EasyChair Preprint

№ 4648

Continuous Distributed Key Generation on Blockchain Based on BFT Consensus

Lei Lei, Ping Ma, Chunjia Lan and Le Lin

December 14, 2020

Continuous Distributed Key Generation on Blockchain Based on BFT Consensus

Lei Lei*, Ping Ma*, Chunjia Lan*†, Le Lin*†
*Neng Lian Tech Ltd., Shanghai, China
†Corresponding Auther
{lei.lei, ping.ma, chunjia.lan, linle}@nenglian.com

***Abstract*—VSS (Verifiable Secret Sharing) protocols are used in a number of block-chain systems, such as Dfinity and Ouroboros to generate unpredicted random number flow, they can be used to determine the proposer list and the voting powers of the voters at each height. To prevent random numbers from being predicted and attackers from corrupting a sufficient number of participants to violate the underlying trust assumptions, updatable VSS protocol in distributed protocols is important. The updatable VSS universal setup is also a hot topic in zkSNARKS protocols such as Sonic [19]. The way that we make it updatable is to execute the share exchange process repeatedly on chain, this process is challenging to be implemented in asynchronous network model, because it involves the wrong shares and the complaints, it requires the participant has the same view towards the qualified key generators, we take this process on chain and rely on BFT consensus mechanism to solve this. The group secret is thus updatable on chain. This is an enhancement to Dfinity. Therefore, even if all the coefficients of the random polynomials of epoch n are leaked, the attacker can use them only in epoch n+2. And the threshold group members of the DKG protocol can be updated along with the updates of the staked accounts and nodes.**

Keywords—blockchain, distributed key generation, consensus, verifiable secret sharing

I. INTRODUCTION

Many BFT algorithms, such as Tendermint [3], use a pseudo-random algorithm (see §D) to compute next proposer. As method of calculating sequence is open to everyone, a malicious adversary can compute the sequence in advance and predict proposer of any future height/round. Thus future block proposer can be easily attacked by DDoS or network split.

Blockchain is a closed system, it cannot query information from the outside world, thus it cannot generate random number through the unpredicted input source like the index of the stock market. And it is a deterministic system, it expects all nodes run the same sequence of transactions can get the same result, and all inputs have to be verifiable, thus it cannot generate random number through random input source like the local timestamp or noise of the proposer node. If the input source is verifiable but predictable, for example, the random number is determined by the block header hash of some future height, the block proposer of that height can choose to contain different sets of transactions in order to obtain a favorable random number for him.

The "true random number flow" on blockchain is important to make the proposers priority and validators of each height unpredictable.

II. RELATED WORK

There are already some famous architectures devoted to generate "true random number flow" on blockchain, such as Algorand, Dfnity and Ouroboros.

The VRF processes in Algorand [2] are: At each height, based on the last random number, each node computes the VRF to obtain a new verifiable random number and participates in arriving at a consensus to confirm the hash with the highest priority—for example, the greatest one. However, a latent issue is that the generator itself can choose to not to broadcast the random number if it is not favorable to him, although he cannot control the other participants' random numbers, it does somehow affect the random number flow.

If we want to make the random number unpredictable to generator itself, each generator can only generate a piece of random number, and only pieces above the threshold can recover the group signature that can be used as random number. Thus each participant cannot predict next random number but has to participate in signature process honestly. Dfinity fits goal well. The DKG protocol—Joint-Feldman [1] is based on BLS threshold group signature [9,18] and has several advantages:

A. The group signature can be recovered by aggregating [8,10] the signature slices of any t participants. Usually the threshold $t = n/2$, which means the protocol is tolerant to at most half nodes offline.

B. Group public key can be computed by every participant, so once group signature is recovered, it can be sent to any node and be verified directly. Thus it's not always case t signature slices are needed for a participant to compute the group signature. This save a large amount of network propagation.

Our work can be summarized as enhanced Dfinity, that is: we do the share exchange process continuously at each epoch, rather than only doing once at the start of the chain. A continuous share exchange mechanism is necessary for two reasons: First, the participants might change, some may want to withdraw its stake, and others want to stake its deposit and join. If a participant has already withdrawn and get out, and is still a valid participant in the DKG protocol, this is clearly not reasonable. Second, the coefficients that generated once at a time might be leaked by human reasons, we cannot depend the system security on no person making mistake.

As for Ouroboros [7], its consensus protocol — Cardano, in first stage, it uses follow-satoshi(fts) and secure multiparty computation to randomly generate the slots for the proposers of

next epoch. In second stage, it changes to use VSS protocol to handle pause or network problem of the protocol in the first stage. We view VSS as an alias of DKG, and one part of our work is to solve the network problem in implementation of VSS protocol. And we do a step further, we consider the participant might be Byzantine thus they can collude to deduce secret, the analysis is in Section V.D.

III. Motivation

In order to make signature slice, the participant has to firstly compute its private key from the received shares of the generators in the "qualified set". There are some issues related to this process and is the key part that this paper wants to discuss.

A. Implementation

- To aggregate the received shares, each participant should have the same view towards the "qualified set". This makes DKG protocol challenging to be adopted in a weak synchronous network. A participant is considered as unqualified if it has been complained against by enough other participants. In a weak synchronous network, maybe some participants collect enough complaints toward a certain participant, but others do not, this lead they have different result of the qualified set.
- Even if we solve the issue, based on what rule, what number of complaints is "enough" to kick a participant out from the qualified set?

