.ProposerId = d.ProposerId

    l := len(d.EnShare)
    df.EnShare = make([][]byte, l)
    for i := 0; i < l; i++ {
        df.EnShare[i] = d.EnShare[i].Marshal()
    }

    l = len(d.Proof)
    df.Proof = make([]rbselection.DLEQproofFlat, l)
    for i := 0; i < l; i++ {
        df.Proof[i] = rbselection.ProofToProofFlat(&d.Proof[i])
```

```
    }

    return &df
}

//
// in file help functions
//
// check time in the right stage, dkg1 --- 1k,2k slot, dkg2 --- 5k,6k slot, sig --- 8k,9k slot
func isValidEpochStage(epochId uint64, stage int, time uint64) bool {
    eid, sid := util.CalEpochSlotID(time)
    if epochId != eid {
        return false
    }

    ss, _, _ := GetRBStage(sid)
    if ss != stage {
        return false
    }

    return true
}

func isInRandomGroup(pks []bn256.G1, epochId uint64, proposerId uint32, address common.Address) bool {
    if len(pks) <= int(proposerId) || int(proposerId) < 0 {
        return false
    }
```

```
    ep := util.GetEpocherInst()
    if ep == nil {
        return false
    }
    pk1 := ep.GetProposerBn256PK(epochId, uint64(proposerId), address)
    if pk1 != nil {
        return bytes.Equal(pk1, pks[proposerId].Marshal())
    }
    return false
}

func buildError(err string, epochId uint64, proposerId uint32) error {
    return errors.New(fmt.Sprintf("%v epochId = %v, proposerId = %v ", err, epochId,
proposerId))
}

func logError(err error) error {
    log.SyslogErr(err.Error())
    return err
}

func getSignorsNum(epochId uint64, evm *EVM) uint32 {
    tmpKey := common.Hash{0}
    bs := evm.StateDB.GetStateByteArray(randomBeaconPrecompileAddr, tmpKey)
    if bs != nil {
        eid := ByteSliceToUInt(bs)
        if eid == epochId {
            num := ByteSliceToUInt32(bs[8:12])
            return num
```

```
        }
    }
    return 0
}

func setSignorsNum(epochId uint64, num uint32, evm *EVM) {
    tmpKey := common.Hash{0}
    dataBytes := make([]byte, 12)
    copy(dataBytes[0:], UIntToByteSlice(epochId))
    copy(dataBytes[8:], UInt32ToByteSlice(num))
    evm.StateDB.SetStateByteArray(randomBeaconPrecompileAddr, tmpKey, dataBytes)
}

//
// variables for mock
//
var getRBMVar = GetRBM
var isValidEpochStageVar = isValidEpochStage
var isInRandomGroupVar = isInRandomGroup
var getRBProposerGroupVar = getRBProposerGroup

//
// contract abi methods
// 成都链安 // 合约的 abi 方法
// dkg1: happens in 0~2k-1 slots, send the commits to chain
// 成都链安 // 第一阶段,发送委托到链上
func (c *RandomBeaconContract) dkg1(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {
```

```
    log.Debug("dkg1")

    dkg1FlatParam, err := validDkg1(evm.StateDB, evm.Time.Uint64(),
contract.CallerAddress, payload)

    if err != nil {

        return nil, err

    }


    eid := dkg1FlatParam.EpochId

    pid := dkg1FlatParam.ProposerId


    // save cij

    hash := GetRBKeyHash(kindCij, eid, pid)

    // 成都链安 // 计算 cij 承诺

    cijBytes, err := rlp.EncodeToBytes(dkg1FlatParam.Commit)

    if err != nil {

        return nil, logError(errRlpCij)

    }

    evm.StateDB.SetStateByteArray(randomBeaconPrecompileAddr, *hash, cijBytes)


    log.Debug("vm.dkg1", "dkg1Id", dkg1Id, "epochID", eid, "proposerId", pid,
"hash", hash.Hex())

    return nil, nil

}


// dkg2: happens in 5k~7k-1 slots, send the proof, enShare to chain

// 成都链安 // 第二阶段,发送证明,共享到链上

func (c *RandomBeaconContract) dkg2(payload []byte, contract *Contract, evm *EVM)
([]byte, error) {

    log.Debug("dkg2")

    dkg2FlatParam, err := validDkg2(evm.StateDB, evm.Time.Uint64(),
```

```
contract.CallerAddress, payload)
    if err != nil {
        return nil, err
    }

    eid := dkg2FlatParam.EpochId
    pid := dkg2FlatParam.ProposerId

    // save encrypt share
    hash := GetRBKeyHash(kindEns, eid, pid)
    encryptShareBytes, err := rlp.EncodeToBytes(dkg2FlatParam.EnShare)
    if err != nil {
        return nil, logError(errRlpEncryptShare)
    }
    evm.StateDB.SetStateByteArray(randomBeaconPrecompileAddr, *hash,
encryptShareBytes)

    log.Debug("vm.dkg2", "dkgId", dkg2Id, "epochID", eid, "proposerId", pid, "hash",
hash.Hex())
    return nil, nil
}

// sigShare: sign, happens in 8k~10k-1 slots, generate R if enough signers
// 成都链安 // 如果有足够多的 signer 就生成随机数 R
func (c *RandomBeaconContract) sigShare(payload []byte, contract *Contract, evm
*EVM) ([]byte, error) {
    log.Debug("sigShare")
    sigShareParam, pks, dkgData, err := validSigShare(evm.StateDB,
evm.Time.Uint64(), contract.CallerAddress, payload)
    if err != nil {
```

```
        return nil, err
    }

    eid := sigShareParam.EpochId
    pid := sigShareParam.ProposerId

    // save
    hash := GetRBKeyHash(sigShareId[:], eid, pid)
    evm.StateDB.SetStateByteArray(randomBeaconPrecompileAddr, *hash, payload)

