5ire Yellowpaper

1 Nested-Chains

Nested-Chains 5irechain addresses the issue of scalability by maintaining parallel chains. These chains are created on the need basis depending on the load on the network. However, once a chain is created it will continue adding blocks, until the chain is joined with the 5irechain using a 5ire block. Figure 1 shows the structure of the nested-chain. The nested chains not only support the scalability in the blockchain, but it also enables us to support creation of parallel chains without adding new nodes (Assemblers, Attesters/voters, ESG Experts). This essentially means that nodes will be running multiple parallel chains on a single node, but as a separate process. 5ire will use the scheduling algorithms to make sure the maximum utilisation of the nodes. Therefore, unlike the conventional blockchains where nodes will sit idle and wait for their turn to create the block, the nodes in the 5ire ecosystem may get turns to create blocks into another parallel chain in the nested-chain. The nodes in all the parallel chains will be selected in a similar way i.e. based on their weights (Reliability Score, Stake, ESG score and Randomized voting).

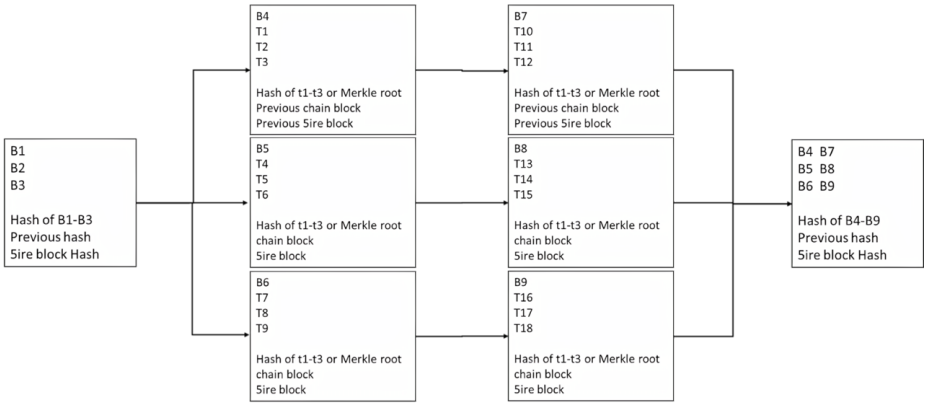

Figure 1: Nested chains

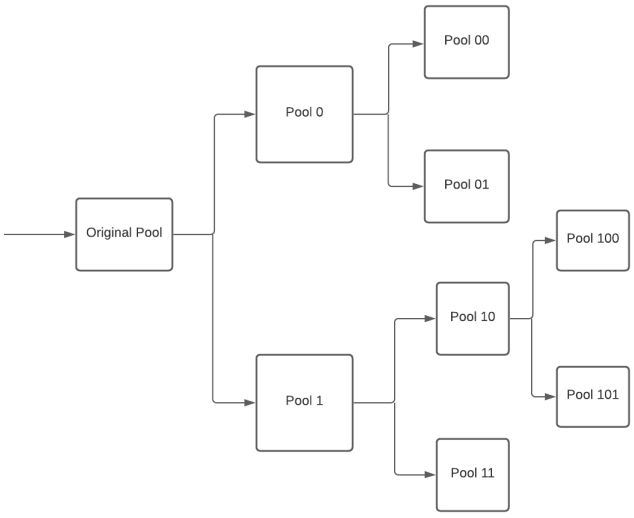

Figure 2: Multiple transaction pools for nested chains.