B. Security

- The shares that generated by participant are determined by the coefficients of the random generated polynomials. The coefficients can be considered as the initial private key of each participant and they should not be leaked. If we discuss the protocol in the Byzantine environment, we need to analyze the safety conditions that the adversary corrupts how many participants can it deduce to the coefficients of every participant, then the adversary is able to predict all the future random numbers flow.
- We should redo the share exchange process when the staked nodes change, for the new staked node should join the DKG process and the shares generated by old nodes should not be valid any more. Even if they do not change, constantly doing this process makes the system securer, though this adds extra network load, we can setup the strategy such as doing the share exchange process every 1000 blocks to make the extra load acceptable.

IV. Implementing DKG on Tendermint

Among BFT consensus protocols, some like CasperFFG [5] doesn't have the feature of instant finality, while Tendermint has this feature [3], we require the blocks of each epoch be finalized at the end of the epoch. Though HoneyBadger [15] also has this feature, its process is much more complicated, thus we decide to integrate the DKG process on Tendermint.

A. Modified consensus process of Tendermint

In Tendermint, a proposal block can be committed if there are $n - f$ precommits of the same round that are for it, then the node can enter the next height [4]. We now present another rule for this mechanism: the node needs to recover the group signature of each height either by gathering the signature slices of t nodes or by receiving a verified group signature before the block can be committed. Thus, at each height, the two conditions—$n - f$ precommits of the same round for this block and a group signature of this height—should be met simultaneously for the node to commit the block of this height.

The signature slice of P_j: $\sigma_{ij} = H_0(m)^{sk_j}$ where m should be a message that every node has a same view to. We use H_0 (group signature) as random number for each height. Since we want to have a "true random number flow", message should be relevant only with the previous random number, we use the concatenated string of group signature of previous height and its hash as the message. The first initial message is configured in initial config file. In Fig 2, We see that the message m is irrelevant with block data. Thus the block proposer cannot manipulate the block data to manipulate the random numbers.

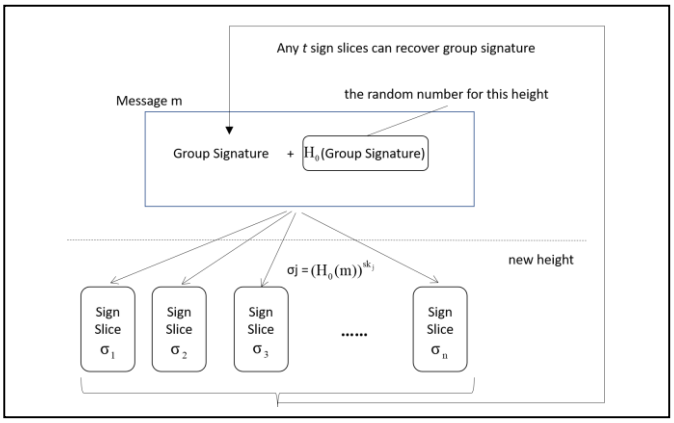

Fig. 1. Continuous random number flow

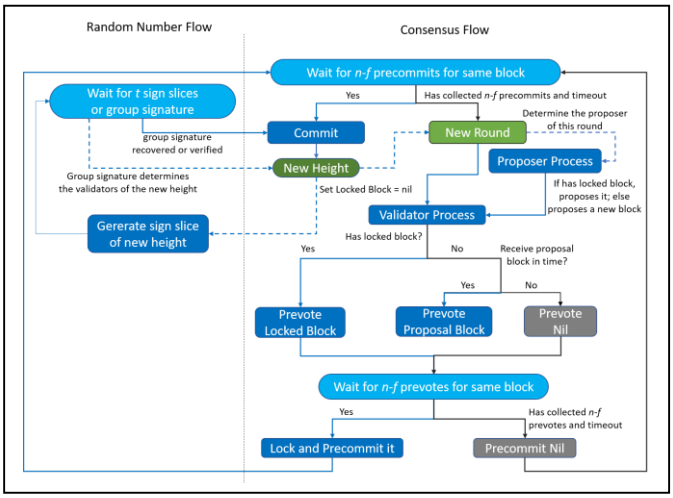

Fig. 2. Consensus flow combined with random number flow

B. Block format

As shown in Table I, a block contains a header that contains the hashes of relevant parts of this block and the state root of previous height. We add "DkgData" that contains the commitments and approvals to the block format of Tendermint.

