Genaro Network

Genaro Network

Roadmap Towards a Multi-Source Data Governance Framework

Yellow Paper v1.0.1

Abstract

The Genaro Network is a new public blockchain platform based on peer-to-peer encryption and sharing. The platform aims to realize highly efficient node management on the public chain based on PoS (Proof of Stake) and SPoR (Sentinel Proof of Retrievability). The vision of Genaro is to establish a new medium of distributed and encrypted storage, and to enable each user to use and share data, and to establish abundant distributed applications (DApps) on the blockchain and provide stable support for these. Compared with other public chains, Genaro has the following advantages: (1) Genaro modified the use of file sentinels to better suit distributed systems through the combination of PoS and SPoR, thus enhancing the ability to defend against replay attacks; (2) in the design of chain-style PoS, Genaro studied famous PoS methods such as Casper (CFFG, CTFG)， Tendermint, and Ouroboros, analyzed the major ways of attacking PoS and proposed relevant schemes; and (3) in terms of management structure, Genaro combines the proof of data integrity and PoS, and provides effective methods of defense against potential problems in PoS. In addition, in terms of the data structure in the public chain, Genaro has developed the GSIOP protocol in line with up-to-date methods of storage encryption, so as to settle different layers of data usage. Finally, in terms of adding data, Genaro has also added relevant VM order sets.

Table of Contents

1 Genaro's Vision

Genaro is a public chain that combines peer-to-peer encrypted storage. The original intention of the design is to allow data to be shared after encryption, giving users different data rights, and ultimately allowing a rich ecosystem of decentralized applications (DApps) to be built on the public chain. As of now, most DApps are centered around transactions in the trading market, gambling, and gaming.

Two data-related features of DApps: (1) there is limited data storage on the chain and (2) off-chain data is not directly usable, making the DApp not as ubiquitous as the current mobile phone or web app, resulting in a narrow functionality for the blockchain.

Genaro is designed to allow data to have a decentralized storage medium and to complement the functionality of its encrypted data, incorporating the idea of encrypted data processing via the blockchain. To achieve this goal, Genaro's system is designed in three parts: the storage network, the public chain, and the consensus governance (Figure 1). This yellow paper will elaborate on these three elements.

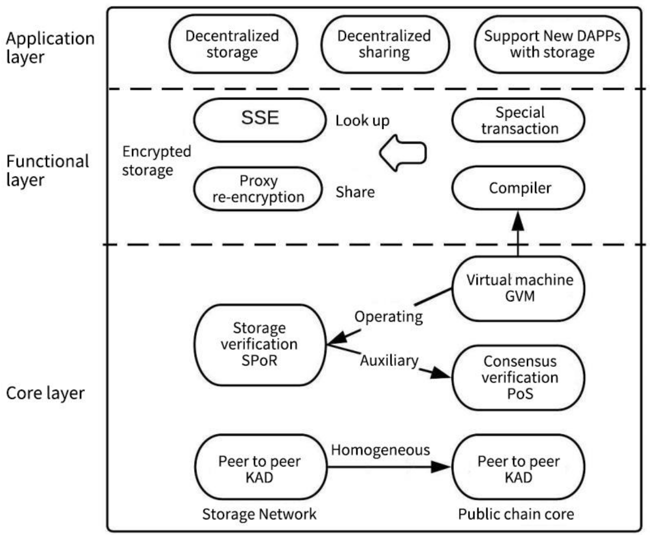

Figure 1. Genaro system architecture

2 Storage Network

2.1 DHT-Distributed Hash Table

P2P [1], that is, peer-to-peer, can be said to be a very concentrated embodiment of the Internet philosophy: common participation, transparent openness, and equal sharing. There are many applications based on P2P technology, including file sharing, instant messaging, collaborative processing, streaming media communication, etc. [2]. Through the contact, analysis and understanding of these applications, the essence of P2P is a new network communication technology. This new communication technology breaks with traditional structures, becoming gradually decentralized and flattened, thus moving towards the future trend of equal nodes all working together.

The application of P2P file sharing (BTs/eMules, etc.) is the most concentrated embodiment of P2P technology. Genaro is a P2P file sharing network as an entry point, around a file network system, and its operability combined with the blockchain formula algorithm to design a new flat, decentralized cloud while retaining the blockchain’s open, transparent characteristics.

The development of the P2P file sharing network generally has the following stages, including the network of the tracker server, the pure DHT network without any server, and the hybrid P2P network.

The Distributed Hash Table (DHT) is a distributed storage method. In DHT, a type of information that can be uniquely identified by a key value is stored on multiple nodes according to a certain convention/protocol, which can effectively avoid the issues of the single failure of a “centralized” server (such as a tracker). As distinct from a centralized node server, each node in the DHT network does not need to store the information of the entire network, but only stores data from its neighboring subsequent node, which greatly reduces the bandwidth occupation and resource consumption. The DHT network also backs up redundant information on the node closest to the keyword, avoiding the single node failure problem.

There are many techniques/algorithms for implementing DHT, such as Chord [3], Pastry [4], Kademlia [5], etc. Genaro uses the Kademlia algorithm because P2P file sharing software such as BT and BT derivatives (Mainline, Btspilits, Btcomet, uTorrent), eMule and eMule Mods (verycd, easy emules, xtreme) are based on this algorithm. Since the protocols implemented by each of these are not the same, there will be incompatibility issues. BT uses Python's Kademlia implementation called Khashmir. eMule uses the C++ Kademlia implementation called Kad. Among the many P2P protocols, Genaro's implementation is based on eMule's Kad, which is the DHT network implementing Kad protocol, for the reason that the point-to-point library and the storage network are isomorphic. In the following introduction, we will gradually explain the advantages of the Kademlia algorithm itself.

Kademlia technology, often referred to as third-generation P2P technology, is a P2P universal protocol for all distributed peer-to-peer computer networks. Kademlia defines the structure of the network and plans communication between nodes and specific information interaction processes. In Kademlia, network nodes use UDP to communicate, and a distributed hash table is used to store data. Each node has its own ID, which is used to identify the node itself and also to help implement Kademlia algorithms and processes.

2.2 KAD Network and Nodes

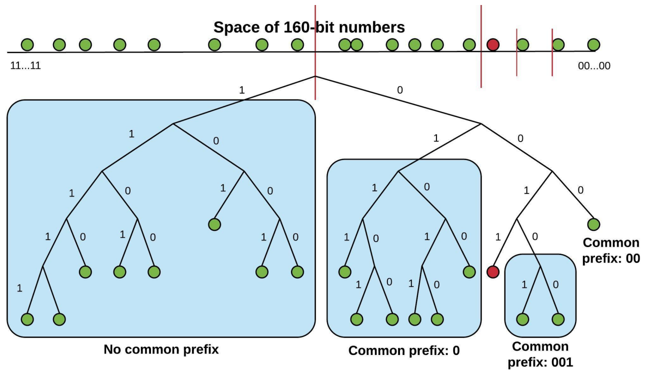

Figure 2. KAD network

The nodes in the KAD DHT storage network include the following features:

- The NodeId needs to be 160 bits or 20 bytes in the KAD;
- Contact contains NodeID (nodeId), Address(string), UDP port number;
- Bucket [VaugeKConst]*Contact is used in the routing of nodes. A bucket can contain k nodes and all nodes will disappear after an hour;
- VaugeKConst is set to 20;
- Router contains Contact and KBucket. KBucket has a bucket in each bit of the ID.

