成都链安
BEOSIN

公链安全审计报告

成都链安 BEOSIN

审计编号：202105100040

审计公链名称：TOPChain

审计开始日期：2021.04.01

审计完成日期：2021.05.10

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	审计类型	审计项	审计子项	审计结果
1	节点安全	RPC 接口审计	RPC 功能实现审计	**通过**
			RPC 接口权限审计	**通过**
		节点测试	畸形数据测试	**通过**
			节点缓冲区溢出攻击审计	**通过**
			DDoS 攻击审计	**通过**
2	钱包及账户安全	私钥/助记词审计	生成算法审计	**通过**
			存储安全审计	**通过**
			使用安全审计	**通过**
3	交易模型安全	交易处理逻辑审计	交易或收据重放攻击审计	**通过**
			畸形交易、伪造交易、重复交易攻击审计	**通过**
			粉尘攻击审计	**通过**
			交易泛滥攻击审计	**通过**
			双花、Over Spend 攻击审计	**通过**
		其他交易安全审计	交易延展性攻击审计	**通过**
			假充值攻击审计	**通过**
			命令行转账方法审计	**通过**
4	共识安全	共识机制设计	Leader 选举和 VRF 机制审计	**通过**
			Shard 节点轮换机制审计	**通过**

			共识算法(包括 xBFT)审计	**通过**
		共识验证实现	是否能使用少于预期成本构造合法区块	**通过**
5	签名安全	签名校验审计	多重签名校验安全	**通过**
			非法签名攻击	**通过**
			节点双签、重签攻击	**通过**
			签名伪造	**通过**
6	智能合约安全	系统合约安全审计	合约执行逻辑审计	**通过**
			节点奖励计算	**通过**
			节点 Slash 合约	**通过**
			节点选举	**通过**
			链上治理	**通过**
7	分片安全	分片机制安全性审计	单片攻击	**通过**
			分片重启	**通过**
			分片 staking 和算力安全	**通过**
			分片数据可用性	**通过**
			分片数据一致性	**通过**
		分片交易安全性审计	分片交易可靠性审计	**通过**
			分片交易完整性审计	**通过**

免责声明：本报告系针对项目代码而作出，本报告的任何描述、表达或措辞均不得被解释为对项目的认可、肯定或确认。本次审计仅针对本报告载明的审计类型及结果表中给定的审计类型范围进行审计，其他未知安全漏洞不在本次审计责任范围之内。成都链安科技仅根据本报告出具前已经存在或发生的攻击或漏洞出具本报告，对于出具以后存在或发生的新的攻击或漏洞，成都链安科技无法判断其对公链安全状况可能的影响，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于公链提供者在本报告出具前已向成都链安科技提供的文件和资料，且该部分文件和资料不存在任何缺失、被篡改、删减或隐瞒的前提下作出的；如提供的文件和资料存在信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符等情况或提供文件和资料在本报告出具后发生任何变动的，成都链安科技对由此而导致的损失和不利影响不承担任何责任。成都链安科技出具的本审计报告系根据公链提供者提供的文件和资料依靠成都链安科技现掌握的技术而作出的，由于任何机构均存在技术的局限性，成都链安科技作出的本审计报告仍存在无法完整检测出全部风险的可能性，成都链安科技对由此产生的损失不承担任何责任。
本声明最终解释权归成都链安科技所有。

审计结果说明：

本公司对公链TOP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，TOP公链通过所有审计项，审计结果为通过(优)。

重点代码模块：

- 系统合约代码：src/xtopcom/xvm
- 共识代码：src/xtopcom/xBFT
- 签名代码：src/xtopcom/xcertauth
- 多签代码：src/xtopcom/xmutisig
- key 生成代码：src/xtopcom/xcrypto
- 交易执行：src/xtopcom/xtxexecutor
- 节点私钥管理：src/xtopcom/xtopcl

目录

1. 节点安全

1.1. RPC 接口审计

1. RPC 功能实现审计

RPC 接口列表

查询类：

```
xcluster_query_manager::xcluster_query_manager(observer_ptr<store::xstore_face_t> st
ore,

observer_ptr<base::xvblockstore_t> block_store,

xtxpool_service::xtxpool_proxy_face_ptr const & txpool_service)

: m_store(store), m_block_store(block_store), m_txpool_service(txpool_service), m_bh
(m_store.get(), m_block_store.get(), nullptr) {

    CLUSTER_REGISTER_V1_METHOD(getAccount);

    CLUSTER_REGISTER_V1_METHOD(getTransaction);

    CLUSTER_REGISTER_V1_METHOD(get_transactionlist);

    CLUSTER_REGISTER_V1_METHOD(get_property);

    CLUSTER_REGISTER_V1_METHOD(getBlock);

    CLUSTER_REGISTER_V1_METHOD(getChainInfo);

    CLUSTER_REGISTER_V1_METHOD(getIssuanceDetail);

    CLUSTER_REGISTER_V1_METHOD(getTimerInfo);

    CLUSTER_REGISTER_V1_METHOD(queryNodeInfo);

    CLUSTER_REGISTER_V1_METHOD(getElectInfo);

    CLUSTER_REGISTER_V1_METHOD(queryNodeReward);

    CLUSTER_REGISTER_V1_METHOD(listVoteUsed);

    CLUSTER_REGISTER_V1_METHOD(queryVoterDividend);

    CLUSTER_REGISTER_V1_METHOD(queryProposal);

    CLUSTER_REGISTER_V1_METHOD(getStandbys);

    CLUSTER_REGISTER_V1_METHOD(getCGP);
```

```
}
```

发送交易：

```
template <class T>

xedge_method_base<T>::xedge_method_base()

: m_edge_local_method_ptr(top::make_unique<xedge_local_method<T>>(elect_main, xip2))
, m_archive_flag(archive_flag) {

    m_edge_handler_ptr = top::make_unique<T>(edge_vhost, ioc, election_cache_data_ac
cessor);

    m_edge_handler_ptr->init();

    EDGE_REGISTER_V1_ACTION(T, sendTransaction);

}
```

请求参数：

参数	描述
body	业务参数
identity_token	身份令牌（未使用，未校验）
method	请求方法
sequence_id	会话次数
target_account_addr	账户地址（未使用，未校验）
version	RPC API 版本（固定为 1.0）

主要的请求参数 account_addr 包含在 body 中，body 是编码后的 json 字符串，例如：

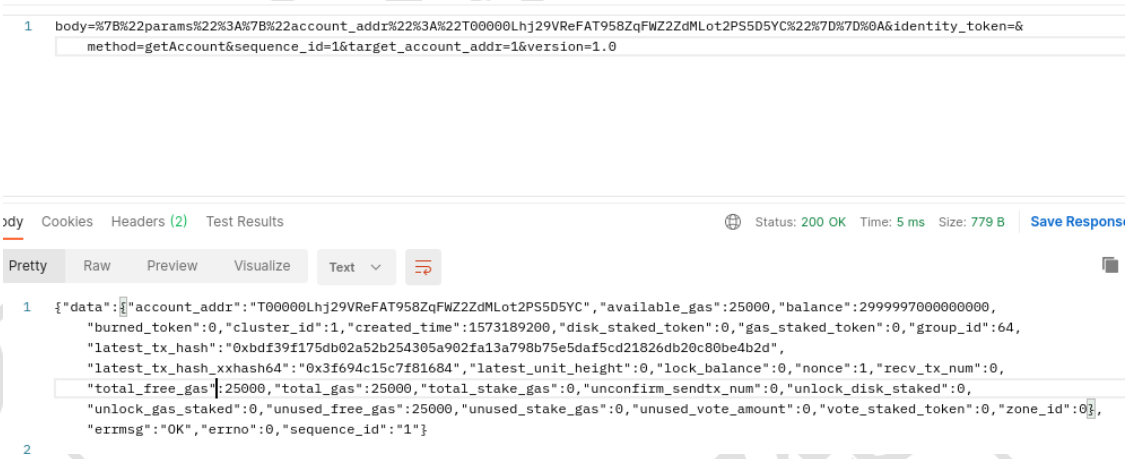

图 1 RPC 请求

target_account_addr 和 identity_token 目前未使用，但是 target_account_addr 不能留空；参数 sequece_id 未做校验，可以为任意字符&字符串；version 固定为 1.0。

除使用 curl/postman 等工具直接调用 RPC 请求外，还可以使用官方提供的客户端 topio，其本质也是组合构造各个 RPC 请求。

2. RPC 接口权限审计

主要检查有没有能越权操作的 RPC API，有没有敏感信息泄露、任意签发交易的问题。上链操作（转账、节点 staking、节点注册、节点投票、节点领取奖励、调用合约等）都由 sendTransaction 完成，通过校验签名来验证请求发起方的身份。

TOP Chain 中 RPC 请求都由 edge 矿工节点来完成，其他角色的节点不开放 RPC 服务接口。

RPC 服务初始化代码：

```
// src/xtopcom/xrpc/xrpc_init.cpp

xrpc_init::xrpc_init(

              //...

)

{

    assert(nullptr != vhost);

    assert(nullptr != router_ptr);

    // 判断节点类型

    switch (node_type) {

     // 验证节点

    case common::xnode_type_t::consensus_validator:

        assert(nullptr != txpool_service);

        assert(nullptr != store);

        init_rpc_cb_thread();

         // 分片间接口

        m_shard_handler = std::make_shared<xshard_rpc_handler>(vhost, txpool_service
, make_observer(m_thread));

        m_shard_handler->start();

        break;
```

```
    // 审计节点、ZEC 选举委员会、REC 选举委员会
    case common::xnode_type_t::committee:
    case common::xnode_type_t::zec:
    case common::xnode_type_t::consensus_auditor:
        assert(nullptr != txpool_service);
        init_rpc_cb_thread();
        m_cluster_handler = std::make_shared<xcluster_rpc_handler>(vhost, router_ptr, txpool_service, store, block_store, make_observer(m_thread));
        m_cluster_handler->start();
        break;
    // 边缘节点
    case common::xnode_type_t::edge: {
        init_rpc_cb_thread();
        m_edge_handler = std::make_shared<xrpc_edge_vhost>(vhost, router_ptr, make_observer(m_thread));
        auto ip = vhost->address().xip2();
        // 开启 http 服务
        shared_ptr<xhttp_server> http_server_ptr = std::make_shared<xhttp_server>(m_edge_handler, ip, false, store, block_store, elect_main, election_cache_data_accessor);
        http_server_ptr->start(http_port);
        //开启 websocket 服务
        shared_ptr<xws_server> ws_server_ptr = std::make_shared<xws_server>(m_edge_handler, ip, false, store, block_store, elect_main, election_cache_data_accessor);
        ws_server_ptr->start(ws_port);
        break;
    }
    case common::xnode_type_t::archive: {
        xassert(false);
    }
```

```
    default:
        break;
    }
}
```

edge 节点提供的 RPC 接口对于请求发起方的校验主要是 method 为 sendTransaction 的签名检查

```
// src/xtopcom/xdata/src/xtransaction.cpp
bool xtransaction_t::sign_check() const {
    static std::set<uint16_t> no_check_tx_type { xtransaction_type_lock_token, xtransaction_type_unlock_token };
    std::string addr_prefix;
    // 获取操作地址
    if (std::string::npos != get_source_addr().find_last_of('@')) {
        uint16_t subaddr;
        base::xvaccount_t::get_prefix_subaddr_from_account(get_source_addr(), addr_prefix, subaddr);
    } else {
        addr_prefix = get_source_addr();
    }

    utl::xkeyaddress_t key_address(addr_prefix);
    uint8_t     addr_type{255};
    uint16_t    network_id{65535};
    //get param from config
    uint16_t config_network_id = 0;//xchain_param.network_id
    if (!key_address.get_type_and_netid(addr_type, network_id) || config_network_id != network_id) {
        xwarn("network_id error:%d,%d", config_network_id, network_id);
        return false;
```

```
    }

    if (no_check_tx_type.find(get_tx_type()) != std::end(no_check_tx_type)) {  // no
 check for other key

        return true;

    }

    // 交易结构中的签名体

    utl::xecdsasig_t signature_obj((uint8_t *)m_authorization.c_str());

    // 判断地址类型、签名校验

    // verify_signature 内部使用 SECP256K1 的 API 校验 ECDSA 签名

    if (data::is_sub_account_address(common::xaccount_address_t{ get_source_addr() }
) || data::is_user_contract_address(common::xaccount_address_t{ get_source_addr() })
) {

        return key_address.verify_signature(signature_obj, m_transaction_hash, get_p
arent_account());

    } else {

        return key_address.verify_signature(signature_obj, m_transaction_hash);

    }

}
```