TABLE I. BLOCK FORMAT

Block	*Field*	*Description*
Header	Height	Height of current block
	DataHash	Hash of Data
	EvidenceDataHash	Hash of EvidenceData
	DkgDataHash	Hash of DkgData
	ProposerAddress	Proposer of this block
	LastGroupSign	BLS group signature of the previous height
	AppHash	Merkle root of state tree after executing previous block
LastCommit	Precommits []	2/3 + precommits of the previous height block
	BlockID	Merkle root of the header, each node can verify if this block links to the same previous block as its
Data	Txs []	Transactions in this block
EvidenceData	Evidences []	Verifiable evidences of malicious behaviors
DkgData	Commitments[]	t commitments of block proposer
	Approvals []	Each approval includes index of share sender (SrcID) and recipient (DestID), its signature

C. Approval table

Executing the DKG algorithm in an asynchronous network causes different nodes to have different views of complaints, e.g: node A receives a complaint from C against D, but node B does not.To solve this, we define every epoch as a statistic cycle of the approval table. Every block is allowed to contain some approvals (as shown in Table I), and each approval in the block corresponds to a unit in the approval table. For example, the approval (SrcID: 5, DestID: 3, Sig:...) means that the third key aggregator has received and verified the share from the fifth key generator. Thus unit of row 3 and column 5 is marked with "✓".

The signature in approval message is not the BLS signature, but is based on other asymmetric cryptography such as ECDSA. Every node has a public key to indicate its identity, and the share that is supposed to be sent to a recipient node needs to be encrypted using the public key of the recipient node to prevent other nodes from peeping at it. A node sends out an approval message after verifying a share from another node. If a node fails to verify a share, it refrains from sending out an approval message rather than make a complaint.

At the end of an epoch, if a unit in the approval table remains blank, this is equivalent to a complaint from the key aggregator (the row) against the key generator (the column).

Since the approval table is formed through blocks of an epoch, consensus ensures that every node has the same view of the approvals. The unit that does not have the approval is marked a "✖", i.e: there is a complaint in this unit. Table II shows two complaints from D and F against E. This can happen if E sent incorrect shares to D and F, or if the shares from E did not arrive at D and F in time.

If leave E as a qualified key generator, D and F cannot make signatures in the $n+2$ epoch because they do not get correct shares from E;

Else, choose deleting column E, there is no "✖" in the table, and the qualified key generators are A, B, C, D, and F. The threshold group still contains six players, and each aggregate shares from only the qualified key generators. Each player makes signatures using its own aggregated shares. Although E has been kicked out from the qualified key generators, it is still a valid threshold group member in the $n+2$ epoch.

$$\mathrm{A(index\,1)}:(s_{11},s_{21},s_{31},s_{41},s_{61})=(\mathrm{f}_1(1),\mathrm{f}_2(1),\mathrm{f}_3(1),\mathrm{f}_4(1),\mathrm{f}_6(1))$$
$$\mathrm{B(index\,2)}:(s_{12},s_{22},s_{32},s_{42},s_{62})=(\mathrm{f}_1(2),\mathrm{f}_2(2),\mathrm{f}_3(2),\mathrm{f}_4(2),\mathrm{f}_6(2))$$
$$\mathrm{C(index\,3)}:(s_{13},s_{23},s_{33},s_{43},s_{63})=(\mathrm{f}_1(3),\mathrm{f}_2(3),\mathrm{f}_3(3),\mathrm{f}_4(3),\mathrm{f}_6(3))$$
$$\mathrm{D(index\,4)}:(s_{14},s_{24},s_{34},s_{44},s_{64})=(\mathrm{f}_1(4),\mathrm{f}_2(4),\mathrm{f}_3(4),\mathrm{f}_4(4),\mathrm{f}_6(4))$$
$$\mathrm{E(index\,5)}:(s_{15},s_{25},s_{35},s_{45},s_{65})=(\mathrm{f}_1(5),\mathrm{f}_2(5),\mathrm{f}_3(5),\mathrm{f}_4(5),\mathrm{f}_6(5))$$
$$\mathrm{F(index\,6)}:(s_{16},s_{26},s_{36},s_{46},s_{66})=(\mathrm{f}_1(6),\mathrm{f}_2(6),\mathrm{f}_3(6),\mathrm{f}_4(6),\mathrm{f}_6(6))$$

For each "✖" unit, we have two choices: delete the column or delete the row, which are equivalent to deleting a key generator from the qualified set or a threshold group member in the $n+2$ epoch, respectively.

TABLE II. THE APPROBAL TABLE

Recv \ *Send*	*A*	*B*	*C*	*D*	*E*	*F*
A	✓	✓	✓	✓	✓	✓
B	✓	✓	✓	✓	✓	✓
C	✓	✓	✓	✓	✓	✓
D	✓	✓	✓	✓	✖	✓
E	✓	✓	✓	✓	✓	✓
F	✓	✓	✓	✓	✖	✓

V. IMPLEMENT VSS ON BLOCKCHAIN

A. Description of DKG protocol

The DKG protocol [9] is as follows: There are n parties $P_1,\dots,P_n$,

1) P_i generates a random polynomial:

$$f_i(x)=a_{i0}+\cdots+a_{i(t-1)}x^{t-1},\ (a_{i0},\dots,a_{i(t-1)})\in\mathbb{Z}_q^t.$$

2) P_i generates commitments for polynomial coefficients, $(A_{i0},A_{i1},\dots,A_{it-1})=(g^{a_{i0}},g^{a_{i1}},\dots,g^{a_{it-1}})$, then it broadcasts the commitments.

3) P_i computes shares $s_{ij}=f_i(j)\ mod\ q, j\in[1,n]$, and sends each share s_{ij} secretly to P_j.