Kademlia uses key values to identify nodes and data on the KAD network. The key value of KAD is not transparent and has a length of 160 bits. Each computer that comes in will have a key, called a NodeID, generated in a 160-bit key-value space. Since the KAD stores content by a KV (key-value) pair, data in the KAD DHT is also independent of the space corresponding to the key in the 160-bit key.

At the beginning, a node is not linked to any other nodes. When a new node is registered, the nearest node will find the link to this node and save the new NodeId. The contacts are selectively removed and then organized inside the bucket when the store overflows. The way to find a NodeId from a node is to find another near node from a known routing table until the requested node is found.

KAD provides several features that other DHTs cannot provide spontaneously, including:

- KAD minimizes the amount of intro information in a node;
- The configuration information includes node information and presence information in the network, and is

automatically propagated through the relationship of side effects in the search for the key;

- The nodes in the KAD are aware of other nodes. This feature allows routing queries through lower latency paths;

- KAD uses both parallel and asynchronous requests to avoid timeout delays for failed nodes;

- KAD is resistant to some DOS attacks.

Another reason for Genaro's choice of KAD is that Genaro's own peer-to-peer system uses KAD. It can be processed by the same system, making it relatively easy to implement later when managing accounts.

2.3 XOR Matrix and Distance Calculation

Each KAD node has a 160-bit NodeId and the key value of each data is also a 160-bit identifier. To determine which node a KV pair exists in, KAD uses the concept of the distance between two identifiers. Given two 160-bit identifiers, x and y, KAD determines the distance between them by their XOR and expresses them as an integer. XOR gets the definition of distance in the system binary tree frame. In a full 160-bit ID binary tree ID, the size of the two ID distances is the smallest subtree containing two nodes. The leaf closest to IDx is the leaf that shares the longest common prefix with x when the tree is not a fully binary tree. For example, the distance between 0011 and 1001 is 1010 and the integer expression is 10, so the distance between the two nodes is 10.

2.4 Node Status (contact)

Using NND (node network data) to represent the information in the KAD node contains:

- IP Address

- UDP port

- Node Id

In the KAD, the nodes store contact information for routing information. At each $0 < i < 60$, each node will keep a list of NNDs at the distance between itself and the node. This list is called k-buckets.

Therefore, the node knows some of the nodes in its subtree. Each k-bucket is stored in chronological order: the node that was received last but not recently seen is in the head, and the node that was most recently seen is at the end. The node updates the appropriate node's NodeId with the appropriate k-bucket when the KAD node receives any information (request or reply) from other nodes. There are three cases when updating k-buckets:

- If the sending node already exists in the recipient's k-bucket, the receiver places it at the very end of the list.

- If the node is not in the appropriate k-bucket and the bucket is made up of fewer key-value pairs than k, then the receiver will directly add the new sender to the end of the list.

- If the appropriate k-bucket is full, the receiver decides what to do by pinging the node that was seen before the k-bucket. If the last recently seen node receives a failed reply, it will be stripped from the k-bucket and the new sender will be placed at the end. Conversely, if the node that was seen recently gets a reply, it is placed at the very end of the list and the new send contact is discarded.

KAD's k-bucket has two benefits:

First, k-bucket implements the the most recently observed node eviction policy, except that the retained node is never removed from the list. Analysis of the P2P system indicates that the more a node is on top, the more likely it is to be on the next hour. The k-bucket maximizes the possibility of staying online by preserving the old surviving contact.

Second, the k-bucket provides resistance to certain DoS attacks. A malicious user cannot swipe the routing state of each node by using a large number of new nodes. The KAD node will only join the new node when the old node leaves them. Each KAD node also has a routing table. The routing table is a binary tree, and the leaf nodes are k-buckets. The node's routing table contains all the k-buckets of the node, that is, the neighbors of the current node in different subtrees.

2.5 Kademlia Protocol [15]

The Kademlia protocol contains four RPC (remote proce -dure calls): PING, STORE, FIND_NODE, and FIND_VALUE, where:

- PING RPC detects if the node is online;
- STORE RPC notifies the node to store a key-value pair [key, value] to ensure subsequent retrieval;
- FIND_NODE RPC contains 160 bits of the key to an arg, and the receiver of FIND_NODE RPC returns an NND (contact) information about the nearest k node of the target id;
- FIND_VALUE RPC functions like FIND_NODE and returns the nearest k-node corresponding to the target id. There is one exception: if the RPC receiver receives a STORE for a given key, this will return the stored value.

2.6 Lookup Algorithm [15]

The node lookup process in KAD is to locate the k nearest nodes by KAD according to the given key value. KAD chooses to use a recursive algorithm in node lookup. The party that initiated the lookup first finds a node from the non-empty k-bucket (or, if the bucket has fewer key-value pairs than α , then it can only get a nearest node by key). The initiator sends the FIND_NODE RPC to the selected nodes by parallel asynchronous means. α is a concurrency parameter of a system. In the recursive phase, the initiator resends the find node to the node that previously sent the RPC. Nodes that cannot respond quickly will be removed unless they are replied to. If a round of looking for a node does not find any closer than the nearest observed node, the initiator will resend the find node to find the k most recent ones that have not been requested. When the initiator receives a reply from the k most recently observed nodes, the search process ends. Each node knows at least one node in each of its subtrees, and each node can be located to other nodes through the NodeID. To store a KV pair, the node needs to locate the k nearest nodes by key value and then send the STORE RPC. To find a KV pair, the node needs to find the k nodes with the closest key value. However, the value lookup uses FIND_VALUE instead of FIND_NODE and this process stops as soon as any node returns a value.

2.7 Cache

In terms of caching, once the lookup is successful, the node that sent the request stores a key-value KV pair that it observed for the nearest node that did not return a value. Because of the unidirectional nature of the topology, it is possible to encounter an array of cached key-value pairs before finding the nearest node in the future when looking for the same key value. With the high frequency of a certain key value, the system may eventually be cached in many nodes. To prevent excessive caching, the validity period of a KV pair in any node database is set to be inversely exponential with the number of nodes, which is the number between the current node and the key and the node with the most recent key ID.

2.8 Bucket refresh

The KAD bucket is generally refreshed through the request transfer between nodes. On the issue of resolving a table for a particular ID segment, each node refreshes the bucket that was not requested by any node during the last hour. Refreshing a bucket is done by first randomly selecting an ID to the bucket segment and then using node lookup on that ID.

In order to join the network, node u must have a contact to the existing node w, which is usually the bootstrap node on each network. u inserts w in the appropriate k-bucket. u will then look up via its own node ID. In the end, u refreshes all the k-buckets, farther than they are now. In the refresh, u fills its own k-bucket and also adds itself to the k-bucket of other nodes. An existing node, by similarly using the complete information of its surrounding subtree, knows that the new node will hold that KV pair.

Any node needs to know the new node, so the STORE RPC is sent to transmit the relevant KV pair to the new node. In order to avoid reusing STORE RPC to save the same content to different nodes, the node will only transmit KV pairs when its own ID is closer than the other nodes to the key value.

2.9 Key Value Redistribution

To ensure the persistence of KV pairs, nodes must periodically redistribute key values. Otherwise, in two cases it may cause the search for a limited key value to fail:

First, some k-nodes get KV pairs first, and they leave the network when the KV pair is distributed.