除 sendTransaction 接口外，其他查询类接口属于公开查询，不存在越权访问或泄露敏感信息的问题

1.2. 节点测试

1. fuzzing 测试

针对开放的 RPC 服务，使用基于 Sulley 构建的工具进行模糊测试，测试过程如下

图 2 Fuzzing 测试

最终测试结果中，未检测到能导致节点崩溃的畸形数据。经模糊测试和代码审计，未发现畸形数据可导致的缓冲区溢出和 DOS 攻击。

2. 非法交易测试

批量生成 sender 和 receiver 地址，持续大量的发送非法交易，由于 edge 节点的流量限制，在持续发送非法交易过程中无法发送正常交易：

```
asus@ubuntu /opt/TOP-chain/runtime  feat/new_horizons • ./topio transfer T00000LX4mzWt7VWTPNhCgnUhoYojjy8HHuHrcFF  5 tx_test
[debug]edge_domain_name: http://unnecessary.org
[debug]edge_ip: 127.0.0.1:19081
[debug]get seeds from  failed
[debug]task_dispatcher init
[debug]register protocols
T00000Lhj29VReFAT958ZqFWZ2ZdMLot2PS5D5YC not found on chain!

asus@ubuntu /opt/TOP-chain/runtime  feat/new_horizons • ./topio transfer T00000LX4mzWt7VWTPNhCgnUhoYojjy8HHuHrcFF  5 tx_test
[debug]edge_domain_name: http://unnecessary.org
[debug]edge_ip: 127.0.0.1:19081
[debug]get seeds from  failed
[debug]task_dispatcher init
[debug]register protocols
Transaction hash: 0x12fcd35e5d48729be4017922a7fd8825f88a7aefacade630601d818a3e008772
Please use command 'topio querytx' to query transaction status later on!!!
```

图 3 非法交易测试

使用 noratelimit 重新编译节点后，在持续发送非法交易的情况下，正常交易可以成功被处理。

2. 钱包及账户安全

2.1. 私钥生成算法

```
// src/xtopcom/xcrypto/src/xckey.cpp:xecprikey_t::xecprikey_t()

xecprikey_t::xecprikey_t()  //sha256(32bytes random)->private key

{

    memset(m_publickey_key,0,sizeof(m_publickey_key));
```

```
    // 随机 buffer 生成
    xrandom_buffer(m_private_key,sizeof(m_private_key));
    uint256_t   hash_value;
    xsha2_256_t hasher;
    // 时间种子
    auto now = std::chrono::system_clock::now();
    auto now_nano = std::chrono::time_point_cast<std::chrono::nanoseconds>(now);
    int64_t time_seed = now_nano.time_since_epoch().count();
    // sha256 散列
    hasher.update(&time_seed,sizeof(time_seed));
    hasher.update(m_private_key, sizeof(m_private_key));
    hasher.get_hash(hash_value);
    const int over_size = std::min((int)hash_value.size(),(int)sizeof(m_private_key)
);
    for(int i = 0; i < over_size; ++i)
    {
        m_private_key[i] += ((uint8_t*)hash_value.data())[i];
    }
    //paired with BN_bin2bn() that converts the positive integer in big-
endian from binary
    m_private_key[0]  &= 0x7F; //ensure it is a positve number since treat is big-
endiam format for big-number
    m_private_key[31] &= 0x7F; //ensure it is a positve number
    generate_public_key();
}
```

随机数的种子使用了系统用作伪随机数生成的特殊文件/dev/urandom，是使用设备驱动程序和其他来源的环境噪声生成的，是类 Unix 系统下推荐的随机数种子。

```
static uint32_t xrandom32()
{
```

```
    //get_sys_random_number might be replaced by std::random_device without xbase li
b

    const uint64_t seed = base::xsys_utl::get_sys_random_number() + base::xtime_utl:
:get_fast_random();

    return (uint32_t)(seed >> 8);

}
```

另提供了一种附加随机种子的私钥生成函数

```
xecprikey_t::xecprikey_t(const std::string rand_seed) //sha256(rand_seed.32bytes ran
dom)->private key

{

            //...

            hasher.update(rand_seed);

            //...

}
```

2.2. 存储安全

TOP Chain 提供的客户端 TOPIO 使用文件形式存储密钥，写入文件前会先对 keystore 信息做 AES-256 加密

```
// src/xtopcom/xtopcl/src/xcrypto.cpp

void aes256_cbc_encrypt(const std::string & pw, const string & raw_text, std::ofstre
am & key_file) {

    AES_INFO aes_info;

    fill_aes_info(pw, raw_text, aes_info);

    // 写入加密后的信息（初始化向量、密文等）

    writeKeystoreFile(key_file, aes_info.iv, aes_info.ciphertext, aes_info.info, aes
_info.salt, aes_info.mac);

}
```

keystore 文件示例

```
{
   "account address" : "T00000LdizAZhkjv3CYxrtsEHUvwFp5MGcEcJ1Kb",
   "crypto" : {
      "cipher" : "aes-256-cbc",
      "cipherparams" : {
         "iv" : "0xc89e0ebcbaf8bb158375f57424f94df7676cc1236a70d5cf76a2f3ead8a57b50"
      },
      "ciphertext" :
      "0x7005354811d30601c1a3da30e82b18008afb50a9e59ad75d2cf1b8b0de3e54a32376e3b9fd25f5e3fd8b8d58c988ed6e",
      "kdf" : "hkdf",
      "kdfparams" : {
         "dklen" : 64,
         "info" : "0xdc53f0c521010cb0",
         "prf" : "sha3-256",
         "salt" : "0x3604c14afec4f9d2e47575bf1de15c264f1c2074d5061598972224a7f886c741"
      },
      "mac" : "0xe373e864e8b3325549d5a5ff3495b33f40c7c3f5d75df5e99f496f8eaeec2c5a"
   },
   "hint" : "name",
   "key_type" : "owner",
   "public_key" : "BHo1Tm7nIRP9EEdXsBg1NSSkJ7zNBBGLHEYexXFGRq1QKrWmLHE1q2NGsOHXmZre/KmUIfspWLErxOpZXUTCaHY="
}
```


图 4 keystore

如果用户在创建账户时设置了密码，在重置密码（resetkeystorepwd）、导入 keystore（importKey）时需要密码解锁

```
// src/xtopcom/xtopcl/src/xcrypto.cpp

string import_existing_keystore(const string & cache_pw, const string & path, bool a
uto_dec) {

    auto key_info = parse_keystore(path);

    if (key_info.empty()) {

        return "";

    }

    // 解密

    auto decrypttext = aes256_cbc_decrypt(cache_pw, key_info);

    if (decrypttext.empty()) {

        if (!auto_dec) {

            cout << "Password error! " << endl;

            cout << "Hint: " << key_info["hint"].asString() << endl;

        }

    }

    return decrypttext;

}
```

2.3. 私钥使用

经过各类 RPC 接口测试，在密钥的使用过程中（如导入 keystore、私钥签名），私钥未在日志、文件等驻留，符合使用安全。

例如命令行转账时的私钥使用

```
// src/xtopcom/xtopcl/src/api_method_imp.cpp

bool api_method_imp::transfer(

                                //...

                               ) {

    // ...

    // 私钥签名

    if (!hash_signature(info->trans_action.get(), uinfo.private_key)) {

        delete info;

        return false;

    }

    task_dispatcher::get_instance()->post_message(msgAddTask, (uint32_t *)info, 0);

    auto rpc_response = task_dispatcher::get_instance()->get_result();

    out_str << rpc_response;

    return true;

}
```

3. 交易模型安全

3.1. 交易处理逻辑审计

3.1.1. 交易类型和流程

```
// src/xtopcom/xdata/xtransaction.h

enum enum_xtransaction_type {

    xtransaction_type_create_user_account = 0, // create user account
```

```
    xtransaction_type_create_contract_account = 1, // create contract account

    xtransaction_type_run_contract = 3, // run contract

    xtransaction_type_transfer = 4, // transfer asset

    xtransaction_type_alias_name = 6, // set account alias name, can be same with ot
her accunnt

    xtransaction_type_set_account_keys = 11, // set account's keys, may be elect key
, transfer key, data key, consensus key

    xtransaction_type_lock_token = 12, // lock token for doing something

    xtransaction_type_unlock_token = 13, // unlock token

    xtransaction_type_create_sub_account = 16, // create sub account


    xtransaction_type_vote = 20,

    xtransaction_type_abolish_vote = 21,


    xtransaction_type_pledge_token_tgas = 22, // pledge token for tgas

    xtransaction_type_redeem_token_tgas = 23, // redeem token

    xtransaction_type_pledge_token_disk = 24, // pledge token for disk

    xtransaction_type_redeem_token_disk = 25, // redeem token

    xtransaction_type_pledge_token_vote = 27, // pledge token for disk

    xtransaction_type_redeem_token_vote = 28, // redeem token


    xtransaction_type_max

};

// src/xtopcom/xdata/src/xtransaction.cpp

bool xtransaction_t::transaction_type_check() const {

    switch (get_tx_type()) {

        #ifdef DEBUG // debug use

        case xtransaction_type_create_user_account:

        case xtransaction_type_set_account_keys:
```

```
        case xtransaction_type_lock_token:
        case xtransaction_type_unlock_token:
        case xtransaction_type_alias_name:
        case xtransaction_type_create_sub_account:
        case xtransaction_type_pledge_token_disk:
        case xtransaction_type_redeem_token_disk:
        #endif
        // 部署用户合约
        case xtransaction_type_create_contract_account:
        // 调用合约
        case xtransaction_type_run_contract:
        // 普通转账
        case xtransaction_type_transfer:
        // 给高级矿工投票
        case xtransaction_type_vote:
        // 取消投票
        case xtransaction_type_abolish_vote:
        // 锁定 token 兑换 gas
        case xtransaction_type_pledge_token_tgas:
        // 解锁兑换 gas 的 token
        case xtransaction_type_redeem_token_tgas:
        // 锁定 token 的兑换选票
        case xtransaction_type_pledge_token_vote:
        // 解锁兑换选票的 token
        case xtransaction_type_redeem_token_vote:
        return true;
        default:
        return false;
```

```
    }

}
```

以客户端发起 RPC 请求为例，edge 节点将调用 do_local_method 处理交易请求，主要是校验 request 中的签名部分和哈希。

```
// src/xtopcom/xrpc/xedge/xedge_method_manager.hpp:sendTransaction_method

// 计算交易哈希，验证是否与 request 中的 hash 一致

if (!tx->digest_check()) {

    throw xrpc_error{enum_xrpc_error_code::rpc_param_param_error, "transaction hash error"};

}

// 如果接收者地址不等于系统地址 sys_contract_rec_standby_pool_addr 或 target_action 不等于 nodeJoinNetwork，就检查签名

if (!(target_action.get_account_addr() == sys_contract_rec_standby_pool_addr && target_action.get_action_name() == "nodeJoinNetwork")) {

    if (!tx->sign_check()) {

        throw xrpc_error{enum_xrpc_error_code::rpc_param_param_error, "transaction sign error"};

    }

}
```

初步校验完后，edge 节点会调用 forward_method 将请求转发，根据接收方账户所属的网络分片将交易发送至对应 Audit Network。

```
int32_t xtxpool_service::request_transaction_consensus(const data::xtransaction_ptr_t & tx, bool local) {

    // ...

    // 校验交易发送来源，可能来自本地或网络。由本地发送的交易只能是系统合约交易，并且不应具有 authorization 字段；非本地交易不能是系统合约交易，并且必须具有 authorization 字段

    int32_t ret = xverifier::xtx_verifier::verify_send_tx_source(tx.get(), local);

    if (ret) {

        // ...

        return ret;

    }
```

```
    // 交易 source 地址映射到 table id
    auto tableid = data::account_map_to_table_id(common::xaccount_address_t{tx->get_source_addr()});
    // 通过 table id 判断是否属于当前网络
    if (!is_belong_to_service(tableid)) {
        // ...
        return xtxpool::xtxpool_error_transaction_not_belong_to_this_service;
    }
    // 判断交易 target 地址是否是系统合约账户并且属于共识 zone
    if (is_sys_sharding_contract_address(common::xaccount_address_t{tx->get_target_addr()})) {
    // 如果是，获取子地址添加到交易 target 地址中
        tx->adjust_target_address(tableid.get_subaddr());
    }
    // 添加交易到源账户网络的交易池
    return m_txpool->push_send_tx(tx);
}
```