4) Each key aggregator verifies:

$$g^{s_{ij}}=\prod_{k=0}^{t-1}(A_{ik})^{j^k}\ modq, i,j\in[1,n].$$

Define Q as set of nodes that are qualified key generators after Handle Complaints Algorithm (Section V.D). Then,

a) P_iThe group public key*:*

$$PK=\prod_{i\in Q}A_{i0}=\prod_{i\in Q}g^{a_{i0}}\ modq.$$

b) P_j*'*s aggregated secret key*:*

$$sk_j=\sum_{i\in Q}s_{ij}\ modq=\sum_{i\in Q}f_i(j)\ modq.$$

c)P_j*'*s public key*:* $pk_j=g^{\sum_{i\in Q}s_{ij}}=\prod_{i\in Q}g^{s_{ij}}$.

*d)*No party can compute the group's private key SK, but it is equal to $\sum_{i\in Q}a_{i0}\ mod\ q$.

5) When P_j's secret key has been aggregated, it can start generating the signature slice: $\sigma_j=\left(H_0(m)\right)^{sk_j}$.

6) Each party can recover a group signature by collecting t signature slices: $\sigma=\prod_{l_i\in S}{\sigma_{l_i}}^{a_{l_i}}$, $a_{l_i}=\prod_{l_i\in S,l_j\in l_i}l_i/(l_i-l_j)$or

receive a group signature directly and verify it by: $e(g_1, \sigma) = e(PK, H_0(m))$, where e is a non-degenerate, efficiently computable, bilinear pairing.

B. An issue in protocol Joint-Feldman

In Protocol Joint-Feldman [1], only if number of complaints is greater than *t* can we mark the key generator as disqualified. This implies that if the number of complaints is equal to or less than *t*, the key generator still qualifies. The question then arises concerning the nodes that complain against it. They do so because they do not receive the share, or receive an incorrect share, and thus cannot aggregate a correct private key. Thus letting the key generator qualify means letting the complainers out of the game. If we use this rule to deal with Table III, all key aggregators are out. To solve this, we can simply delete columns D, E, and F and get a 6*3 table, which means that we have six players in the $n + 2$ epoch and each aggregates three shares. Although we lose half of key aggregators, at least the game can continue.

TABLE III. A CASE WHERE ALL KEY GENERATORS HAVE BEEN COMPLAINED AGAINST

Recv \ Send	A	B	C	D	E	F
A						×
B						×
C					×	
D					×	
E				×		
F				×		

But if we change the rule to simply eliminate all key generators that have been complained against, it is problematic in some cases. See Table IV, E and F are two malicious nodes that are eventually left as the only qualified key generators; if they are controlled by a malicious party, the party can predict all random numbers because it controls all private keys.

TABLE IV. TWO MALICIOUS NODES COMPLAIN AGAINST ALL OTHER KEY GENERATORS

Recv \ Send	A	B	C	D	E	F
A						
B						
C						
D						
E	×	×	×	×		
F	×	×	×	×		

C. Simple way to deal with approval table

A simple method is to iterate the unit at the diagonal, to find the unit of which the row and the column contain the most "×" units, then delete the row and the column, keep doing this until there is no "×" unit in the whole table.

D. Security analysis of DKG algorithm

Though the simple way can solve the issue mentioned in Section V.B, if we analyze the protocol in the Byzantine environment, which means the adversary can corrupt all generators to obtain all coefficients, or control enough sibyl nodes to figure out all the coefficients of all the generators, the simple method is not secure enough, thus we state three conditions:

1) [Liveness] The number of remaining key aggregators should be equal or greater than *t*.

2) [Safety] All qualified key generators should not be controlled by a malicious party.

3) [Safety] There should not be *t* or more than *t* key aggregators controlled by a malicious party.

The reason for Condition 1) is clear, as otherwise, the group signature cannot be recovered. The Condition 2) and 3) are in response to collusion attack. However, it is hard to judge a participant is controlled by an adversary or not, but with premise of BFT, we can find a way to judge them

Premise of BFT Malicious stakes should be less than 1/3 of all the staked money.

Using this premise, we can deduce Conditions 2) and 3) to:

2) [Safety] Stake belonging to the remaining key generators should be no less than 1/3 of the total stakes

3) [Safety] There should not be *t* key aggregators have stakes less than 1/3 of the total stake

In this way, we could use concrete figure to measure, to eliminate the possibility that a malicious party control all the key generators or control more than *t* key aggregators.

E. Using the knapsack model to deal with approval table

To handle a "×" unit, deleting the column decreases the possibility that Condition 2) is met, while deleting the row decreases the possibility that Condition 1) is met.

Conditions 1) and 2) are analogous to conditions of volume and the value in the 0-1 knapsack problem. We consider that each column in the approval table is like an item in the knapsack model. Its value is the stake of the key generator of that column and its volume is the number of "×" that the column has. While the 0-1 knapsack problem is to find a solution whereby the knapsack is filled with columns with the highest total value, the knapsack problem in our model is to find a solution that reaches a target total value with the minimal total volume.

We call the method used to deal with the approval table the "Handle Complaints Algorithm." The goal is to eliminate all "×" units from the table. It is described in §A

The algorithm can obtain a solution if there is one, it might not be the most optimal one, that is, the minimum possible total volume of items in the knapsack that could satisfy a goal total value, for we use greedy approximation algorithm to obtain the result. That is not a problem, as long as the result is indeed a solution and every node run in the algorithm can get the same

result. If the algorithm ends with failure, all configurations of the threshold group members stay unchanged.