Second, the new node may join the network with an ID that is closer than the node that is now distributing the key.

In both cases, the node with the KV pair must be redistributed to ensure that it is always available to the k most recent key-valued nodes. To compensate for the node leaving the network, the KAD redistributes the KV key once per hour. KAD proposes two mechanisms for optimizing key-value redistribution while ensuring efficient use of computing resources:

First, when a node receives a STORE RPC for a given KV pair, it assumes that the RPC is also sent to the other k-1 nearest nodes, and then the receiver will not redistribute the KV pair within the next hour. This guarantees that as long as the time period of the redistribution is not fully synchronized, each node will be redistributed every hour for a given KV pair.

Second, optimization avoids node lookups before redistribu -ting key values. In KAD, a node has full information about the surrounding subtrees of at least k nodes. If node u flushes all k-buckets in the k-node subtree before redistributing the KV pair, it will automatically find the k nearest nodes with the given key value.

3 Genaro Public Chain

3.1 Searchable Encryption

Before we talk about the entire public chain, we need to introduce a technique used in the public chain, namely searchable encryption, and its role in the encrypted storage of the public chain. First, searchable encryption, as its name implies, is a scheme for searching and querying based on ciphertext, using cryptography to protect the user's data privacy. Second, searchable encryption has four advantages: provable security, control search, hidden query, and query independence. In decentralized storage, searchable encryption can provide privacy for personal data. Third, searchable encryption can be very simple and fast. It does not require a lot of pre-interaction and can achieve higher real-time operations.

In data protection, the privacy of personal information needs to be prioritized. The second thing to support is the change of dynamic data, that is, the way DApps modify decentraliz -ed data storage. Searchable encryption technology is an indispensable part of Genaro's I/O Streaming Protocol (GSIOP).

In the common scenarios of searchable encryption, searchable encryption is generally divided into the following four stages:

(1) Data encryption phase: The data owner encrypts the plaintext data locally using the key and then uploads the encrypted data to the data storage hosting unit.

(2) Generating a search trapdoor stage: The user generates a corresponding search trapdoor using the key and the keyword and sends the trapdoor to the data storage

hosting unit, wherein the search trapdoor can secretly contain the keyword content contained therein.

(3) Ciphertext retrieval stage: According to the received keyword retrieval trapdoor, the data storage escrow unit retrieves the ciphertext and sends the ciphertext satisfying the retrieval condition to the user. The data storage hosting server cannot obtain more information than the search results during execution.

(4) Ciphertext decryption phase: After the user obtains the returned ciphertext from the data storage escrow unit, the key is used to decrypt the relevant data.

Depending on the type of key, searchable encryption can be further classified into Searchable Symmetric Encryption (SSE) and Public-key Encryption with Keyword Search (PEKS). Genaro's current scenario is to provide the data owner with a functional service for encrypted data search, so the solution is based on SSE. Here we are not discussing PEKS for the time being. The SSE scheme consists of five algorithms:

(1) $\mathrm{K} = \mathrm{KeyGen}(\mathrm{k})$: Input the security parameter k and output the randomly generated key K. This operation is usually performed locally on the data owner side.

(2) $(\mathrm{I}, \mathrm{C}) = \mathrm{Enc}(\mathrm{K}, \mathrm{D})$: Input the key K and the plaintext file set, and output the index and ciphertext file set. This operation is performed locally on the data owner side.

(3) $\mathrm{Tw} = \mathrm{Trapdoor}(\mathrm{K},\mathrm{W})$: Input the key K and the keyword W to output the trapdoor corresponding to the keyword. This operation is performed locally on the data owner side.

(4) $\mathrm{D}(\mathrm{W}) = \mathrm{Search}(\mathrm{I},\mathrm{Tw})$: Input index I and the trapdoor Tw of the keyword to be searched, and output a set of identifiers of the file containing the keyword W. The Search operation is performed by the key distribution control in Genaro.

(5) $\mathrm{Di} = \mathrm{Dec}(\mathrm{K}, \mathrm{Ci})$: Input the key K and the ciphertext file Ci, and output the corresponding plaintext file Di after decryption. This operation is performed locally on the data owner side.

3.2 Genaro I/O Streaming Protocol (GSIOP)

In the data storage of the blockchain, the size limitation of the storage on the chain and the inability of the under-chain data to be self-certified have always been the bottleneck of DApp design. Genaro itself has an entry point outside the chain to the chain and the storage part is responsible for this part. Therefore, the data holder can store and access the out-of-chain data by keeeping it under the chain through the Genaro stream protocol, initializing access on the Genaro chain. Compared with the existing oracle mode and the original direct storage on the chain, the ownership and privacy of the data can be guaranteed with more storage space. With such an indication, the Genaro Streaming Protocol is designed to achieve multi-faceted file changes while ensuring privacy through encrypted storage. This is also what Genaro has always aimed for, as a Dapp’ s stored information needs to be private. GSIOP customizes the part of the encrypted storage to a widely usable protocol by using the features of Genaro storage, so that it can encrypt the data stream and obtain the special function of the relevant encryption algorithm. In the actual use of the blockchain, the size of the data on the chain and the synchronization of the data under the chain are difficult to guarantee. In other words, if the data remains on the chain, the size of the data must be considered. The amount of data that each block can retain is limited. Increasing the size of the data affects the size of the block.

First of all, whether the data under the chain can guarantee the certainty of access is a key issue. Due to the non-uniqueness of the identity on the chain, it is difficult to judge whether the chain operation after the chain identity is operated by the credible part. Secondly, The data under the chain can be processed multiple times by different nodes when uploading. Because the data is public, it is difficult to sort out which parts are available.

For example, a malicious node can change public data through A, B, and C accounts without any penalty. Even if the A, B, and C accounts are prohibited from being changed at the same time, a new account, D, will be maliciously operated, so the users of the chain data before being uploaded need to be differentiated.

The current solutions include the Lightning Network and a Raiden network ,which are called chain-based solutions. When the data link is down to the uplink, it is verified by means of self-verification and then verification of the nodes on the chain. The chain is provided in a centralized manner whether it is off-chain or on-chain. We prefer to pass the data under the chain through a protocol, and then synchronize this part of the data in the same way. To put it simply, it is the channel that gives the next data in the chain. The user only needs to initialize the data on the Genaro, and then the reward and punishment is solved through the chain management.

GSIOP is a protocol scheme that guarantees the reliability of storage under the chain through encryption. First, the encryption algorithm will be used to selectively change the storage unit when the data is initialized. In the storage account, it is divided into: Data Owner, Data Modifier and Data Monitor.

This corresponds to three users who have different roles in the data. The data owner has a fully modified key and a fully modified operation. This part of the data is maintained by the owner in the system, so the data can be related to a specific storage address and the data has privacy. The data modifier refers to when the data owner chooses to change the relevant part of the data. The data can be segmented, so that the address of the data modifiers corresponds to the data change action and so that the person who has changed the data and made a problem can be found. The data monitor has the right to view the data without changing the data. A function of the entire system is to exclude the corrupted data related to bad address uploads. In the early days, there will be a large number of data owners, who can decide the modifiers and monitors. After the system iteration, some data owners who provide stable data will have specific accounts, as well as modifiers and monitors who can be trusted.

