Audit Report for

Public Chain Security

Audit number: 202105100040

Public chain name: TOP Chain

Audit start date: 2021.04.01

Audit completion date: 2021.05.10

Audit result: Passed (Excellent)

Audit team: Chengdu Lianan Technology Co., Ltd.

Audit type and results:

S/N	Audit type	Audit item	Audit sub-items	Audit results
1	Node Security	RPC Interface	RPC Function Implementation	**Passed**
			RPC Interface Permissions	**Passed**
		Node Tests	Malformed Data Test	**Passed**
			Node Buffer Overflow Attack	**Passed**
			DDoS attack	**Passed**
2	Wallet and Account Security	Private Key/ Mnemonic Words	Generation algorithm	**Passed**
			Storage security	**Passed**
			Usage security	**Passed**
3	Transaction Model Security	Transaction Processing Logic	Transaction or Receipt Replay Attack	**Passed**
			Attack Through Malformed Transaction, Forged Transaction, or Repeated Transaction	**Passed**
			Dusting Attack	**Passed**
			Transaction Flood Attack	**Passed**
			Double Spend and Over Spend Attack	**Passed**
		Other Transaction Security Tests	Transaction Malleability Attack	**Passed**
			Fake Recharge Attack	**Passed**
			Command Line Transfer Method	**Passed**

4	Consensus Security	Consensus Mechanism Design	Leader Election and VRF Mechanism	**Passed**
			Shard Node Rotation Mechanism	**Passed**
			Consensus Algorithm (including xBFT)	**Passed**
		Consensus Verification Implementation	Is it possible to construct a legal block with less than the expected cost	**Passed**
5	Signature Security	Signature Verification	Multi-Signature Verification Security	**Passed**
			Illegal Signature Attack	**Passed**
			Node Double-Sign and Re-Sign Attack	**Passed**
			Signature Forgery	**Passed**
6	Smart Contract Security	System Contract Security	Contract Execution Logic	**Passed**
			Node Reward Calculation	**Passed**
			Node Slash Contract	**Passed**
			Node Election	**Passed**
			On-chain Governance	**Passed**
7	Shard Security	Sharding Mechanism Security	Single Shard Attack	**Passed**
			Shard Restart	**Passed**
			Shard Staking and Computing Power Security	**Passed**
			Sharded Data Availability	**Passed**
			Sharded Data Consistency	**Passed**
		Shard Transaction Security	Shard Transaction Reliability	**Passed**
			Shard Transaction Integrity	**Passed**

Disclaimer: This report is an audit of the project code. Any description, statement, or wording in this report shall not be interpreted as an endorsement, affirmation, or confirmation of the project as a whole. This audit is only conducted for the audit types specified in this report and the scope of the audit types given in the results table. Any other potential security vulnerabilities not specified in the report are not included in the scope of this audit. Chengdu Lianan Technology only issues this report in reference to the security status from potential attacks or vulnerabilities before the issuance of this report. Chengdu Lianan Technology cannot judge the possible impact on the security status of the public chain for new attack vectors or vulnerabilities that may exist in the future, and therefore is not responsible for them. The security audit analysis and other content generated by this report are based solely on the documents and materials provided to Chengdu Lianan Technology by the public chain provider before the issuance of this report, with the assumption that there are no missing, tampered, deleted, or concealed documents or materials. If there are missing, tampered, deleted, or concealed documents or materials, or the documents and materials provided have any changes made to them after the issuance of this report, Chengdu Lianan Technology will not bear any responsibility for the losses or adverse effects caused thereby. This audit report issued by Chengdu Lianan Technology is generated in reference to the documents and materials provided by the public chain provider, which is based on technology of which Chengdu Lianan Technology has ample expertise. Due to the technical limitations that exist in any organization, there is always the possibility that not all risks are fully accounted for. As such, Chengdu Lianan Technology will not bear any responsibility for the resulting consequences of any risks not detected in this audit report.

The final interpretation rights of this disclaimer belongs to Chengdu Lianan Technology.

Description of audit results:

Our company conducted multi-dimensional and comprehensive security audits on the three aspects of TOP public chain: Code standardization, Security, and Business logic. After completing the audit, TOP public chain passed all audit items, and the audit result is **Passed (Excellent)**.

Key code modules:

- System contract code: src/xtopcom/xvm
- Consensus code: src/xtopcom/xBFT
- Signature code: src/xtopcom/xcertauth
- Multi-signature code: src/xtopcom/xmutisig
- Key generation code: src/xtopcom/xcrypto
- Transaction execution: src/xtopcom/xtxexecutor
- Node private key management: src/xtopcom/xtopcl

BEOSIN Blockchain Security

Contents

1. Node Security

1.1. RPC Interface

1.1.1. RPC Function Implementation

RPC interface list

Query class:

```
xcluster_query_manager::xcluster_query_manager(observer_ptr<store::xstore_face_t> store,
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);
}
```

Send transaction:

```
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_accessor);
    m_edge_handler_ptr->init();
    EDGE_REGISTER_V1_ACTION(T, sendTransaction);
}
```

Request parameters:

Parameter	Description
body	Business parameters
identity_token	Identity token (unused, not verified)
method	Request method
sequence_id	Times of session
target_account_addr	Account address (unused, not verified)
version	RPC API version (fixed to 1.0)

The main request parameter account_addr is contained in body, which is an encoded JSON string, for example:


```
body=%7B%22params%22%3A%7B%22account_addr%22%3A%22T00000Lhj29VReFAT958ZqFWZ2ZdMLot2PS5D5YC%22%7D%7D%0A&identity_token=&
    method=getAccount&sequence_id=1&target_account_addr=1&version=1.0
```



```
{"data":{"account_addr":"T00000Lhj29VReFAT958ZqFWZ2ZdMLot2PS5D5YC","available_gas":25000,"balance":2999997000000000,
    "burned_token":0,"cluster_id":1,"created_time":1573189200,"disk_staked_token":0,"gas_staked_token":0,"group_id":64,
    "latest_tx_hash":"0xbdf39f175db02a52b254305a902fa13a798b75e5daf5cd21826db20c80be4b2d",
    "latest_tx_hash_xxhash64":"0x3f694c15c7f81684","latest_unit_height":0,"lock_balance":0,"nonce":1,"recv_tx_num":0,
    "total_free_gas":25000,"total_gas":25000,"total_stake_gas":0,"unconfirm_sendtx_num":0,"unlock_disk_staked":0,
    "unlock_gas_staked":0,"unused_free_gas":25000,"unused_stake_gas":0,"unused_vote_amount":0,"vote_staked_token":0,"zone_id":0},
    "errmsg":"OK","errno":0,"sequence_id":"1"}
```


Figure 1 RPC request

target_account_addr and identity_token are currently not used, but target_account_addr cannot be left blank; the parameter sequence_id has not been checked and can be any character or string; version is fixed to 1.0.

In addition to using tools such as curl/postman to directly call RPC requests, you can also use the official topio client, which essentially combines and constructs each RPC request.

1.1.2. RPC Interface Permissions

In this section, we check whether there is any RPC API request that can be operated without authorization, and whether there is any leakage of sensitive information or arbitrary transaction issuance. On-chain operations (transfer, node staking, node registration, node voting, node rewards, contract calling, etc.) are all completed by sendTransaction. The identity of the request initiator is checked by verifying the signature.

RPC requests in TOP Chain are all completed by edge miner nodes. RPC service interfaces are not used/visible to nodes of other roles.