2 Transaction Pools

5irechain addresses the issue of scalability by maintaining parallel chains. In order to support this, 5irechain allows multiple transaction pools, one for each parallel chain. Whenever the number of transactions in a transaction pool surpasses a threshold, that particular transaction pool is divided into two different pools. For this purpose, we make use of hash functions. A transaction goes into one of the pools depending upon the hash value of the public key of the transaction-sender. Roughly speaking, if there are n transaction pools, each pool is dynamically assigned a number which is a bit-string of size at most $\log n$. Figure 2 shows how transaction pools are spit. The original pool is divided into two pools: pool 0 and pool 1. When pool 0 is split, two distinct pools are generated, pool 00 and pool 01. When pool 1 is split, two pools, namely pool 10 and pool 11 are generated, and so on. Now, let us assume that a new transaction T has been sent by a sender in the 5irechain ecosystem. The public key of the sender of T is denoted as Pk_T. In order to decide which pool T should be include into, we need to first compute the hash of Pk_T using a well-known hash function such as SHA-256. Assume that $H = SHA\text{-}256(Pk_T)$. If the id. of a pool matches the first few bits of H, then T should go into that pool. For example, if the first three bits of H is 101, then T will go into pool 101. Similarly, if H starts with 01, then T will go into pool 01. It is easy to see that any transaction can go into a unique pool depending upon the first few bits the hash value of the public key of the sender.

Joiner nodes merge two parallel chains by creating a block out of two blocks, each one belonging to a different chain. Thus, joiner nodes merge two different

parachains. They do so when the transaction pools corresponding to the two pools drop, making it unnecessary to maintain two different chains. Joiner nodes do not merge chains randomly. They only merge two chains that correspond to a pair of transaction pools whose ids differ in the least significant bit. In other words, the joiner nodes merge two transaction pools that have come into existence due to the splitting of a particular transaction pool. Let us explain it with an example. There are two transaction pools in Figure 2, namely pool 100 and pool 101. They were formed by dividing the pool 10. So, if the chains corresponding to the pool 100 and 101 are merged by the joiner nodes, it will yield a chain that will correspond to the pool 10. Joiner nodes do not merge chains that correspond to two transaction pools that spun out of different pools.

3 Block Verification

Assume there are n nodes who want to participate in the consensus protocol in order to become block assemblers. These nodes are denoted as $P_1, P_2, \cdots, P_n$. At the time of locking the stakes, each participating node selects an $x_i \xleftarrow{\$} \mathcal{Z}_p$, and computes $X_i = SHA\text{-}256(x_i)$. Each node P_i publishes X_i along with other transactions that enables participation in the race. After a certain time, the set of winning block assemblers are determined depending upon their total weights. Let S be the set of indices of the winning block assemblers, where $|S| = \eta$. Each winning block-assembler $P_i : i \in S$ then unlocks X_i by presenting the pre-image x_i. Let us assume that $\alpha = HASH(T)$, where $T = \bigcup_{i \in S} x_i$. We use α to distribute the task of block-assembly among the winning block-assemblers. All blocks produced in an epoch have to be attested by the attester nodes which are the same as the block assemblers. While one block assembler produces a block, all other assemblers perform as attester nodes. Each block needs to be attested by at least t attester nodes out of the $\eta - 1$ nodes. If at least t of them verify the block, then it will be incorporated into the 5irechain.

4 Scheduling Block Assembly

In the consensus mechanism of 5irechain ecosystem, block assemblers are selected on the basis of weights computed using various factors. A set of nodes offer to become block-assemblers at an epoch by locking their stakes. Each of these nodes select a random number h_i, and provides the hash H_i of h_i as part of its transactions. The consensus mechanism selects a set of n nodes who will share the right of block assembly in that epoch. Let τ be the hash of the blocks generated exactly half an hour ago. There are slots in an epoch. Let us assume that there are n nodes $U_1, U_2, \cdots, U_n$ who are the block assemblers during the current epoch. The weights associated with node U_i is given by w_i, for $i \in [1, n]$. These are the top n nodes in terms of weight. The epoch starts at time $t = t_0$. Each node $U_i : i \in [1, n]$ publishes h_i well before time $t = t_0$. Once all h_i's are available, everyone calculates $\alpha = SHA\text{-}256(h_1, h_2, \cdots, h_n)$. In our 5irechain

ecosystem, there can be multiple concurrent chains running in parallel. The number of parallel blockchains depends upon the number of incoming transactions and cannot be predicted a priori. We discuss how the 5irechain ecosystem assigns one assembler the task of assembling a particular block. We define two arrays Stake, and L, each of size n. We define Stake as the array of weights of assembler nodes. Again, for each $i \in [1, n]$, $L[i]$ stores the number of blocks assembled by U_i since time $t = t_0$ as they are updated dynamically. In other words, when the algorithm assigns the blocks to the assemblers for time $t = t_c$, $L[i]$ should hold the number of blocks assembled by U_i between time $[t_0, t_{c-1}]$. L is initialised to 0 in the beginning. The allocation algorithm takes the array L as an input and also updates the same. Based on this, we distribute the task of block assembly among the nodes in time $t = t_c$.