接收方接收到 receipt 后会调用 push_recv_tx 或 push_recv_ack_tx

```
//src/xtopcom/xtxpool/src/xtxpool.cpp
int32_t xtxpool_t::on_receipt(const data::xcons_transaction_ptr_t & cons_tx) {
    int32_t ret;
    // 如果是 tx 接收方，添加交易收据到交易池
    if (cons_tx->is_recv_tx()) {
        XMETRICS_COUNTER_INCREMENT("txpool_receipt_recv_total", 1);
        return push_recv_tx(cons_tx);
    } else {
        // 如果是 tx 发送方，此时接收的是接收方的接受交易 receipt
        return push_recv_ack_tx(cons_tx);
    }
```

```
}
```

交易执行的主要代码位于 src/xtopcom/xtxexecutor 中，在 xtransaction_context.h 定义了不同交易类型对应的类。交易在被打包（make_block）以及被验证（verify_block）时会被执行

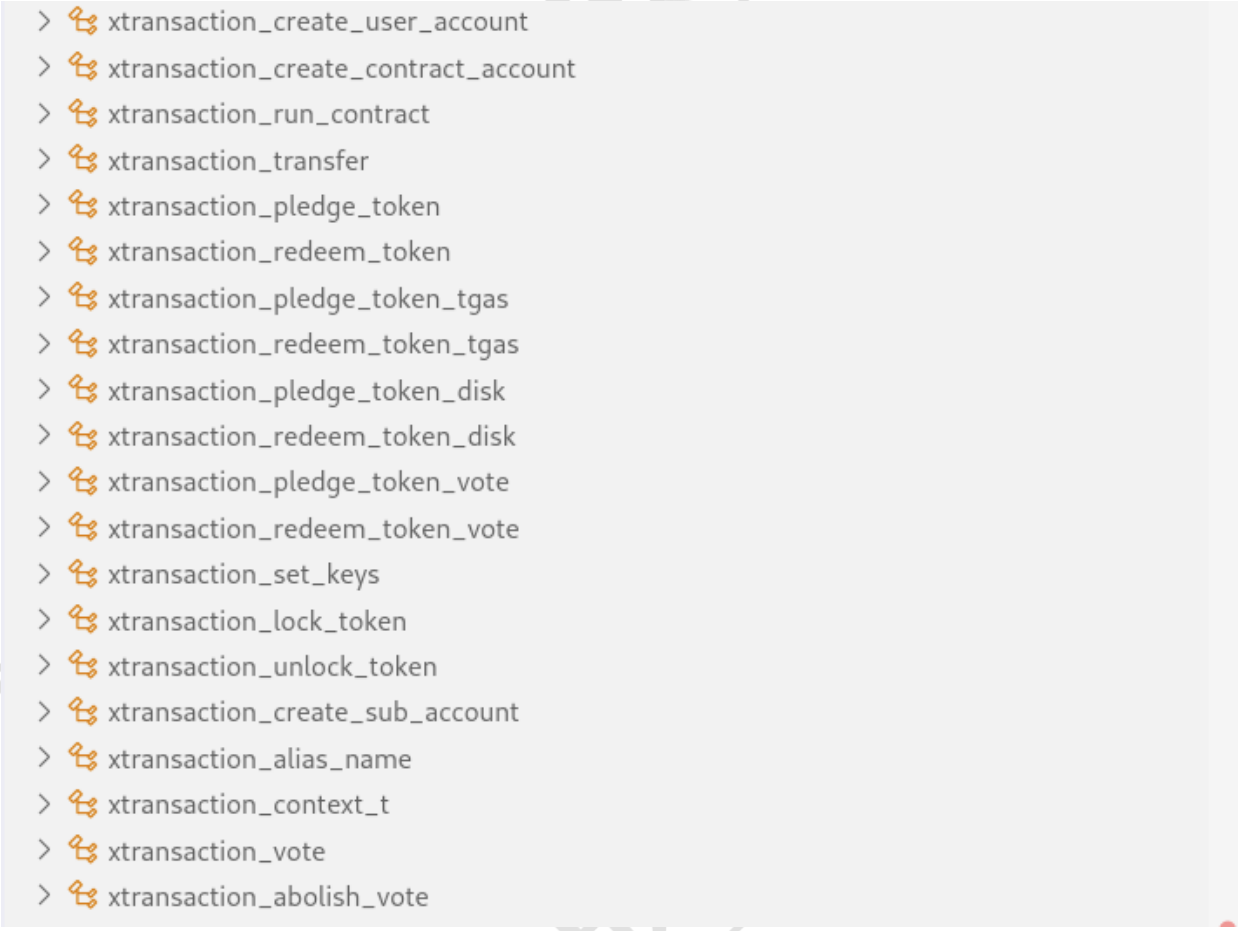

图 5 交易执行的函数

3.1.2. 交易及收据重放攻击

本项审计 TOP Chain 上的交易是否可以在同类型的不同链或同一条链上重放。在 Top Network 中，如果在同一条链重放完全相同的两笔交易，当重放间隔超过一定时间时，无法通过时间戳校验

```
// src/xtopcom/xverifier/src/xtx_verifier.cpp

// verify trx duration expiration

int32_t xtx_verifier::verify_tx_duration_expiration(const data::xtransaction_t * trx
_ptr, uint64_t now) {

    uint32_t trx_fire_tolerance_time = XGET_ONCHAIN_GOVERNANCE_PARAMETER(tx_send_tim
estamp_tolerance);

    uint64_t fire_expire = trx_ptr->get_fire_timestamp() + trx_ptr-
>get_expire_duration() + trx_fire_tolerance_time;
```

```
    if (fire_expire < now) {

        xwarn("[global_trace][xtx_verifier][verify_tx_duration_expiration][fail], tx
:%s, fire_timestamp:%" PRIu64 ", fire_tolerance_time:%" PRIu32 ", expire_duration:%"
 PRIu16 ", now:%" PRIu64,

        trx_ptr->dump().c_str(), trx_ptr-
>get_fire_timestamp(), trx_fire_tolerance_time, trx_ptr-
>get_expire_duration(), now);

        return xverifier_error::xverifier_error_tx_duration_expired;

    }



    xdbg("[global_trace][xtx_verifier][verify_tx_duration_expiration][success], tx h
ash: %s", trx_ptr->get_digest_hex_str().c_str());

    return xverifier_error::xverifier_success;

}
```

重放交易时间戳校验失败

```
xbase-15:19:42.941-T134328:[Warn ]-(verify_tx_duration_expiration:206): [global_tr
ace][xtx_verifier][verify_tx_duration_expiration][fail], tx:{transaction:hash=3eb8
afff40f4832203431f0f0e9a789b2134ba6c74971111d14ae6e30eca3a50,type=4,subtype=2,from
=T00000LhPZXie5GqcZqoxu6BkfMUqo7e9x1EaFS6,to=T00000LRauRZ3SvMtNhxcvoHtP8JK9pmZgxQC
e7Q,nonce=2,refcount=2,this=0x7fccc8041610}, fire_timestamp:1618814156, fire_toler
ance_time:300, expire_duration:100, now:1618816782
```

图 6 时间戳校验

以相近的时间重放交易，会在重复交易校验不通过时被丢弃。

```
// src/xtopcom/xtxpool/src/xaccountobj.cpp

// 查找 m_tx_map 中是否已经存在交易

auto map_it = m_tx_map.find(tx->get_transaction()->get_digest_str());

if (map_it != m_tx_map.end()) {

    // 丢弃不合法交易，错误码为 xtxpool_error_request_tx_repeat

    drop_invalid_tx(tx, xtxpool_error_request_tx_repeat);

    return xtxpool_error_request_tx_repeat;

}
```

对于同类型不同链的情况，例如发生链分叉，对账户 nonce 以及 last_tx_hash 的检查可以防范该类型重放攻击

```
// src/xtopcom/xtxpool/src/xaccountobj.cpp
// 当前交易的 nonce 值与账户发送交易数量对比
if (tx->get_transaction()->get_last_nonce() < m_latest_send_trans_number) {
    // 丢弃不合法交易，错误码为 xtxpool_error_tx_nonce_too_old
    drop_invalid_tx(tx, xtxpool_error_tx_nonce_too_old);
        return xtxpool_error_tx_nonce_too_old;
    }
    // ...
    // 如果发送队列为空
    if (m_send_queue.empty()) {
        // 当前交易 nonce 必须等于账户发送交易数量
        if (tx->get_transaction()->get_last_nonce() != m_latest_send_trans_number) {
            drop_invalid_tx(tx, xtxpool_error_tx_nonce_incontinuity);
            return xtxpool_error_tx_nonce_incontinuity;
        }
        // 交易 last_tx_hash 与账户 m_latest_send_trans_hash 须相等
        if (!check_send_tx(tx, m_latest_send_trans_hash)) {
            drop_invalid_tx(tx, xtxpool_error_tx_last_hash_error);
            return xtxpool_error_tx_last_hash_error;
        }
    } else {
    // 如果发送队列不为空
    // 获取队列最后一个交易
    auto iter = m_send_queue.rbegin();
    auto cons_tx_tmp = iter->m_tx->get_transaction();
    // 交易 nonce 须与最后一个交易的 nonce 连续（+1）
    if (tx->get_transaction()->get_last_nonce() == cons_tx_tmp-
>get_last_nonce() + 1) {
```

```
    // 交易的 last_tx_hash 须等于最后一个交易的 hash
        if (!check_send_tx(tx, cons_tx_tmp->digest())) {
            drop_invalid_tx(tx, xtxpool_error_tx_last_hash_error);
            return xtxpool_error_tx_last_hash_error;
        }
    } else if (tx->get_transaction()->get_last_nonce() > cons_tx_tmp-
>get_last_nonce() + 1) {
        // nonce 不连续的情况，丢弃
        drop_invalid_tx(tx, xtxpool_error_tx_nonce_incontinuity);
        return xtxpool_error_tx_nonce_incontinuity;
    } else {
        // nonce 小于规定值，比较当前交易与队列中的其他交易，如果 nonce 重复但当前交易时间戳
较新，则丢弃相应的队列交易
        int32_t ret = check_and_erase_old_nonce_duplicate_tx(tx);
        if (ret != xsuccess) {
            drop_invalid_tx(tx, ret);
            return ret;
        }
    }
    // tx sendqueue is full, drop it
    if (m_send_queue.size() >= m_send_tx_queue_max_num) {
        drop_invalid_tx(tx, xtxpool_error_send_tx_queue_over_upper_limit);
        return xtxpool_error_send_tx_queue_over_upper_limit;
    }
}
```

收据处理 push_recv_tx 时重复收据会删除

```
int32_t xtxpool_table_t::push_recv_tx(const xcons_transaction_ptr_t & cons_tx) {
    int32_t ret = verify_receipt_tx(cons_tx);
```

```
    if (ret) {

        XMETRICS_COUNTER_INCREMENT("txpool_push_tx_fail", 1);

        return ret;

    }

    xtransaction_t * tx = cons_tx->get_transaction();

    uint64_t tx_timer_height = cons_tx->get_clock();

    std::vector<std::pair<std::string, uint256_t>> committed_recv_txs;

    ret = m_consensused_recvtx_cache.is_receipt_duplicated(cons_tx-
>get_clock(), tx, committed_recv_txs);

    // 删除重复收据

    delete_committed_recv_txs(committed_recv_txs);

    if (ret != xsuccess) {

        xwarn("xtxpool_table_t::tx_push fail. table=%s,timer_height:%ld,tx=%s,fail-
%s", m_table_account.c_str(), tx_timer_height, cons_tx-
>dump().c_str(), get_error_str(ret).c_str());

        return ret;

    }

    // ...

    return ret;

}
```