RPC service initialization code:

```
// src/xtopcom/xrpc/xrpc_init.cpp
xrpc_init::xrpc_init(
                //...
)
{
    assert(nullptr != vhost);
    assert(nullptr != router_ptr);
    // Determine node type
    switch (node_type) {
    // Verify node
    case common::xnode_type_t::consensus_validator:
        assert(nullptr != txpool_service);
        assert(nullptr != store);
        init_rpc_cb_thread();
        // Interface between shards
        m_shard_handler = std::make_shared<xshard_rpc_handler>(vhost, txpool_service, make_observer(m_thread));
        m_shard_handler->start();
        break;
    // Audit node, ZEC election committee, REC election committee
    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;
    // Edge node
    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();
        // Start http service
        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);
        // Start websocket service
        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;
        }
}
```

The verification of the RPC interface provided by the edge node for the request initiator is primarily the signature check performed in the sendTransaction method.

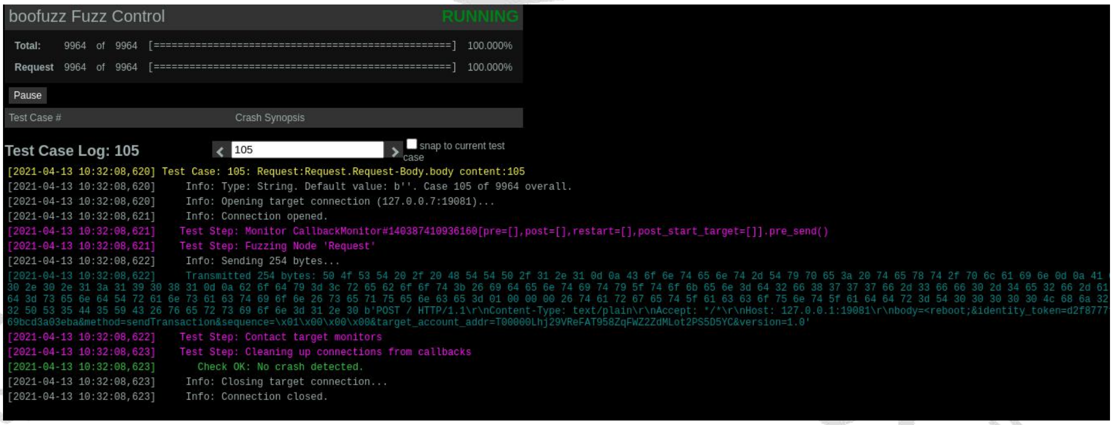
```
// src/xtopcom/xdata/src/xtransaction.cpp
bool xtransaction_t::sign_check() const {
        static std::set<uint16_t> no_check_tx_type { xtransaction_type_lock_token,
xtransaction_type_unlock_token };
        std::string addr_prefix;
        // Obtain operation address
        if (std::string::npos != 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;
        }
        // Signature body in the transaction structure
        utl::xecdsasig_t signature_obj((uint8_t *)m_authorization.c_str());
        // Determine address type and signature verification
        // verify_signature internally uses the API of SECP256K1 to verify ECDSA signature
        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_parent_account());
        } else {
            return key_address.verify_signature(signature_obj, m_transaction_hash);
        }
}
```

Except for the sendTransaction interface, all other query interfaces are public queries. After auditing, no unauthorized access or leakage of sensitive information was found.

1.2. Node Test

1.2.1. Fuzz Testing

A tool built on Sulley was used to fuzz test open RPC services. The test process is as follows:



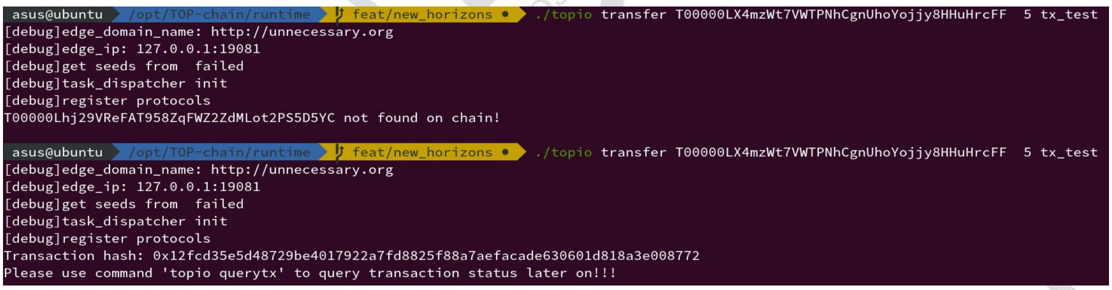
```
boofuzz Fuzz Control                                              RUNNING
Total:    9964  of  9964  [==================================================]  100.000%
Request   9964  of  9964  [==================================================]  100.000%
Pause
Test Case #                          Crash Synopsis

Test Case Log: 105          105        snap to current test case
[2021-04-13 10:32:08,620] Test Case: 105: Request:Request.Request-Body.body content:105
[2021-04-13 10:32:08,620]     Info: Type: String. Default value: b''. Case 105 of 9964 overall.
[2021-04-13 10:32:08,620]     Info: Opening target connection (127.0.0.7:19081)...
[2021-04-13 10:32:08,621]     Info: Connection opened.
[2021-04-13 10:32:08,621]    Test Step: Monitor CallbackMonitor#140387410936160[pre=[],post=[],restart=[],post_start_target=[]].pre_send()
[2021-04-13 10:32:08,621]    Test Step: Fuzzing Node 'Request'
[2021-04-13 10:32:08,622]     Info: Sending 254 bytes...
[2021-04-13 10:32:08,622]     Transmitted 254 bytes: 50 4f 53 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c 61 69 6e 0d 0a 41
30 2e 30 2e 31 3a 31 39 30 38 31 0d 0a 62 6f 64 79 3d 3c 72 65 62 6f 6f 74 3b 26 69 64 65 6e 74 69 74 79 5f 74 6f 6b 65 6e 3d 64 32 66 38 37 37 37 66 2d 33 66 66 30 2d 34 65 32 66 2d 61
64 3d 73 65 6e 64 54 72 61 6e 73 61 63 74 69 6f 6e 26 73 65 71 75 65 6e 63 65 3d 01 00 00 00 26 74 61 72 67 65 74 5f 61 63 63 6f 75 6e 74 5f 61 64 64 72 3d 54 30 30 30 30 30 4c 68 6a 32
32 50 53 35 44 35 59 43 26 76 65 72 73 69 6f 6e 3d 31 2e 30 b'POST / HTTP/1.1\r\nContent-Type: text/plain\r\nAccept: */*\r\nHost: 127.0.0.1:19081\r\nbody=<reboot;&identity_token=d2f8777
69bcd3a03eba&method=sendTransaction&sequence=\x01\x00\x00\x00&target_account_addr=T00000Lhj29VReFAT958ZqFWZ2ZdMLot2PS5D5YC&version=1.0'
[2021-04-13 10:32:08,622]    Test Step: Contact target monitors
[2021-04-13 10:32:08,623]    Test Step: Cleaning up connections from callbacks
[2021-04-13 10:32:08,623]      Check OK: No crash detected.
[2021-04-13 10:32:08,623]     Info: Closing target connection...
[2021-04-13 10:32:08,623]     Info: Connection closed.
```

Figure 2 Fuzz test

In the final test results, no malformed data that could cause a node to crash was detected. After fuzz testing and code auditing, no buffer overflow and/or DoS attack caused by malformed data was found.