F. Commitment region

Handle Complaints Algorithm guarantees only Conditions 1) and 2), below, we show how to guarantee Condition 3).

If we simply let all staked nodes to participate in DKG process, the system will be vulnerable to sibyl-attack. Currently we set the barrier of becoming a staked node to that it should stake at least 1/1000 of the whole chain's coins. A malicious party can split his deposit into multiple accounts and stake on a huge amount of nodes, once he stakes t nodes, he controls t key aggregators, thus Condition 3) is violated. To reject those sibyl nodes. We can define a region of beginning blocks of each epoch, we call it the commitment region, where each proposer is allowed attach its commitments into the block, and the block proposer itself becomes a key generator and a key aggregator. Condition 3) can be satisfied by setting a commitment length n close to threshold t (see §B). In each epoch, When the i_{th} block proposer P_i in the commitment region is going to make a block, it generates a random polynomial function containing t coefficients, and generates t commitments for each coefficient, then it includes the commitments into the block. When its block has been committed by consensus, it computes shares $s_{ij} = f_{i(j)}$ mod q for player P_j, in which $j = 1, \dots, n$ and P_j is the j_{th} block proposer in the commitment region. It waits five heights to send the shares, as other nodes may not receive the commitment block immediately, and thus they cannot verify the shares.

There might be duplicate block proposers within the commitment region. In this case, we treat them as different share recipients but as the same key generator.

We treat different block proposers as the same key generator because they constitute one physical node, and there is no need for them to generate two random polynomial functions and let the aggregators aggregate them because this complicates the implementation to no benefit. We treat the block proposers as different share recipients because if we consider them as the same share recipient, the number of malicious nodes will not follow a binomial distribution.

For the shares need to be encrypted and sent to the recipient, we need another asymmetric encryption algorithm, such as ECC, to use the recipient's public key to encrypt the share, so that the nodes in the middle cannot peek into the data of share. Duplicate nodes that are logically different key aggregators are physically one node. Therefore, multiple shares sent to it are encrypted by the same public key of ECC of the recipient node. In the $n + 2$ epoch, this node acts as different threshold group members and sends multiple signature slices at each height.

The received shares of each epoch are stored in a data file "dkg-state.json", the file also stores the coefficients generated at the start of this epoch, and the private key from the aggregated shares of last epoch. This file is stored when a new share is received and verified, and at the start of each.

VI. Continuous Key Rotation

A. Continuous Share Exchange process

We define Share Exchange Process as that which includes proposing a commitment block, broadcasting shares, verifying shares and broadcasting approvals, building up an approval table and running the Handle Complaints Algorithm to obtain a new threshold group of epoch $n + 2$, i.e. the implementation of Step 1 to 4 in Section V.A on blockchain. We define the Signature Process as at each height every threshold group node generates a signature slice and every node tries to recover the group signature, i.e. the implementation of Step 5 to 6 in Section V.A. The two processes overlap in the time base. The Share Exchange process in epoch n involves preparing for the DKG signature process in epoch $n + 2$; because it determines the group's public key and private key, and the aggregated private key of each threshold group node of epoch $n + 2$.

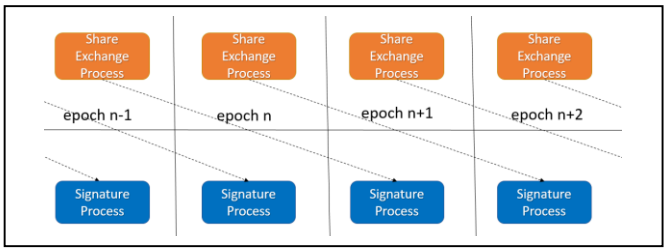

Fig. 3. The overlapping of the two prosesses

At the end of every epoch, each node runs Handle Complaints Algorithm to obtain a new threshold group of epoch $n + 2$. Each node in the group uses aggregated shares as its private key, computes the public key of every other threshold group node and the group public key based on the commitments of the qualified key generators. If the Handle Complaints Algorithm fails (it cannot eliminate all "×" units from the given approval table), the threshold group and the group public key in epoch $n + 2$ remain unchanged from those in the previous epoch $n + 1$. This is a minor case since if the network is good and all shares are received and verified, the approval table of each epoch should always be full filled.

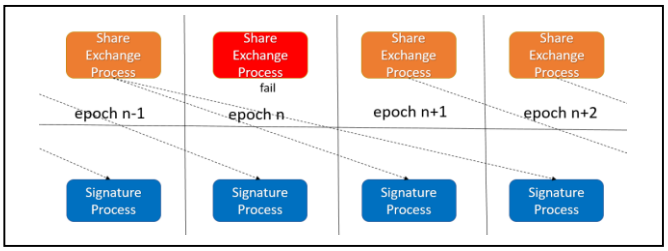

Fig. 4. Cases in which the Handle Complaints Algorithm of one epoch fails

The first Share Exchange process still relies on synchronous communication. It is executed when the chain starts, we assume that no complaint is generated because the initial nodes when we launch the chain should be ones that we deploy ourselves, they must be honest nodes. If a node has verified all shares sent to it, it broadcasts a FINISH message. If a node has gathered n FINISH messages, it finishes the first Share Exchange process and switches to the consensus process. We assume that all

shares are eventually verified by every node, if a network issue arises, we simply relaunch the chain.

