LunarX: Universal Middleware for Data Driven Decentralized Applications, Part I (draft version 0.6.6)

(https://github.com/LunarX-ONE/White-Paper)

Ben Fei
Email: feiben@lunarion.com

Jiangtao Li
Email: silver.lijt@gmail.com

***Abstract*—LunarX is an universal middleware, designed to build a decentralized, infinitely-scalable, temper proof, anonymous and autonomous data management infrastructure for decentralized applications(DApp) and their users. It resembles database systems, but without centralized servers. Every peer joining the LunarX network becomes a part of the whole data service system, providing data access functionality to clients.**

In real world businesses, not all the data is transactional that has to be put into a blockchain, but these great amount of data still requires properties of anonymous, autonomous, temper proof, scalability and traceability, in an untrusted peer to peer network. Some of these data is structured, other is unstructured. We propose an algebraic definition of commutative map to study these data structures within a uniform framework, and to see when given a data structure, which kind of encryption schemes are commutative with it, and how to construct a possible scheme. In order to store, update and track these data structures, we developed DAG model for these purposes. Built upon several key distributed computation, cryptography and database techniques, LunarX is designed for serving such data management purpose. It is a developing pilot project, hoping to contribute some ideas to the community.

1. Introduction

Centralized online platforms provide convenient tools for users to access information and services from anywhere and anytime. These platforms normally manage big data centers, driven by variant database systems, from simple embedded k-v modules, to structured relational db systems, and new-sql big data clusters. But the greater concern in long term is privacy. These platforms not only have all the knowledge of their users' data, but also share users' identities and user's characteristics. Users have no way but to TRUST these platforms that their data and identities are safe. On the other hand, end users eventually pay for the high cost of these data centers.

In contrast to centralized platforms, decentralized applications(DApp for short) do not own users' data, users themselves do. cryptographic currency [1] and blockchain [2] technology solved the transaction part of data exchange. In fact, it is a data structure that chains blocks, and with each block, there are predefined fields used in recording properties of transactions, preventing "double spent" or other malicious actions on an account.

Transactional consistency is essential, but from a global perspective, only a little proportion of data is transactional. Other data has weaker requirement of globally consistency, but still requires a model with properties of secure, temper proof, anonymous, autonomous and traceability. It is a much more general and scalable data model for decentralized applications in describing the real world businesses.

This model is not as strict as transaction model. It is a distributed table that records users activities as rows, and does not collecte user data inside a data center, but distribute to possible peers joining in the network. Each individual user can only access his own data entries in the whole table, and these entries may be stored in multiple remote peers.

In addition, columns need to be scalable(vertical scalability), since new services may be developed in the future. Although the total data may be thousands of Tera bytes, the overhead is balanced, since every participant, including the applications' providers will only store and compute the part that they have been authorized to access. The data model is highly localized, like a huge sparse but locally dense matrix covering the world.

This is the reason why we also call LunarX a World Wide Table. This distributed table design is a middleware that knits individual nodes together, serving upper layer applications with uniform data management functionality.

There are several supporting techniques helping us to build the network: distributed hash table [8] [9], symmetric/asymmetric/homoporphic/non-homomorphic cryptography, consistent hash [11] [12], merkle tree/DAG [10] [27] [28], proof of storage and its variants, Reed-Solomon erasure coding [14], and some other peer to peer techniques.

N.Carmack
May 6, 2018

2. Background

Databases collect data together, mapping them to structures for future update and analysis. Normally they are deployed in one or multiple centers, and machines, which run db instances, trust each other. But in P2P network environment, since there is no trust and no stable service can be expected and provided, data security and availability become essential and critical.

2.1. Homomorphic Encryption

Homomorphic encryption is a form of encryption that allows computation on ciphertexts. Cloud computing platforms perform complex computations on homomorphically encrypted data, therefor clients sensitive data will not need to be exposed. Most recent advance on this field includes: The Brakerski-Gentry-Vaikuntanathan cryptosystem (BGV) [15], scale-invariant cryptosystem [17], the NTRU-based cryptosystem [18], and The Gentry-Sahai-Waters cryptosystem [19].

Fully homomorphic encryption schemes have weaker security properties than non homomorphic schemes. In our distributed data computing framework, we need to operate non-homomorphically encrypted data from remote, which is a much more strong constraint, leading to the algebra concept of commutative map(definition 5), but we redefine it for this context.

2.2. BlockChain

In fact, a blockchain is a decentralized sequential transactional database. .....

But we should think again that do we really need everything to be a part of transactions? In decentralized applications, we need to record immutable facts, which part requires no globally consensus, but a consensus reached by all the participants who involved in the facts. And in some cases, we do allow double spent. For example, one publishes his article and shares it to his subscribers. In this scenario, this article has been "spent" many times. What we have to do is recording these immutable activities, reaching consensus of all the subscribers and the author, and making sure they can not be tampered with.

2.3. Proof of Storage

Since users and the service providers have no trust to each other, they both need to delegate someone else to verify the validity of the data. Most importantly, the delegated verifiers have no knowledge of users data. This is called publicly verifiable [3]. To users, it is also hard to judge whether or not the storage providers and verifiers collude in generating the proof for rewards. Even all the parties are honest, it is a much more unstable computational task. Peers may encounter hardware damage, or have their devices severed from the network. The complete data still needs to be reconstructed from the network during the time that some peers are out of service.