    ////////////////
    // calc r if not exist
    sigNum := getSignorsNum(eid, evm) + 1
    setSignorsNum(eid, sigNum, evm)
    if uint(sigNum) >= posconfig.Cfg().RBThres {
        r, err := computeRandom(evm.StateDB, eid, dkgData, pks)
        if r != nil && err == nil {
            hashR := GetRBRKeyHash(eid + 1)
            evm.StateDB.SetStateByteArray(randomBeaconPrecompileAddr, *hashR,
r.Bytes())
            evm.StateDB.AddLog(&types.Log{
                Address: contract.Address(),
                Topics: []common.Hash{common.BigToHash(new(big.Int).SetUint64(eid)),
common.BytesToHash(r.Bytes())},
                // This is a non-consensus field, but assigned here because
                // core/state doesn't know the current block number.
                BlockNumber: evm.BlockNumber.Uint64(),
            })
        }
```

```
    }

    log.Debug("contract do sig end", "epochId", eid, "proposerId", pid)

    return nil, nil
}

//
// calc random
//
// compute random[epochId+1] by data of epoch[epochId]
// 成都链安 // 用当前 epoch id 去计算下一个 epochid 的随机数 r
func computeRandom(stateDB StateDB, epochId uint64, dkgData []RbCijDataCollector,
pks []bn256.G1) (*big.Int, error) {
    randomInt := GetStateR(stateDB, epochId+1)
    if randomInt != nil && randomInt.Cmp(big.NewInt(0)) != 0 {
        return randomInt, errors.New("random exist already")
    }

    if len(pks) == 0 {
        return nil, logError(errors.New("can't find random beacon proposer group"))
    }

    // collect DKG SIG
    sigData := make([]RbSIGDataCollector, 0)
    for id := range pks {
        sigDatum, err := GetSig(stateDB, epochId, uint32(id))
        if err == nil && sigDatum != nil {
            sigData = append(sigData, RbSIGDataCollector{sigDatum, &pks[id]})
```

```
        }
    }

    if uint(len(sigData)) < posconfig.Cfg().RBThres {
        return nil, logError(errors.New("insufficient sign proposer"))
    }

    gSignatureShare := make([]bn256.G1, len(sigData))
    xSig := make([]big.Int, len(sigData))
    for i, data := range sigData {
        gSignatureShare[i] = *data.data.GSignShare
        xSig[i].SetBytes(GetPolynomialX(data.pk, data.data.ProposerId))
    }

    // Compute the Output of Random Beacon
    gSignature := rbselection.LagrangeSig(gSignatureShare, xSig,
int(posconfig.Cfg().PolymDegree))
    random := crypto.Keccak256(gSignature.Marshal())

    // Verification Logic for the Output of Random Beacon
    // Computation of group public key
    nr := len(pks)
    c := make([]bn256.G2, nr)
    for i := 0; i < nr; i++ {
        c[i].ScalarBaseMult(big.NewInt(int64(0)))
        for j := 0; j < len(dkgData); j++ {
            c[i].Add(&c[i], dkgData[j].cij[i])
        }
    }
```

```
    xAll := make([]big.Int, nr)
    for i := 0; i < nr; i++ {
        xAll[i].SetBytes(GetPolynomialX(&pks[i], uint32(i)))
        xAll[i].Mod(&xAll[i], bn256.Order)
    }
    gPub := rbselection.LagrangePub(c, xAll, int(posconfig.Cfg().PolymDegree))

    // mG
    mBuf, err := getRBMVar(stateDB, epochId)
    if err != nil {
        return nil, logError(err)
    }

    m := new(big.Int).SetBytes(mBuf)
    mG := new(bn256.G1).ScalarBaseMult(m)

    // Verify using pairing
    pair1 := bn256.Pair(&gSignature, rbselection.Hbase)
    pair2 := bn256.Pair(mG, &gPub)
    if pair1.String() != pair2.String() {
        return nil, logError(errors.New("final pairing check failed"))
    }

    log.SyslogInfo("compute random success", "epochId", epochId+1, "random",
common.ToHex(random))
    return big.NewInt(0).SetBytes(random), nil
}
```

```
func GetValidDkg1Cnt(db StateDB, epochId uint64) uint64 {
    if db == nil {
        return 0
    }

    count := uint64(0)
    for i := 0; i < posconfig.RandomProperCount; i++ {
        c, err := GetCji(db, epochId, uint32(i))
        if err == nil && len(c) != 0 {
            count++
        }
    }

    return count
}

func GetValidDkg2Cnt(db StateDB, epochId uint64) uint64 {
    if db == nil {
        return 0
    }

    count := uint64(0)
    for i := 0; i < posconfig.RandomProperCount; i++ {
        c, err := GetEncryptShare(db, epochId, uint32(i))
        if err == nil && len(c) != 0 {
            count++
        }
    }
```

```
    return count
}

func GetValidSigCnt(db StateDB, epochId uint64) uint64 {
    if db == nil {
        return 0
    }

    count := uint64(0)
    for i := 0; i < posconfig.RandomProperCount; i++ {
        c, err := GetSig(db, epochId, uint32(i))
        if err == nil && c != nil {
            count++
        }
    }

    return count
}
```

4、 slot_leader_select_contracts.go

```
// 成都链安 // file:go-wanchain/core/vm/pos_control_contracts.go
package vm

import (
    "bytes"
    "crypto/ecdsa"
    "errors"
```

```
    "fmt"
    "math/big"
    "strings"
    "time"

    "github.com/wanchain/go-wanchain/pos/uleaderselection"
    "github.com/wanchain/go-wanchain/rlp"