B. Independent true random number flow

In Section I, we mentioned that we want to obtain an "independent true random number flow" that is irrelevant to the blocks data. But to implement distributed key rotation, we still rely on consensus to ensure that each participant has the same view of the approval table. Thus the participant can choose to send or not send an approval to affect the approval table, and affect the random number flow. Since it still cannot predict the random numbers, this is safe.

If the premise of $n \geq 3f + 1$ is violated, i.e. the chain is possible to be forked by malicious voters. Since random number flow is irrelevant to blocks data, it doevs not fork, it can determine proposer priority list and voters of later heights, thus we can rely on independent true random flow to solve the forks.

But the random number flow in our implementation is not fully independent to the blocks data, if the chain forks, the approval table forks as well, this will cause the random numbers in epoch $n + 2$ begin to fork. That is the reason that we set new threshold group take effect in epoch $n + 2$. The epoch $n + 1$ is supposed to detect the forks and choose one. To prevent long-range attack [16], the change of validators should not take effect in the next epoch, so does the change of threshold group members in our protocol.

The random number of each height cannot be used to select a subset of validators from the full set, because it undermines the premise of BFT (see §C).

VII. Experimental Data

Compare performance with Tendermint, the performance in our protocol should be slower, because of the extra network load of the shares and approvals in each epoch, and the signature slices at each height. Our contribution is not to boost performance but to obtain better security.

We formulated a system of twelve validator nodes using Tendermint with the continuous DKG process. The average time to commit an empty block is approximately two seconds, and each block carries 10,000 transactions that takes three to four seconds. We compared it with a Tendermint cluster without implementing DKG and obtained nearly identical latency. This is because Tendermint requires two steps of voting and collecting. Recovering a group signature needs only one step of sending signature slices and collecting them, and this step can be performed simultaneously or ahead of the prevote step of Tendermint. So group signature is often recovered earlier than *n-f* precommits are collected. The shares and approval messages broadcasted in network can occupy some bandwidth. But if they are placed in an epoch length of 200 blocks, the effect is acceptable.

What's more important of the experiment is to show the proposers priority and validators of each height are randomly selected, rather than followed by a predicted pseudo-random sequence. The experimental data in §F shows that the total counts of a node be selected as proposer in a range of blocks is proportional to its stake, as shown in §E.

VIII. Applications

Other than generating random numbers on blockchain, VSS protocol is also used to enhance the security of the wallet [18], and in electronic voting [20], and in the setup phase of the zkSNARKs protocols, such as Groth16 [13] and Plonk [14]. While Groth16 needs setup for each program, Plonk needs only one universal setup, in the premise that there is at least one honest participant that does not collude to deduce the group secret. The schema of continuous key rotation fits for the systems that consider the group secret in the initial setup phase could be leaked, and the group members can change, thus a new setup phase is needed because the new joined member does not trust the public parameters generated by the old ones.

IX. Conclusion

We propose a modified DKG protocol on blockchain based on the consensus algorithm Tendermint.

The enhancement to Dfinity is: continuously execute the exchange of the shares in every epoch, rather than only on the start of the chain. Threshold group members are selected from all staked nodes, thus they are updated if staked nodes change. Even if the staked nodes don't change, the coefficients of the polynomials of each key generator are regenerated each epoch, thus the group public key and the aggregated private key of each threshold group member of the DKG protocol are rotated in each epoch. It is difficult for the attacker to continuously learn the updated polynomials of each participant in each epoch, the security of the distributed system is then enhanced.

The contribution to DKG protocol is: we present a way to execute it on blockchain in asynchronous and Byzantine environment, we consider the participants may collude to compute the group private key and take measures to prevent it.

References

[1] Rosario Gennaro, Stanis law Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. Journal of Cryptology 20(1):51-83.

[2] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. ACM, 2017, pp. 51-68. https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf

[3] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. arXiv preprint arXiv:1807.04938, 2018. https://arxiv.org/abs/1807.04938.

[4] Terdermint Organization. Application Architecture Guide. https://tendermint.com/docs/app-dev/app-architecture.html

[5] Vlad Zamfir. Casper the friendly ghost: A "correct-by-construction" blockchain consensus protocol, 2017. arXiv:1710.09437. https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf

[6] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY Technology Overview Series Consensus System. CoRR abs/1805.04548 (2018).https://dfinity.org/static/ dfinity-consensus0325c35128c72b42df7dd30c22c41208.pdf

[7] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Annual International Cryptology Conference. Springer, 2017, pp. 357-388.

[8] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. In Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in

Cryptology, ASIACRYPT '01, pages 514–532, London, UK, UK, 2001. Springer-Verlag.

[9] B. Libert, M. Joye, and M. Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. Theoretical Computer Science, 645:1–24, 2016.

[10] Dan Boneh, Manu Drijvers, Gregory Neven. BLS Multi-Signatures With Public-Key Aggregation. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html.