Existing projects on this domain are FileCoin [5], IPFS [4], Sia [6] and Storj [7]. IPFS defines a world wide content addressable file system with its innovative content-addressed block storage model and with content-addressed hyper links. It saves data in a Merkle DAG [27] (directed acyclic graph), which is versioned and similar to the architecture what the git version control system [28] is using. IPFS designed key components including identity management, networking, data exchanging and routing using established peer-to-peer protocols. In addition, it introduces a novel BitSwap protocol, inspired by BitTorrant [24] for the exchange of data blocks and relies on a Merkle DAG to manage its content-addressable storage. FileCoin lays on top of IPFS and builds an incentive system in rewarding disk space providers and auditors.

The supporting component of these systems is proof of storage. Filecoin invented proof of Replication and proof of spacetime [25] for preventing Sybil attacks, Generation attacks and Outsourcing attacks. If they steal the replica from other nodes, the attack is called outsourcing attack. Or, if they somehow can generate a replica, or something else that can be sufficient in proof, the attack is called generation attack. But these attacks are scheme-specific, meaning that if we adopt data replication as our storage model, then malicious nodes will probably generate the proof of a replica on demand but actually they do not have the data in their local storage. LunarX may encounter some other attacks, which will be addressed after we define the data model and architecture in the next section.

3. Definitions and Data Model

Outsource computational tasks from a thin client with relatively weak computational device to a more powerful computation services (worker) risks the leak of information. In this section, we formalize the concepts that support information confidentiality, and in the next section we focus on computational verification.

3.1. Commutative Map

Definition 1. *A **map** is an operation that transforms a data set D to another structure without loss of information.*

$$m : D \quad \rightarrow \quad D \tag{1}$$

We define:

Definition 2. *A **unit map** I is an operation that keeps the data structure unchanged.*

Example 1. *The following examples do not change the data structure, hence are unit maps:*
replication of one data set.
query certain value from a data set.

Definition 3. *if $a \circ b = I$, we call a is the **inverse map** of b, and vice versa.*

For example, encryption $enc(.)$ is the inverse map of decryption $dec(.)$.

All maps, with the join $\circ$ of any two maps, form a semi-group M. Let $m_1, m_2 \in M$, $d \in D$, the properties of associativity and identity are satisfied:

$$m_1 \circ (m_2(d)) = (m_1 \circ m_2)(d) \quad (2)$$
$$I \circ (d)) = d \quad (3)$$

Since M is not a ring, and D is not an abelian group, D with an action of M does not form a $M - module$. Hence there is no distributivity(or bilinearity) of this structure.

Definition 4. *A **projection** prj is a mapping of a data structure into a subset.*

Projection may lose information, but will extract some meaningful results out of the whole original data set. Mathematical functions in data statistics and analysis are normally projections, like: multiplication, addition, summing, etc. Set operations like union and intersection are projections. And so are their combinations.

Definition 5. *We define that two maps $m_1, m_2 \in M$ are **commutative** if $m_1 \circ m_2 = m_2 \circ m_1$.*

Apparently, only those maps that are commutative with encryption, can be distributed to untrusted peers.

Lemma 1. *Every map is commutative with the unit map I:*

$$I \circ m = m \circ I \quad (4)$$

Proof 1. *Obvious.*

Example 2. ***Homomorphic Encryption**: Some encryption schemes are homomorphic with respect to their specific operations. In our context, these operations are commutative with certain encryptions. For example, if an encryption algorithm $h\_enc(.)$ has homomorphic properties for string concatenation $cat(.)$, which is a projection of a string space, then one can concatenate the ciphertext:*

$$s = cat(h\_enc(Str1), h\_enc(Str2)) \quad (5)$$

and

$$h\_dec(s) = cat(Str1, Str2) \quad (6)$$

. where $h\_dec(s)$ is the inverse map of $h\_enc(.)$. Apparently, it is commutative:

$$cat(.) \circ h\_enc(Str1, Str1) = h\_enc(.) \circ cat(Str1, Str2) \quad (7)$$

.

The math concept defined in this section has fruitful intrinsic structures to be discovered. As we mentioned above, it is not a $M - module$, then what it is exactly? Since this is a system design article, we will not go deep on this subject, but will draft another paper to elaborate this theory.

3.2. Typical DB Structures

Online data-related services usually depend on the basic data access capability provided by the database systems. Starting from analyzing typical data structures, we will discuss the design of an universal middleware.

3.2.1. Numerical Calculations. Numerical calculations are projections mapping data from a high dimensional space to a discrete space. For example:

addition of two numbers is not commutative with non-homomorphic encryption. But addition may be commutative with homomorphic encryption schemes.

Ordering of a random number sequence is not commutative with encryption.

There are many data structures invented in the past decades for various data management purposes. We here discusses some typical structures that are mostly used in database implementations. Both relational databases and new-sql column based big data systems may have different indexes on columns in order to be able to efficiently find rows that match a particular query. Indexes are auxiliary data structures that represents the information in different ways. Hence they are maps between data structures. For numerical columns, B+ tree is a standard choice for range queries. And on string or text columns, inverted index and hash table are normally used to perform keyword search in constant complexity.