1.2.2. Illegal Transaction Test

To test the handling of illegal transactions, sender and receiver addresses were generated in batches, after which large numbers of illegal transactions were continuously sent. Due to the flow restrictions imposed by edge nodes, normal transactions could not be sent during the continuous sending of illegal transactions:



```
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!!!
```

Figure 3 Illegal transaction test

After recompiling the node with the noratelimit flag, normal transactions could be successfully processed during the continuous sending of illegal transactions.

2. Wallet and Account Security

2.1. Private Key Generation Algorithm

```
// 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));
	// Generate random buffer
	xrandom_buffer(m_private_key,sizeof(m_private_key));
	uint256_t hash_value;
	xsha2_256_t hasher;
	// Time seed
	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
	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();
}
```

The special file /dev/urandom is used by the system for pseudo-random number generation and is used to create random number seeds. Seeds are generated using device drivers and environmental noise from other sources, which is the recommended random number seed generation mechanism for Unix-like systems.

```
static uint32_t xrandom32()
{
	//get_sys_random_number might be replaced by std::random_device without xbase lib
	const uint64_t seed = base::xsys_utl::get_sys_random_number() +
base::xtime_utl::get_fast_random();
	return (uint32_t)(seed >> 8);
}
```

In addition, a private key generation function based on a random seed is also provided.

```
xecprikey_t::xecprikey_t(const std::string rand_seed) //sha256(rand_seed.32bytes random)-
```

```
>private key
{
	//...
	hasher.update(rand_seed);
	//...
}
```

2.2. Storage Security

The TOPIO client provided by TOP Chain stores private keys in the form of a file. The keystore information is encrypted with AES-256 before writing to the file.

```
// src/xtopcom/xtopcl/src/xcrypto.cpp
void aes256_cbc_encrypt(const std::string & pw, const string & raw_text, std::ofstream & key_file) {
	AES_INFO aes_info;
	fill_aes_info(pw, raw_text, aes_info);
	// Write encrypted information (initialization vector, ciphertext, etc.)
	writeKeystoreFile(key_file, aes_info.iv, aes_info.ciphertext, aes_info.info, aes_info.salt, aes_info.mac);
}
```

Example of keystore file:


```
{
   "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="
}
```


Figure 4 keystore

If the user sets a password when creating the account, the password is required when resetting the password (resetkeystorepwd) and importing the keystore (importKey).

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

```
string import_existing_keystore(const string & cache_pw, const string & path, bool auto_dec) {
	auto key_info = parse_keystore(path);
	if (key_info.empty()) {
		return "";
	}
	// Decrypt
	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. Use/Visibility of Private Key

In the process of using private keys for various RPC interface tests, (such as importing a keystore file, private key signing etc.), private keys do not appear in any logs or files, which meets the safety criteria for private key use.

For example, the use of a private key when transferring via command line.

```
// src/xtopcom/xtopcl/src/api_method_imp.cpp
bool api_method_imp::transfer(
					//...
				) {
	// ...
	// private key signature
	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. Transaction Model Security

3.1. Transaction Processing Logic

3.1.1. Transaction Type and Procedure

```
// 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 other 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
		// Deploy user contract
		case xtransaction_type_create_contract_account:
		// Call the contract
		case xtransaction_type_run_contract:
```

```
            // Normal transfer
            case xtransaction_type_transfer:
            // Vote to advanced miners
            case xtransaction_type_vote:
            // Cancel the voting
            case xtransaction_type_abolish_vote:
            // Lock token to swap for gas
            case xtransaction_type_pledge_token_tgas:
            // Unlock the token swapped for gas
            case xtransaction_type_redeem_token_tgas:
            // Lock token to swap for votes
            case xtransaction_type_pledge_token_vote:
            // Unlock the token swapped for votes
            case xtransaction_type_redeem_token_vote:
            return true;
            default:
            return false;

        }
}
```

If, for example, the client initiates an RPC request, the edge node will call the do_local_method to process the transaction request, which mainly verifies the signature and hash in the request.

```
// src/xtopcom/xrpc/xedge/xedge_method_manager.hpp:sendTransaction_method
// Calculate the transaction hash to verify if it is consistent with the hash in request.
if (!tx->digest_check()) {
        throw xrpc_error{enum_xrpc_error_code::rpc_param_param_error, "transaction hash
error"};
}
// Check the signature if the receiver address is not equal to the system address
sys_contract_rec_standby_pool_addr, or target_action is not equal to nodeJoinNetwork.
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"};
        }
}
```

After the initial verification, the edge node calls forward_method to forward the request and send the transaction to the corresponding Audit Network according to the network shard to which the receiver's account belongs.

```
int32_t xtxpool_service::request_transaction_consensus(const data::xtransaction_ptr_t & tx,
bool local) {
        // ...
        // Verify the source of the transaction, which may come from local or external network.
The transactions sent locally can only be system contract transactions and should not contain
authorization field. Non-local transactions cannot be system contract transactions and must
contain authorization field.
        int32_t ret = xverifier::xtx_verifier::verify_send_tx_source(tx.get(), local);
```

```
    if (ret) {
        // ...
        return ret;
    }
    // Transaction source address mapping to table id
    auto tableid = data::account_map_to_table_id(common::xaccount_address_t{tx-
>get_source_addr()});
    // Determine whether it belongs to the current network by table id
    if (!is_belong_to_service(tableid)) {
        // ...
        return xtxpool::xtxpool_error_transaction_not_belong_to_this_service;
    }
    // Determine whether the transaction target address is a system contract account and
belongs to the consensus zone.
    if (is_sys_sharding_contract_address(common::xaccount_address_t{tx-
>get_target_addr()})) {
    // If yes, obtain the sub-address and add it to the transaction target address.
        tx->adjust_target_address(tableid.get_subaddr());
    }
    // Add the transaction to the transaction pool of the source account network.
    return m_txpool->push_send_tx(tx);
}
```

The receiver then calls push_recv_tx or push_recv_ack_tx after receiving the receipt.

```
//src/xtopcom/xtxpool/src/xtxpool.cpp
int32_t xtxpool_t::on_receipt(const data::xcons_transaction_ptr_t & cons_tx) {
    int32_t ret;
    // If it is the transaction receiver, add the transaction receipt data to the transaction pool.
    if (cons_tx->is_recv_tx()) {
        XMETRICS_COUNTER_INCREMENT("txpool_receipt_recv_total", 1);
        return push_recv_tx(cons_tx);
    } else {
        // If it is the transaction sender, what is received at this time is the receipt of receiver
accepting the transaction.
        return push_recv_ack_tx(cons_tx);
    }

}
```

The main code for transaction execution is in src/xtopcom/xtxexecutor, and classes corresponding to different transaction types are defined in xtransaction_context.h. The code will be executed when the transaction is packaged (make_block) and verified (verify_block).