In the first version, GSIOP first changed the value of UINT type and for such a function, the first type of application, that is, an encrypted side chain system, can be derived. The data owner in this system is the user who holds the GNX. He can use the GNX in the chain by locking and the GNX is the Coin on the Genaro chain. Information about the use of GNX for another chain can be accepted and encrypted in the storage network by means of a user who owns account information for another account other than GNX as a partial data changer. Finally, the entire node of the other chain acts as a monitor. It can see that there is indeed information coming in this way and stores the intermediate steps. At the end of the user's recovery of the remaining GNX, you only need to call all relevant storage information, in this way you can guarantee a simple cross-chain operation.

In later versions, GSIOP will provide a call change operation for the string and the related function extension application will be written later. Due to the addition of related storage features, Genaro has made corresponding changes in the smart contracts and virtual machine parts.
Genaro will add opcodes to existing stacking virtual machines in a way that is compatible with existing EVMs, as well as adding related types and instructions to the grammar. Finally, through special transactions to change the state in the chain database, the following is the specific implementation process.

3.3 File Sharing Based on Proxy Re-Encryption

One of Genaro's goals is to allow everyone who uses Genaro to share in an encrypted environment.

There are two main types of sharing in this encryption environment: one is to use someone else's public key to encrypt directly. The other is via proxy re-encryption schemes. Both types of encryption algorithm can achieve our characteristics of data encryption and sharing. Relatively speaking, proxy re-encryption is a more scalable approach that Genaro is currently preparing to adopt in addition to public key encryption. In addition, Genaro is also considering and evaluating the use of attribute-based encryption in the future to achieve encrypted data for multi-user group sharing with different attributes [8], to achieve a more granular sharing effect.

A proxy re-encryption scheme is an asymmetric encryption scheme that permits a proxy to transform ciphertexts under user A ’ s public key into ciphertexts under user B ’ s public key. In order to do this, the proxy is given a re-encryption key $r_{A\to B}$, which makes this process possible.

The notion of proxy re-encryption was introduced in 1998 by Blaze et al. [9]. Their proposal, which is usually referred to as BBS scheme, is bidirectional (it is trivial to obtain $r_{B\to A}$ from $r_{A\to B}$) and multihop (the re-encryption process is transitive), but not resistant to collusions.

Ateniese， Fu， Green and Hohenberger proposed in [10], new proxy re-encryption schemes based on bilinear pairings. In particular, they provided an initial basic scheme (AFGH scheme), which is subsequently extended throughout the paper to support additional functionalities. As the AFGH scheme is unidirectional, unihop and collusion-resistant, it was adopted by Genaro.

3.3.1 AFGH scheme

This scheme is based on the ElGamal cryptosystem. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two groups pf prime order q, with a bilinear map $e:\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_2$; the global parameters are $g \in \mathbb{G}_1$ and $Z = e(g,g) \in \mathbb{G}_2$。

- Key Generation (KG): User A selects a random integer$a \in \mathbb{Z}_q$, and generates her pair of secret and public keys: $s_A = a$ and $p_A = g^a$。
- Re-Encryption Key Generation (RKG): User A takes user B ’ s public key, and together with its secret key, it computes the re-encryption key $r_{A\to B} = (P_B)^{1/S_A} = g^{b/a} \in \mathbb{G}_1$。
- First-level Encryption: Anyone is able to encrypt messages intended only for Alice using her public key p_A. To encrypt a message $m \in \mathbb{G}_2$, one selects a random integer $k \in \mathbb{Z}_q$, and computes $c = \left(e\left(p_A,g^k\right),mZ^k\right) = (Z^{ak},mZ^k) \in \mathbb{G}_1 \times \mathbb{G}_2$。
- Second-level Encryption: To create second-level ciphertexts, which are re-encryptable, one computes $c = \left(p_A^k,mZ^k\right) = (g^{ak},mZ^k) \in \mathbb{G}_1 \times \mathbb{G}_2$。
- Re-Encryption: Anyone in possession of the reencryption key $r_{A\to B}$can transform a second-level ciphertexts for user A into a first-level ciphertext for user B, by computing: $c_B = \left(e\left(\alpha, r_{A\to B}\right), \beta\right) = (Z^{bk},mZ^k) \in \mathbb{G}_1 \times \mathbb{G}_2$。
- First-level Decryption: User A uses her secret key s_A to transform a ciphertext $c_A = (\alpha, \beta) \in \mathbb{G}_1 \times \mathbb{G}_2$ into the original message$m = \frac{\beta}{\alpha^{\frac{1}{s_A}}} = \frac{\beta}{\alpha^{1/a}}$。
- Second-level Decryption: For decrypting a ciphertext $c_A = (\alpha, \beta) \in \mathbb{G}_1 \times \mathbb{G}_2$, user A computes: $m = \frac{\beta}{e(\alpha,g)^{\frac{1}{s_A}}} = \frac{\beta}{e(\alpha,g)^{1/a}}$。

This scheme uses two different ciphertext spaces; first-level ciphertexts are intended for non-delegatable messages, whereas second-level ciphertexts can be transformed into first-level ones through re-encryption. The AFGH scheme has the following properties:

- Unidirectional: The re-encryption key $r_{A\to B}$ cannot be used to derive the reverse one $r_{B\to A}$. This property is useful in settings where the trust relationship between user A and user B is not symmetrical.

- Resistant to collusions: If the proxy and user B collude, they are not able to extract the secret key of user A; at most, they can compute the weak secret $g^{1/a}$, but this information does not represent any gain to them.

- Unihop: This scheme is not transitive with respect to re-encryption; the re-encryption process transforms a ciphertext from one space to another, so this process cannot be repeated.

3.4 New VM opcodes

In the choice of opcodes, Genaro uses some new instructions based on the original Ethereum VM, which are storage-dependent. For each instruction, we still use α to represent the number of extra parts added to the stack and δ to represent the part subtracted from the stack.

Table 1. Genaro's opcodes

Value	Mnemonic	δ	α	Description
0x5c	STORAGE_GAS	2	1	Get the storage space duration $\mu_s'[0] \equiv \begin{cases} \sigma[\mu[0],\mu[1]]_{sg} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]_{sg} \\ 0 otherwise \end{cases}$
0x46	STORAGE_GAS_USED	2	1	Get the storage slice size $\mu_s'[0] \equiv \begin{cases} \sigma[\mu[0],\mu[1]]_{gu} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]_{gu} \\ 0 otherwise \end{cases}$
0x3f	STORAGE_GAS_PRICE	2	1	Get the number of stored backups $\mu_s'[0] \equiv \begin{cases} \sigma[\mu[0],\mu[1]]_{gp} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]_{gp} \\ 0 otherwise \end{cases}$
0x5d	SSIZE	2	1	Get storage size $\mu_s'[0] \equiv \begin{cases} \sigma[\mu[0],\mu[1],]_{ss} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]_{ss} \\ 0 otherwise \end{cases}$
0x47	SENTINEL_HEFT	1	1	Get the user's heft attribute $\mu_{sh}'[0] \equiv \sigma[\mu[0]]_{sh}$
0x21	DATAVERSIONREAD	3	17	Obtain the version status of the sidechain dataVersion according to dataVersion, file, and address.
0x22	DATAVERSIONUPDATE	2	0	According to fileID, address setting to enable KDC synchronization sidechain status to the public chain $\mu_s'[0] \equiv \begin{cases} 1 if \sigma[u[0] mod 2^{160},\mu[1],\mu[2],]_{du} \neq \emptyset \\ 0 otherwise \end{cases}$

3.5 New Instructions

After adding new opcodes, Genaro added new instructions to the original solidity language and kept compatibility while increasing the operational possibilities of the data.