3.2.2. B tree. B tree and its variant B+ tree [26] are data structures that maintain the order of numerical random data sequences, and perform searches, insertions and deletions in logarithmic time. The process building a B+ tree is a map that attaches each element to a path from the root of the tree. The leaf nodes of the tree are ordered, so that we can find any range of data without going through all the data set. It is quite efficient when the data set grows huge.

But if encryption schemes can not preserve the order of the original data, applying B+ tree upon an encrypted data set holds no meaning. Order-preserving encryption

scheme(OPES for short [20]) enables to apply comparison operation directly on encrypted data. But this advantage is accompanied by a weakness of the system. An attacker does not have to determine the exact sensitive data d, on the contrary, it is still a successful attack if he can estimate a tight enough interval including d with a high enough probabilistic confidence level, say, 95%.

An attacker can repeatedly generate random numbers to test the encryption system and get the encrypted data, by comparing with the users' encrypted data, he is able to estimate what the original plain data is. Hence the domain of users' data must keep secrete.

Monotonic functions are order preserving, so are their composites. Here we construct an example function, which is piece-wise linear with randomly generated scaling parameters. Users should construct their own in practical applications.

Example 3. *let* $L = \{l_0, l_1, l_2, ..., l_n\}$*, where* $l_i \in (0, \infty)$ *are randomly generated.*
Define $[0, 1]$ *the domain of* $f(x)$*, and randomly split it into* n *intervals:* $[x_i, x_{i+1})$*,* $i \in \{0, 1, ..., n-1\}$*:*

$$f(x) = \begin{cases} 0 & x = 0, \\ \Sigma_{j=0}^{i} l_j + \frac{l_{i+1}}{x_{i+1} - x_i}(x - x_i) & x_i \leq x < x_{i+1}. \end{cases}$$

Insert and Query:

1) For a randomly incoming data sequence $D = \{d_1, d_2, ...\}$, user select a locally monotonic function f with domain $[x_1, x_2]$, calculate $f(d_i)$.

2) Transforms d_i linearly into the monotonic domain of f. Since the rang of d_i is unknown, we may multiply a sufficient small factor ϵ to the data set to make sure all the possible values fall into $[x_1, x_2]$. ϵ is application specific. Or we can choose a function with monotone domain from $-\infty$ to ∞.

3) Sends $f(d_i)$ and the signature to the remote peer.

4) On remote peer, verify signature firstly, if fails, reject the request.

5) Remote peer inserts into its local B+ tree. Remote peer has no knowledge of the function the user construct, he will not possible to reconstruct the function.

6) In any range query from x_i to x_j, user send $[f(x_i), f(x_j)]$ to the remote peer and get the list of result points. Use the inverse function of f to get the original ranged data set.

3.2.3. Hash Table. The complexity of accessing data by hash table is $O(1)$. $Put(D)$ stores data D at the array index computed by hash function $hash(.)$, and $get(D)$ fetches D from the index. So actually $get(D) = query(.) \circ hash(D)$, where $query(.)$ gets data from the index.

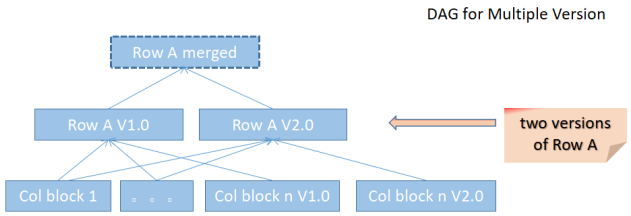

Figure 1. DAG for Local Data Structure

Lemma 2. *$get(.)$ is commutative with encryption $enc(.)$:*

Proof 2.

$$dec(.) \circ query(.) \circ hash(.) \circ enc(.) = query(.) \circ hash(.) \quad (8)$$

since $enc(.)$ is the inverse element of $dec(.)$, we left multiply $enc(.)$ on both sides and get:

$$query(.) \circ hash(.) \circ enc(.) = enc(.) \circ query(.) \circ hash(.) \quad (9)$$

3.2.4. Inverted Index. Actually an inverted index is a hash table with each entry attached a list of content ids(or addresses). It is popular and wide adopted by large scale search engines [22]. When in query, one can quickly locate where the keywords are, and fetch all the content including them by visiting the storage in constant complexity. Hence the following lemma is apparent.

Lemma 3. *Operations, including union, intersection and any combination of the two, on this data structure is commutative with encryption.*

$$union \circ enc = enc \circ union \quad (10)$$
$$intersection \circ enc = enc \circ intersection \quad (11)$$

3.3. Data Model

Multiple version control: a git-like DAG model for multiple versions of a data entry(figure 1).

Tamper resistance: to be done.

As we mentioned before, The data model is highly localized, like a huge sparse but locally dense matrix covering the world. to be more precisely, this locally dense part is a DAG (directed acyclic graph), which is not only used for version tracking, but also used to meet many other requirements of structured data processing. We will keep updating data model during project developing.