并且还会检查收据 cert 的开始/创建时间是否重复，意味着不能重放 cert 在同一时间创建的收据

```
int32_t xtransaction_recv_cache_t::check_duplicate_in_cache(uint64_t tx_timer_height, const xtransaction_t * receipt_tx) {
    auto iter = m_transaction_recv_cache.find(tx_timer_height);
    if (iter != m_transaction_recv_cache.end()) {
        const auto & tx_hash_set = iter->second;
        auto         iter1 = tx_hash_set.find(receipt_tx->digest());
        // found from cache, the receipt is duplicate.
        if (iter1 != tx_hash_set.end()) {
            xwarn("xtransaction_recv_cache table=%s tx send receipt duplicate.timer_height:%ld txHash:%s", m_table_account.c_str(),
            tx_timer_height, receipt_tx->get_digest_hex_str().c_str());
            return xtxpool_error_sendtx_receipt_duplicate;
        }
    }
    xdbg("xtransaction_recv_cache table=%s tx is not duplicate. timer_height:%ld txHash:%s", m_table_account.c_str(), tx_timer_height,
    receipt_tx->get_digest_hex_str().c_str());
    return xsuccess;
}
```

图 7 检查区块 clock 是否重复

3.1.3. 粉尘攻击

粉尘攻击指攻击者通过向用户钱包发送极少量的代币，称为“粉尘”，攻击者通过追踪被粉尘化的钱包资金和所有的交易，继而连接上这些地址，确定这些钱包地址所属的公司或个人，破坏区块链的匿名性；或滥用区块链资源，造成区块链内存池紧张。如果粉尘资金不被移动，则攻击者无法与其建立连接，就无法完成钱包或地址所有者的去匿名化。

TOP Chain 使用账户模型，与 Bitcoin 的 UTXO 模型不同，UTXO 模型中用户钱包的“余额”由若干个未花费交易输出构成，在用户转账时粉尘 UTXO 会一直在用户转账的交易中有所表现，Bitcoin 的交易由输入和输出组成，所以交易可以通过 UTXO 串联起来，通过粉尘攻击达到去匿名化的目的。在 TOP Chain 中，粉尘资金发送到用户钱包后，在用户余额中添加金额，没有独立于用户余额，攻击者无法达到去匿名化的目的，并且 TOP Chain 中每笔交易都要消耗一定量的 gas，当免费 gas 配额用完后需要锁定 token 来兑换，防止粉尘交易无限制消耗区块链资源。

```
// user fire each transactions linked as a chain
uint64_t    m_latest_send_trans_number{0};  // heigh or number of transactior
uint256_t   m_latest_send_trans_hash{};  // all transaction fired by account,

// consensus mechanisam connect each received transaction as a chain
uint64_t    m_latest_recv_trans_number{0};  // heigh or number of transactior
uint256_t   m_latest_recv_trans_hash{};   // all receipt contruct a mpt tree,

// note: the below properties are not allow to be changed by outside,it only
uint64_t    m_account_balance{0};       // token balance,
uint64_t    m_account_burn_balance{0};
xpledge_balance m_account_pledge_balance;
uint64_t    m_account_lock_balance{0};
//uint64_t    m_account_lock_balance{0};
uint64_t    m_account_lock_tgas{0};
uint64_t    m_account_unvote_num{0};    // unvoted number
int64_t     m_account_credit{0};        // credit score,it from the contributi
uint64_t    m_account_nonce{0};         // account 'latest nonce,increase atom
uint64_t    m_account_create_time{0};      // when the account create
int32_t     m_account_status{0};        // status for account like lock,suspen
std::map<std::string, std::string>  m_property_hash;  // [Property Name as ke
xnative_property_t                  m_native_property;
uint16_t                            m_unconfirm_sendtx_num{0};
std::map<uint16_t, std::string>     m_ext;
```

图 8 账户属性

3.1.4. 交易泛滥攻击

对于普通交易，分配的 disk 空间用尽后，必须通过抵押 token 来获得额外的 disk 空间，用于发起和永久储存新的交易。

对于 Beacon 系统合约交易，除了 gas 消耗之外，交易发送方还会被自动扣除一笔手续费并销毁，费用由链上治理参数 beacon_tx_fee 决定，当前为 100*10^6 uTOP，可使系统免受交易泛滥攻击。

```
// src/xtopcom/xtxexecutor/src/xtransaction_fee.cpp

uint64_t xtransaction_fee_t::cal_service_fee(const std::string& source, const std::s
tring& target) {

    uint64_t beacon_tx_fee{0};
```

```
    #ifndef XENABLE_MOCK_ZEC_STAKE

    // 当源地址不是系统合约地址，目的地址是 beacon 合约地址时设置 beacon_tx_fee

    if (!is_sys_contract_address(common::xaccount_address_t{ source })

    && is_beacon_contract_address(common::xaccount_address_t{ target })){

        beacon_tx_fee = XGET_ONCHAIN_GOVERNANCE_PARAMETER(beacon_tx_fee);

    }

    #endif

    return beacon_tx_fee;

}
```

3.1.5. 双花攻击

对于 TOP Chain 而言，账户的每个交易都包含一个唯一的、递增的 nonce 以及已确认的前一个交易的哈希值，正常情况下攻击者无法发生双花攻击。对于 Bitcoin 等使用算力竞争的区块链而言，当攻击者拥有超过 50%算力， 构造双花交易后竞争算力使交易所在区块链成为最长链，会导致双花攻击成功。对于使用 hpPBFT-PoS*共识的 TOP Chain 来说，双花攻击基本不存在。

3.1.6. 非法交易

本项审计是否存在畸形交易、伪造交易攻击漏洞。畸形交易在节点畸形数据测试部分已覆盖。

用户在发起交易时会对整个交易数据签名，对交易中任何数据的修改都会导致 edge 节点在校验签名时不通过。

```
// src/xtopcom/xrpc/xedge/xedge_method_manager.hpp:sendTransaction_method

if (!(target_action.get_account_addr() == sys_contract_rec_standby_pool_addr && targ
et_action.get_action_name() == "nodeJoinNetwork")) {

    if (!tx->sign_check()) {

        throw xrpc_error{enum_xrpc_error_code::rpc_param_param_error, "transaction s
ign error"};

    }

}
```

如果 edge 节点为恶意，此处签名校验失效，但验证节点也会对交易签名再一次校验，伪造的交易会验证失败。

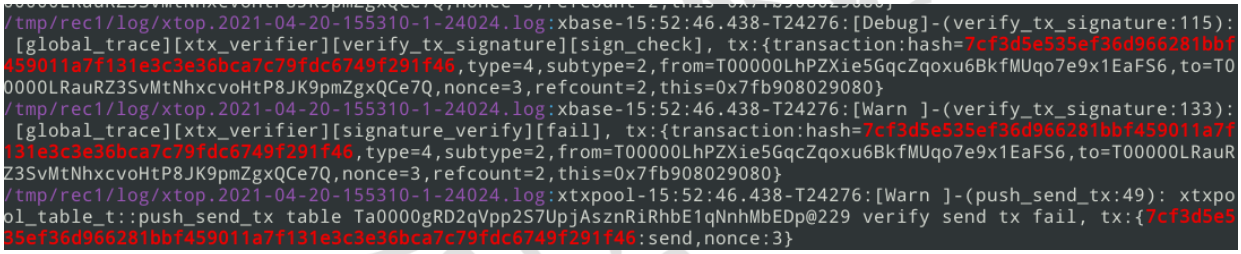

图 9 签名校验

3.2. 其他交易安全审计

3.2.1. 交易延展性攻击

交易延展性也被成为可锻性，会造成交易 ID 的不一致，导致用户找不到发送的交易，影响一些钱包充值或提现的状态。TOP Chain 中交易签名于其他交易数据是分开的，改变交易签名不会改变交易哈希，如果改变其他交易数据，那么签名校验就无法通过。并且签名算法使用的是 Schnorr，不具有 ECDSA 签名的可延展性。

```
// src/xtopcom/xdata/src/xtransaction.cpp:digest_check

bool xtransaction_t::digest_check() const {

    base::xstream_t stream(base::xcontext_t::instance());

    // 哈希计算

    do_write_without_hash_signature(stream, true);

    uint256_t hash = utl::xsha2_256_t::digest((const char*)stream.data(), stream.siz
e());

    if (hash != m_transaction_hash) {

        xwarn("xtransaction_t::digest_check fail. %s %s",

        to_hex_str(hash).c_str(), to_hex_str(m_transaction_hash).c_str());

        return false;

    }

    return true;

}
```

3.2.2. 假充值攻击

客户端获取交易时会返回交易状态，响应结果中的交易状态对应四种：success, fail, queue, pending

```
// src/xtopcom/xrpc/xgetblock/get_block.cpp

void get_block_handle::update_tx_state(xJson::Value & result_json, const xJson::Valu
e & cons) {

    if (cons["confirm_unit_info"]["exec_status"].asString() == "success") {

        result_json["tx_state"] = "success";

    } else if (cons["confirm_unit_info"]["exec_status"].asString() == "failure") {

        result_json["tx_state"] = "fail";

    } else if (cons["send_unit_info"]["height"].asUInt64() == 0) {

        result_json["tx_state"] = "queue";

    } else {

        result_json["tx_state"] = "pending";

    }

}
```

在交易执行后会更新对应的状态

```
// src/xtopcom/xtxexecutor/src/xtransaction_executor.cpp:exec_batch_txs

for (auto & tx : txs) {

    xtransaction_result_t result;

    int32_t action_ret = xtransaction_executor::exec_tx(account_context, tx, result)
;

    if (action_ret) {

        tx->set_current_exec_status(enum_xunit_tx_exec_status_fail);

        // receive tx should always consensus success, contract only can exec one tx
 once time, TODO(jimmy) need record fail/success

        if (tx->is_recv_tx() || tx->is_confirm_tx()) {

            xassert(txs.size() == 1);

        } else {

            txs_result.m_exec_fail_tx = tx;

            txs_result.m_exec_fail_tx_ret = action_ret;

            // if has successfully txs, should return success

            xwarn("xtransaction_executor::exec_batch_txs tx exec fail, %s result:fai
```

```
l error:%s",
            tx-
>dump().c_str(), chainbase::xmodule_error_to_str(action_ret).c_str());
            return action_ret; // one send tx fail will ignore success tx before
        }
    } else {
        tx->set_current_exec_status(enum_xunit_tx_exec_status_success);
        txs_result.succ_txs_result = result;
    }
    txs_result.m_exec_succ_txs.push_back(tx);
    xkinfo("xtransaction_executor::exec_batch_txs tx exec succ, tx=%s,total_result:%
s",
    tx->dump().c_str(), result.dump().c_str());
}
```

客户端在进行充值校验时只有 success 状态是交易执行成功，难以发生假充值攻击。

3.2.3. 命令行转账方法审计

官方提供的客户端 topio 命令行转账，topio 转账的命令是：

```
./topio transfer TARGET_ADDRESS AMOUNT NOTE
```

命令行转账方法中对 note 做了长度限制，不能大于 128 个字节

```
void ApiMethod::transfer1(std::string & to, double & amount_d, std::string & note, d
ouble & tx_deposit_d, std::ostringstream & out_str) {
    std::ostringstream res;
    if (update_account(res) != 0) {
        return;
    }
    std::string from = g_userinfo.account;
    if (note.size() > 128) {
        std::cout << "note size: " << note.size() << " > maximum size 128" << endl;
        return;
```

```
    }

    uint64_t amount = ASSET_TOP(amount_d);

    uint64_t tx_deposit = ASSET_TOP(tx_deposit_d);

    if (tx_deposit != 0) {

        api_method_imp_.set_tx_deposit(tx_deposit);

    }

    api_method_imp_.transfer(g_userinfo, from, to, amount, note, out_str);

    tackle_send_tx_request(out_str);

}
```

topio 将交易签名后通过 RPC API 发送给节点，后续的节点处理流程与其他 RPC 一致。

4. 共识安全

TOP Chain 采用 hpPBFT-PoS*作为共识算法，hpPBFT 为高速并行的实用拜占庭容错，参考 HotStuff 实现。

Hotstuff 理论中，连续三个阶段的确认就可以认为这个块不可更改。hotstuff 每个阶段对应 prepareQC, lockedQC, commitQC，三阶段完成后交易可以 100%确定完成，即要证明不存在两个冲突的 commitQC。