    "github.com/wanchain/go-wanchain/pos/posconfig"
    "github.com/wanchain/go-wanchain/pos/posdb"
    "github.com/wanchain/go-wanchain/pos/util"
    "github.com/wanchain/go-wanchain/pos/util/convert"

    "github.com/wanchain/go-wanchain/functrace"

    "github.com/wanchain/go-wanchain/common"
    "github.com/wanchain/go-wanchain/crypto"
    "github.com/wanchain/go-wanchain/log"

    "github.com/wanchain/go-wanchain/accounts/abi"
    "github.com/wanchain/go-wanchain/core/types"
)

const (
    SlotLeaderStag1 = "slotLeaderStag1"
    SlotLeaderStag2 = "slotLeaderStag2"

    SlotLeaderStag1Indexes = "slotLeaderStag1Indexes"
    SlotLeaderStag2Indexes = "slotLeaderStag2Indexes"
```

```
)

var (
    slotLeaderSCDef = `[
            {
                "constant": false,
                "type": "function",
                "inputs": [
                    {
                        "name": "data",
                        "type": "string"
                    }
                ],
                "name": "slotLeaderStage1MiSave",
                "outputs": [
                    {
                        "name": "data",
                        "type": "string"
                    }
                ]
            },
            {
                "constant": false,
                "type": "function",
                "inputs": [
                    {
                        "name": "data",
                        "type": "string"
```

```
                }
            ],
            "name": "slotLeaderStage2InfoSave",
            "outputs": [
                {
                    "name": "data",
                    "type": "string"
                }
            ]
        }
    ]`
    slotLeaderAbi, errSlotLeaderSCInit =
abi.JSON(strings.NewReader(slotLeaderSCDef))
    stgOneIdArr, stgTwoIdArr [4]byte

    scCallTimes = "SLOT_LEADER_SC_CALL_TIMES"
)

// 成都链安 // 错误定义
var (
    ErrEpochID = errors.New("EpochID is not valid")
    ErrIllegalSender = errors.New("sender is not in epoch leaders ")
    ErrInvalidLocalPublicKey = errors.New("getLocalPublicKey error, do not found
unlock address")
    ErrInvalidPreEpochLeaders = errors.New("can not found pre epoch leaders return
epoch 0")
    ErrInvalidGenesisPk = errors.New("invalid GenesisPK hex string")
    ErrSlotLeaderGroupNotReady = errors.New("slot leaders group not ready")
    ErrSlotIDOutOfRange = errors.New("slot id index out of range")
    ErrPkNotInCurrentEpochLeadersGroup = errors.New("local public key is not in
```

```
current Epoch leaders")

    ErrInvalidRandom = errors.New("get random message error")

    ErrNotOnCurve = errors.New("not on curve")

    ErrTx1AndTx2NotConsistent = errors.New("stageOneMi is not equal
sageTwoAlphaPki")

    ErrEpochLeaderNotReady = errors.New("epoch leaders are not ready")

    ErrNoTx2TransInDB = errors.New("tx2 is not in db")

    ErrCollectTxData = errors.New("collect tx data error")

    ErrRlpUnpackErr = errors.New("RlpUnpackDataForTx error")

    ErrNoTx1TransInDB = errors.New("GetStg1StateDbInfo: Found not data of key")

    ErrVerifyStg1Data = errors.New("stg1 data get from StateDb verified failed")

    ErrDleqProof = errors.New("VerifyDleqProof false")

    ErrInvalidTxLen = errors.New("len(mi)==0 or len(alphaPkis) is not right")

    ErrInvalidTx1Range = errors.New("slot leader tx1 is not in invalid range")

    ErrInvalidTx2Range = errors.New("slot leader tx2 is not in invalid range")

    ErrInvalidProof = errors.New("proof is bigZero")

    ErrPowRcvPosTrans = errors.New("pow phase receive pos protocol trans")

    ErrDuplicateStg1 = errors.New("stage one transaction exists")

    ErrDuplicateStg2 = errors.New("stage two transaction exists")

)
```

// 成都链安 // 每个 Slot 内，在 Epoch Leader 组中等概率选出一名出块者（这里的选择不再受权益权重等因素影响），即 Slot Leader。选择过程基于 Random Number Proposer 组产生的随机数

// 成都链安 // slot_leader 的选举合约(slot_leader 就是出块者)

```
func init() {

    if errSlotLeaderSCInit != nil {

        panic("err in slot leader sc initialize :" + errSlotLeaderSCInit.Error())

    }
```

```
    // 成都链安 // 转化函数名称
    stgOneIdArr, _ = GetStage1FunctionID(slotLeaderSCDef)
    stgTwoIdArr, _ = GetStage2FunctionID(slotLeaderSCDef)
}

type slotLeaderSC struct {
}

func (c *slotLeaderSC) RequiredGas(input []byte) uint64 {
    return 0
}
// 成都链安 // 入口函数
func (c *slotLeaderSC) Run(in []byte, contract *Contract, evm *EVM) ([]byte, error)
{
    functrace.Enter()
    log.Debug("slotLeaderSC run is called")

    if len(in) < 4 {
        return nil, errParameters
    }

    var methodId [4]byte
    copy(methodId[:], in[:4])
    var from common.Address
    from = contract.CallerAddress
    // 成都链安 // 执行判断，选择进入不同函数
    if methodId == stgOneIdArr {
```