4. Network Definition, Attacks and Solutions

4.1. Properties

A scheme of proving something(not only storage), in an environment absent trustiness of each other, must has such properties:

P1) Transparent: no prover is able to generate a valid proof if it has no original data.

P2) Zero knowledge: a verifier must validate a proof without knowing the original data. Verifier can be anyone, and this is called publicly verifiable [3].

P3) No conspiracy: a verifier has no chance to collude with any prover, generating positive confirmation of fake storage, or negative proof of an honest service provider.

P4) Distinguishable identities: Identify every peer that participates in the network, ensuring that for a certain data piece, different peer has different signature, hence stops outsourcing attack [25].

P5) Robustness: in a globally distributed system, peers may often be disconnected from the network, no responding to any challenger or data request. Obviously this is not malicious. After it is online again, auditors should be notified and begin the data possession validation again. So a criteria to distinguish malicious actions or just normal disconnection is quite necessary.

P6) Retrievability: proof of retrievability [29] [30](POR for short) is a scheme to make sure a service provider has no chance to blackmail users. In one hand, a provider can prove he has the data belonging to a user, but in the other hand, alters the code so that he will not release the data to the owner unless the owner pay something to him. PORs hence enable the data owner to challenge the provider several times, and at each time gets one piece of data along with the proof. After a round of challenges, the owner can reconstruct the complete data. POR is an enhancement of PODP(Proof of Data Possession) [23] [31]. But according to P2), any verifier must not has the ability to reconstruct data by repeatedly challenging a provider. So in LunarX, we require that users have to encrypt their original data before submitting them to the network.

P7) Memoryless: if a peer proves its storage of a data D in time slot t, its probability $p(t+s)$ of possessing the data for extra time slot s is the same to $p(s)$. This motivates us to use a exponential distribution to generate challenges to peers. The promising point is that when the time slot gets denser, the cost of attack grows exponentially, due to the advantage of exponential distribution.

4.2. Roles

The network consists of three major roles: DApp and their users, service provider, tracker. The following scenarios elaborate how these roles cooperate with each other in the LunarX network.

DApps and their users:
DApps run on LunarX and their users who subscribe the services they published. Both sides are data senders, and the receiver is no longer a private database, but the LunarX open network. For example, a storage business provides a cloud-storage application, its clients may choose to submit files to LunarX, and the business has no way to access these files. Only thing it possibly have is the data that clients authorized it to have.

Service Providers:
Any user can join the LunarX network to be a data service provider, selling his spare disk space to other peers in the network. The LunarX daemon automatically accepts records from the network and insert them into local table shard. At regular intervals, the service provider earns incentive tokens for his service, as long as automatic proofs show that the data structure is still available and complete, without being altered or damaged.

LunarX employs an incentive blockchain system to mine tokens. During the repeatedly proofing of valid data computing, blocks will be created, and with them a certain amount of tokens as the mining reward will be sent to the provider's wallet address. Also, the blockchain manages exchanges of tokens and services.

Not only local file systems can be of service, the distributed file systems (DFS) are also good candidates for professional users. There are many industrial level DFS implementations, for example the hadoop distributed file system(HDFS) [32]. Typically, this option is for traditional data center operators.

Trackers:
Trackers validate transactions between service users and service providers. In addition, each of them drives a bunch of verifiers to audit db service providers of the retrievability of the data and availability of the services they deployed. And trackers will be rewarded by tokens for their service as well.

4.3. Rules

The following rules are common sense, but we still list them here for the completeness of this work. Readers shall keep these in mind in the following discussion to avoid possible logical fallacies.

1)Service providers have no original data. What they are permitted to have is the encrypted data only.

2)Verifier has zero knowledge of the original data, neither it can reconstruct the original data via repeatedly challenging. Although we expect the auditors to be honest, the protocols of verification/audition are open. Then any peer has the chance to imitate a verifier to steal data.

3)Most of the possible attacks or cheats we talk about have a prerequisite that the attackers have the ability to tamper with the code and disguise their nodes as normal nodes in the network, following all the LunarX protocols in communication, but with fabricated data encapsulated in the messages for malicious purposes.

According to the above rules, together with the properties in section 4.1, we may be able to design our defence system against attacks. We firstly begin to discuss the possible attacks in an untrusted network and how our defence works.

4.4. Attacks/Flaws

From now on, we use data D to represent a collection of table entries, and $enc(D)$ is the encrypted version of the data by symmetric or asymmetric cryptographic methods. Normally, some entries may serialize very big binary or text data, We will not address how a big entries being segmented to relatively small pieces, since it is straight forward. What we will focus is how $enc(D)$ is going to be saved(in merkle tree or DAG), secured, verified, and reconstructed. Since if every piece of a $enc(D)$ is safe and retrievable, the whole $enc(D)$ is safe and retrievable.

Sybil attack [33] [34]: Sybil identities $S1, S2, ..., Sn$, controlled by an attacker, claim they all have a copy of $enc(D)$, and generate valid proofs when challenges come from verifiers. But actually, they only have one(or significantly lesser) copy of $enc(D)$.