假设 A 和 B 是两个冲突的块，不可能存在 AB 高度相同的情况，因为 proposal 的提交需要多数节点的投票，每个节点每个阶段只会投一个 proposal，不可能出现同一高度两个 proposal 票数都过半的情况。

假设 A 和 B 高度不相同，设 qc1.node=A, qc2.node=B，v1=qc1.viewNumber, v2=qc2.viewNumber, 假设 v1<v2。qcs 是高度大于 A 且与 A 冲突的、高度最小的那个合法的 preparaQC 证书，qcs.viewNumber=vs，伪代码表示为：

```
E(prepareQC):=(v1<prepareQC.viewNumber<v2) ∧ (prepareQC.node conflicts with A)
```

现在可以设置一个切换点 qcs，qcs 可以看作是“冲突”的起始位置:

```
qcs:=argmin{prepareQC.viewNumber|prepareQC is valid∧E(prepareQC)}
```

一个正确的副本会发送部分签名的结果 tsign(<qc1.type,qc1.viewNumber,qc1.node>)给 leader，令 r 成为对于 tsign(<qcs.type, qcs.viewNumber,qcs.node>)有贡献的第一个副本，这样的 r 必须存在，否则 qc1.sig 和 qcs.sig 的其中一个就不能创建。

在视图 v1 中，副本 r1 使用 precommitQC 阶段更新 lockedQC，对应 A。由于 vs 的最小化定义，副本 r 在 A 处生成的 lockedQC 在 qcs 形成之前不会改变，否则 r 一定看到了其他视图的 prepareQC，就不符合 vs 的最小化假设了。副本 r 在视图 vs 的 prepare 阶段调用 safeNode，其中消息 m 包含 m.node=qcs.node。假设 m.node 与 lockedQC.node 冲突，就无法通过 safeNode 的 safety 校验（返回 false）

```
// hotstuff 算法

function safeNode(node, qc):

    return (node extends from lockedQC .node)// safety rule

    (qc.viewNumber > lockedQC .viewNumber ) // liveness rule
```

此外，m.justify.viewNumber>v1 将违反 vs 最小值的假设，因此 safeNode 校验中的 liveness 也是失败的。所以 r 不能针对 vs 进行 prepare 投票，即 qcs 根本无法生成，不可能存在两个冲突的 commitQC。

为了弥补在无许可链上使用 BFT 算法的安全性问题，TOP Chain 使用了股权证明来提高节点参与共识的门槛。节点 stake 综合考虑多个因素：保证金（TOP token）、信誉分、得票数。节点 stake（包括 auditor 和 validator）计算公式如下：

auditor stake = (矿工保证金+矿工得票总数/2) * auditor 信誉分

```
// src/xtopcom/xstake/xstake_algorithm.h

uint64_t get_auditor_stake() const noexcept {

    uint64_t stake = 0;

    if (is_auditor_node()) {

        stake = (m_account_mortgage / TOP_UNIT + m_vote_amount / 2) * m_auditor_cred
it_numerator / m_auditor_credit_denominator;

    }

    return stake;

}
```

validator stake=sqrt[（矿工保证金+矿工得票总数/2）* validator 信誉分]

```
// src/xtopcom/xstake/xstake_algorithm.h

uint64_t get_validator_stake() const noexcept {

    uint64_t stake = 0;

    if (is_validator_node()) {
```

```
        // 链上治理参数 最大 validator stake

        auto max_validator_stake = XGET_ONCHAIN_GOVERNANCE_PARAMETER(max_validator_s
take);

        stake = (uint64_t)sqrt((m_account_mortgage / TOP_UNIT + m_vote_amount / 2) *
 m_validator_credit_numerator / m_validator_credit_denominator);

        stake = stake < max_validator_stake ? stake : max_validator_stake;

    }

    return stake;

}
```

一个共识集群包括一个 auditor group 和两个 validator group，不同集群的选举彼此独立。共识集群通过三阶段提交范式完成 BFT 轮次。Leader 的选择通过 VRF-FTS（Follow-the-satoshi）确定。通过 VRF 生成随机数种子，经过综合 Stake 加权。跟踪节点的工作负载贡献、总保证金、投票等由 Beacon 上的一系列合约完成。

```
template <typename RNG>
std::shared_ptr<node_type> select(RNG & prng) const {
    auto node = this->root();
    while (!node->is_leaf()) {
        auto left = std::dynamic_pointer_cast<node_type>(node->left());
        auto right = std::dynamic_pointer_cast<node_type>(node->right());
        assert(left != nullptr);

        auto const r = std::uniform_int_distribution<std::uint64_t>{0, node->stake()}(prng);
        node = (r <= left->stake()) ? left : right;
        assert(node != nullptr);
    }

    assert(node->is_leaf());
    return node;
}
```

图 10 随机选举

综合 stake 考虑了各种因素，降低了恶意节点被选举为 leader 的概率。并且共识节点会定时轮入轮出 shard。shard 轮换机制由系统智能合约实现

```
bool elect_auditor_validator(common::xzone_id_t const & zone_id,
                             common::xcluster_id_t const & cluster_id,
                             common::xgroup_id_t const & auditor_group_id,
                             std::uint64_t const random_seed,
                             common::xlogic_time_t const election_timestamp,
                             common::xlogic_time_t const start_time,
                             data::election::xelection_association_result_store_t const & association_result_store,
                             data::election::xstandby_network_result_t const & standby_network_result,
                             std::unordered_map<common::xgroup_id_t, data::election::xelection_result_store_t> & all_cluster_election_result_store);

bool elect_auditor(common::xzone_id_t const & zid,
                   common::xcluster_id_t const & cid,
                   common::xgroup_id_t const & gid,
                   common::xlogic_time_t const election_timestamp,
                   common::xlogic_time_t const start_time,
                   std::uint64_t const random_seed,
                   data::election::xstandby_network_result_t const & standby_network_result,
                   data::election::xelection_network_result_t & election_network_result);

bool elect_validator(common::xzone_id_t const & zid,
                     common::xcluster_id_t const & cid,
                     common::xgroup_id_t const & auditor_gid,
                     common::xgroup_id_t const & validator_gid,
                     common::xlogic_time_t const election_timestamp,
                     common::xlogic_time_t const start_time,
                     std::uint64_t const random_seed,
                     data::election::xstandby_network_result_t const & standby_network_result,
                     data::election::xelection_network_result_t & election_network_result);
```

图 11 auditor 和 validator 选举

一个共识集群包括 auditor group 和 validator group，由参数 auditor_group_count 和 validator_group_count 决定，根据综合 stake 选择节点轮入/轮出共识集群。

```
static void normalize_stake(common::xrole_type_t const role, std::vector<xelection_awared_data_t> & input) {
    auto & result = input;
    switch (role) {
    case common::xrole_type_t::advance: {
        std::sort(std::begin(result), std::end(result), [](xelection_awared_data_t const & lhs, xelection_awared_data_t const & rhs) { return lhs > rhs; }
        );
        for (auto i = 0u; i < result.size(); ++i) {
            if (result[i].stake() > 0) {  // special condition check for genesis nodes.
                result[i].comprehensive_stake(calc_comprehensive_stake(i));
            } else {
                assert(result[i].stake() == 0);
                result[i].comprehensive_stake(minimum_comprehensive_stake);
            }
        }

        std::sort(std::begin(result), std::end(result), [](xelection_awared_data_t const & lhs, xelection_awared_data_t const & rhs) { return lhs < rhs; }
        );
        break;
    }

    case common::xrole_type_t::consensus: {
        for (auto & standby_node : result) {
            standby_node.comprehensive_stake(std::max(standby_node.stake(), minimum_comprehensive_stake));
        }

        std::sort(std::begin(result), std::end(result), [](xelection_awared_data_t const & lhs, xelection_awared_data_t const & rhs) { return lhs < rhs; }
        );
        break;
    }
```

图 12 计算综合 stake

1. 共识流程

一轮 BFT 由 leader 发起 proposal 开始，由其他节点对 proposal 投票，当收集到足够多的 vote 之后，leader 向其他节点广播 commit，完成一轮 BFT。一轮 BFT 后视图更改，视图更改事件会触发出块。

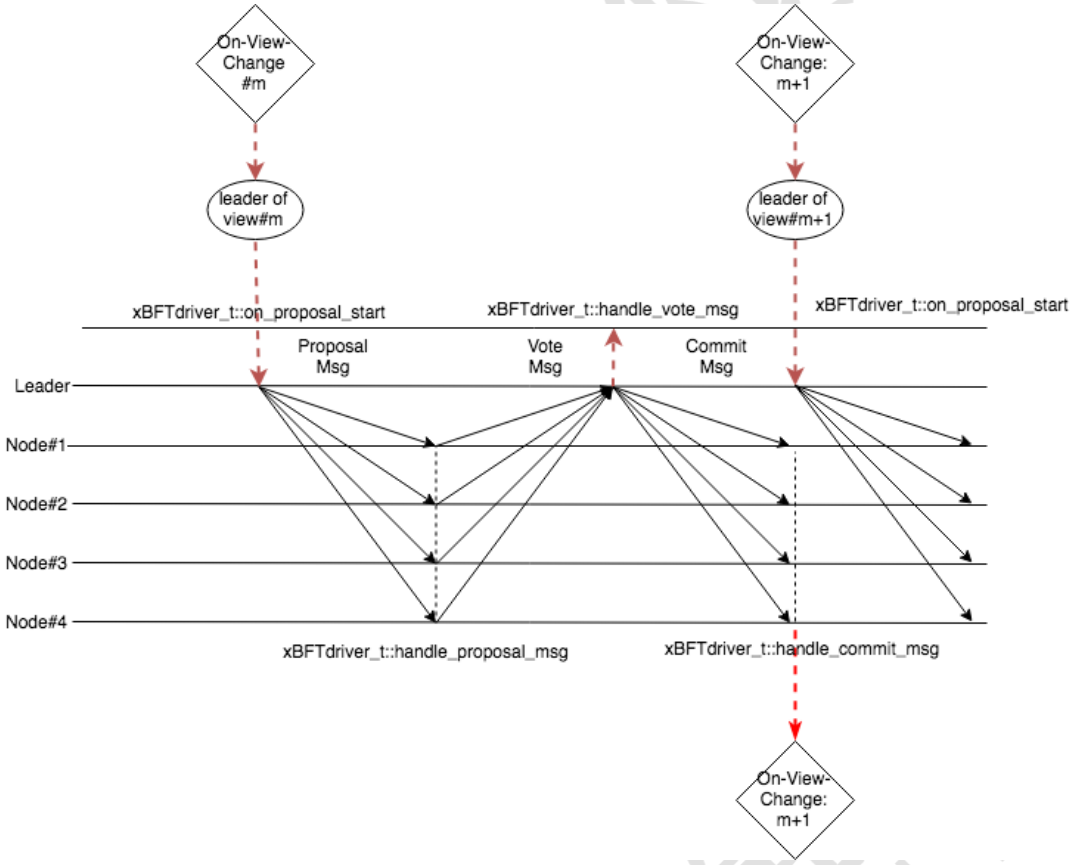

图 13 共识流程

leader 接收到 vote 消息后会检查 vote 数量是否已经满足 2f+1（包括 validator 和 auditor），接着广播 commit 消息，一轮 xBFT 结束。区块 1 为 HighQC 状态，可能会被 fork 或丢弃；当第二轮 xBFT 结束后产生区块 2 为 HighQC，此时区块 1 变为 locked 状态，不允许被 fork；第三轮 xBFT 结束后区块 1 变为 commit 状态，区块 2 变为 locked 状态，新产生的区块 3 为 HighQC 状态。每轮 BFT 都会发生 view change，三轮 BFT 后一个区块才完全确认，并且审计网络也会执行检查，因此即使 Leader 作恶，无效的交易也会被阻止。