Let us assume that there are l parallel chains at time $t = t_c$. That is to say that, we need to find l block assemblers for creating l parallel blocks in time $t = t_c$. The Algorithm 1 shows how the ecosystem allocates blocks to assemblers in time $t = t_c$. In Algorithm 1, the function RAND-FRAC takes a seed , and generates random fractional numbers in the range [0,1].

The Algorithm 1 allocates blocks to assemblers in proportion to their weights, while at the same time ensuring that the same assembler does not get to assemble two blocks at the same time from two parallel chains. This is done to optimise the CPU usage of block assemblers.

We have shown in Figure 3 how Algorithm 1 distributes slots to 13 block assemblers. Here, we have chosen to have 15 time-slots in an epoch. We have also taken into consideration The plausibility of existence of multiple concurrent chains. In this example we have randomly varied the number of parallel chains between 1 and 5. Note that in real life, the number of parallel chains would not vary randomly, rather they would follow the rules of the 5irechain consensus mechanism. However, here the number of parallel blocks are randomised to exhibit the property of Algorithm 1 that it can distribute slots to the assemblers correctly in all circumstances. In Figure 3, the columns represent time-slots and the rows represent parallel chains. The algorithm distributes time-slots to block assemblers in a way that no assembler gets to assemble more than one block from more than one chain at the same time-slot.

5 Homomorphic Encryption

Homomorphic encryption scheme is a type of encryption scheme that allows users to perform computations on an encrypted data without decrypting it. The properties of a homomorphic encryption scheme ensures that if a computation is performed on an encrypted data, it results in a ciphertext which would be equivalent to the ciphertext of a plaintext obtained through performing the same computation on the unencrypted data. Moreover, under a homomorphic encryption scheme, the secret decryption key is not required to do any processing on a ciphertext, rather all the computations can be done using the public key. Homomorphic encryptions are of two types: partially homomorphic encryption

Algorithm 1 An algorithm for slot allocation

Require: $Stake, L, \alpha$
Ensure: $X_c^J : J \in [1, l], L$
  $\phi = \emptyset$
  for $i = 1 \to l$ **do**
    $S = [1, n] \setminus \phi$
    $tmp1 = \sum_{r \in S} Stake[r]$
    $tmp2 = \sum_{r \in S} L[r]$
    for $r \in S$ **do**
      $B[r] = Stake[r]/tmp1$
      $Y[r] = \max(0, B[r] - L[r]/tmp2)$
    end for
    $\pi =$ **RAND-FRAC**(α, i)
    for $j = 1 \to n$ **do**
      if $j \in \phi$ **then**
        Continue
        $S_0 = [1, j] \setminus \phi$
        $\mu_j = \frac{\sum_{k \in S_0} Y[k]}{\sum_{k \in S} Y[k]}$
        if $\mu_j \geq \pi$ **then**
          break
        end if
      end if
    end for
    $\phi = \phi \cup \{j\}$
    $X_c^i = j$
    $L[j] = L[j] + 1$
  end for

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	5	10	13	11	12	6	10	12	11	6	1	4	9	5
2	-	-	8	-	2	6	1	5	10	-	9	11	10	6	2
3	-	-	9	-	3	11	13	13	3	-	12	13	3	8	12
4	-	-	7	-	12	-	9	4	2	-	5	7	13	11	7
5	-	-	-	-	-	-	-	-	8	-	-	-	-	-	10