Natural Damage: In a globally distributed environment, services provided by nodes from everywhere is unstable. They may have disk errors in running, network jam in communicationor whatever problems of devices that they can not response any request in time. So for the robustness of the system, we leverage Reed-Solomon erasure coding [13] [14] for recovering data from error nodes. That is to say, if some of the nodes fail to respond data request, we still can recover the complete data from the remaining nodes. In our test, Reed-Soloman coding reaches 400MB/G/core on our desktop computer.

Forging Attack: Provider does not honestly store $enc(D)$, but generates proof on demand, i.e. when it receives a challenge, it steals the data from another peer. If a scheme relies on multiple replication of the original data, a fake storage could be forms of outsourcing or generation just in time [5]. Since LunarX has no duplication of the original data in any untrusted peers, attackers then have nowhere to steal the data in responding challenges. This attack has no chance to perform.

Instead, a possible cheat from a provider is in this way: provider gets the data and its merkle tree, but he only persists the merkle tree and discards the data. Hence he can claim more space to auditors for more rewards. We defend this by randomly injecting water print in the encrypted data, and we name it proof of random injection(PORI):

Setup:

1) User U submits $E = enc(D)$ to LunarX.

2) LunarX segments E into pieces $E_i, i = 1, 2, 3, ..., n$, constructs Reed-Soloman data blocks $RS_j(E), j = 1, 2, 3, ..., m$, where $m > n$, depending on what code rate we choose.

3) Injects in random positions $p_k(RS_j), k \in Z$ in some of these pieces RS_j with special water print $c_k(RS_j)$.

4) LunarX tells U these $c_k(RS_j)$ and $p_k(RS_j)$.

Challenge:

5) Verifier issues a challenge with $p_k(RS_j)$ for RS_j, k and $j \in \{1, 2, ..., n\}$ are randomly selected.

6) Prover, i.e. the provider, answers this challenge by seek in RS_j and read the water print $c_{j,k}(P)$.

Verify:

7) For all j and k, Verifier receives the code $c_{j,k}(P)$.

8) Compare $c_{j,k}(P)$ and $c_k(RS_j)$, equals then the proof is valid.

Hostage Attack: A provider holds the data piece but does not release it to its owner(the user who submit the data). The retrievability (P6, section 4.1) prevents this attack.

Conspiracy Attack: Verifiers and providers collude. A provider offers no service but collude with a bunch of verifiers for proofing the integrity and retrievability of the data he does not really has. This is the why the rule 2 in section 4.3 claims that a verifier can not have any piece of the original data. (to be more specific)

Ciphertext only Attack [20] [21]: An attacker only has access to all the encrypted data, but known nothing about the domain of the original data.

5. Design and Implementation

The protocol design for LunarX is meticulous about information protection(retrievability and confidentiality), public auditability and fault tolerance. As the incentive model that drives the operation of the network, blockchain technology is integrated for rewarding and punishment. Peers, providing computer resource to LunarX, are rewarded

with tokens, while those who cheat, will be severed from the network, even their tokens will be deducted as punishment. In this section we take a more detailed look at each of these aspects.

5.1. Network

As mentioned before, LunarX network has three types participants: DApps and their users, trackers and service providers.

Trackers are a group that runs auditors to validate the authentication of data storage. Any individual who has computational resource can join the group, receive tasks of audition, issuing challenges to providers and verify their proofs. In auditing a huge amount of data, it is a resource consuming process, so officially we need the device connected to be an auditor meets some particular computing criteria.

Peers, who have spare resource, especially big disk and big I/O band width, contribute the capacity to LunarX in exchange for service rewards. They are organized in a DHT network, e.g Kademlia DHT [8] and its improvement S/Kademlia DHT [9], which stores nodes and resource locations throughout the network. Kademlia DHT uses XOR(exclusive or) operator to calculate the logical distance between any two peers. All nodes in a massive network according to the XOR metric construct a prefix binary tree, hence has the logarithmic complexity in searching through. Suppose we have 30 million nodes, in the worst case, it only costs 25 hops to reach the searched node.

A much more encouraging advantage of Kademlia DHT is the ability against the DOS(denial of service) attack. Even if some of nodes is flooded, routing procedure will seek around the jammed area, therefor it will have limited effect on network availability.

DApps, who seek extra bigger and trustworthy data service from LunarX for their clients, have two basic requirements: record data and query(download if data is some kind of files) data. Before submitting data to LunarX, the owner has to encrypt it by a selected symmetric/asymmetric encryption method. If one loses his key (private key for asymmetric method or secret key for symmetric method), then he exposes his data to the public. The data D exists in LunarX is the encrypted version $enc(D)$, which part we have already explained in the property of retrievability(P6, section 4.1).