```
        vldReset := validStg1Reset(evm.StateDB, from, in[:], evm.Time.Uint64())
        vldService := validStg1Service(from, in[:])

        if !(vldReset && vldService) {
            return nil, errors.New("ValidTx stg1")
        }
        return handleStgOne(in[:], contract, evm) //Do not use [4:] because it has
do it in function
    } else if methodId == stgTwoIdArr {
        vldReset := validStg2Reset(evm.StateDB, from, in[:], evm.Time.Uint64())
        vldService := validStg2Service(evm.StateDB, from, in[:])

        if !(vldReset && vldService) {
            return nil, errors.New("ValidTx stg2")
        }
        return handleStgTwo(in[:], contract, evm) //Do not use [4:] because it has
do it in function
    }

    functrace.Exit()
    log.SyslogErr("slotLeaderSC:Run", "", errMethodId.Error())
    return nil, errMethodId
}
```

// 成都链安 // 交易验证

```
func (c *slotLeaderSC) ValidTx(stateDB StateDB, signer types.Signer, tx
*types.Transaction) error {

    if posconfig.FirstEpochId == 0 {
```

```
        log.SyslogErr("slotLeaderSC:ValidTx", "", ErrPowRcvPosTrans.Error())
        return ErrPowRcvPosTrans
    }

    from, err := types.Sender(signer, tx)
    if err != nil {
        return err
    }

    payload := tx.Data()
    if len(payload) < 4 {
        return errParameters
    }

    var methodId [4]byte
    copy(methodId[:], payload[:4])

    // 成都链安 // 选择不同的验证阶段
    if methodId == stgOneIdArr {
        vldReset := validStg1Reset(stateDB, from, payload,
uint64(time.Now().Unix()))
        vldService := validStg1Service(from, payload)

        if vldReset && vldService {
            return nil
        } else {
            return errors.New("ValidTx stg1")
        }
    } else if methodId == stgTwoIdArr {
```

```
        vldReset := validStg2Reset(stateDB, from, payload,
uint64(time.Now().Unix()))
        vldService := validStg2Service(stateDB, from, payload)

        if vldReset && vldService {
            return nil
        } else {
            return errors.New("ValidTx stg2")
        }
    } else {
        log.SyslogErr("slotLeaderSC:ValidTx", "", errMethodId.Error())
        return errMethodId
    }
}

// valid common data
// 成都链安 // 验证公共的数据，包括参数个数等
func ValidPosELTx(stateDB StateDB, from common.Address, payload []byte) error {

    if len(payload) < 4 {
        return errParameters
    }

    var methodId [4]byte
    copy(methodId[:], payload[:4])

    if methodId == stgOneIdArr {
        if validStg1Reset(stateDB, from, payload, uint64(time.Now().Unix())) {
            return nil
```

```
        } else {
            return errors.New("ValidPosELTx stage1 error")
        }
    } else if methodId == stgTwoIdArr {
        if validStg2Reset(stateDB, from, payload, uint64(time.Now().Unix())) {
            return nil
        } else {
            return errors.New("ValidPosELTx stage2 error")
        }
    } else {
        log.SyslogErr("slotLeaderSC:ValidTx", "", errMethodId.Error())
        return errMethodId
    }
}


// 成都链安 // 验证 stg1reset 阶段
func validStg1Reset(stateDB StateDB, from common.Address, payload []byte, time
uint64) bool {
    epochIDBuf, selfIndexBuf, err := RlpGetStage1IDFromTx(payload)
    if err != nil {
        log.Error("validStg1Reset failed")
        return false
    }

    // epoch stage
    invalidStage := isInValidStage(convert.BytesToUint64(epochIDBuf), time,
posconfig.Sma1Start, posconfig.Sma1End)
    if !invalidStage {
        return false
```

```
    }

    // duplicated

    dup := isDuplicateTrans(stateDB, convert.BytesToUint64(epochIDBuf),
convert.BytesToUint64(selfIndexBuf), SlotLeaderStag1)

    if dup {

        return false

    }


    return true

}


// 成都链安 // 验证 stg2reset 阶段

func validStg2Reset(stateDB StateDB, from common.Address, payload []byte, time
uint64) bool {

    epochIDBuf, selfIndexBuf, err := RlpGetStage2IDFromTx(payload)

    if err != nil {

        log.Error("validStg1Reset failed")

        return false

    }


    // epoch stage

    invalidStage := isInValidStage(convert.BytesToUint64(epochIDBuf), time,
posconfig.Sma2Start, posconfig.Sma2End)

    if !invalidStage {

        return false

    }


    //duplicated

    dup := isDuplicateTrans(stateDB, convert.BytesToUint64(epochIDBuf),
convert.BytesToUint64(selfIndexBuf), SlotLeaderStag2)
```

```
    if dup {
        return false
    }

    return true
}

// 成都链安 // 验证 stg1 的 service 阶段
func validStg1Service(from common.Address, payload []byte) bool {
    epochIDBuf, selfIndexBuf, err := RlpGetStage1IDFromTx(payload[:])
    if err != nil {
        log.Error("validStg1Service failed")
        return false
    }

    if !InEpochLeadersOrNotByAddress(convert.BytesToUint64(epochIDBuf),
convert.BytesToUint64(selfIndexBuf), from) {
        log.SyslogErr(ErrIllegalSender.Error())
        return false
    }

    _, _, _, err = RlpUnpackStage1DataForTx(payload[:])
    if err != nil {
        log.Error(err.Error())
        return false
    }
    return true
}
```