Figure 3: Slot Allocation

and fully homomorphic encryption scheme. Partially homomorphic encryption schemes allow some but not all operations on encrypted data. For example, El Gamal encryption scheme only allows addition but not multiplication. A fully homomorphic encryption scheme allows both addition and multiplication operations to be performed on encrypted data. Let us assume that E is a fully homomorphic encryption scheme. E_{pk} denotes public key encryption under the public key pk. As such the following operations hold under E.

$$E_{pk}(x_0) \odot E_{pk}(x_1) = E_{pk}(x_0 \oplus x_1)$$

$$E_{pk} \otimes E_{pk}(x_1) = E_{pk}(x_0 \circ x_1)$$

Homomorphic encryption allows privacy preserving computation on private data. Our 5irechain ecosystem will enable users to store their private data in IPFS nodes. Let us assume that there is an organization A that possesses private information about all clients registered with it. All the private information about clients are stored in encrypted form in IPFS nodes. Let us denote by m_i, the information about a client u_i stored in the ecosystem. Also assume that there are δ clients. This information can be supplied to a third party to perform some processing on the data. Let us suppose that the public and the private key of the authority A is (sk, pk).The IPFS nodes hold $\Gamma_i = E_{pk}(m_i)$, for all $i \in [1, \delta]$. Now A wants a third party P to securely perform an algorithm on the inputs of all clients in the set $[1, \delta]$. That is to say, A wants to learn the value of $F = \xi(m_1, m_2, \cdots, m_\delta)$, without letting P learn the values of m_i, for any $i \in [1, \delta]$.This is done as follows. P runs the algorithm ξ on the encrypted ciphertexts Γ_i, for $i \in [1, \delta]$. In other words, P computes $C = \xi(\Gamma_1, \Gamma_2, \cdots, \Gamma_\delta)$. Due to the fully homomorphic property of the encryption scheme, C will be same as $E_{pk}(\xi(m_1, m_2, \cdots, m_\delta)) = E_{pk}(F)$. Now, A can decrypt it to get the value of F.

6 Practical Aspects of Homomorphic Encryption

When combined with blockchain, these features of homomorphic encryption could usher in a new era of agile and yet highly resilient secure computing strategies. Though conceptualized in 1978, the first fully homomorphic encryption scheme, in short, FHE, was becoming a reality following the phenomenal success of the research work done by Gentry et al. in 2009 titled "A fully homomorphic encryption scheme". The goal of homomorphic encryption is to allow an infinite number of additions or multiplications of encrypted data with the target of having the ciphertext that would be produced, had the same same operations been performed on the corresponding plaintexts. The problem is that designing such an encryption algorithm is really hard. As a result, there are a few different "types" of homomorphic encryption that describe how close a particular algorithm is to that final goal. Despite being an ideal solution for solving a variety of major business challenges, there is no commercial use of FHE so far. One major problem with FHE is that it is not efficient. Depending on the degree of freedom in terms of achieving full functionality over the cipher text domain, there are three categories of homomorphic encryption; Fully homomorphic encryption, Somewhat homomorphic encryption and partial homomorphic encryption.

7 Decentralized Identity Verification

In our 5ire ecosystem, we propose to deploy a decentralized mechanism to verify the identity of a person. Any person running a node can associate her identity with the node. This is done by creating a transaction. The content of these transactions is the encrypted information about his identity, like social security number, passport number etc. This information is encrypted and stored in IPFS nodes and the hash of the ciphertext is included in a transaction which becomes part of the blockchain. If later the person needs to prove her identity to the authority, then she can provide it to the authority with a non-interactive zero knowledge proof of the fact that the encrypted information stored in the IPFS node is actually a correct encryption of her identity under her public key. Let us consider that $m = (m_0, m_1, \cdots, m_{k-1})$ is the identity information of a person. The person has a private-public key pair (sk, pk). The encryption of m under pk is denoted as $T = E_{pk}(m_i) : i \in [k-1]$. T is stored in an IPFS node, and $Hash(T)$ is stored in the blockchain. The person can prove her identity by providing m, and non-interactive proofs $\pi = (\pi_0, \pi_1, \cdots, \pi_{k-1})$, where $\pi_i = ZKP[sk : m_i, pk, E_{pk}(m_i)]$, for $i \in [k-1]$.