2. 共识算法一致性

HighQC 状态的区块不能保证有 2f+1 个节点都 commit 了这个区块，因为在一轮 BFT 的最后一阶段可能发生异常无法确保大多数节点 commit 了 HighQC；当区块状态变为 Locked，可以表明有 2f+1 个节点对 proposal 投票了，但是在第二阶段 commit 时，也无法确保大多数共识节点都收到 commit 消息，TOP Chain 在此阶段做了检查使得 locked block 无法分叉；当第三次确认，区块状态是 commited，可以保证大多数节点的一致性，可以执行交易中的内容修改对应账号的状态。

```
// 当前block高度==locked block高度
else if(_test_for_block->get_height() == locked_block_height)
{
    // 但是hash不相等，错误
    if(_test_for_block->get_block_hash() != get_lock_block()->get_block_hash())
    {
        xwarn("xBFTRules::safe_check_follow_locked_branch,fail-block with same height of locked,but different hash of proposal=%s vs
        locked=%s at node=0x%llx",_test_for_block->dump().c_str(), get_lock_block()->dump().c_str(),get_xip2_addr().low_addr);
        return -1;
    }
    return 1;
}
// 当前block高度在locked block之后
else if(_test_for_block->get_height() == (locked_block_height + 1) )
{
    // 上一个block hash与lock block hash不相等，处于分叉中
    if(_test_for_block->get_last_block_hash() != get_lock_block()->get_block_hash())
    {
        xwarn("xBFTRules::safe_check_follow_locked_branch,fail-proposal try to fork at locked block of prev, proposal=%s vs locked=%s at
        node=0x%llx",_test_for_block->dump().c_str(), get_lock_block()->dump().c_str(),get_xip2_addr().low_addr);
        return -1;
    }
    return 1;
}
```

图 14 分叉校验

3. 共识算法活性

xBFTdriver_t::on_view_fire:当收到比 proposal 序号更大的 viewid 事件后，将 proposal 加入 timeout 列表，safe_check_for_block 失败加入 outofdate 列

xBFTdriver_t::on_clock_fire:全局时钟调用 safe_check_for_block 对 proposal 进行检查，检查失败时放入 outofdate 列表

```
bool xBFTRules::safe_check_for_block(base::xvblock_t * _block)
{
    // 必须存在locked block
    base::xvblock_t * lock_block = get_lock_block();
    if( (NULL == _block) || (NULL == lock_block) )
        return false;
    // heigh,viewid,chainid,account
    if(   (_block->get_viewid()  < _block->get_height())
       || (_block->get_height()  <= lock_block->get_height())
       || (_block->get_viewid()  <= lock_block->get_viewid())
       || (_block->get_chainid() != lock_block->get_chainid())
       || (_block->get_account() != lock_block->get_account())
       )
    {
        return false;
    }
    return true;
}
```

图 15 safe_check_for_block

notify_proposal_fail 处理 timeout_list 和 outofdate_list，当 timeout_list 中的 proposal 位于它的 leader 节点上，广播 commit 消息；如果不是 leader，就标记这一轮共识状态为 timeout，outofdate_list 中的标记为 cancel。

cancel 和 timout 的 proposal 将不做处理，共识轮次由 clock block 驱动，可以保证共识算法的活性。

5. 签名安全

签名算法以 Schnorr 门限签名算法为基础，提供单签名、多签名合并和校验签名的功能。Schnorr 签名存在现有的安全证明：当使用足够随机的哈希函数并且签名中使用的椭圆曲线离散对数问题足够困难时，可以证明 Schnorr 的安全性，安全性优于 ECDSA。

一轮共识开始时，Leader 调用 do_sign()对 proposal 签名

```
//step#4: do signature here
if(proposal->get_cert()->is_validator(get_xip2_addr().low_addr))
{
    proposal->set_verify_signature(get_vcertauth()->do_sign(get_xip2_addr(), proposal->get_cert(),
    base::xtime_utl::get_fast_random64()));//bring leader 'signature
}
else
{
    proposal->set_audit_signature(get_vcertauth()->do_sign(get_xip2_addr(), proposal->get_cert(),
    base::xtime_utl::get_fast_random64()));//bring leader 'signature
}
```

图 16 签名

其他共识节点在处理 proposal 消息时对签名校验，并对 proposal 签名，添加投票信息

```
if(get_vcertauth()->verify_sign(leader_xip,_proposal->get_block()) == base::enum_vcert_auth_result::enum_successful)//first verify
leader'signature as well
{
    const int result_of_verify_proposal = verify_proposal(_proposal->get_block(),_bind_xclock_cert,this);
    _proposal->set_result_of_verify_proposal(result_of_verify_proposal);
    if(result_of_verify_proposal == enum_xconsensus_code_successful)//verify proposal then
    {
        std::string empty;
        _proposal->add_voted_cert(leader_xip, _proposal->get_cert(),get_vcertauth());//add leader'cert to list
        _proposal->get_block()->set_verify_signature(empty);//reset cert
        _proposal->get_block()->set_audit_signature(empty); //reset cert

        const std::string signature = get_vcertauth()->do_sign(replica_xip, _proposal->get_cert(),base::xtime_utl::get_fast_random64
        ());//sign for this proposal at replica side

        if(_proposal->get_cert()->is_validator(replica_xip.low_addr))
            _proposal->get_block()->set_verify_signature(signature); //verification node
        else  if(_proposal->get_cert()->is_auditor(replica_xip.low_addr))
            _proposal->get_block()->set_audit_signature(signature);  //auditor node
        else //should not happen since has been tested before call
        {
```

图 17 签名校验

Leaders 处理 vote 消息时，如果当前 proposal 的投票已满足 2f+1，聚合其他共识节点的签名，并调用 verify_muti_sign()验证多重签名

```
// 投票完成
if(_proposal->is_vote_finish()) //check again
{
    // validator不为空，聚合validator签名
    if(false == _proposal->get_voted_validators().empty())
    {
        const std::string merged_sign_for_validators = get_vcertauth()->merge_muti_sign(_proposal->get_voted_validators()
        , _proposal->get_block());
        _proposal->get_block()->set_verify_signature(merged_sign_for_validators);
    }
    // auditor不未空，聚合auditor签名
    if(false == _proposal->get_voted_auditors().empty())
    {
        const std::string merged_sign_for_auditors = get_vcertauth()->merge_muti_sign(_proposal->get_voted_auditors(),
        _proposal->get_block());
        _proposal->get_block()->set_audit_signature(merged_sign_for_auditors);
    }
    // 验证多重签名
    if(get_vcertauth()->verify_muti_sign(_proposal->get_block()) == base::enum_vcert_auth_result::enum_successful) //
    quorum certification and  check if majority voted
    {
```

图 18 聚合签名并校验

多重签名校验过程：检查 validator 和 auditor 是否来自同一 group、同一集群，调用 xschnorrsig_t::verify_muti_sign 校验 validator 和 auditor 的签名。

```
// 签名者数量检查
if(muti_signers_pubkey.size() < sig_threshold)
{
    xerror("xschnorrsig_t::verify_muti_sign,fail-too little signers(%d) < sig_threshold(%u)",(int32_t)muti_signers_pubkey.size(),
    sig_threshold);
    return false;
}
// 聚合多签公钥
std::shared_ptr<xmutisig::xpubkey> vote_agg_pub = xmutisig::xmutisig::aggregate_pubkeys_2(muti_signers_pubkey,
xmutisig::xschnorr::instance());
if(vote_agg_pub == nullptr)
{
    xerror("xschnorrsig_t::verify_muti_sign,fail-aggregate pubkeys with size(%d)",(int32_t)muti_signers_pubkey.size());
    return false;
}
return xmutisig::xmutisig::verify_sign(target_hash,
                                       *vote_agg_pub.get(),
                                       aggregated_sig_obj.get_mutisig_seal(),
                                       aggregated_sig_obj.get_mutisig_point(),
                                       xmutisig::xschnorr::instance());
```

图 19 多签名校验

节点对 proposal 执行 xproposal_t::add_voted_cert()过程中，确保了一个节点没有重复的投票者

```
if(m_all_votors.find(account_addr_of_node) == m_all_votors.end()) //filter any duplicated account
{
    if(get_cert()->is_validator(voter_xip))
    {
        const std::string& signature = qcert_ptr->get_verify_signature();
        auto set_res = m_all_voted_cert.emplace(signature);//emplace do test whether item already in set
        if(set_res.second)//return true when it is a new element
        {
            auto map_res = m_voted_validators.emplace(voter_xip,signature);
            if(map_res.second)
            {
                m_all_votors.emplace(account_addr_of_node);//record account to avoid duplicated nodes of same account
                ++m_voted_validators_count;
                return true;
            }
            xerror("xproposal_t::add_replica_cert,received two different verify certificate from replica_xip=%llu",voter_xip.low_addr);
        }
        else
        {
            xerror("xproposal_t::add_replica_cert,received a duplicated validator'certificate from replica_xip=%llu",voter_xip.low_addr);
            /possible attack
        }

    }
```

图 20 validator 记录

6. 智能合约安全

系统合约包含节点注册、奖励业务、tcc 委员会、选举的接口。节点注册合约包括节点注册、节点注销、设置分红比例、更新节点类型、赎回节点存款、设置节点昵称等。

1. 节点注册

注册与更新节点时会检查节点是否存在、节点类型、节点昵称是否合法、节点分红比例是否在 0 到 100 的范围内，并且节点注册需要有类型为 xaction_type_asset_out 的 source_action，m_amount 作为节点注册的抵押

```
xreg_node_info node_info;
auto ret = get_node_info(account, node_info);
XCONTRACT_ENSURE(ret != 0, "xrec_registration_contract::registerNode2: node exist!");
common::xrole_type_t role_type = common::to_role_type(node_types);
XCONTRACT_ENSURE(role_type != common::xrole_type_t::invalid, "xrec_registration_contract::registerNode2: invalid node_type!");
XCONTRACT_ENSURE(is_valid_name(nickname) == true, "xrec_registration_contract::registerNode: invalid nickname");
XCONTRACT_ENSURE(dividend_rate >= 0 && dividend_rate <= 100, "xrec_registration_contract::registerNode: dividend_rate must be >=0 and be <= 100");

const xtransaction_ptr_t trans_ptr = GET_TRANSACTION();
XCONTRACT_ENSURE(trans_ptr->get_source_action().get_action_type() == xaction_type_asset_out && !account.empty(),
                 "xrec_registration_contract::registerNode: source_action type must be xaction_type_asset_out and account must be not empty");

xstream_t stream(xcontext_t::instance(), (uint8_t *)trans_ptr->get_source_action().get_action_param().data(), trans_ptr->get_source_action().
get_action_param().size());
```

图 21 节点注册

获取节点角色和对应需要的最小抵押，比较 mortgage 是否满足最小抵押条件

```
if (node_info.is_validator_node()) {
    if (node_info.m_validator_credit_numerator == 0) {
        node_info.m_validator_credit_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(min_credit);
    }
}
if (node_info.is_auditor_node()) {
    if (node_info.m_auditor_credit_numerator == 0) {
        node_info.m_auditor_credit_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(min_credit);
    }
}
uint64_t min_deposit = node_info.get_required_min_deposit();
xdbg(("[xrec_registration_contract::registerNode2] call xregistration_contract registerNode() pid:%d, transaction_type:%d, source action type: %d,
m_deposit: %" PRIu64
      ", min_deposit: %" PRIu64 ", account: %s\n"),
     getpid(),
     trans_ptr->get_tx_type(),
     trans_ptr->get_source_action().get_action_type(),
     asset_out.m_amount,
     min_deposit,
     account.c_str());
//XCONTRACT_ENSURE(asset_out.m_amount >= min_deposit, "xrec_registration_contract::registerNode2: mortgage must be greater than minimum deposit");
XCONTRACT_ENSURE(node_info.m_account_mortgage >= min_deposit, "xrec_registration_contract::registerNode2: mortgage must be greater than minimum
deposit");
```

图 22 节点注册抵押检查

节点注销时会获取节点是否处于惩罚时期，惩罚时期的节点无法注销

```
xslash_info s_info;
if (get_slash_info(account, s_info) == 0 && s_info.m_staking_lock_time > 0) {
    XCONTRACT_ENSURE(cur_time - s_info.m_punish_time >= s_info.m_staking_lock_time, "[xrec_registration_contract::unregisterNode]
    : has punish time, cannot deregister now");
}
```

图 23 节点注销检查

赎回 stake 时会校验赎回的时间间隔（72 小时）

```
xrefund_info refund;
auto ret = get_refund(account, refund);
XCONTRACT_ENSURE(ret == 0, "xrec_registration_contract::redeemNodeDeposit: refund not exist");
XCONTRACT_ENSURE(cur_time - refund.create_time >= REDEEM_INTERVAL, "xrec_registration_contract::redeemNodeDeposit: interval must
be greater than or equal to REDEEM_INTERVAL");
```