```
// 成都链安 // 验证 stg2 的 service 阶段

func validStg2Service(stateDB StateDB, from common.Address, payload []byte) bool {

    epochID, selfIndex, _, alphaPkis, proofs, err :=
RlpUnpackStage2DataForTx(payload[:])

    if err != nil {

        log.Error("validTxStg2:RlpUnpackStage2DataForTx failed")

        return false

    }


    if !InEpochLeadersOrNotByAddress(epochID, selfIndex, from) {

        log.SyslogErr("validTxStg2:InEpochLeadersOrNotByAddress failed")

        return false

    }


    for _, proof := range proofs {

        if proof.Cmp(bigZero) == 0 {

            log.SyslogErr("validTxStg2ByData", "proofs", ErrInvalidProof.Error())

            return false

        }

    }


    //log.Info("validTxStg2 success")


    mi, err := GetStg1StateDbInfo(stateDB, epochID, selfIndex)

    if err != nil {

        log.Error("validTxStg2", "GetStg1StateDbInfo error", err.Error())

        return false

    }
```

```
    //mi
    if len(mi) == 0 || len(alphaPkis) != posconfig.EpochLeaderCount {
        log.SyslogErr("validTxStg2", "len(mi)==0 or len(alphaPkis) not equal",
len(alphaPkis))
        return false
    }
    if !util.PkEqual(crypto.ToECDSAPub(mi), alphaPkis[selfIndex]) {
        log.SyslogErr("validTxStg2", "mi is not equal alphaPkis[index]", selfIndex)
        return false
    }
    //Dleq

    ep := util.GetEpocherInst()
    if ep == nil {
        log.Error(ErrEpochID.Error())
        return false
    }
    buff := ep.GetEpochLeaders(epochID)
    if buff == nil || len(buff) == 0 {
        log.SyslogWarning("epoch leader is not ready at validStg2Service",
"epochID", epochID)
        return false
    }
    epochLeaders := make([]*ecdsa.PublicKey, len(buff))
    for i := 0; i < len(buff); i++ {
        epochLeaders[i] = crypto.ToECDSAPub(buff[i])
    }

    if !(uleaderselection.VerifyDleqProof(epochLeaders, alphaPkis, proofs)) {
```

```
        log.SyslogErr("validTxStg2", "VerifyDleqProof false self Index", selfIndex)
        return false
    }
    return true
}

// 成都链安 // 处理 stg1
func handleStgOne(in []byte, contract *Contract, evm *EVM) ([]byte, error) {
    log.Debug("slotLeaderSC handleStgOne is called")

    epochIDBuf, selfIndexBuf, err := RlpGetStage1IDFromTx(in)
    if err != nil {
        return nil, err
    }

    keyHash := GetSlotLeaderStage1KeyHash(epochIDBuf, selfIndexBuf)

    evm.StateDB.SetStateByteArray(slotLeaderPrecompileAddr, keyHash, in)

    err = updateSlotLeaderStageIndex(evm, epochIDBuf, SlotLeaderStag1Indexes,
convert.BytesToUint64(selfIndexBuf))

    if err != nil {
        return nil, err
    }

    addSlotScCallTimes(convert.BytesToUint64(epochIDBuf))

    log.Debug(fmt.Sprintf("handleStgOne save data addr:%s, key:%s, data len:%d",
```

```
slotLeaderPrecompileAddr.Hex(),
        keyHash.Hex(), len(in)))
    log.Debug("handleStgOne save", "epochID", convert.BytesToUint64(epochIDBuf),
"selfIndex",
        convert.BytesToUint64(selfIndexBuf))

    return nil, nil
}

// 成都链安 // 处理 stg2
func handleStgTwo(in []byte, contract *Contract, evm *EVM) ([]byte, error) {

    epochIDBuf, selfIndexBuf, err := RlpGetStage2IDFromTx(in)
    if err != nil {
        return nil, err
    }

    keyHash := GetSlotLeaderStage2KeyHash(epochIDBuf, selfIndexBuf)

    evm.StateDB.SetStateByteArray(slotLeaderPrecompileAddr, keyHash, in)

    err = updateSlotLeaderStageIndex(evm, epochIDBuf, SlotLeaderStag2Indexes,
convert.BytesToUint64(selfIndexBuf))

    if err != nil {
        return nil, err
    }
    addSlotScCallTimes(convert.BytesToUint64(epochIDBuf))

    log.Debug(fmt.Sprintf("handleStgTwo save data addr:%s, key:%s, data len:%d",
```

```
slotLeaderPrecompileAddr.Hex(),
        keyHash.Hex(), len(in)))
    log.Debug("handleStgTwo save", "epochID", convert.BytesToUint64(epochIDBuf), "selfIndex",
        convert.BytesToUint64(selfIndexBuf))

    functrace.Exit()
    return nil, nil
}

// GetSlotLeaderStage2KeyHash use to get SlotLeader Stage 1 KeyHash by epochid and selfindex
// 成都链安 // 通过 epochid 和 selfindex 去获取 slotleader stg2 阶段的 keyhash
func GetSlotLeaderStage2KeyHash(epochID, selfIndex []byte) common.Hash {
    return getSlotLeaderStageKeyHash(epochID, selfIndex, SlotLeaderStag2)
}

func GetSlotLeaderStage2IndexesKeyHash(epochID []byte) common.Hash {
    return getSlotLeaderStageIndexesKeyHash(epochID, SlotLeaderStag2Indexes)
}

// GetSlotLeaderSCAddress can get the precompile contract address
// 成都链安 // 获取预编译合约的地址
func GetSlotLeaderSCAddress() common.Address {
    return slotLeaderPrecompileAddr
}

// GetSlotLeaderScAbiString can get the precompile contract Define string
func GetSlotLeaderScAbiString() string {
```

```
    return slotLeaderSCDef
}

// GetSlotScCallTimes can get this precompile contract called times
func GetSlotScCallTimes(epochID uint64) uint64 {
    buf, err := posdb.GetDb().Get(epochID, scCallTimes)
    if err != nil {
        return 0
    } else {
        return convert.BytesToUint64(buf)
    }
}