Figure 2 illustrates the network topology, showing how trackers and providers are organized in LunarX. Tracker are labeled with $A_i, i = 1, 2, 3.....$, service providers are those around the world, marked by $P_j, j = 1, 2, 3...$, as well as a node with label C is a user. Accodring to publicly verifiability (P2, section 4.1), any third parties, labeled with

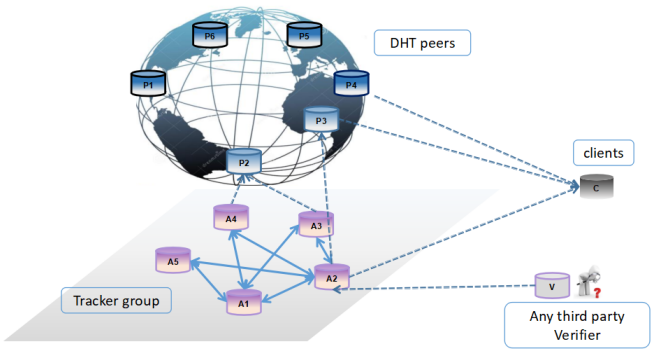

Figure 2. Network Topology

V, will register themselves to the tracker group to take tasks of audition.

Maps between data and nodes who store them, are maintained by tracker group in a big hash table. The trackers use data hashes as hash table keys. Each key maps to a unique node ID as the value which is responsible to store the corresponding data. Since storer nodes can dynamically join and leave the network, the hash table maintained by trackers is dynamic, periodically updated. When a node is not accessible for whatever reasons, the trackers who have the affected part of the hash table, have to update the table, replacing the dead nodes ID by another who has a copy of the data. Trackers themselves can be the backup nodes, if users choose them to be.

Trackers drive a bunch of verifiers to periodically audit peers for the validity of the data. Then also, every tracker maintains a list of registered verifiers.

5.2. Client Protocol and Fault Recovery

DApps may have big entries. For example, a P-2-P storage DApp has each data row which records users' files and metadata. As we mentioned before, this row storage sharding is straight forward. On client side, if one row has size less than 1Mb, we directly append it to the end of the table. If one entry is binary type and has bigger size, say 5GB, we will not store it in the table. We segment the entry data into each piece of 500MB, and record this segmentation and content links in the entry. These links will be used to patch up all the data pieces when the row is retrieved from LunarX. It is straight forward, then when we talk about entry data D, we suppose it is less than 500MB.

To a globally distributed system, nodes are unstable. They may be disconnected, jammed, hacked or destroyed purposely. Some solutions save multiple replicas as data redundancy. One fails, gets another. While our design leverages Reed-Solomon erasure coding for recovering data from error nodes. It is able to detect and correct multiple

symbol errors. Data D is divided into a list of m parts, and by adding t check parts to the list, a ReedCSolomon code can detect any combination of up to t erroneous parts, or correct up to $\lfloor t/2 \rfloor$ parts [13] [14].

While data replication is still an option. User may choose to have a back up of his original data, but with different metadata, water prints for proof, timestamps and some other information that identify the data. Only user himself and the trackers know the two copies are actually the same. Replica can be stored on trusted node, for example, on the trackers. We will go back to this point in the next section.

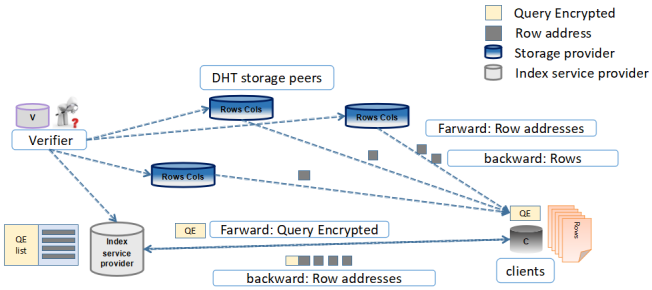

Figure 3. P-2-P Service Engine

5.2.1. Insert into Table. :

Insert(Row=$\{Col_1 = D_1, Col_2 = D_2, ...\}$**):**

1) Via LunarX client, user U encrypts D_i to $enc(D_i)$, keeps his private key in safe.

2) Selects a code rate of error eraser coding.

3) For each column, LunarX client divides $E(D_i) = enc(D_i)$ into pieces $E_j = enc_j(D_i), j = 1, 2, ..., n$, constructs Reed-Soloman data blocks $RS_k(E_j) = enc_j(D_i), k = 1, 2, ..., m$, where $m > n$, depending on the code rate he chooses.

4) Injects in random positions $p_k(RS_j), k \in Z$ in some of these pieces RS_j with special water print $c_k(RS_j)$.

5) LunarX tells U these $c_k(RS_j)$ and $p_k(RS_j)$.

6) Seek available storage peers via trackers for RS_j.

7) LunarX client calls:
$(key, P) = reserve(hash(RS_j(E)), RS_j(E))$,
directly connects to these peers and transfer RS_j to them by $(key, P, address) = store(key, hash(RS_j(E)), RS_j(E))$.

Before Step 6), we have to mention that the client has already joined the DHT network with its NodeID, generated by hashing his public key, and download the DHT routing table. At the same time, it connects to known trackers for peers which are available and have the space it needs.

In Step 7):
$(key, P) = reserve(hash(RS_j(E)), RS_j(E))$

The reserve operator invokes trackers to reserve space for $RS_j(E)$, with its hash $hash(RS_j(E))$ as the parameters, in response to a client request. The peer P generates a globally unique string which serves as a key that can be used by the client to insert and query his data $RS_j(E)$. Reserved space may expire if terms that both sides agree upon can not be satisfied. For example, after a certain time slot, P has not receive his payment, he may discard this reservation.