[11] Alistair Stewart. Grandpa Byzantine Finality Gadgets. https://github.com/w3f/consensus/blob/master/pdf/grandpa.pdf

[12] Pedersen T.P. (1992) Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In: Feigenbaum J. (eds) Advances in Cryptology — CRYPTO '91. CRYPTO 1991. Lecture Notes in Computer Science, vol 576. Springer, Berlin, Heidelberg https://www.cs.cornell.edu/courses/cs754/2001fa/129.PDF

[13] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: Proceedings of the 35th Annual International Conference on Theory and Applications of Cryptographic Techniques. EUROCRYPT '16. 2016, pp. 305–326. .

[14] Kate A., Zaverucha G.M., Goldberg I. (2010) Constant-Size Commitments to Polynomials and Their Applications. In: Abe M. (eds) Advances in Cryptology - ASIACRYPT 2010. ASIACRYPT 2010. Lecture Notes in Computer Science, vol 6477. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-17373-8_11

[15] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 31–42. DOI:https://doi.org/10.1145/2976749.2978399

[16] Vitalik Buterin. Long-range attacks: The serious problem with adaptive proof of work. https://blog.ethereum.org/2014/05/15/long-range-attacks-the-serious-problem-with-adaptive-proof-of-work/, 2014.

[17] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

[18] R. Gennaro, S. Goldfeder, and A. Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security. In International Conference on Applied Cryptography and Network Security, pages 156–174. Springer, 2016.

[19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings". In: Proceedings of the 26th ACM Conference on Computer and Communications Security. CCS '19. 2019..

[20] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael J. Wiener, editor, Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, volume 1666 of Lecture Notes in Computer Science, pages 148–164. Springer, 1999.

APPENDIX

A. HANHLE COMPLAINTS ALGORITHM

Input: An $n * m$ approval table with threshold t.

Output: A $j * k$ sub-table with no "×" unit, where $j \geq t$ and the total stake of the k columns should surpass $1/3$ of the total stake.

We perform the following steps to find the solution:

Step 1. Put the columns with zero "×" into the knapsack. Now the volume of the knapsack is still 0, but it has a total value.

Step 2. Check if the total value meets Condition 2, that is, the total value surpass 1/3 of all stakes. If it does: The algorithm ends successfully. The solution is a *n*k* table, which has *k* columns that have no complaint. Otherwise: Delete the columns with more than *t* "×" unis, then goto **Step 3**.

Step 3. For each column c := range the remaining columns {

We define:

Additive columns of c:= the other columns that have no "×" due to the deletion of the "×" rows of c (e.g., in.Table V, column F's additive columns are B, D, and J)

An item is an abstract concept; it is the column c itself and its additive columns. We call c the main column of this item.

The item's value is the sum of stakes of the column c itself and its additive columns

The item's volume is the number of "×"units in column c

The item's ratio := its value/its volume

}

Note that although different items have different main columns, their additive columns may overlap, (e.g, The items in Table V are {B}, {D}, {F,{B,D,J}}, {H,{B,D}},and {J})

Sort items by their ratios. If the ratios of two items are equal, sort them by the address of the node of the main column.

Put the top item into the knapsack, which means placing the columns of this item into the knapsack and deleting rows with "×"of the main column of this item

Step 4. Check if the knapsack meets Condition 1 and 2, that is:

If the total volume of knapsack has exceeded the limit, goto **Step 5**

Otherwise:

If the total value in the knapsack has reached the target value, the algorithm ends with success, return: the columns in the knapsack as the qualified set Q and the remaining rows as the threshold group members.

Otherwise: Modify the volumes and values of the remaining items (because after the deletion of some rows, the volumes of the remaining columns may decrease), then repeat Step 3 and Step 4.

(Step 3 and 4 are akin to depth-first search in a binary tree. When it reaches the leaf of a path and has not found a solution, it needs to go backward and try other paths. Every node is a choice between placing a column into knapsack and not)

Steps 5. This step is the backtracking step in the recursive algorithm. It sends the search back to higher tree nodes. It tries not to put the item into the knapsack and tries the next item in the sorted list, then goto **Step 4** to check the result.

Keep the depth-first search until it finds a solution or ends the search without a solution, which means that the algorithm fails**.**

An example of executing the algorithm is given below:

TABLE V. AN INPUT APPROVAL TABLE

Slots	*3*	*1*	*2*	*2*	*1*	*1*	*1*	*7*	*18*	*4*
Send / Recv	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A								×	×	
B								×	×	
C				×		×		×	×	
D								×	×	
E		×				×		×	×	
F									×	
G						×				×
H										
I										
J										

Total slots=40, target value=40 ×1/3=12, t=5

Table VI is the execution process that handles the approval table in Table V

TABLE VI. AN EXAMPLE OF EXECUTING HANDLE COMPLAINTS ALGORITHM

Step	Items in knapsack	Sorted remaining items by their ratios	Total Value	Total Volume	Delete d Rows
1	A,C,E,G	J: $4/1 = 4$ F: $(1+1+2+4)/3 = 8/3$ H: $(7+1+2)/5 = 2$ D: $2/1 = 2$ B: $1/1 = 1$	7	0	
2	$7 \geq 12$? N. Goto Step 3. Put J to knapsack				
3	A,C,E,G,J	H: $(7+1+2+1)/5 = 2.2$ F: $(1+1+2)/2 = 2$ D: $2/1 = 2$ B: $1/1 = 1$	11	1	G
4	$11 \geq 12$? N ; $1 > 5$? N; Goto Step 3. Put H and B,D,F into knapsack				
3	A,B,C,D,E, F,G,H,J		22	6	A,B,C, D,E,G
4	$6 > 5$? Y; Goto Step 5. Revert last putting, change to put F and B,D into napsack				
5	A,B,C,D,E, F,G,J		15	3	C,E,G
The algorithm ends successfully. The remaining key generators are A, B, C, D, E, F, G, J. The remaining key aggregators are A, B, D, F, H, I, J.					