// GetSlotLeaderStage1KeyHash use to get SlotLeader Stage 1 KeyHash by epoch id and
self index
// 成都链安 // 获取一阶段的 keyhash
func GetSlotLeaderStage1KeyHash(epochID, selfIndex []byte) common.Hash {
    return getSlotLeaderStageKeyHash(epochID, selfIndex, SlotLeaderStag1)
}

func GetStage1FunctionID(abiString string) ([4]byte, error) {
    var slotStage1ID [4]byte

    abi, err := util.GetAbi(abiString)
    if err != nil {
        log.SyslogErr("slotLeaderSC", "GetStage1FunctionID:GetAbi", err.Error())
        return slotStage1ID, err
    }
```

```
    copy(slotStage1ID[:], abi.Methods["slotLeaderStage1MiSave"].Id())

    return slotStage1ID, nil
}

func GetStage2FunctionID(abiString string) ([4]byte, error) {
    var slotStage2ID [4]byte

    abi, err := util.GetAbi(abiString)
    if err != nil {
        log.SyslogErr("slotLeaderSC", "GetStage2FunctionID:GetAbi", err.Error())
        return slotStage2ID, err
    }

    copy(slotStage2ID[:], abi.Methods["slotLeaderStage2InfoSave"].Id())

    return slotStage2ID, nil
}

// PackStage1Data can pack stage1 data into abi []byte for tx payload
// 成都链安 // 把 stg1 阶段的数据打包到 tx 中
func PackStage1Data(input []byte, abiString string) ([]byte, error) {
    id, err := GetStage1FunctionID(abiString)
    outBuf := make([]byte, len(id)+len(input))
    copy(outBuf[:4], id[:])
    copy(outBuf[4:], input[:])
    return outBuf, err
}
```

```
func InEpochLeadersOrNotByAddress(epochID uint64, selfIndex uint64, senderAddress
common.Address) bool {

    ep := util.GetEpocherInst()

    if ep == nil {

        return false

    }

    epochLeaders := ep.GetEpochLeaders(epochID)

    if len(epochLeaders) != posconfig.EpochLeaderCount {

        log.SyslogWarning("epoch leader is not ready at
InEpochLeadersOrNotByAddress", "epochID", epochID)

        return false

    }


    if int64(selfIndex) < 0 || int64(selfIndex) >= posconfig.EpochLeaderCount {

        log.SyslogErr("InEpochLeadersOrNotByAddress", "selfIndex out of range",
int64(selfIndex))

        return false

    }


    if crypto.PubkeyToAddress(*crypto.ToECDSAPub(epochLeaders[selfIndex])).Hex() ==
senderAddress.Hex() {

        return true

    }


    addr1 :=
crypto.PubkeyToAddress(*crypto.ToECDSAPub(epochLeaders[selfIndex])).Hex()

    addr2 := senderAddress.Hex()


    log.Info("epochleader not match", "epochleader array address", addr1, "sender",
addr2)
```

```
    return false
}

type stage1Data struct {
    EpochID uint64
    SelfIndex uint64
    MiCompress []byte
}

// RlpPackStage1DataForTx
func RlpPackStage1DataForTx(epochID uint64, selfIndex uint64, mi *ecdsa.PublicKey,
abiString string) ([]byte, error) {
    pkBuf, err := util.CompressPk(mi)
    if err != nil {
        return nil, err
    }
    data := &stage1Data{
        EpochID: epochID,
        SelfIndex: selfIndex,
        MiCompress: pkBuf,
    }

    buf, err := rlp.EncodeToBytes(data)
    if err != nil {
        log.SyslogErr("RlpPackStage1DataForTx", "rlp.EncodeToBytes", err.Error())
        return nil, err
    }

    return PackStage1Data(buf, abiString)
```

```
}


// RlpUnpackStage1DataForTx
// 成都链安 // 从 tx 中解包出 stg1 的数据
func RlpUnpackStage1DataForTx(input []byte) (epochID uint64, selfIndex uint64, mi
*ecdsa.PublicKey, err error) {
    var data *stage1Data


    buf := input[4:]


    err = rlp.DecodeBytes(buf, &data)
    if err != nil {
        log.SyslogErr("RlpUnpackStage1DataForTx", "rlp.DecodeBytes", err.Error())
        return
    }


    epochID = data.EpochID
    selfIndex = data.SelfIndex
    // UncompressPK has verified the point on curve or not.
    mi, err = util.UncompressPk(data.MiCompress)
    if err != nil {
        log.SyslogErr("RlpUnpackStage1DataForTx", "util.UncompressPk", err.Error())
    }
    return
}


// RlpGetStage1IDFromTx
func RlpGetStage1IDFromTx(input []byte) (epochIDBuf []byte, selfIndexBuf []byte, err
error) {
```

```
    var data *stage1Data

    buf := input[4:]

    err = rlp.DecodeBytes(buf, &data)
    if err != nil {
        log.SyslogErr("RlpGetStage1IDFromTx", "rlp.DecodeBytes", err.Error())
        return
    }
    epochIDBuf = convert.Uint64ToBytes(data.EpochID)
    selfIndexBuf = convert.Uint64ToBytes(data.SelfIndex)
    return
}

type stage2Data struct {
    EpochID uint64
    SelfIndex uint64
    SelfPk []byte
    AlphaPki [][]byte
    Proof []*big.Int
}

func RlpPackStage2DataForTx(epochID uint64, selfIndex uint64, selfPK
*ecdsa.PublicKey, alphaPki []*ecdsa.PublicKey,
    proof []*big.Int, abiString string) ([]byte, error) {
    pk, err := util.CompressPk(selfPK)
    if err != nil {
        log.SyslogErr("RlpPackStage2DataForTx", "util.CompressPk", err.Error())
        return nil, err
```