Value type - increase part:

Storage address d_storage: Saves a 32-byte value corresponding to the space address in the store. The storage address type also has members as the basis for all storage parts.

Storage address member:

- read and update

Call method:

```
d_storage ds =
0x12345678901234567890123456789012;
//Read data for a specific version of a specific space
uint256 result = ds.read.version(0x03);
uint256 newresult = result + 3;
assert(newresult < expectation);
return;
//Enable data timing update
assert(ds.update);
return;
```

Store the address to be queried, because in the smart contract, you can stop the contract execution by using it as a judgment point:

```
d_storage ds =
0x12345678901234567890123456789012;
uint32 bucketID =
0x12345678901234567890123456789012;
mapping(d_storage => uint32 []) public file;
file[ds].push(bucketID);
//Check if it is greater than 3 months
sgas(ds,bucketID)< 3 month;
//Check if the number of backups is more than 6
require (ds.slice >6);assert(ds.update);
//Check if the fragment size is more than 1GB
require(ds.sused > 1 GB);
// Check if the space is 3GB
require(ds.ssize == 3 GB);
// Check if the total number of file fingerprints owned by
an address is more than 10000
Check_sentinel (user_address)> 10000;
```

3.6 New Special Transaction

The special transaction is added to accommodate the bet of the public chain and the storage verification attribute. The following special transactions are added in the system:

(1) Bet transactions (user-initiated transactions). Its role is to support the action of betting in PoS. The user uses the balance in his account to place a bet. After the bet, the amount of the bet will be added into the user's stake attribute.

(2) Storage weight synchronization transaction (storage network initiated transaction). Its role is to synchronize the stored results in the public chain as an aid. The special account specified by the storage network initiates a synchronous transaction of the heft value, synchronizing the heft to the chain.

(3) The transaction in which the user purchases the storage space. Its main role is to support users to directly pay the storage capacity corresponding price on the chain. In the transaction, the purchase amount is determined according to the space parameter provided by the user, and the successful deduction of the amount indicates that the transaction is successful. During the transaction process, some records of expired historical storage space can be cleaned up. After purchasing the storage space, some traffic can be bundled according to the purchase quota.

(4) The transaction in which the user purchases the traffic. Traffic in the store represents upload and download traffic. The traffic purchased by the user will be added to the total traffic, for example, 1G space is purchased and 10G traffic can be downloaded 10 times.

4 Consensus Governance

Before introducing Genaro's consensus, first we wish to establish some common-sense points about consensus mechanisms. The consensus is basically there to organize the distributed processes in a way that will determine a specific value, that is, the consensus permits participants to vote for a specific value. Given a series of processors, each with an initial value:

- All non-error handling eventually reaches a value;
- All processing decisions do things at this value;
- The value determined needs to be raised by some processors.

These three characteristics are called termination, agreement, and validity. Any algorithm with these three characteristics can solve the problem of consensus. [16]

Termination and agreement are relatively easy to explain. We just need to avoid operations from the wrong nodes. For validity, we need to remove the nodes that give the wrong choice regardless of the initial value. This algorithm certainly satisfies the requirements of termination and agreement, but it is useless.

4.1 FLP Impossible Features

The FLP (Fischer, Lynch and Patterson) impossibility [11] actually involves a slightly weaker form of consensus: for termination, only some non-erroneous process decisions are sufficient. It's important to note that having a powerful process to determine value is not enough, because the process may fail and the other will have to take it where it is, or even a weak termination to meet the requirements.

The conclusion of FLP's impossible feature is that error detection in non-synchronous situations cannot detect the problem of the processor without borders and then return a specific value, that is, it is impossible to judge whether the processor is down or not replying for a long time. It has been proven in research that no asynchronous algorithm can guarantee certainty when protecting downtime errors and it can not be guaranteed without downtime.

FLP's impossible feature proves that the asynchronous algorithm without any validation “0-1” can guarantee that the process will be terminated as soon as the downtime occurs and that it is still correct when no similar error occurs. Because in the case of asynchronous, the asynchronous distributed system will converge the state in the system to a “two-valued state”. In this state, any 0 input or 1 input will destroy the balance, but in the end will go back to the dual value state. In this way, the system is proven to be infinite and non-terminal, so the system will go into an infinite loop. The end of the loop is completely out of this state, which means that you need to add a state that can be guided. Fault-Tolerant consensus can be done by vote, but we still have a hard time proving that it can end, so if the need is between “0-1” the result is a guarantee of 1 possibility. We can prove it through a process, but if we need a logically perfect guarantee, we can't reach it. In the design of the Genaro Consensus, considering that there is no such thing as the ability to break the two-valued state without adding additional information, we need to add additional information when designing the forked approach in the chain consensus. What is added here is storage related information for Genaro.

4.2 CAP Theorem

Genaro must follow the CAP theorem in the design of blockchain consensus, which is a specific distributed computer data system. The CAP theorem is also known as Brewer's theorem. The theorem proves three parts that cannot be guaranteed simultaneously for a distributed computer system: Consistency, Availability, and Partition tolerance [12].

This consistency is not primarily used for ACID database consistency. The specific performance feature is that each read has the latest written result or an error. Availability means that every time you pass a request in the system, you get a reply, not an error, but you don't need to ask for this to be the latest one. The partition tolerance is that when the information between some nodes in the network is lost, the system still works. When designing consensus, the CAP theorem needs to be considered in advance, that is, in the case of consistency, the finality of the block can be executed synchronously. And in the case of availability, only one asynchronous execution termination system can be implemented. In this system, the FLP impossible feature needs to be emphasized. That is, in the CAP, after the fault tolerance is guaranteed, choose synchronization (class BFT consensus) or asynchronous (Nakamoto chain consensus).

4.2.1 System Fault Tolerance

As the only part that can't be guaranteed in the CAP at the same time, we need to discuss it separately. In the design of Genaro, we must give priority to the availability of the chain, that is, to achieve the multi-node high concurrency standard of the public chain. In this case, we need to abandon the consistency feature and to preserve the system fault tolerance. The general way to reduce fault tolerance is through special data or state checking, that is, through a special broadcast that is a special transaction; and another timeout or a specific protocol to do fault tolerance.

In a distributed storage system, when a system failure causes information to be lost or duplicated and there is no malicious node (ie, no error information), the consensus problem is called CFT (Crash Fault Tolerance). When there is a malicious node, it is called BFT (Byzantine Fault Tolerance). In order to solve the problem of Byzantine Fault Tolerance common in distributed systems, Genaro draws on the industry's most classical PBFT (Practical Byzantine Fault Tolerance) algorithm, which was proposed by Miguel Castro and Barbara Liskov in 1999 [13]. The core of this consensus is to design an asynchronous approach to consistency and Byzantine fault tolerance. Then we need to understand and graft PBFT on the compatibility issue of Byzantine fault tolerance in the design. In the previous clarification of the CAP theorem, it has been explained that the chain consensus blockchain is also an AP-type distributed system, that is to say, the consistency is different and the Byzantine fault-tolerant part in the asynchronous case is similar. The following is a brief description of PBFT.

PBFT assumes an asynchronous distributed system model and node failures occur independently. PBFT uses encryption to prevent spoofing attacks and replay attacks and to detect corrupted information. The information includes the public key signature, the message validation code, and the message digest generated by the collision-free hash function. All nodes know the public key of the other node for signature verification.