Figure 5 Functions of transaction execution

3.1.2. Transaction and Receipt Replay Attacks

In this section, we check whether the transactions on TOP Chain can be replayed on different chains of the same type or on the same chain. In Top Network, if two identical transactions are replayed on the same chain, when the replay interval exceeds a certain amount, the time stamp verification will not pass.

```
// 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_timestamp_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 hash: %s",
trx_ptr->get_digest_hex_str().c_str());
    return xverifier_error::xverifier_success;
}
```

Timestamp verification failing for a replay transaction:

```
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
```

Figure 6 Timestamp verification

If a transaction is replayed at a similar time, it will be discarded when it fails to pass the repeated transaction verification.

```
// src/xtopcom/xtxpool/src/xaccountobj.cpp
// Check if there is a transaction in m_tx_map
auto map_it = m_tx_map.find(tx->get_transaction()->get_digest_str());
if (map_it != m_tx_map.end()) {
    // Discard illegal transaction, the error code is xtxpool_error_request_tx_repeat
    drop_invalid_tx(tx, xtxpool_error_request_tx_repeat);
    return xtxpool_error_request_tx_repeat;
}
```

In the case of different chains of the same type, such as chain forks, checking the account nonce and last_tx_hash can prevent this type of replay attack.

```
// src/xtopcom/xtxpool/src/xaccountobj.cpp
// Compare the nonce value of the current transaction with the number of transactions sent by
the account
if (tx->get_transaction()->get_last_nonce() < m_latest_send_trans_number) {
    // Discard illegal transaction, the error code is xtxpool_error_tx_nonce_too_old
    drop_invalid_tx(tx, xtxpool_error_tx_nonce_too_old);
        return xtxpool_error_tx_nonce_too_old;
    }
    // ...
    // If the sending queue is empty
    if (m_send_queue.empty()) {
        // The current transaction nonce must be equal to the number of transactions sent by
the account.
        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;
        }
        // Transaction last_tx_hash must be equal to the m_latest_send_trans_hash of the
account.
        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 {
        // If the sending queue is not empty
        // Obtain the last transaction in the queue
        auto iter = m_send_queue.rbegin();
        auto cons_tx_tmp = iter->m_tx->get_transaction();
        // The nonce of the transaction must be continuous with the nonce of the last transaction
(+1).
        if (tx->get_transaction()->get_last_nonce() == cons_tx_tmp->get_last_nonce() + 1) {
        // The last_tx_hash of the transaction must be equal to the hash of the last transaction.
            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) {
            // Discard if nonce is not continuous.
            drop_invalid_tx(tx, xtxpool_error_tx_nonce_incontinuity);
            return xtxpool_error_tx_nonce_incontinuity;
        } else {
            // If the nonce is smaller than the specified value, compare the current transaction
with other transactions in the queue. If the nonce is repeated but the current transaction
timestamp is newer, the corresponding queue transaction will be discarded.
            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;
        }
}
```

In terms of receipt processing via push_recv_tx, duplicate receipts will be deleted.

```
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);
        // Delete duplicate receipt
        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;
}
```

And it will also check whether the start/create time of the receipt cert is duplicated, which means that receipts of which the cert is created at the same time cannot be replayed.

```
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;
}
```

Figure 7 Check whether block clocks are duplicated

3.1.3. Dusting Attack

A dusting attack is an attack wherein the attacker sends a very small amount of tokens to a user's wallet, which is termed "dust." The attacker traces the dusted wallet funds and all transactions, and then traces these addresses to determine the companies and/or individuals to which these wallet addresses belong, undermining the anonymity of the blockchain. Dusting can also consume the available resources of a blockchain, causing a shortage in the memory pool of the blockchain. If dust funds are not transferred, the attacker cannot establish a connection with the receiving wallet, and the anonymity of the wallet or address owner will not be compromised.

TOP Chain uses an account model different from Bitcoin's UTXO model. In the UTXO model, the "balance" of the user's wallet is composed of several unspent transaction outputs. When the user makes a transfer, the dust UTXO will always be involved in the user's transfer transaction. A Bitcoin transaction is composed of inputs and outputs, so a transaction can be connected in series through UTXOs to eventually achieve the purpose of de-anonymization via dusting attack.

In TOP Chain, after dusted funds are sent to the user's wallet, the amount is added to the user's balance. It is not independent of the user's balance, so attackers cannot achieve de-anonymization. Additionally, each transaction in TOP Chain consumes a certain amount of gas. When the free gas is used up, tokens need to be locked to swap for additional gas to prevent unrestricted consumption of the blockchain's resources by dust transactions.

```
    // user fire each transactions linked as a chain
    uint64_t     m_latest_send_trans_number{0};  // heigh or number of transaction
    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 transaction
    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,susper
    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;
```

Figure 8 Account attributes

3.1.4. Transaction Flood Attack

For normal transactions, after the allocated disk space is used up, tokens must be deposited to obtain additional disk space for initiating and permanently storing new transactions.

For Beacon system contract transactions, in addition to gas consumption, a handling fee will be automatically deducted from the sender of the transaction and then subsequently burned. The fee is determined by the on-chain governance parameter beacon_tx_fee, which is currently 100*10^6 uTOP. This can protect the system from transaction flood attacks.

```
// src/xtopcom/xtxexecutor/src/xtransaction_fee.cpp
uint64_t xtransaction_fee_t::cal_service_fee(const std::string& source, const std::string& target) {
    uint64_t beacon_tx_fee{0};
    #ifndef XENABLE_MOCK_ZEC_STAKE
    // Set beacon_tx_fee if the source address is not the system contract address and the target address is the beacon contract address.
    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. Double Spend Attack

For TOP Chain, each transaction for an account contains a unique & incremental nonce and the hash value of the previously confirmed transaction. Under normal circumstances, an attacker does not have the chance to launch double spend attacks. For Bitcoin and other blockchains which operate via computing power competition (e.g. PoW), when an attacker has more than 50% of the total computing power, it is possible to successfully launch a double spend attack by racing via compute power to create a longer chain where the double spend transaction is included. TOP Chain uses the hpPBFT-PoS* consensus mechanism where double spend attacks essentially do not exist.

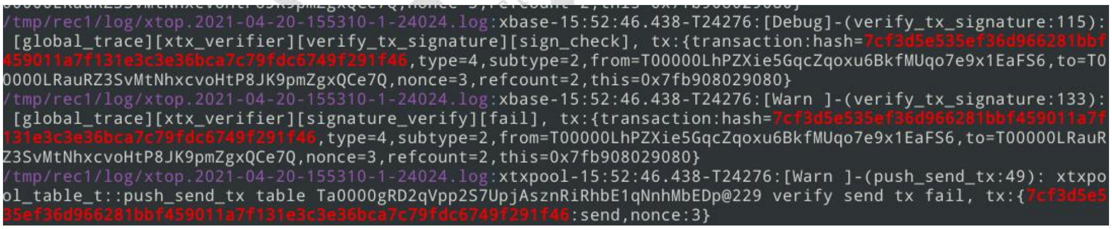
3.1.6. Illegal Transaction

In this section, we check whether there are any vulnerabilities from attacks centered around malformed or forged transactions. Malformed transactions were covered in the node malformed data test.

A user signs the entire transaction when initiating a transaction, and so any modification to the data within the transaction will cause it to fail the signature verification check at the edge node.

```
// src/xtopcom/xrpc/xedge/xedge_method_manager.hpp:sendTransaction_method
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"};
        }
}
```

If the edge node is malicious, its signature verification is of course ineffective. However, the verification node will also verify the transaction signature, and so the forged transaction will still fail verification.