```
    }

    pks := make([][]byte, len(alphaPki))
    for i := 0; i < len(alphaPki); i++ {
        pks[i], err = util.CompressPk(alphaPki[i])
        if err != nil {
            log.SyslogErr("RlpPackStage2DataForTx", "util.CompressPk", err.Error())
            return nil, err
        }
    }

    data := &stage2Data{
        EpochID: epochID,
        SelfIndex: selfIndex,
        SelfPk: pk,
        AlphaPki: pks,
        Proof: proof,
    }

    buf, err := rlp.EncodeToBytes(data)
    if err != nil {
        log.SyslogErr("RlpPackStage2DataForTx", "rlp.EncodeToBytes", err.Error())
        return nil, err
    }

    id, err := GetStage2FunctionID(abiString)
    if err != nil {
        return nil, err
```

```
    }

    outBuf := make([]byte, len(id)+len(buf))
    copy(outBuf[:4], id[:])
    copy(outBuf[4:], buf[:])

    return outBuf, nil
}

func RlpUnpackStage2DataForTx(input []byte) (epochID uint64, selfIndex uint64,
selfPK *ecdsa.PublicKey,
    alphaPki []*ecdsa.PublicKey, proof []*big.Int, err error) {
    inputBuf := input[4:]

    var data stage2Data
    err = rlp.DecodeBytes(inputBuf, &data)
    if err != nil {
        log.SyslogErr("RlpUnpackStage2DataForTx", "rlp.DecodeBytes", err.Error())
        return
    }

    epochID = data.EpochID
    selfIndex = data.SelfIndex
    selfPK, err = util.UncompressPk(data.SelfPk)
    if err != nil {
        log.SyslogErr("RlpUnpackStage2DataForTx", "util.UncompressPk", err.Error())
        return
    }
```

```
    alphaPki = make([]*ecdsa.PublicKey, len(data.AlphaPki))
    for i := 0; i < len(data.AlphaPki); i++ {
        alphaPki[i], err = util.UncompressPk(data.AlphaPki[i])
        if err != nil {
            log.SyslogErr("RlpUnpackStage2DataForTx", "util.UncompressPk",
err.Error())
            return
        }
    }

    proof = data.Proof
    return
}

func RlpGetStage2IDFromTx(input []byte) (epochIDBuf []byte, selfIndexBuf []byte, err
error) {
    inputBuf := input[4:]

    var data stage2Data
    err = rlp.DecodeBytes(inputBuf, &data)
    if err != nil {
        log.SyslogErr("RlpGetStage2IDFromTx", "rlp.DecodeBytes", err.Error())
        return
    }

    epochIDBuf = convert.Uint64ToBytes(data.EpochID)
    selfIndexBuf = convert.Uint64ToBytes(data.SelfIndex)
    return
}
```

```
func GetStage2TxAlphaPki(stateDb StateDB, epochID uint64, selfIndex uint64)
(alphaPkis []*ecdsa.PublicKey,
    proofs []*big.Int, err error) {

    slotLeaderPrecompileAddr := GetSlotLeaderSCAddress()

    keyHash := GetSlotLeaderStage2KeyHash(convert.Uint64ToBytes(epochID),
convert.Uint64ToBytes(selfIndex))

    data := stateDb.GetStateByteArray(slotLeaderPrecompileAddr, keyHash)
    if data == nil {
        log.Debug(fmt.Sprintf("try to get stateDB addr:%s, key:%s",
slotLeaderPrecompileAddr.Hex(), keyHash.Hex()))
        log.SyslogErr(fmt.Sprintf("try to get stateDB addr:%s, key:%s",
slotLeaderPrecompileAddr.Hex(), keyHash.Hex()))
        return nil, nil, ErrNoTx2TransInDB
    }

    epID, slfIndex, _, alphaPki, proof, err := RlpUnpackStage2DataForTx(data)
    if err != nil {
        return nil, nil, err
    }

    if epID != epochID || slfIndex != selfIndex {
        log.SyslogErr("GetStage2TxAlphaPki", "error", ErrRlpUnpackErr.Error())
        return nil, nil, ErrRlpUnpackErr
    }

    return alphaPki, proof, nil
```

```
}


func GetStg1StateDbInfo(stateDb StateDB, epochID uint64, index uint64) (mi []byte,
err error) {

    slotLeaderPrecompileAddr := GetSlotLeaderSCAddress()

    keyHash := GetSlotLeaderStage1KeyHash(convert.Uint64ToBytes(epochID),
convert.Uint64ToBytes(index))


    // Read and Verify

    readBuf := stateDb.GetStateByteArray(slotLeaderPrecompileAddr, keyHash)

    if readBuf == nil {

        log.SyslogErr("GetStg1StateDbInfo", "error", ErrNoTx1TransInDB.Error())

        return nil, ErrNoTx1TransInDB

    }


    epID, idxID, miPoint, err := RlpUnpackStage1DataForTx(readBuf)

    if err != nil {

        return nil, ErrRlpUnpackErr

    }

    mi = crypto.FromECDSAPub(miPoint)

    //pk and mi is 65 bytes length


    if epID == epochID &&

        idxID == index &&

        err == nil {

        return

    }


    return nil, ErrVerifyStg1Data
```

```
}

func getSlotLeaderStageIndexesKeyHash(epochID []byte, slotLeaderStageIndexes string)
common.Hash {
    var keyBuf bytes.Buffer
    keyBuf.Write(epochID)
    keyBuf.Write([]byte(slotLeaderStageIndexes))
    return crypto.Keccak256Hash(keyBuf.Bytes())
}

func getSlotLeaderStageKeyHash(epochID, selfIndex []byte, slotLeaderStage string)
common.Hash {
    var keyBuf bytes.Buffer
    keyBuf.Write(epochID)
    keyBuf.Write(selfIndex)
    keyBuf.Write([]byte(slotLeaderStage))
    return crypto.Keccak256Hash(keyBuf.Bytes())
}

func addSlotScCallTimes(epochID uint64) error {
    buf, err := posdb.GetDb().Get(epochID, scCallTimes)
    times := uint64(0)
    if err != nil {
        if err.Error() != "leveldb: not found" {
            log.SyslogErr("addSlotScCallTimes", "error", err.Error())
            return err
        }
    } else {
        times = convert.BytesToUint64(buf)
```