The PBFT algorithm is a form of state machine replication: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. The replicas move through a succession of configurations called ‘Views’. In a view one replica is the primary and the others are backups. We denote the set of replicas by $\mathbb{R}$, f is the maximum number of replicas that may be faulty and $\mathfrak{R} = 3f + 1$. View changes are carried out when it appears that the primary has failed. PBFT algorithm works roughly as follows:

(1) A client sends a request to invoke a service operation to the primary, and timestamp is used to ensure exactly once semantics for the execution of client requests;

(2) The primary atomically multicasts the request to all the backups. When the primary receives a client request, , it starts a three-phase protocol to atomically multicast the request to the replicas: pre-prepare, prepare, and commit. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views.

(3) A replica sends the reply to the request directly to the client. The reply has the form $< \mathrm{reply}, v, t, c, i, r >$, where v is the current view number, t is the timestamp of the corresponding request, i is the replica number, and is r the result of executing the requested operation, allowing the client to track the view and hence the current primary. A client sends a request to what it believes is the current primary using a point-to-point message.

(4) The client waits for $f + 1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r. This ensures that the result is valid, since at most f replicas can be faulty. If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

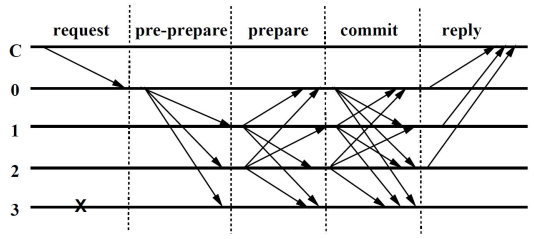

Figure 3. PBFT algorithm process

The resiliency of PBFT algorithm is optimal: $3f + 1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty.

Safety means that the replicated service satisfies linearizability: it behaves like a centralized implementation that executes operations atomically one at a time. Safety requires the bound on the number of faulty replicas because a faulty replica can behave arbitrarily, e.g., it can destroy its state. Safety is provided regardless of how many faulty clients are using the service (even if they collude with faulty replicas): all operations performed by faulty clients are observed in a consistent way by non-faulty clients. The safety property is insufficient to guard against faulty clients, e.g., in a file system a faulty client can write garbage data to some shared file. However, PBFT limit the amount of damage a faulty client can do by providing access control: it authenticates clients and deny access if the client issuing a request does not have the right to invoke the operation. Also, services may provide operations to change the access permissions for a client. Since the algorithm ensures that the effects of access revocation operations are observed consistently by all clients, this provides a powerful mechanism to recover from attacks by faulty clients.

PBFT algorithm does not rely on synchrony to provide safety. Therefore, it must rely on synchrony to provide liveness; otherwise it could be used to implement consensus in an asynchronous system, which is not possible. We guarantee liveness, i.e., clients eventually receive replies to their requests, provided at most f replicas are faulty and delay(t) does not grow faster than t indefinitely, i.e. we add control over the time in design.

Since PBFT itself has a limit on the total number of nodes in the public communication, the idea of asynchronous BFT is time control and node control, which provides a design idea in the design of asynchronous BFT compatibility of Genaro. That is to say, a related measurement and control of the number of node synchronizations and the time sampling is required.

4.3 Sentinel Proof of Retrievability (SPoR)

Genaro's ultimate goal is to have a public chain that can be encrypted and shared. So first we need to ensure that the encrypted shared files themselves can be proven to have storage integrity and can be retrieved. We need a reliable and efficient way to verify the integrity of the file storage and the relevant proof that it can be fully retrieved. Genaro chose a representative SPoR algorithm [6], which provides a complete, provable, and secure theoretical system for file storage integrity verification. We can use this set of storage integrity verification algorithms to provide important auxiliary information for the Genaro Chain style consensus algorithm to be introduced later, to achieve organic integration and complementary advantages. Here we focus on Sentinel, which means that this part of the information is our focus in the whole process. Compared to other consensus mechanisms for replacing PoW content, such a design will have lower consumption, because the essence is a consensus to verify the hash combination. This part of the auxiliary information is then given the corresponding heft in the chain governance, which means that our governance is not a simple stake and offline governance. We use the storage integrity verification algorithm to audit the contribution of storage nodes on the chain and combine the online governance of stakes to gradually optimize and maintain the stability of the system. The advantage of this approach is that it does not interfere with the stability of the system due to some uncertainties under the chain. SPoR (Sentinel Proof of Retrievability) is an algorithm of traditional PoR that detects data verifiability by setting up a specific file fingerprint (sentinel). A file fingerprint is a block of random values and is indistinguishable from encrypted data. The SPoR protocol structure includes the following three parts:

- Setup phase: The verifier V encrypts the file F. It then embeds sentinels in random positions in file F, sentinels being randomly constructed check values. Let $\tilde{F}$ denote the file F with its embedded sentinels.

- Verification phase: To ensure that the archive has retained F, V specifies the positions of some sentinels in $\tilde{F}$ and asks the archive to return the corresponding sentinel values.

- Security: Because F is encrypted and the sentinels are randomly valued, the archive cannot feasibly distinguish a priori between sentinels and portions of the original file. Thus we achieve the following property: If the archive deletes or modifies a substantial of $\tilde{F}$, it will with high probability also change roughly a substantial of sentinels. Provided that the verifier V requests and verifies enough sentinels， it can detect whether the archive has erased or altered a substantial fraction of $\tilde{F}$.

Suppose that the file comprises b blocks: F[1],⋯,F[b]. The function entailes four steps:

(1) Error correction: We carve our file F into k-block "chunks." To each chunk we apply an (n,k,d)-error correcting code C， which expands each chunk into n blocks and therefore yields a file $F' = F'[1],\cdots,F'[b]$, with $b'=bn/k$ blocks.

(2) Encryption: We apply a symmetric-key cipher E to F', yielding file F''. Our protocols require the ability to decrypt data blocks in isolation, as our aim is to recover F even when the archive deletes or corrupts blocks. Thus we require that the cipher E operate independently on plaintext blocks.

(3) Sentinel creation: Let $f:\{0,1\}^j \times \{0,1\}^* \rightarrow \{0,1\}^l$ be a suitable one-way function, we compute a set of s sentinels $\{a_\omega\}_{\omega=1}^{s}$ as $a_\omega = f(k,\omega)$. We append these sentinels to F'', yielding F'''。

(4) Permutation: Let $g:\{0,1\}^j \times \{1,\cdots,b'+s\}^* \rightarrow \{1,b'+s\}$ be a pseudorandom permutation. We apply g to permulate the blocks of F''', yielding the output file $\tilde{F}$。

For example: a block size of 128; 128 bits is the size of an AES block and yields sentinels of sufficient size to protect against brute-force sentinel-guessing attacks. Let us consider use of the common (255, 223, 32)*-Reed-Solomon code, By means of the standard technique of "striping", we can obtain a* (255, 223, 32) *code over* GF^{218}. *A chunk consists then of* $n = 255$ *blocks.*

Let us consider a file F with $b = 2^{27}$ *blocks, i.e. a 2GBfile. This file expands by just over 14% under error-coding to a size of* $b' = 153{,}477{,}87$. *Suppose that we add* $s = 1{,}000{,}000$ *sentinels.Thus the total number of blocks to be stored is* $b' + s = 154{,}477{,}870$. *The total file expansion is around 15%。*