```
/tmp/rec1/log/xtop.2021-04-20-155310-1-24024.log:xbase-15:52:46.438-T24276:[Debug]-(verify_tx_signature:115):
 [global_trace][xtx_verifier][verify_tx_signature][sign_check], tx:{transaction:hash=7cf3d5e535ef36d966281bbf
459011a7f131e3c3e36bca7c79fdc6749f291f46,type=4,subtype=2,from=T00000LhPZXie5GqcZqoxu6BkfMUqo7e9x1EaFS6,to=T0
0000LRauRZ3SvMtNhxcvoHtP8JK9pmZgxQCe7Q,nonce=3,refcount=2,this=0x7fb908029080}
/tmp/rec1/log/xtop.2021-04-20-155310-1-24024.log:xbase-15:52:46.438-T24276:[Warn ]-(verify_tx_signature:133):
 [global_trace][xtx_verifier][signature_verify][fail], tx:{transaction:hash=7cf3d5e535ef36d966281bbf459011a7f
131e3c3e36bca7c79fdc6749f291f46,type=4,subtype=2,from=T00000LhPZXie5GqcZqoxu6BkfMUqo7e9x1EaFS6,to=T00000LRauR
Z3SvMtNhxcvoHtP8JK9pmZgxQCe7Q,nonce=3,refcount=2,this=0x7fb908029080}
/tmp/rec1/log/xtop.2021-04-20-155310-1-24024.log:xtxpool-15:52:46.438-T24276:[Warn ]-(push_send_tx:49): xtxpo
ol_table_t::push_send_tx table Ta0000gRD2qVpp2S7UpjAsznRiRhbE1qNnhMbEDp@229 verify send tx fail, tx:{7cf3d5e5
35ef36d966281bbf459011a7f131e3c3e36bca7c79fdc6749f291f46:send,nonce:3}
```

Figure 9 Signature verification

3.2. Other Types of Transaction Security

3.2.1. Transaction Malleability Attack

Transaction malleability will cause inconsistencies in transaction IDs, resulting in users not being able to locate sent transactions, and affecting recharges or withdrawals from wallets. The transaction signature in TOP Chain is separated from other transaction data. Changing the transaction signature will not change the transaction hash. If other transaction data is changed, the signature verification will fail. Additionally, the Schnorr signature algorithm is used, which does not have the malleability issues of ECDSA signatures.

```
// src/xtopcom/xdata/src/xtransaction.cpp:digest_check
bool xtransaction_t::digest_check() const {
	base::xstream_t stream(base::xcontext_t::instance());
	// Hash calculation
	do_write_without_hash_signature(stream, true);
	uint256_t hash = utl::xsha2_256_t::digest((const char*)stream.data(), stream.size());
	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. Fake Recharge Attack

When the client receives a transaction, the transaction status will be returned. There are four types of transaction statuses possible in the response: success, fail, queue, and pending.

```
// 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";
	}
}
```

The corresponding status will be updated after the transaction is executed.

```
// 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:fail
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());
}
```

When the client performs the recharge verification, only transactions with the success status can be executed, therefore making it difficult to launch a fake recharge attack.

3.2.3. Command Line Transfer Method

In the official topio client, the transfer command is:

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

The length of the note is restricted to a maximum of 128 bytes in the command line transfer method.

```
void ApiMethod::transfer1(std::string & to, double & amount_d, std::string & note, double &
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);
}
```

the topio client signs the transaction and sends it to the node through the RPC API, and the subsequent node processing flow is consistent with other RPCs.

4. Consensus Security

TOP Chain uses the hpPBFT-PoS* consensus mechanism. HpPBFT stands for High-speed Parallel Practical Byzantine Fault Tolerance. Refer to HotStuff for implementation details.

In HotStuff theory, a block is considered as final after three consecutive stages of confirmation. The three stages in HotStuff are: prepareQC, lockedQC, and commitQC. After these three stages, a transaction can be considered completed with 100% certainty, that is, it is necessary to prove that there are no two conflicting commitQCs.

Assuming that A and B are two conflicting blocks, it is impossible for A and B to have the same block height, because the submission of a proposal requires a majority of nodes to vote. Each node will only vote for one proposal at each stage, and it is impossible to have two proposals with more than half of the votes at the same height.

Assuming that A and B have different block heights, set qc1.node=A, qc2.node=B, v1=qc1.viewNumber, v2=qc2.viewNumber, and suppose v1<v2. qcs is the legal prepareQC certificate with the smallest height that is greater than A and conflicts with A. qcs.viewNumber=vs, and the pseudo code is expressed as:

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

Now we can set a switch point qcs, which can be regarded as the starting position of the "conflict":

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

Part of the signed result tsign(<qc1.type,qc1.viewNumber,qc1.node>) of a correct copy will be sent to the leader, so that r becomes the first copy that contributes to tsign(<qcs.type, qcs.viewNumber,qcs.node>). Such r must exist, otherwise one of qc1.sig or qcs.sig cannot be created.

In view v1, copy r1 updates lockedQC in the precommitQC phase, which corresponds to A. Due to the minimization definition of vs, the lockedQC generated at A by copy r will not change before qcs is formed, otherwise r must have seen the prepareQC of other views, which does not meet the minimization assumption of vs. Copy r calls safeNode in the prepare phase of view vs, where message m contains m.node=qcs.node. Assuming that m.node conflicts with lockedQC.node, it cannot pass the safety check of safeNode (return false).

```
// hotstuff algorithm
function safeNode(node, qc):
      return (node extends from lockedQC.node)// safety rule
      (qc.viewNumber > lockedQC .viewNumber ) // liveness rule
```

In addition, m.justify.viewNumber>v1 will violate the assumption of the minimum value of vs.

Therefore, the liveness in safeNode verification also fails. So r cannot perform prepare voting for vs, which means qcs cannot be generated at all, and there cannot be two conflicting commitQCs.

In order to solve the security issues of using the BFT algorithm on a permissionless chain, TOP Chain uses proof of stake to raise the barrier for nodes to participate in consensus. A node's "Comprehensive Stake" is affected by multiple factors: deposit (TOP token), credit score, and the number of votes received. The formulae used to calculate a node's Comprehensive Stake (both auditor and validator) are as follows:

Auditor stake = (miner's deposit + miner's total number of votes / 2) * auditor credit score

```
// 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_credit_numerator / m_auditor_credit_denominator;
        }
        return stake;
}
```

Validator stake = sqrt [(miner's deposit + miner's total number of votes / 2) * validator credit score]

```
// src/xtopcom/xstake/xstake_algorithm.h
uint64_t get_validator_stake() const noexcept {
        uint64_t stake = 0;
        if (is_validator_node()) {
                // on-chain governance parameter maximum validator stake
                auto max_validator_stake =
XGET_ONCHAIN_GOVERNANCE_PARAMETER(max_validator_stake);
                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;
}
```

A consensus cluster includes an auditor group and two validator groups, and the elections of clusters are independent of each other. The consensus cluster completes the BFT rounds through a three-phase submission paradigm. Leader selection is determined by VRF-FTS (Follow-The-Satoshi). Random number seeds are generated through VRF and weighted by Comprehensive Stake. Tracking of the node's workload, contribution, total deposit and votes is completed by a series of contracts on the Beacon chain.

```
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;
}
```

Figure 10 Random election

Various factors are considered in the Comprehensive Stake, reducing the probability of malicious nodes being elected as the leader. Additionally, consensus node will regularly take turns in and out of shards. The shard rotation mechanism is implemented by the system smart contract:

```
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);
```

Figure 11 Election of auditor and validator

A consensus cluster includes auditor groups and validator groups, which are determined by the parameters auditor_group_count and validator_group_count. Nodes are selected to enter/leave the consensus cluster according to their Comprehensive Stake.

```cpp
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;
    }
```

Fig 12 Calculate comprehensive stake

4.1. Consensus Procedure

A round of BFT starts with the leader initiating a proposal. Other nodes vote on the proposal, and when enough votes have been collected, the leader broadcasts a commit message to the other nodes to complete a round of BFT. A view change occurs after a round of BFT, which will trigger a new block to be generated.