B. CHOOSING AN APPROPRIATE COMMITMENT LENGTH

Let n be the length of the commitment region, in which each block proposer has a lower than $1/3$ probability to be a malicious one. We use p to represent the probability, and m to represent the number of malicious key aggregators in the region. Then, m follows a binomial distribution $B(n,p)$, and the cumulative distribution function $CDF(t,n,p)$ stands for the probability of $m < t$. Given an acceptable failure probability ρ and threshold t, we can deduce the maximum length of the region n such that CDF$(t,n,p)<$ ρ:

TABLE VII. MAXIMUM LENGTH OF COMMITMENT REGION FOR A THRESHOLD t

$-\lg\rho$	p	n	t
10	1/5	47	30
12	1/4	36	30
12	1/4	56	40
12	1/3	47	40

If we set $t = 40$, we can choose n from [47,56).

C. ANALYSIS OF MINIMAL NUMBER OF SELECTIONS TO GET A QUALIFIED SUBSET

To use a random number to select a subset of validators for consensus, we need to ensure that the subset also satisfies the premise of BFT: Less than $1/3$ of the subset's voting power is malicious. A validator's voting power is equal to the number of times it has been selected. To ensure this in terms of probability, the analysis is similar to that in (§B). Let n be the total number of selections. Each selection has a lower than $1/3$ probability to obtain a malicious validator. We use p to represent the probability and m to represent the total voting power of the malicious validators in the subset. Then, m follows a binomial distribution $B(n,p)$ and the cumulative distribution function $CDF(n/3,n,p)$ stands for the probability of $m < n/3$. Given an acceptable failure probability ρ, we can deduce the minimal number of selections n such that $CDF(n/3,n,p) < \rho$:

TABLE VIII. MINIMAL NUMBER OF SELECTIONS TO GET A QUALIFIED SUBSET BASED ON DIFFERENT PROBABILITIES TO GET MALICIOUS NONE IN A SELECTION

$-lg\rho$	p	n
12	1/5	600
10	1/4	1200
12	1/4	2000

If there are u staked accounts, and their stakes are evenly distributed, the probability that an account is not selected in n selections is $C_u^1 * \left(u - 1/u\right)^n$. If $u = 100$ and $n = 2000$, this is 1.86×10^{-7}.

It thus turns out that if u is far smaller than n, the subset is often equal to the full set, the scale of the nodes doing consensus is not reduced, but voting powers are randomly assigned to them.

D. EXAMPLES SHOWING THE PSEUDO-RANDOM SEQUENCE OF THE PROPOSER IN TENDERMINT

TABLE IX. AN EXAMPLE OF 4-NODE PRIORITY INCREMENT IN TENDERMINT

	A: Power 20	**B:Power 18**	**C:Power 12**	**D:Power 10**
Initial Priority	20	18	12	10
1st proposer is A (max is 20)	20-60=-40	18	12	10
All increase	-40+20=-20	18+18=36	12+12=24	10+10=20
2nd proposer is B (max is 36)	-20	36-60=-24	24	20
All increase	-20+20=0	-24+18=-6	24+12=36	20+10=30
3rd proposer is C (max is 36)	0	-6	36-60=-24	30

TABLE X. AN EXAMPLE OF 4-NODE PRIORITY INCREMENT IN TENDERMINT

	A: Power 99	**B: Power 1**
Initial Priority	99	1
1st proposer is A (max is 99)	99- 100=-1	1
All increase	-1+99=98	1+1=2
2nd proposer is A(max is 98)	98-100=-2	2
All increase	-2+99=97	2+1=3
3rd proposer is A(max is 97)	97-100=-3	3

E. EXPERIMENTAL DATA

TABLE XI. EXPERIMENTAL DATA

Round / slots	1	2	3	4
7	205	201	215	203
7	211	202	209	207
14	419	403	417	412
14	421	410	423	411
22	615	624	611	621
22	627	611	623	617
29	832	822	831	828
29	839	838	835	829
44	1251	1260	1255	1257
44	1255	1259	1254	1241
58	1660	1698	1658	1703
58	1665	1672	1669	1671

We make the stakes of the twelve nodes 1, 1, 2, 2, 3, 3, 4, 4, 6, 6, 8, 8×10^{10}, and their slots are equal to their stakes divide 1/1000 of whole money of the system, as there are other initial accounts, so their slots in the stake table are 7, 7, 14, 14, 22, 22, 29, 29, 44, 44, 58, 58. We do four rounds of tests, each round is 10’000 blocks, and see the probability of each node be selected as proposer is more or less proportional to their slots.