$(key, P, address) = store(key, hash(RS_j(E)), RS_j(E))$

The store operation sends $RS_j(E)$ and its hash directly to the peer P. The client must submit the key returned by $reserve(hash(RS_j(E)), RS_j(E))$ to authenticate with the peer. If some columns have big size data, the inserting procedure may be time consuming, LunarX supports broken-point continuingly transferring for big data segments.

5.2.2. Append Index for Columns. Since privacy protection is the top priority of the system design, some of the indexes is commutative, others don't. Then as we known, inverted index and hash table can be stored remotely. In LunarX, users may choose to store the index of their column data locally or delegate a provider to maintain their indexes. The delegated providers then become remote index service providers for their clients.

Since the storage is distributed, we have to use the separated architecture for a embedded database engine in P-2-P network(figure 3).

Here we use fulltext index as an example.

AppendFulltextIndex($Col_i = D_i$**):**

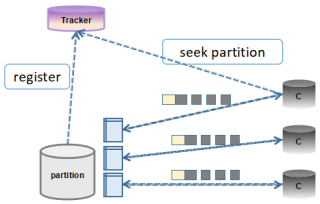

Figure 4. One to Many Partition

1) For Data D_i, $Inser(D_i)$ as in 5.2.1.

2) If D_i is a document or any object with abstracted features, tokenize D or its features, then we get a list of words: $list(w_1, w_2, ...)$.

3) Encrypt each words in the list, and get $List(D) = list(E(w_1), E(w_2), ...)$.

4) Insert $List(D)$ and the data address$(key, P, address)$ returned by $store(key, hash(RS_j(E)), RS_j(E))$ into remote index service provider.

5) On the side of index provider, select a tokenizer that separates $List(D)$ via simple comma, and index this content.

6) Return client the index provider ID, which will be used in search.

Normally, users are thin clients, having limited space for meta-data only. In contrast with centralized DB clusters which are heavy, LunarX partition the whole database into peers around the world, with each partition serving multiple clients(Figure 4). Since indexes are normally compressed structure, an index provider with mainstream desktop computer configuration may be able to serve tens of clients with each having 50 million records.

We must emphasize that only data with property of autonomous can be partitioned in the way LunarX does. For example, data from UGC(user generated content) DApps.

5.2.3. Query. :

Query(.):

1) connect to index provider, send query and get result set including the addresses of rows that match the particular query.

2) For each row D, which has been encoded in $RS_j(E), j = 1, 2, ..., m$, query DHT by calling $retrieve(hash(RS_j(E)))$, which locates peers that have the data.

3) Till enough $RS_j(E)$ have been found and downloaded, stops finding.

4) If a node holding any $RS_j(E)$ that is not accessible, just seek the next one $RS_{j+1}(E)$.

5) If the error node exceeds the upper bound of code rate, this file damages. Seek via trackers if there are replicas. If not, this row is lost. Procedure Ends.

6) Get $E(D)$ by decoding $RS_j(E)$.

7) Decrypt $E(D)$ to get D.

$retrieve(hash(RS_j(E)))$: Given the hash of $RS_j(E)$, the retrieve operation locates peers who have the data, and retrieves data from them.

In the case of hostage attack briefed in section 3.3, clients may repeatedly challenge the peer, and each time get a piece of $RS_j(E)$. After sufficient times of challenging, clients will reconstruct the complete $RS_j(E)$. Hackers will intentionally use the same methodology to get the data, but what they get is the encrypted version of the original data, and the client(the owner) is the only one who holds the key for decryption.

There is another way against the hostage attack. Just ignore $RS_j(E)$ and seek the next piece $RS_{j+1}(E)$, we will recover E as long as most of these pieces are correctly downloaded. This is what a download protocol does in above **Query(.)** procedure.

5.3. Tracker Protocol

Trackers have lot of things to do, including: maintaining data hash table, synchronizing for global consistency, managing verifiers for audition, detecting health of nodes via heart-beating, etc. Among these functionality, we give the details of the most important two operations: resolvePeer(.) for finding the resource, and audit(.) for proving the data validity.

$(P, key) = resolvePeer(hash(RS_j(E)))$:
This operation is invoked by a client to find a peer which is able to store the data piece $RS_j(E)$. The client sends the request to the tracker to which it is connected, with the hash of $RS_j(E)$ as a parameter. The tracker calculates the

real position A according to this hash $hash(RS_j(E))$ by some strategies, for example the consistent hashing [**?**] [**?**], then forward the request to it. When the request reaches the tracker A, it checks its list of registered storer nodes, and their capacity, QOS and credits to determine a qualified provider P for $RS_j(E)$.

$audit(P_i, hash(RS_j(E)), c_i)$ via PORI:
A tracker audits a peer P_i by issuing challenge c_i, together with the hash $hash(RS_j(E))$ of the data, to that peer. This procedure calls PORI that is defined in section 4.3 to proof the data possession. audit(.) will be invoked randomly and periodically as what we have explained before.