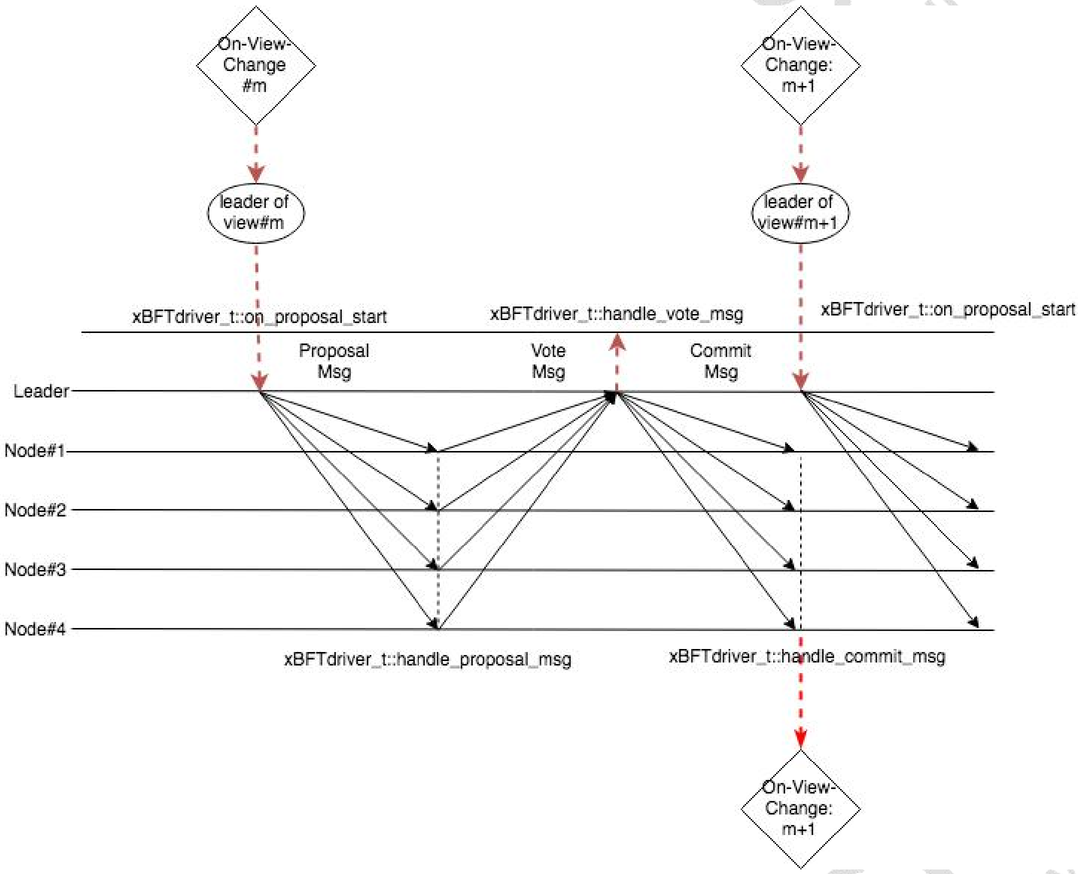


Figure 13 Consensus procedure

After receiving each vote message, the leader checks whether the number of votes has met the 2f+1 requirement (including validator and auditor nodes). If this requirement is met, the leader then broadcasts a commit message, and the round of xBFT ends. Block 1 is in the HighQC state and may be forked or discarded. When the second round of xBFT ends, the generated block 2 is HighQC, and block 1 becomes locked at this time and cannot be forked. After the third round of xBFT, block 1 becomes the commit state, block 2 becomes the locked state, and the newly generated block 3 becomes the HighQC state. The view changes in each round of BFT, and a block is fully confirmed after three rounds of BFT. Additionally, the audit network will also perform checks. Therefore, even if the leader is malicious, invalid transactions will still be blocked.

4.2. Consensus Algorithm Consistency

For a block in the HighQC state, it is not guaranteed that 2f+1 nodes have committed this block, because an abnormality may occur in the last stage of a round of BFT. The system cannot be sure that most nodes committed HighQC. When the block state becomes Locked, it means that 2f+1 nodes voted on the proposal, but in the second phase of commit, it still cannot be sure that a majority of consensus nodes have received the commit message. TOP Chain performs checks at this stage so that the locked block cannot be forked. After the third confirmation, the block status is committed, which is enough to ensure consistency between a majority of nodes. At this point, the system can execute the content in a transaction to modify the status of an account.

```
// 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;
}
```

Figure 14 Fork verification

4.3. Consensus Algorithm Activity

xBFTdriver_t::on_view_fire: after receiving an event with a viewid greater than the sequence number of the proposal, the proposal will be added to the timeout list. If the safe_check_for_block fails, it will be added to the outofdate list.

xBFTdriver_t::on_clock_fire: global clock calls safe_check_for_block to check the proposal, and puts it in the outofdate list when the check fails.

```
bool xBFTRules::safe_check_for_block(base::xvblock_t * _block)
{
    base::xvblock_t * lock_block = get_lock_block();
    if( (NULL == _block) || (NULL == lock_block) )
        return false;

    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;
}
```

Figure 15 safe_check_for_block

notify_proposal_fail processes the timeout_list and outofdate_list. When the proposal in timeout_list is on its leader node, it broadcasts a commit message. If it is not on the leader, it will mark the status of this round of consensus as timeout, and mark the one in outofdate_list as canceled.

The canceled and timeout proposals will not be processed. The consensus round is driven by the clock block, which ensures the continued activity of the consensus algorithm.

5. Signature Security

The signature algorithm used is based on the Schnorr threshold signature algorithm. The functions of single signature, multi-signature merging, and signature verification are provided. There are existing security proofs for Schnorr signatures: when a sufficiently random hash function is used and the elliptic curve discrete logarithm used in the signature is difficult enough, it can be proven that Schnorr is the safer alternative when compared to ECDSA.

At the beginning of a round of consensus, the Leader calls do_sign() to sign the proposal.

```
//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
}
```

Figure 16 Signature

Other consensus nodes verify the signature when processing the proposal message, sign the proposal, and then add voting information.

```
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
        {
```

Figure 17 Signature verification

When the leader processes a vote message, if the vote of the current proposal has reached the 2f+1 quorum requirement, the signatures of other consensus nodes are aggregated, and a call is made to verify_muti_sign() to verify the multi-signature.

```
if(_proposal->is_vote_finish()) //check again
{
    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);
    }
    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);
    }
    if(get_vcertauth()->verify_muti_sign(_proposal->get_block()) == base::enum_vcert_auth_result::enum_successful) //quorum
    certification and  check if majority voted
    {
```

Figure 18 Aggregate signature and verification

Procedure of multi-signature verification: Check whether validators and auditors are from the same group and cluster, then call xschnorrsig_t::verify_muti_sign to verify the signatures of validators and auditors.

```
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;
}
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());
```

Figure 19 Multi-signature verification

When a node executes xproposal_t::add_voted_cert() on the proposal, it is ensured that there are no duplicate voters.

```
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
        }

    }
```

Figure 20 Validator record

6. Smart Contract Security

The system contract includes interfaces for node registration, rewards, TCC committee, and node election. The node registration contract includes functions for node registration, node termination, setting dividend ratios, updating node types, withdrawing node deposits, setting node nicknames, etc.

6.1. Node Registration

When registering and updating a node, the system will check whether the node exists, whether the node type and node nickname is legal, whether the node dividend ratio is within the range of 0 to 100, and whether the node registration requires a source_action of type xaction_type_asset_out. m_amount is used as the deposit for node registration.

```
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());
```

Figure 21 Node registration

Obtain the node role and the corresponding minimum required deposit and check whether the deposit meets the minimum deposit conditions:

```
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");
```

Figure 22 Check node registration mortgage

When a node is terminated, the system will check whether the node is within the penalty period. A node in the penalty period cannot be terminated.

```
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");
}
```

Figure 23 Check node termination

The withdraw interval (72 hours) will be checked when a node attempts to withdraw its stake.

```
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");
```

Figure 24 Check stake withdrawal

6.2. Incentives

20% of the reward pool is allocated for node votes, 76% of which is rewarded based on node workload, and 4% of which is rewards for on-chain governance committees.