Consider $\epsilon = 0.005$*, i.e., an adversary that has corrupted* 1/2% *of the data blocks and unused sentinels in* F. *Now* $C = b' / n = 601{,}87$ *and* $\mu = n\epsilon(b' + s)/(b' - \epsilon(b' + s)) \approx 1.29$ *(*μ *is an upper bound on the mean number of corrupted blocks per chunk). By theorem 1 in [6],* $Ce^{(d/2-\mu)}(d/2\mu)^{-d/2} \approx 601{,}874 \times e^{14.71}(12.41)^{-16} \approx 4.7 \times 10^{-6}$, *i.e. the probability that the adversary renders the file unretrievable.*

Suppose that we let $q = 1{,}000$*, i.e., the verifier queries 1,000 sentinels with each challenge. Since the total number of sentinels is* $s = 1{,}000{,}000$*, the verifier can make 1,000 challenges over the life of the file (a challenge per day for about three years). The probability of detecting adversarial corruption of the file is at least* $1 - (1 - \epsilon/4)^q \approx 71.3\%$. *This is not overwhelmingly large, of course, but probably suffices for most purposes, as detection of file-corruption is a cumulative process. A mere 12 challenges, for instance, provides detection-failure probability of less than 1 in 1,000,000.*

4.3.1 Better Sampling Mechanism

Genaro's innovation was to make changes to the previous SPoR algorithm based on the characteristics of the public chain. In the operation of the public chain, there will be pseudo-random hashes. In the storage process, if the number of fingerprints per inspection is small, the miners will have a relatively low cost when attacking. Combining these two features, Genaro designed a sampling mechanism that conforms to the public chain storage. In the previous SPoR, we used a specific file fingerprint, which is the sentinel, to ensure the validity of file storage backtracking. In the new version, we will use a better sampling mechanism instead of the previous one. For details, refer to [7]. In the following, we replace the sentinel with a file fingerprint, and the granularity refers to the size of the file fingerprint. Specifically:

- We turned the original single file fingerprint into a random multi-selection type, that is, to have a finer granularity for each of the previous file fingerprints, so that it is easier for the miners to contribute to the long-term contribution when selecting miners.

  For example: when the file fingerprint granularity is 6 , the miner can get a unit of file fingerprint as long as it exceeds 1/6 contribution capacity. To get the next need to contribute plus 1/6, when the granularity is 12, the miners who contributed 1/6 before got 2 fingerprints, but if you want to get a file fingerprint, you only need to increase the contribution by 1/12, so when you have a large granularity, for good miner who wants to make more contributions is an incentive.

- Selects unified check multiple times each time you select a file fingerprint, and perform the traceable test by randomly selecting the fingerprint group. Relative to the verification of a single fingerprint, the miner (that is, the part responsible for receiving the storage in the storage system) predicts the verification problem given by the verification node relatively easily, thereby performing a replay attack in which a specific fingerprint is given in advance to delete a specific file. In order to technically reduce the possibility of replay attacks, we have adopted a random selection of fingerprint group verification. This can improve the single check by random group class, which greatly reduces the chance of replay attacks, and increases the difficulty of attack by finer granularity.

  For example, in the original case, if a miner has a fingerprint of No. 4 file in a single verification, he only needs to submit the fingerprint of No. 4 file as a receipt of the verification signal to continue the replay attack without considering whether to save the No. 4 file. The only thing that can be ensured is that the file generated by the image is not attacked. Otherwise, if the mirror node is used at the same time, then the file cannot be retrieved; in the fingerprint set verification, if the miner has 6 fingerprints of a single file of 4, 5, 6, 7, 8, and 9, once a fingerprint set that requires 5, 7, and 9 and a fingerprint set that requires 4, 7 at a time are quite different in the verification signal giving the required receipt. In this case, if the replay attack is made, the miner needs to generate 63 groups, which means that if the three verifications fail, it can be concluded that the miner has not contributed and discharged the miner, so the difficulty for the attacker is greatly increased. In fact, we use a finer granularity than previous versions in our development, which can increase the difficulty.

- The random number is selected by the specific byte of the hash generated by the block, which ensures that the file fingerprint of each test cannot be predicted. The generation of a random number in the blockchain is relatively limited. We choose the partial bytes of the hash generated by each block evolution as the basis for selection, and the relative probability of these fields is relatively small. That is to say, if a miner wants to do something wrong in a particular block, they first need to stake in the system and contribute to a specific standard and at the same time just for the replay attack and the miners who are backing up with him at the same time, thus the probability is relatively small and requires a lot of cost to achieve.

Compared with the previous sampling mechanism, the new file fingerprint group sampling mechanism has better characteristics for backtracking proof by miners' replay attacks and has a better incentive mechanism for miners than larger granularity file fingerprints.

4.4 Chain Style PoS

Before we introduce the chained PoS, it is worthwhile to compare the other mainstream chain consensus forms. The most mainstream now is the chained PoW, also known as the Nakamoto, the most representative of which are Bitcoin and Ethereum. The existing PoW has a workload in the 5000/GH class and the memory I/O class of the CPU/GPU class to 3000TH. The CryptoNight form used by Monero and Bytecoin was cracked by ASIC processors, resulting in more and more limited anti-ASIC algorithms, such as blake2b. As the memory configuration of ASICs becomes higher and higher, the ASIC configuration of related PoWs will become more and more serious. As a new public chain, if you continue to use PoW, it will be easily succumb to a 51% attack. Moreover, in terms of energy consumption, the entire system is cost-controlled by means of energy consumption, which was considered unsustainable at the beginning of the design process for Genaro. Based on these considerations, the Genaro public chain was originally designed with the concept of chained PoS.

In the details of the Proof of Stake, we need to consider the choice of the fork condition or quickly eliminate the forked part in a quick way. That is to say, for PoS design, we need to consider two PoS problems: nothing at stake and long range attack [14].

4.4.1 Nothing at Stake

The problem in the design of the PoS system is that in contrast to PoW's continuously increasing work costs, PoS nodes only need to stake to participate in consensus. It is easy to do so in a stake that a user who can pledge a lot of tokens can perform multiple forks for proof, and this situation is also the most profitable option for the prover.

For example, when multiple forks happen simultaneously, simultaneous authentication of the forks is the most profitable choice for the attacker, because no matter which is the longest chain, he will get the benefit for the whole stable system – because FLP is unlikely to be characterized by a constant balance of dual-valued states.

We mentioned earlier that there is a need to provide a state-oriented job in the system. Genaro's work is done by adding additional storage information, which is covered in the Storage Hybrid Consensus section.

4.4.2 Long Range Attack

This attack works thanks to the way we reserve availability in the CAP theorem. What the attacker needs to do is to attack through the attributes of this system. Because the availability is preserved, in this case, consistency is not guaranteed – that is, the attacker can fork through the farthest block. In the entire distributed system, the system also believes that the response before the block has just been recovered, so our new approach needs to be guaranteed by special transactions or status checks. Genaro uses the latter here and this part will be covered in the Storage Hybrid Consensus section.

4.4.3 Storage Overlay Hybrid Consensus

In order to solve the problem of the above two chained PoS problems, Genaro has devised a mixed consensus through the combination of the storage part and the public chain.