6. Applications

Based on LunarX, we have several DApps in developing. The first is a P-2-P storage application, users store their documents via a light client. And the second is in healthcare field, via smart devices, users submit their daily training data to LunarX, get analysis report of his health condition, share the parts they feel proud of and keep other data in secrete. Every one has a great amount of private data generated during his daily life. Due to the locality of LunarX, the overhead is well balanced around the world.

7. Future Work

In current release, LunarX supports mainstream local file systems, like NTFS for windows, ext family on linux, or mac file systems, both of desktop and server. We may offer versions for professional users that file systems for cluster, e.g. HDFS [32], will also be supported.

In the future, we will have more DApps for various businesses, which will greatly benefit from the scalability of LunarX. On fundamental research, we will invest great resource on distributed and encryption commutative data structures, with a much more practical view of how to balance the efficiency, scalability, privacy, how to design more available data functionality that DApps can smoothly plugin.

Acknowledgments

LunarX is inspired by many brilliant existing works, not only the works we mentioned in the reference list, but also our technical team and many of the online-offline discussions that we can hardly list all of them here. As an engineering pilot in this field, we hope this work to be an entry point for building a community that experts, scholars, geeks and whoever have enthusiasm can work together to advance the technology.

References

[1] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf

[2] Blockchian: https://en.wikipedia.org/wiki/Blockchain

[3] Publicly Verifiable Secret Sharing, https://en.wikipedia.org/wiki/Publicly_Verifiable_Secret_Sharing

[4] Juan Benet, *IPFS-Content Addressed, Versioned, P2P File System*, https://ipfs.io/ipfs/

[5] Filecoin: https://filecoin.io/filecoin.pdf

[6] Sia: https://sia.tech/sia.pdf

[7] Storj: https://storj.io/storj.pdf

[8] Kademlia (DHT): https://en.wikipedia.org/wiki/Kademlia

[9] I.Baumgart, S.Mies, S/Kademlia: A practicable approach towards secure key-based routing, in: Parallel and Distributed Systems, 2007 International Conference on, volumn 2, pages 1-8, IEEE, 2007.

[10] Merkle Trees: https://link.springer.com/content/pdf/10.1007/978-3-540-24676-3_32.pdf

[11] Karger.D, Lehman.E,Leighton.T,Panigrahy.R,Levine.M,Lewin.D,(1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. ACM Press New York, NY, USA. pp. 654C663.

[12] Consisting Hashing. https://en.wikipedia.org/wiki/Consistent_hashing

[13] ReedCSolomon error correction, https://en.wikipedia.org/wiki/Reed-Solomon_error_correction

[14] Reed, Irving S.; Solomon, Gustave (1960), Polynomial Codes over Certain Finite Fields, Journal of the Society for Industrial and Applied Mathematics (SIAM),8(2): 300C304.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping. In ITCS 2012.

[16] Z. Brakerski and V. Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In FOCS 2011 (IEEE)

[17] Z. Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In CRYPTO 2012 (Springer)

[18] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan. On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption. In STOC 2012 (ACM)

[19] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In CRYPTO 2013 (Springer)

[20] R.Agrawal, J.Kiernan, R.Srikant, Y.Xu. Order preserving encryption for numeric data, in Proceeding SIGMOD '04 Proceedings of the 2004 ACM SIGMOD international conference on Management of data, Pages 563-574

[21] D.R.Stinson. Cryptography: Theory and Practice. CRC Press, 2nd edtion, 2002.

[22] Knuth, D. E. (1997) [1973]. "6.5. Retrieval on Secondary Keys". The Art of Computer Programming (Third ed.). Reading, Massachusetts: Addison-Wesley. ISBN 0-201-89685-0.

[23] Proof of data possession, http://cryptowiki.net/index.php?title=Proof_of_data_possessio

[24] B.Cohen. Incentives build robustness in bittorrant. in Proceedings of the 1st Internation Workshop on Economics of P2P Systems (P2PECON '03)

[25] Protocol Labs. Technical Report: Proof of Replication. 2007

[26] Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes" (PDF), Acta Informatica, 1 (3): 173C189, doi:10.1007/bf00288683

[27] Merkle DAG: https://github.com/ipfs/specs/tree/master/merkledag

[28] Tommi Virtanen. Git for Computer Scientists: http://eagain.net/articles/git-for-computer-scientists/

[29] A.Juels, B.S. Kaliski Jr, PORs: Proofs of Retrievability for Large Files, in Proceeding CCS '07 Proceedings of the 14th ACM conference on Computer and communications security, Pages 584-597 .

[30] H.Shacham, B.Waters, Compact Proofs of Retrievability, Journal of Cryptology, July 2013, Volume 26, Issue 3, pp 442C483.

[31] G.Ateniese,R.Burns, etc, Provable Data Possession at Untrusted S-tores, CCS07, October 29CNovember 2, 2007, Alexandria, Virginia, USA, Page 598-610

[32] Hadoop Distributed File System, https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[33] Sybil attack, https://en.wikipedia.org/wiki/Sybil_attack

[34] Douceur, John R., The Sybil Attack, International workshop on Peer-To-Peer Systems. Retrieved 23 April 2016.

[35] V.Buterin,etc, Etheruem White Paper, https://github.com/ethereum/wiki/wiki/White-Paper