The vote rewards are issued to nodes in the active state with number of votes > 0, and deposit > 0. The total voting rewards are dispersed every 12 hours and distributed according to the proportion of the total votes each node has. The formula is:

Node vote reward = number of votes / total number of votes in the entire network * 20 billion * M% * 20% (M% is the proportion of incremental issuance that year)

Workload rewards for different node types are as follows:

- Edge (routing): 2%
- Auditor (audit): 10% (Equally divided between each shard, and rewards are distributed according to the node's audit workload within a shard)
- Validator (verification): 60% (Equally divided between each shard, and rewards are distributed according to the node's verification workload within a shard)
- Archiver: 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;
}
```

Figure 25 Reward calculation

6.3. TCC Committee

The types of legal proposals are as follows:

```
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;
    }
}
```

Figure 26 Proposal type verification

The procedure of submitting a proposal via xrec_proposal_contract::submitProposal is as follows:

1. Check whether the proposal type is legal.

- If the type is proposal_update_parameter, determine whether target exists in onchain_params (the update operation requires this parameter), and compare the updated value with the old value.
- If the type is proposal_update_asset, determine whether target is a legal address.
- If the type is proposal_add_parameter, check whether target exists in onchain_params (the adding operation requires that the parameter does not already exist).

- If the type is proposal_delete_parameter, it is required that target exists in onchain_params.

- If the proposal type is proposal_update_parameter_incremental_add/delete, the value of target can only be "whitelist".

2. Get the sender amount in the proposal transaction, and check whether it is greater than the on-chain parameter min_tcc_proposal_deposit.

3. Get the expiration time tcc_proposal_expire_time.

4. Set the proposal structure information (proposal_id, parameter, new_value, deposit, etc.). end_time is set to the current time plus the expiration time.

5. Remove the expired proposal and return the deposit of the proposal.

When a proposal is withdrawn, in order to clear the proposal and return the deposit, the system will first check whether the caller of the contract is the initiator of the proposal.

The main procedure when voting on a proposal is as follows:

1. Check whether the caller of the contract is a member of the board of directors.

2. Check the status of the proposal. The status cannot be of type failed or success (completed status).

3. If the proposal has expired, get the current voting results of the proposal, and calculate whether the proposal has passed according to the priority of the proposal. The higher the priority is, the higher the pass threshold will be.

```
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;
    }
}
```

Figure 27 Threshold of expired proposals

4. If the proposal has not expired, check whether the current caller has already voted. If not, then vote for this proposal, and calculate whether it has passed according to the following algorithm.

5. Regardless of whether the status of the proposal is success or failed, it will be considered as a

completed proposal and removed from the existing proposal set, after which the deposit will be returned.

6. If the proposal status is success, the modification of the relevant on chain parameters is applied. Finally, any expired proposals are removed.

After auditing, the functions implemented by the system's smart contract have been fully verified, and there is no logical security issue.

7. Shard security

7.1. Sharding Mechanism Security

The compute/staking architecture of TOP chain is shown in the following figure:

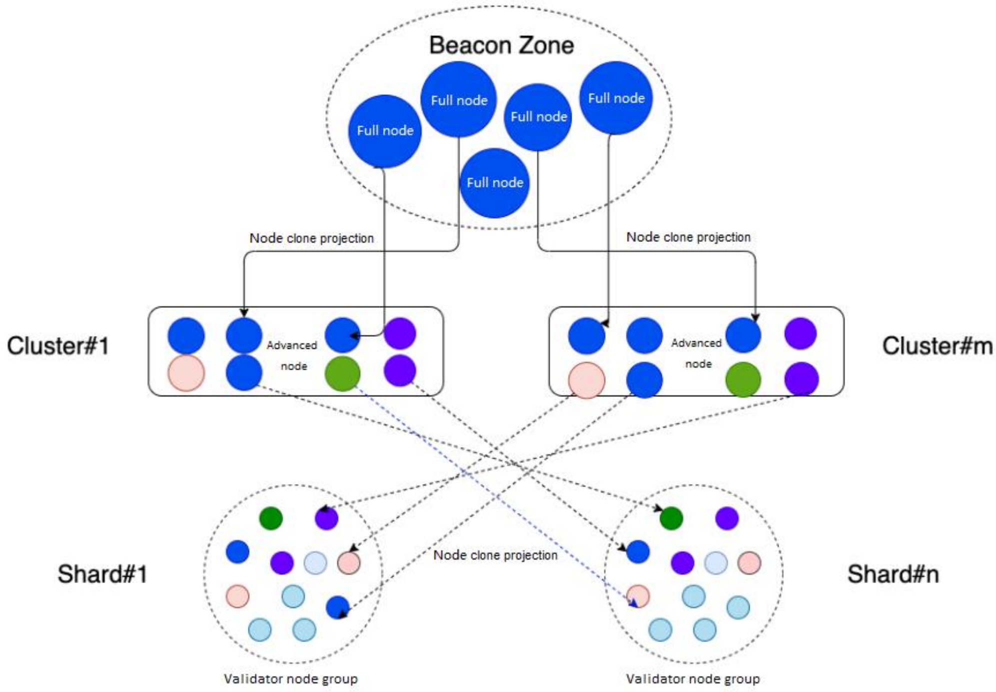


Figure 28 Design of computing power and staking on TOP chain

By combining three layers of staking/compute power, mechanisms such as clone projection, random rotation, and random mapping means that the computing power and staking of the entire network can cover every cluster and shard, ensuring security.

Shards are asynchronous. For example, if shard 1 sends a cross-shard transaction to shard 2, it will not wait for receipt confirmation. Additionally, the failure of one shard will not block other shards. The security of each shard depends on the selection of each shard validator and the consensus mechanism. Sharding on TOP Chain is carried out randomly. A VRF is used to create an unpredictable random seed, and so malicious nodes cannot gain access to a specific shard, which increases the cost of attacks. At regular intervals, some nodes in a shard will be reassigned, and over time, each shard will have completely different nodes than before.

Node's will check the cache regularly and will send uncommitted receive transactions.

```
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));
    }
}
```

Figure 29 Timed inspection

When the receipt status is updated to confirmed, it will be removed from the cache.