First of all, we need to briefly mention the storage-related components, i.e. the file fingerprint group. We ensure the integrity of the file uploaded into the distributed storage network through the random fingerprint pair and PoR algorithm of the file, that means it can be retrieved by a timed random sampling method.

For example, when uploading a file, the file is stored by adding a fingerprint record, and the reliability is increased by a redundant storage related method. Then the related operation of checking the storage side is periodically performed, specifically, in a pseudo-random pattern obtained by block hashing. This checks the random fingerprint pair in the fingerprint group record. If the return value is correct, the verification will pass, and any error or timeout operation will not pass. Mixing the previous practices into our chain, we can see how we solve the two problems mentioned earlier.

For nothing at stake, we provide the total number of inspection fingerprint record groups to the blockchain consensus operation. Since the total records of the inspection are synchronized on the chain, on the multiple forks the only part on the main chain that can synchronize to the latest is the total number of fingerprint inspection records with a relatively large value. This extra piece of information is the part of the information that was previously mentioned to push the balance of the two-valued state. With this information, we can impose additional penalties on the attacker who bet on the wrong fork.

For the long term attack, as with the previous solution, since the fingerprint verification record is introduced, the part of the relatively long block fork can be determined by the total number. Even if the fork can be evolved at the beginning, after the verification point, it will stop because the total number of fingerprints is smaller than the main fork.

After solving these two related problems, a new problem arises. If a long range attack starts from the previous fingerprint verification point, a new attack mode mayarise depending on how quickly the attack is repelled. Genaro will introduce the governance of the consensus and solve the problem through chain governance.

4.5 Data Governance Design

As an indispensable link in the consensus, the governance structure also makes a major difference from the traditionally structured Internet. If in a network statisfying consistency, verification in the system can be proven within the specified time, then in the network statisfying availability, each node in the system needs to follow a specific data governance structure to broadcast in order to maintain the fault tolerance characteristic of the entire system.

In the data governance structure of the Genaro public chain, we can design the following governance methods through the additional information obtained in the storage network:

- Each storage node needs to stake to enter the system contribution storage unit;
- After storage, the ranking is calculated by the combined hefts of the storage contribution and the stake amount combination;
- Highly ranked 101 nodes generate block operations in the public chain. The selection of 101 nodes is based on conclusions derived from successful PoS cases. Compared with the 21 nodes of DPoS in EOS, 101 is a greatly increased number. Inclusiveness also increases the difficulty of adding new node flood attacks;
- Each round of the committee will generate a block in turn and the higher ranked nodes will have more rounds. The existing block generation is divided into a round out block and a random number block. In Ouroboros, the random number is encrypted and communicated, so the block output occurs according to a specific random number, because it has been determined that the PoS is not necessarily comprised of good nodes. Genaro finds good nodes and then ranks them by storing additional information about how data can be retrieved. With Genaro's approach, we can generate blocks faster because you save the random number communication operation;
- Large nodes can choose small nodes to contribute storage for themselves.

Under this governance structure:

(1) Each storage node wants to start contributing in the system and needs to stake to enter the system. In the design, the storage node needs to be relatively stable. This mechanism for entering the threshold for the admission mechanism can be selected in the primary node. Part of the screening has already been obtained, because when the data needs to be stored, the storage space it needs and the space for mapping backups require relatively robust storage nodes to ensure that the data can be retrieved later.

(2) Since the members of the committee are ranked by the comprehensive heft of the storage contribution and the amount of the stake, the nodes of the chain system need to be recognized by both the storage network and the public chain to be accepted by the committee. That is to say, for a single approach to pledge, the ranking of the storage network can be stably kept on the committee relative to the nodes that can guarantee the

stability of the system and avoid being pushed out by people with excessive assets.

(3) The large node can encourage more small nodes to participate in the storage construction of the storage network by letting the small nodes contribute storage, and the large nodes can also improve their ranking in this way. Such a participation mechanism allows medium-sized nodes and large nodes to have a chance to play. Although there are different divisions of work for large nodes, medium nodes, and small nodes, there is a fair part in the overall mechanism design. Medium-sized nodes can compete with large nodes, and small nodes can sell their own storage to obtain additional blocks.

5 Summary and Outlook

The current technical advantages of Genaro Network include: (1) The design of the Sentinel Set, (2) The original hybrid consensus mechanism and the combination of the consensus SPoR and the chained PoS, (3) The unique data management method on the chain, (4) The encryption algorithm which involves specific features of GSIOP, (5) The storage information on the instruction level of the VM stack and the special transaction to meet the storage requirements, (6) The means by which the number of times the miner is asked multiple times is reduced, thus increasing network stability. In the future, Genaro will consider more encryption storage features, including the use of de-duplex encryption to ensure that a single encrypted file is not redundantly processed to increase miners' benefits. Encrypted file stream storage allows users to download and watch streaming media operations. In terms of consensus, Genaro will provide more fine-grained consensus improvements to ensure the participation of small and medium-sized nodes and increase the instruction set of new VMs to expand and increase more application scenarios.

References

[1] Schollmeier, R. (2002). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, *Proceedings of the First International Conference on Peer-to-Peer Computing*, IEEE.

[2] Bandara, H. M. N. D. and Jayasumana, A. P. (2012). Collaborative Applications over Peer-to-Peer Systems – Challenges and Solutions. Peer-to-Peer Networking and Applications. arXiv:1207.0790.

[3] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*. 31(4), 149.

[4] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 329–350.

[5] Maymounkov, P. and Mazieres, D. (2002). Kademlia: a peer-to-peer information system based on the XOR metric, *International Workshop on Peer-to-Peer Systems*, 53-65.

[6] Juels, A., and Kaliski, B. (2007). PORs: Proofs of retrievability for large files. *Proceedings of CCS* 2007, 584–97, ACM Press.

[7] Naor, M. and Rothblum, G. N. (2009). The Complexity of Online Memory Checking. *Journal of the ACM* (JACM), 56(1).

[8] Yu, S., Wang, C., Ren, K. and Lou, W. (2010). Achieving secure, scalable, and fine-grained data access control in cloud computing. INFOCOM, 2010 Proceedings IEEE, 1-9.

[9] Blaze, M., Bleumer, G. and Strauss, M. (1998). Divertible protocolsand atomic proxy cryptography. Advances in Cryptology—EUROCRYPT'98, 127–144.

[10] Ateniese, G., Fu, K., Green, M. and Hohenberger, S. (2005). Improved proxy re-encryption schemes with applications to secure distributed storage. In Proceedings of the 12th Annual Network and Distributed System Security Symposium, 29-44.

[11] Michael J.Fischer, Nancy A.Lynch and Michael S.Paterson, (1985) Impossibility of Distributed Consensus with One Faulty Process, Journal off Association for Computing Machinery.

[12] Lynch, N. and Gilbert, S. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News, 33(2), 51-59.

[13] Castro, M. and Liskov, B. (1999). Practical Byzantine Fault Tolerance, In the Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA.

[14] Bentov, I., Gabizon, A. and Mizrahi, A. (2016). Crytocurrencies without Proof of Work. International Conference on Financial Cryptography and Data Security.142-157

[15] Maymounkov, Mazieres, (2002) Kademlia: A Peer-to-peer information System Based on the XOR Metric. IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems. 53-65

[16] A.D.Kshemkalyani, M.Singal, Distributed Computing: Principles, Algorithms, and Systems, ISBN:9780521189842, paperback edition, Cambridge University Press, March 2011. 756 pages