```
    }

    times++

    posdb.GetDb().Put(epochID, scCallTimes, convert.Uint64ToBytes(times))
    return nil
}

func isInValidStage(epochID uint64, time uint64, kStart uint64, kEnd uint64) bool {
    //eid, sid := util.CalEpochSlotID(evm.Time.Uint64())
    eid, sid := util.CalEpochSlotID(time)
    if epochID != eid {
        log.SyslogWarning("Tx epochID is not current epoch", "epochID", eid,
"slotID", sid, "currentEpochID", epochID)

        return false
    }

    if sid > kEnd || sid < kStart {
        log.SyslogWarning("Tx is out of valid stage range", "epochID", eid,
"slotID", sid, "rangeStart", kStart,
            "rangeEnd", kEnd)

        return false
    }

    return true
}

func isDuplicateTrans(stateDb StateDB, epochID uint64, index uint64, stageName
```

```
string) bool {

    slotLeaderPrecompileAddr := GetSlotLeaderSCAddress()

    var keyHash common.Hash

    if stageName == SlotLeaderStag1 {

        keyHash = GetSlotLeaderStage1KeyHash(convert.Uint64ToBytes(epochID),
convert.Uint64ToBytes(index))

        data := stateDb.GetStateByteArray(slotLeaderPrecompileAddr, keyHash)

        if data == nil {

            return false

        }

        log.SyslogWarning("isDuplicateTrans", "epochID", epochID, "index", index,
"stageName", stageName)

        return true

    }


    if stageName == SlotLeaderStag2 {

        keyHash = GetSlotLeaderStage2KeyHash(convert.Uint64ToBytes(epochID),
convert.Uint64ToBytes(index))

        data := stateDb.GetStateByteArray(slotLeaderPrecompileAddr, keyHash)

        if data == nil {

            return false

        }

        log.SyslogWarning("isDuplicateTrans", "epochID", epochID, "index", index,
"stageName", stageName)

        return true

    }

    return false

}


// 成都链安 // 更新 slotleader 的索引
```

```
func updateSlotLeaderStageIndex(evm *EVM, epochID []byte, slotLeaderStageIndexes
string, index uint64) error {

    var sendtrans [posconfig.EpochLeaderCount]bool

    var sendtransGet [posconfig.EpochLeaderCount]bool


    key := getSlotLeaderStageIndexesKeyHash(epochID, slotLeaderStageIndexes)

    bytes := evm.StateDB.GetStateByteArray(slotLeaderPrecompileAddr, key)


    if len(bytes) == 0 {

        sendtrans[index] = true

        value, err := rlp.EncodeToBytes(sendtrans)

        if err != nil {

            log.SyslogErr("updateSlotLeaderStageIndex", "rlp.EncodeToBytes",
err.Error())

            return err

        }

        // 成都链安 // 改变数据库状态

        evm.StateDB.SetStateByteArray(slotLeaderPrecompileAddr, key, value)


        log.Debug("updateSlotLeaderStageIndex", "key", key, "value", sendtrans)

    } else {

        err := rlp.DecodeBytes(bytes, &sendtransGet)

        if err != nil {

            log.SyslogErr("updateSlotLeaderStageIndex", "rlp.DecodeBytes",
err.Error())

            return err


            sendtransGet[index] = true

            value, err := rlp.EncodeToBytes(sendtransGet)
```

```
            if err != nil {
                log.SyslogErr("updateSlotLeaderStageIndex", "rlp.EncodeToBytes",
err.Error())
                return err
            }
            evm.StateDB.SetStateByteArray(slotLeaderPrecompileAddr, key, value)
            log.Debug("updateSlotLeaderStageIndex", "key", key, "value",
sendtransGet)
        }
        return nil
    }
}

func GetValidSMA1Cnt(db StateDB, epochId uint64) uint64 {
    return getValidIndexCnt(db, epochId, SlotLeaderStag1Indexes)
}

func GetValidSMA2Cnt(db StateDB, epochId uint64) uint64 {
    return getValidIndexCnt(db, epochId, SlotLeaderStag2Indexes)
}

func getValidIndexCnt(db StateDB, epochId uint64, indexKey string) uint64 {
    var sendtransGet [posconfig.EpochLeaderCount]bool
    epochIDBuf := convert.Uint64ToBytes(epochId)

    key := getSlotLeaderStageIndexesKeyHash(epochIDBuf, indexKey)
    bytes := db.GetStateByteArray(slotLeaderPrecompileAddr, key)
    if len(bytes) == 0 {
        return 0
```

```
    }

    err := rlp.DecodeBytes(bytes, &sendtransGet)
    if err != nil {
        log.SyslogErr("GetValidSMA1Cnt, rlp decode fail", "err", err.Error())
        return 0
    }

    cnt := uint64(0)
    for i := range sendtransGet {
        if sendtransGet[i] {
            cnt++
        }
    }

    return cnt
}

func GetSlStage(slotId uint64) uint64 {
    if slotId <= posconfig.Sma1End {
        return 1
    } else if slotId < posconfig.Sma2Start {
        return 2
    } else if slotId <= posconfig.Sma2End {
        return 3
    } else if slotId < posconfig.Sma3Start {
        return 4
    } else if slotId <= posconfig.Sma3End {
```

```
        return 5
    } else {
        return 6
    }
}
```

成都链安
BEOSIN

官方网址

https:// lianantech.com

电子邮箱

vaas@lianantech.com

微信公众号