图 24 赎回 stake 检查

2. 奖励业务

奖励池的 20%为节点选票奖励，76%为节点工作量奖励，4%为链上治理委员会奖励。

选票奖励对象为选票数>0 且为 active 状态，保证金>0 的节点，按照节点的得票数占比分配总节点选票奖励，发放周期为 12 小时。公式为：

节点选票奖励=投票数/全网总票数*200 亿*M%*20% （M%为当年增量发行的比例）

工作量奖励按工作类型分类：

- Edger 路由：2%
- Auditor 审计：10%（各 cluster 平分，按照节点在 cluster 内的审计工作量占比分奖励）
- validator 验证：60%（各 shard 平分，按照节点在 shard 内的验证工作量占比分奖励）
- Archiver 存档：4%

```
top::xstake::uint128_t xzec_reward_contract::get_reward(top::xstake::uint128_t issuance, xreward_type reward_type) {
    uint64_t reward_numerator = 0;
    if (reward_type == xreward_type::edge_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(edge_reward_ratio);
    } else if (reward_type == xreward_type::archive_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(archive_reward_ratio);
    } else if (reward_type == xreward_type::validator_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(validator_reward_ratio);
    } else if (reward_type == xreward_type::auditor_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(auditor_reward_ratio);
    } else if (reward_type == xreward_type::vote_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(vote_reward_ratio);
    } else if (reward_type == xreward_type::governance_reward) {
        reward_numerator = XGET_ONCHAIN_GOVERNANCE_PARAMETER(governance_reward_ratio);
    }
    return issuance * reward_numerator / 100;
}
```

图 25 奖励计算

3. tcc 委员会

合法提案类型如下:

```
bool xrec_proposal_contract::is_valid_proposal_type(proposal_type type) {
    switch ( type)
    {
    case proposal_type::proposal_update_parameter:
    case proposal_type::proposal_update_asset:
    case proposal_type::proposal_add_parameter:
    case proposal_type::proposal_delete_parameter:
    case proposal_type::proposal_update_parameter_incremental_add:
    case proposal_type::proposal_update_parameter_incremental_delete:
        return true;

    default:
        return false;
    }
}
```

图 26 提案类型校验

xrec_proposal_contract::submitProposal 提交提案会进行如下步骤：

- 检查提案类型是否合法
  - 如果是 proposal_update_parameter，判断 target 是否存在于 onchain_params（更新操作要求存在该参数），比较更新的值与旧值
  - 如果是 proposal_update_asset，判断 target 是否为合法地址
  - 如果是 proposal_add_parameter，检查 target 是否存在于 onchain_params（添加操作要求不存在该参数）
  - 如果是 proposal_delete_paramter，要求 onchain_params 存在 target
  - 如果是 proposal_update_parameter_incremental_add/delete，target 仅允许值为”whitelist”
- 获取提案交易中的 sender 发送金额，与链上参数 min_tcc_proposal_deposit 比较，必须大于该参数
- 获取过期时间 tcc_proposal_expire_time
- 设置 proposal 结构信息(proposal_id, parameter,new_value,deposit 等)，end_time 设置为当前时间加上过期时间
- 清除当前已过期的 proposal，清除时退回了 proposal 的抵押

撤回提案时会检查合约调用方是否为提案发起方，才能清除提案并退回抵押。

对提案投票时的主要过程如下：

- 检查合约调用方是否为理事会成员
- 检查提案状态，不能为 failure 或 success（已完成的状态）

- 如果提案已过期，获取提案当前的投票结果，按提案不同的优先级计算提案是否通过，可以看到优先级越高的提案通过的阈值越高。

```
not_yet_voters = voter_committee_size - yes_voters - no_voters;

if (proposal.priority == priority_critical) {
    if (((yes_voters * 1.0 / voter_committee_size) >= (2.0 / 3)) && ((no_voters * 1.0 / voter_committee_size) < 0.20)) {
        proposal.voting_status = voting_success;
    } else {
        proposal.voting_status = voting_failure;
    }
} else if (proposal.priority == priority_important) {
    if ((yes_voters * 1.0 / voter_committee_size) >= 0.51 && (not_yet_voters * 1.0 / voter_committee_size < 0.25)) {
        proposal.voting_status = voting_success;
    } else {
        proposal.voting_status = voting_failure;
    }
} else {
    // normal priority
    if ((yes_voters * 1.0 / voter_committee_size) >= 0.51) {
        proposal.voting_status = voting_success;
    } else {
        proposal.voting_status = voting_failure;
    }
}
```

图 27 过期提案阈值

- 如果提案处于未过期的状态，检查当前调用者是否已经投过票，如果未投票则为提案投票，并按照下面的算法计算是否通过。

```
not_yet_voters = voter_committee_size - yes_voters - no_voters;

if (proposal.priority == priority_critical) {
    // 赞成者比例大于等于2/3，否决者比例小于20%，并且未投票者数量为0，直接通过
    if (((yes_voters * 1.0 / voter_committee_size) >= (2.0 / 3)) && ((no_voters * 1.0 / voter_committee_size) < 0.20) && (not_yet_voters == 0)) {
        proposal.voting_status = voting_success;
    } else if ((no_voters * 1.0 / voter_committee_size) >= 0.20) {
        // 当否决者比例超过20%，直接否决
        proposal.voting_status = voting_failure;
    }
} else if (proposal.priority == priority_important) {
    // 赞成者比例超过一半，并且未投票者比例小于25%，直接通过
    if ((yes_voters * 1.0 / voter_committee_size) >= 0.51 && (not_yet_voters * 1.0 / voter_committee_size < 0.25)) {
        proposal.voting_status = voting_success;
    } else if ((no_voters * 1.0 / voter_committee_size) >= 0.51) {
        // 否决者比例大于一半，直接否决
        proposal.voting_status = voting_failure;
    }
} else {
    // normal priority
    if ((yes_voters * 1.0 / voter_committee_size) >= 0.51) {
        // 赞成者数量大于一半，直接通过
        proposal.voting_status = voting_success;
    } else if ((no_voters * 1.0 / voter_committee_size) >= 0.51) {
        // 否决者数量大于一半，直接否决
        proposal.voting_status = voting_failure;
    }
}
```

图 28 未过期提案阈值

- 当提案状态为成功或失败，都作为已完成提案在现有提案集合中清除并退回抵押
- 如果提案状态是成功，应用对链上参数的修改。过程最后清除过期提案。

经审计，系统智能合约实现的功能在操作中都做了完善的校验，且无逻辑安全问题

7. 分片安全

7.1. 分片机制安全性审计

TOP 链中算力和 staking 的设计如下图所示：

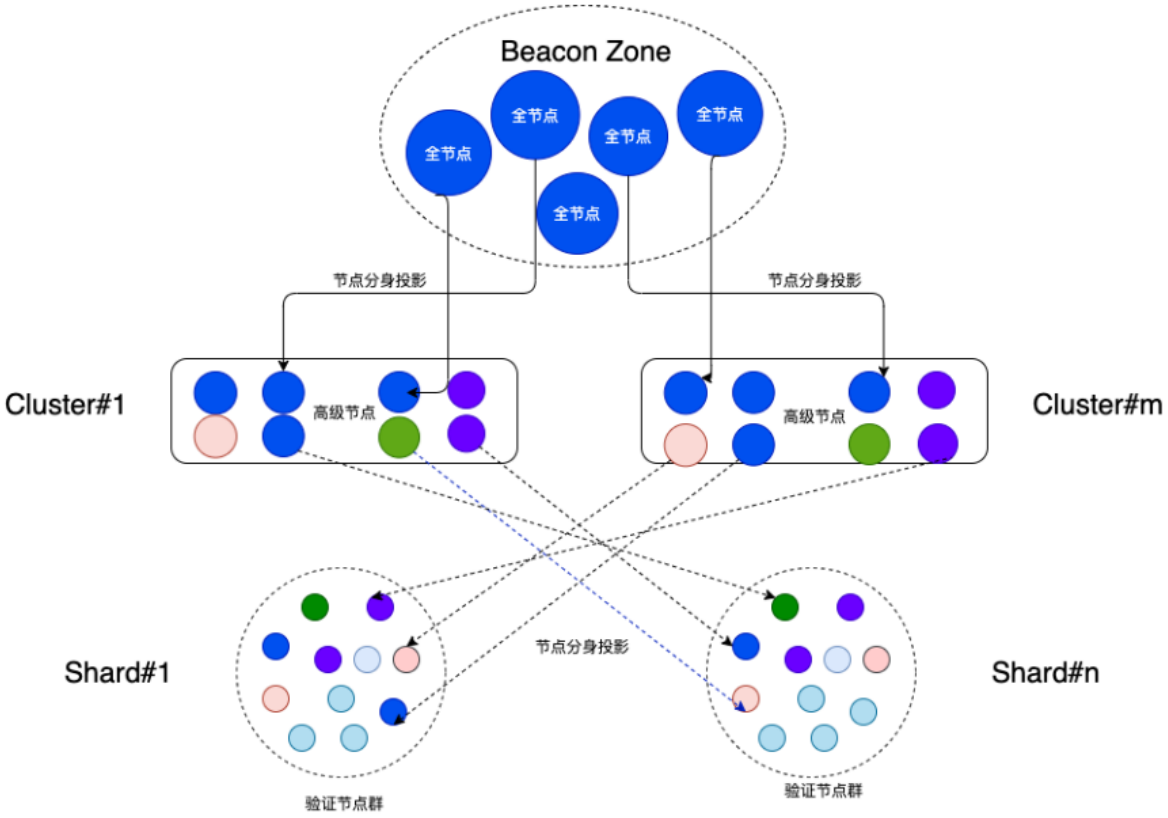

图 29 TOP 链算力和 stacking 设计

通过组合 3 层 Staking/算力，分身投影、随机轮换、随机映射等，让全网的算力和 Staking 覆盖到每一个 cluster 和 Shard，确保 Staking/算力安全。

多个分片是异步的，例如发送分片 shard1 不会等待接收分片 shard2 的收据响应(receipt confirmed)，一个分片的故障也不会导致其他分片阻塞。对于每个分片的安全性取决于每个分片验证者的选择和共识机制，TOP Chain 的分片是随机进行的，使用了 VRF 创建无法预测的随机种子，恶意节点无法控制进入特定某个分片，增加攻击成本。并且每隔一段时间，分片中的一些节点会被重新分配，随着时间推移，每个分片将具有与先前完全不同的节点。

节点会定时检查 cache，将发送未 commit 的 recv 交易

```
void xtxpool_table_t::on_timer_check_cache(xreceipt_tranceiver_face_t & tranceiver, xtxpool_receipt_receiver_counter & counter)
    std::vector<xcons_transaction_ptr_t> receipts;
    uint64_t now = xverifier::xtx_utl::get_gmttime_s();

    receipts = m_unconfirm_sendtx_cache.on_timer_check_cache(now, counter);
    if (receipts.empty()) {
        return;
    }
    for (auto & receipt : receipts) {
        if (!receipt->is_commit_prove_cert_set()) {
            xassert(receipt->is_recv_tx());
            std::string table_account = account_address_to_block_address(common::xaccount_address_t(receipt->get_source_addr()));
            uint64_t justify_table_height = receipt->get_unit_cert()->get_parent_block_height() + 2;
            base::xauto_ptr<base::xvblock_t> justify_tableblock = xblocktool_t::load_justify_block(m_para->get_vblockstore(),
            table_account, justify_table_height);
            if (justify_tableblock == nullptr) {
                xwarn("xtxpool_table_t::on_timer_check_cache can not load justify tableblock. tx=%s,account=%s,height=%ld",
                    receipt->dump().c_str(), table_account.c_str(), justify_table_height);
                continue;
            }
            receipt->set_commit_prove_with_parent_cert(justify_tableblock->get_cert());
        } else {
            xdbg("xtxpool_table_t::on_timer_check_cache tx receipt already set commit prove. tx=%s", receipt->dump().c_str());
        }

        tranceiver.send_receipt(receipt, get_receipt_resend_time(receipt->get_unit_cert()->get_gmtime(), now));
    }
}
```