8 Associating a mobile number with a wallet

Our 5irechain ecosystem supports the association of mobile numbers with wallets. A 5irechain user can send 5ire tokens to her peers using the latter's mobile number as an address. A user U can do this by creating a transaction T. The transaction will specify the mobile number to be included in the wallet. Let G be a finite group of prime order q in which the Decisional Diffie-Hellman problem is intractable. Also assume that g is a random generator of G. The miner selects random $x \xleftarrow{\$} \mathcal{Z}_q$, and includes $b = g^x$ in the block that also contains the transaction T. The miner sends x to the mobile number of the user U. U generates another transaction S that includes a non-interactive zero knowledge(NIZK) proof of knowledge of x given b. If the NIZK proof is correct, another miner will include S in a block. This will finalize the registration of the mobile number of the user. Now, anyone can send coins to the wallet of U, just by using her mobile number, instead of her wallet address. We discuss below how the NIZK proof can be constructed.

1. User U chooses $r \xleftarrow{\$} \mathcal{Z}_q$, and computes a commitment $com = g^r$.
2. U generates the challenge $ch = Hash(g, b, com, pk)$, where pk is the public key of U.
3. U generates the response $res = r - ch * x$.

U includes com, ch and res into S. Anyone can check the validity of the NIZK proof by checking if the following equation holds or not.

$$com = g^{res} * b^{ch}$$

If the user U wants to add more numbers to her wallet, she has to perform the same operations every time.

9 Encrypting the amount in a transaction

In 5irechain network, users hide the monetary value of a transaction by means of encryption. Whenever a user creates a new transaction that sends some tokens to another user, it encrypts the amount to be sent to the receiver using the receiver's public key. The user also provides non-interactive zero knowledge proof of the fact that the user's remaining balance is equal to her previous balance minus the amount of tokens she sent to the receiver. The users use El-Gamal encryption scheme for this purpose. Let us suppose that G_q is a group of order q. Also assume that g is a random generator of G_q. The public key of user U_1 is given by Pk_1, and the public key of user U_2 is given by Pk_2. U_1 has some k amount of 5ire tokens encrypted under her public key Pk_1 as $E_{Pk_1}(k) = (e_1, e_2) = (g^{k+r}, Pk_1^r)$, where $r \xleftarrow{\$} \mathcal{Z}_q$. If U_1 sends tokens to U_2, then it creates a transaction. The transaction includes the encryption of under

the public key Pk_2 of U_2. This ciphertext is denoted as $E_{Pk_2}(\alpha) = (e_3, e_4) = (g^{\alpha+s}, Pk_2^s)$, where $s \xleftarrow{\$} \mathcal{Z}_q$. Let us assume that $\beta = k - \alpha$. U_1 also encrypts β as $E(\beta) = (e5, e6) = (g^{\beta+t}, Pk_1^t)$. We show below how U_1 can generate a NIZK proof of well-formedness of $E(\beta)$.

Let us assume that $e_7 = e_1/e_5$, and $e_8 = e_2/e_6$.

U_1 chooses random $\mu_1 \xleftarrow{\$} \mathcal{Z}_q$, and computes a commitment $com_1 = Pk_1^{\mu_1}$. Let us assume that $ch_1 = Hash(com_1, Pk_1, e_2)$. U_1 generates a response $res_1 = \mu_1 - ch_1 * r$.

U_1 chooses random $\mu_2 \xleftarrow{\$} \mathcal{Z}_q$, and computes a commitment $com_2 = Pk_2^{\mu_2}$. Let us assume that $ch_2 = Hash(com_2, Pk_2, e_4)$. U_1 generates a response $res_2 = \mu_2 - ch_2 * s$.

U_1 chooses random $\mu_3 \xleftarrow{\$} \mathcal{Z}_q$, and computes a commitment $com_3 = Pk_1^{\mu_3}$. Let us assume that $ch_3 = Hash(com_3, Pk_1, e_6)$. U_1 generates a response $res_3 = \mu_3 - ch_3 * t$.