```
// 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;
    }
```

Figure 30 Remove expired transactions

The node receiving the receipt will also save a cache, which will be cleared after confirmation or expiration. Therefore, when all nodes of a shard are down, or any of the three stages are not completed due to any reasons such as network failure, the unfinished sendtx or receipt will be completed after returning to normal.

The data on TOP Chain is separated into 3 layers to guarantee the availability of shard data:

1. Beacon (Full node group): Stores and synchronizes all blocks and transaction data of the entire network to ensure the availability of data in the entire network.

2. Cluster (Advanced node group): Stores and synchronizes the block and transaction data of multiple shards under the same cluster to ensure the availability of data in a cluster.

3. Shard (Verification node group): Stores and synchronizes the block and transaction data on the current shard to ensure the availability of data in a shard.

To ensure consistency of sharded data, TOP Chain uses the following architecture:

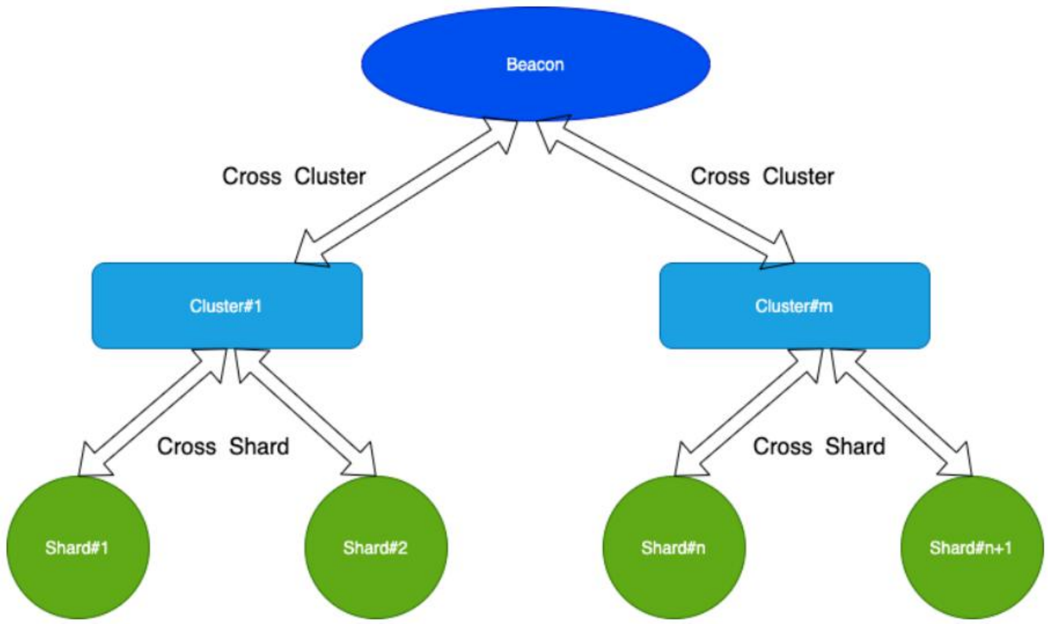


Figure 31 Check data consistency

According to the above figure, it can be seen that the audit and verification of the sharded data in TOP chain also has 3 layers:

1. Beacon node group: Responsible for generating a consistent clock and Drand block across the entire network, generating consistent node rotation and election results across the entire network, Cross-Cluster security, and ensuring global data consistency by auditing the block data of a cluster.

2. Cluster node group: Responsible for Cross-Shard data security, which includes auditing the block data of a shard, and ensuring data consistency within a cluster.

3. Shard node group: Responsible for data consistency checks of all accounts within a shard, as well as transaction execution and state calculation.

With the combination of these 3 layers, Cross-Shard data, Cross-Cluster data, and beacon oversight, the state can remain consistent.

7.2. Shard Transaction Security

The procedure of cross-shard transactions is as follows:

1. Map the Sender address of the transaction to Shard#1. The shard receives and verifies the original transaction, and executes the Sender Action.
2. Shard#1 completes the consensus and execution of Sender Action, generates a block and a certificate, and then broadcasts them to Shard#3 mapped by Receiver.
3. Shard#3 checks the certificate, completes the consensus and execution of Receiver Action, generates a block and a certificate, and then broadcasts them to Sender's Shard#1.
4. Shard#1 checks the certificate, completes the consensus and execution of Confirm Action, completes the entire transaction, and writes the block as the certificate.

At the beginning of a round of consensus, Shard#1 will create a block, including the Unit block and its output, and then verify and execute the Sender Action. When the block is confirmed, Shard#1 analyzes the transaction in the block, constructs a receipt and sends it to 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);
}
```

Figure 32 Processing during block confirmation

Shard#3 then verifies and executes the Receiver Action. Similarly, when the block is confirmed, Shard#3 parses the transaction in the block to generate a receipt for the received transaction (confirmation), which will be sent to Shard#1. Finally, the last consensus round is done, and the Confirm Action is executed. After the block of each stage is confirmed, the receipt can be sent to the next stage.

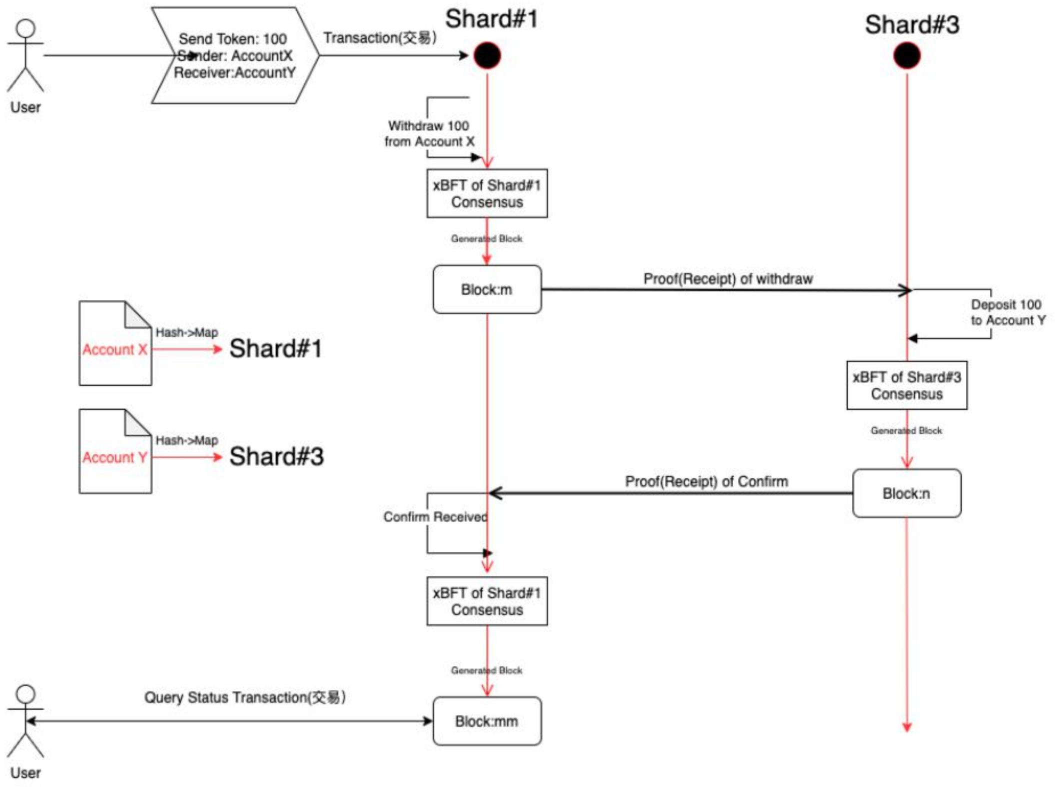


Figure 33 Procedure of shard transaction

When querying the transaction status, it will be queried in the shard where the transaction is initiated. The success and fail status of the transaction are determined by the status in confirm_unit_info. When the height of the unit block on the sending shard is 0, the status is queue, and the status in other cases is pending. When the block commits and calls xstore::set_transaction_hash, it determines the current transaction type. If it is "confirm", the transaction is confirmed, and the height in confirm_unit_info is set.

```
// 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";
        }
}
```

When updating the transaction status, the corresponding block is found through the source_addr and unit height of the transaction (namely the confirm transaction of the entire transaction). Therefore, the shard that initiated the transaction will update the transaction status as success after the entire shard transaction is completed.

8. Summary

Our company conducted multi-dimensional and comprehensive gray-box security audits on the module security and business logic security of the TOP public chain via simulated attacks and code audit. After audit completion, the determination is: **TOP public chain passed all public chain security audit items, and the audit result is Passed (Excellent).**

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

Email

vaas@lianantech.com

WeChat Official Account