图 30 定时检查

当 receipt 状态被更新为 confirmed 后会从 cache 中移除

```
// clear confirmed entries
for (auto iter = m_retry_cache.begin(); iter != m_retry_cache.end();) {
    // always erase first, then insert again
    auto current_entry = *iter;

    if (current_entry.m_receipt->is_confirmed()) {
        // remove confirmed
        xdbg(" [unconfirm cache]on_timer_check_cache is_confirmed");
        XMETRICS_COUNTER_INCREMENT("txpool_receipt_retry_cache", -1);
        m_unconfirm_tx_num--;
        iter = m_retry_cache.erase(iter);
        continue;
    }
```

图 31 清除过期交易

接收 receipt 的节点也会保存一个 cache，确认或过期后才清除。所以当一个分片所有节点宕机，或三个阶段中由于网络等其他因素某个阶段未完成，恢复正常后，未完成的 sendtx 或 receipt 还会继续完成。

对于分片数据的可用性保证，通过对 TOP 链代码的审计分析，在数据存储时主也是由 3 层组成：

1. Beacon 全节点群：存储和同步全网所有块和交易数据，确保全网数据的可用性。
2. Clustter 高级节点群：存储和同步同一 clustter 下的多个 Shard 的块和交易数据，确保一个 Cluster 的数据的可用性。
3. Shard 验证节点群：存储和同步当前 Shard 上的块和交易数据，确保一个 Shard 内的数据的可用性。

TOP 链在确保分片数据的一致性时，具体架构如下图所示：

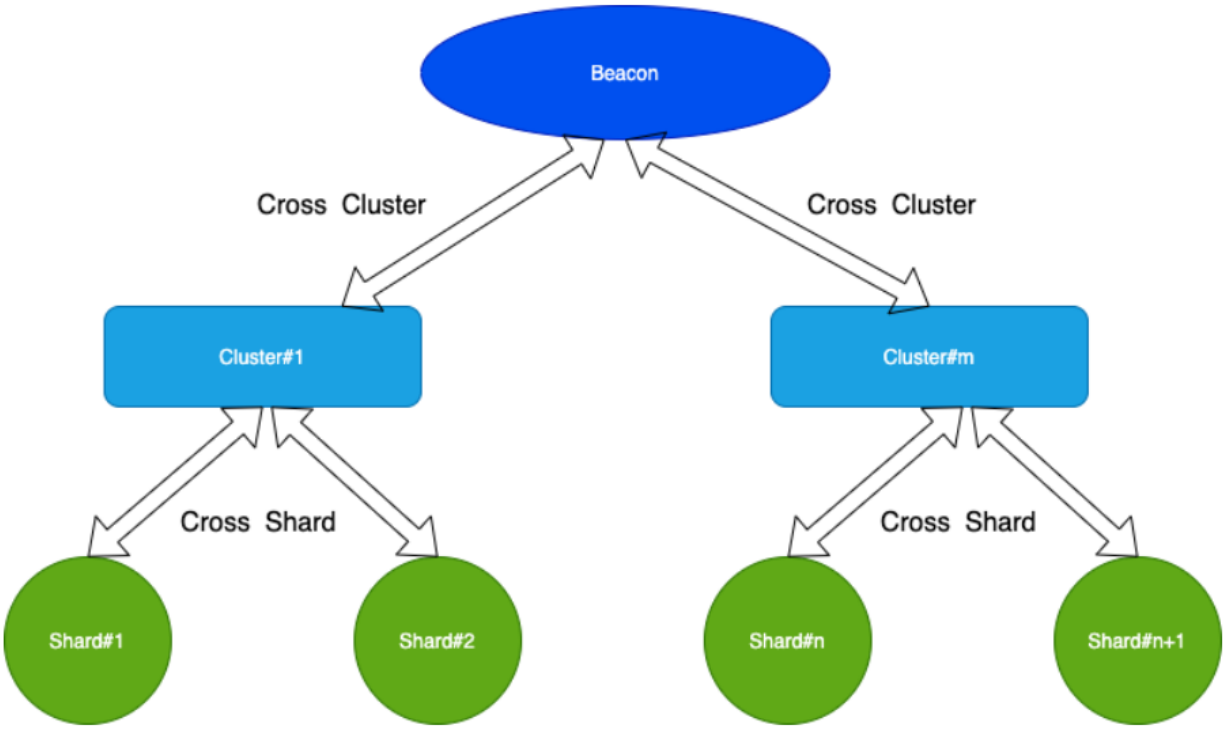

图 32 数据一致性检验

TOP 链对于分片数据的一致性的审计和检验，根据上图可知也是有 3 层组成：

1. Beacon 节点群：负责产生全网一致的时钟和 Drand 块，负责产生全网一致的节点轮换和选举结果，负责 Cross-Cluster 安全，通过对 Cluster 的块数据进行审计确保数据全局一致性。
2. Cluster 节点群：负责 Cross-Shard 的数据安全，对 Shard 的块数据进行审计，确保 Cluster 内数据一致性。
3. Shard 节点群：负责 Shard 内所有账号的数据一致性检验，以及交易执行和 state 计算。

通过这 3 层的组合，可以让 Cross-Shard 数据、Cross-Cluster 数据以及 beacon 在 state 上达成一致。

7.2. 分片交易安全性审计

跨分片交易的流程如下：

1. 根据交易的 Sender 地址映射到分片 Shard#1，分片收到并检验原始交易，执行 Sender Action

2. Shard#1 完成 Sender Action 的共识和执行，产生块和凭证广播给 Receiver 映射的 Shard#3

3. Shard#3 检验凭证，完成 Receiver Action 的共识和执行，产生块和凭证，广播给 Sender 的 Shard#1

4. Shard#1 检验凭证，完成 Confirm Action 的共识和执行，完成整个交易，并写入块作为凭证

Shard#1 一轮共识开始时会创建区块，包含 Unit block 和其输出，校验并执行 Sender Action，当区块确认后解析区块中的交易，构造收据 receipt 发送给 Shard#3

```
void xtxpool_t::on_block_confirmed(xblock_t * block) {
    xinfo("xtxpool_t::on_block_confirmed: block:%s", block->dump().c_str());

    auto handler = [this](base::xcall_t & call, const int32_t cur_thread_id, const uint64_t timenow_ms) -> bool {
        xblock_t * block = dynamic_cast<xblock_t *>(call.get_param1().get_object());
        xinfo("xtxpool_t::on_block_confirmed process, block:%s", block->dump().c_str());
        if (block->is_tableblock() && block->get_clock() + block_clock_height_fall_behind_max > this->m_para->get_chain_timer()->logic_time()) {
            make_receipts_and_send(block);
        }
        xtxpool_table_t * xtxpool_table = this->get_txpool_table_by_addr(block->get_account());
        xtxpool_table->on_block_confirmed(block);
        return true;
    };

    if (is_mailbox_over_limit()) {
        xwarn("xtxpool_t::on_block_confirmed txpool mailbox limit,drop block=%s", block->dump().c_str());
        return;
    }
    base::xcall_t asyn_call(handler, block);
    send_call(asyn_call);
}
```

图 33 区块确认时的处理

Shard#3 校验并执行 Receiver Action，同样当所在区块确认后解析区块中的交易来生成接收交易的收据，也就是 confirm，发送给 Shard#1。最终共识并执行 Confirm Action，每个阶段的区块确认后才发送 receipt 进入下一阶段。

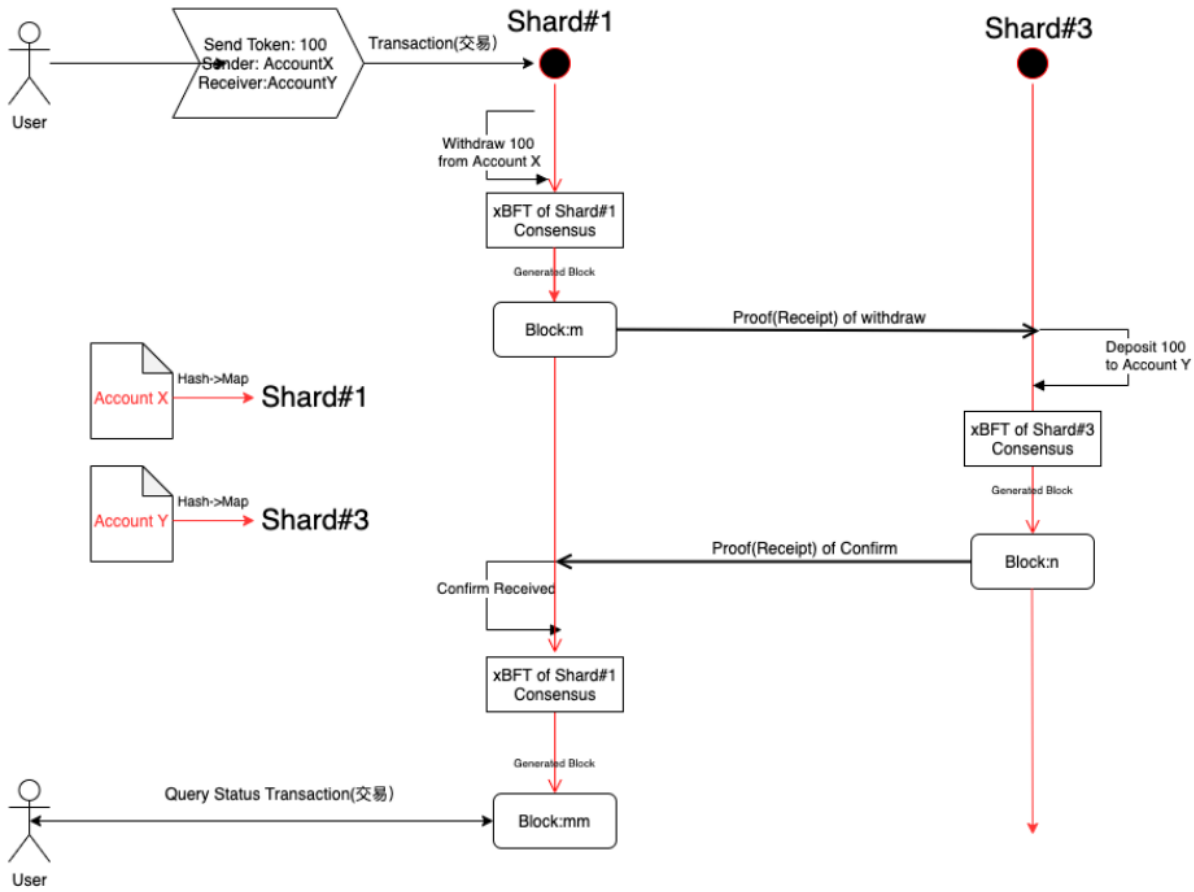

图 34 分片交易流程

查询交易状态时，会在交易的发起 shard 查询。交易的 success 和 fail 状态都通过 confirm_unit_info 中的状态决定，发送分片上 unit block 的高度为 0 时的状态为 queue，其他情况的状态为 pending。在区

块 commit 并调用 xstore::set_transaction_hash 时，判断当前交易类型，如果是 confirm 即交易确认，设置 confirm_unit_info 中的 height。

```
// src/xtopcom/xrpc/xgetblock/get_block.cpp

void get_block_handle::update_tx_state(xJson::Value & result_json, const xJson::Value & cons) {

    if (cons["confirm_unit_info"]["exec_status"].asString() == "success") {

        result_json["tx_state"] = "success";

    } else if (cons["confirm_unit_info"]["exec_status"].asString() == "failure") {

        result_json["tx_state"] = "fail";

    } else if (cons["send_unit_info"]["height"].asUInt64() == 0) {

        result_json["tx_state"] = "queue";

    } else {

        result_json["tx_state"] = "pending";

    }
}
```

更新交易状态时，通过交易的 source_addr 和 unit 高度查找对应的区块（即当前完整交易的 confirm 交易）。所以发起交易的 shard 在整个分片交易完成后才会更新交易状态为成功。

8. 总结

本公司采用模拟攻击和代码审计的方式对公链 TOP 的模块安全性以及业务逻辑安全性等方面进行多维度全面的灰盒安全审计。**经审计，TOP 公链通过所有公链安全审计项，审计结果为通过(优)**

成都链安
BEOSIN

官方网址

https://lianantech.com

电子邮箱

vaas@lianantech.com

微信公众号