U_1 chooses random $\mu_4, \mu_5 \xleftarrow{\$} \mathcal{Z}_q$, and computes a commitment $com_{41} = Pk_2^{\mu_4}, com_{42} = Pk_1^{\mu_5}$, and $com_{43} = g^{\mu_4 - \mu_5}$. Let us assume that

$$ch = Hash(com_{41}, com_{42}, com_{43}, Pk_1, Pk_2, e_7, e_8, e_3, e_4)$$

U_1 computes a response $res_{41} = \mu_4 - ch * s$, and $res_{42} = \mu_5 - ch * b$, where $b = r - t$.

The verification equations are as below.

1. $Pk_1^{res_1} = com_1/e_2^{ch_1}$.
2. $Pk_2^{res_2} = com_2/e_4^{ch_2}$.
3. $Pk_1^{res_3} = com_3/e_6^{ch_3}$.
4. $Pk_2^{res_4} = com_{41}/e_4^{ch}$.
5. $Pk_1^{res_{42}} = com_{42}/e_8^{ch}$.
6. $g^{res_{41} - res_{42}} = com_{43}/(e3/e7)^{ch}$.

10 Reward Mechanism

In the 5ire network, ESG scores are taken into account while selecting validators. ESG scores contribute 20% of the total score of the validators. The calculation of ESG scores involve the node providing a report in support of her compliance with sustainability policies. This report is reviewed by a group of nodes. They also provide a rating to the ESG node that is on the scale of $[1, 10]$. Finally the ratings from different nodes are averaged out, and an overall score is computed. This ESG score is used to calculate the weight of the node for becoming a block validator. Let us assume that there are n different nodes that provide rating for a particular ESG node E. The rating providers are denoted as $P_1, P_2, \cdots, P_n$.

The rating provided by P_i is given by v_i, for all $i \in [1, n]$. Hence, $1 \leq v_i \leq 10$. The overall ESG score of E is given by

$$S_E = \bar{v} = \frac{\sum_{i=1}^{n} v_i}{n}$$

Also assume that σ is the standard deviation of the ratings. That is

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \bar{v})^2}$$

Every rating provider receives some reward for participating in the process. Let the total amount of reward to be disbursed to n participants be M. Also assume that $f_i = |\{P_j : v_j = i\}|, \forall i \in [1, 10]$. The reward won by a participant depends upon the numeric value of the rating v_i. We say that the reward obtained by a participant P_j is m_i, if $v_j = i$. Therefore,

$$\sum_{i=1}^{n} f_i * m_i = M \tag{1}$$

For each $i \in [1, 10]$, we define m_i as

$$m_i = \frac{K}{\sqrt{c + \frac{v_i - \bar{v}}{\sigma}}}$$

Here, both K, and c are constants. We choose $c = 0.1$. The value of K can be found by solving equation 1.

11 Joiner Nodes

In 5ire network, nested chains are used to provide high throughput when the influx of transaction is very high. Nested chains work as parallel chains that emerge from one chain, and progress simultaneously. There are some special actors called joiner nodes that take care of creating and finalizing these nested chains. When the number of transactions in a transaction pool increases beyond a threshold, the joiner nodes send a "split" transaction to the network. This split transaction specifies a chain number. A block assembler includes that transaction in a block. Once it is included, the chain splits into two different chains that progress independently to each other. Once the number of transactions in the two pools drops then the joiner nodes send a "join" transaction aimed at the two different chains. If the transaction is included in both the chains, block assemblers stop mining on either chain until the joiner nodes create a joiner block that merges the two chains into one. Once the two chains get merged by the joiner blocks, mining can resume on the merged chains as usual. The joiner nodes are actually same as the block assemblers. The block assemblers need to send a "split" or a "join" transaction jointly. In order for a "split" or

a “join” transaction to be accepted, the total weight of the joiner nodes should be at least 50% of the total weight of all the joiner nodes in that epoch. When a “split” or a “join” transaction is created, all the joiner nodes who back the transaction provide their signature on the transaction itself. A block assembler needs to check that the total weight of the signers of the transaction is at least 50% of the total weight of all the joiner nodes in that epoch.