Bitkub Chain (BKC) Technical Paper

(V 2.1)

Updated 22 November 2022

Table of contents

Table of contents **1**

1. Introduction **1**

2. Blockchain paradigm **3**

2.1) Native currency units 4

2.2) Bitkub Chain history 4

3. States & Tries **6**

3.1) Merkle Patricia Trie 6

3.1.1) Extension node 6

3.1.2) Branch node 6

3.1.3) Leaf node 6

3.2) States 7

3.2.1) State Root 7

3.2.2) World State 7

3.2.3) Transaction State 9

a) Transaction state components 9

b) Transaction Trie 9

3.2.4) Transaction Receipt 9

a) Transaction Receipt Trie 9

4. GAS **10**

4.1) Gas Limit and Gas Price 10

4.2 Mechanism of gas calculation 10

4.2.1) Native coin transaction 11

4.2.2) Smart contract creation and interaction 11

5. Execution Engine (EVM) **12**

5.1) EVM Instructions Set 14

5.2) Execution Cycle 14

6. Validation & Finalization **15**

6.1) Finalization 16

6.2) Validation 16

6.2.1) Block header validation 16

6.2.2) Block body validation 16

6.2.3) State validation 17

7. Consensus **18**

7.1) Consensus definition 18

7.2) Consensus Mechanism 18

7.2.1) Proof-of-Stake-Authority (PoSA) 18
Staking mechanism 19
Authority mechanism 21
7.2.2) Difference between Clique PoA and Erawan PoSA 21
Block reward distribution 21
Block header contents 21
Block header validation 21

8. Mining and Reward distribution **23**
8.1) Mining mechanism 23
8.2) Block reward distribution mechanism 24

9. Block period **25**
9.1) Performance comparison 26

10. Architecture **29**
10.1) Blockchain layer 29
10.1.1) Network layer 29
10.1.2) Blockchain core 29
10.1.3) Tools & library 30
10.1.4) Application 30
10.2) Network Architecture 30
10.2.1) Validator node pool 30
10.2.2) Boot node pool 31
10.2.3) Remote procedure call (RPC) Node 31
10.2.4) Archive Node 31

11. The advantages of Bitkub Chain **31**
11.1) Blockchain Layer 31
11.1.1) Block size 31
11.1.2) Block time 32
11.1.3) Transaction fees 32
11.1.4) Infrastructure design 33
11.2) Application Layer 35
11.2.1) Registration 35
Registration levels 35
Register procedure 35
11.2.2) KAP token standard 36
11.2.3) KKUB 37
11.2.4) HyperBlock 38
Backend Service 38
Gas Tank 39

Smart Contract 39
Fail-Safe mechanism 39

12. Future Technical Goals **41**
12.1) Scalability 41
12.1.1) Layer1 Scaling 41
a) Increasing block size 41
b) Sharding 41
12.1.2) Layer2 Scaling 43
a) Rollups 43
b) Plasma 43
c) Validium 44
12.2) Consensus Improvement 44
11.2.1) Proof-of-Stake + 3 44
12.3) Bitkub Chain SDK (SDK) 45

13. Conclusion **47**

References **47**

1. Introduction

Blockchain technology, which runs Bitcoin, has developed over the last decade into one of today's biggest ground-breaking technologies with the potential to impact every industry from financial to healthcare, to manufacturing, to educational institutions. However, Bitcoin (the first cryptocurrency) has only used blockchain as the infrastructure and created digital money with the ability to transfer value. Whereas Ethereum made a new paradigm shift in blockchain technology with programmable capability for digital money, global payments, and applications.

In the Blockchain Trilemma, Bitcoin and Ethereum still use the Proof-of-Work (PoW) consensus type, which helps increase transparency and security. However, it generates environmental impact and the high cost of mining nodes that cause power consumption, leading to an implementation challenge, high transaction fees, and causing transaction delays. As a result, it creates a low rate of blockchain adoption and adds a barrier to people who aim to use blockchain technology.

To reconfigure and provide a better consensus solution, "Bitkub Chain" (BKC) introduced a technology platform and system that help bridge the gap, lower impact on the environment, faster processing of transactions with lower fees, helps to save the mining cost, and solve cybersecurity issues.

"Bitkub Chain'' started developing and reapplying the Ethereum blockchain [3] and forked from the Go-Ethereum [21] concept but changed the consensus from Proof-of-Work (PoW) based on Ethash [26] to Proof-of-Authority (PoA) based on Clique [10]. In the initial stage, Bitkub Chain uses the PoA consensus [9] to add the advantage of lower mining or running node cost and faster speed of verifying transactions. However, giving greater weight to scalability and increasing network performance from PoA might lower the decentralization level.

Therefore, to offer a better technology platform and system for blockchain adoption. Our team is developing a new consensus called Proof-of-Stake-Authority (PoSA) to increase the network's decentralization, corresponding and addressing a solution to all concerns from PoA, significantly improving network performance and providing truthful data with less environmental impact to democratize opportunity to use blockchain technology.

To summarize, the Bitkub Chain enhances the opportunity for everyone to participate in the decentralized economy without barriers to entry such as low transaction speed, high transaction fees, etc. Bitkub Chain was designed to be a public network providing a good user experience. Both blockchain and the application layer can increase the network's speed and scalability. It consists of increasing the block size, reducing the block time to maximize speed, and providing a fixed rate of gas price to stabilize the transaction fees. Moreover, Bitkub Chain has a specific infrastructure design for the validator set by connecting the validator node directly through the unlimited bandwidth network, which can help improve the synchronization and stability. In addition, on the application layer, we have a KAP which is a standard to implement the token and Wrapped KUB (KKUB), the KAP-20 version of KUB, which will be an important part of driving the De-Fi and DApp on the Bitkub Chain, and the HyperBlock system that combined off-chain and on-chain solutions together to improve the transaction scalability. Finally, all solutions provide the best experience for users, developers, and everyone to engage with the decentralized world seamlessly and efficiently through Bitkub Chain.

2. Blockchain paradigm

Bitkub Chain, a forked Ethereum base blockchain [3], can be interpreted as a transaction-based state machine. It starts from the first state, called the genesis state, and executes the transactions to produce some current state. The visualization can be shown in Figure-01.

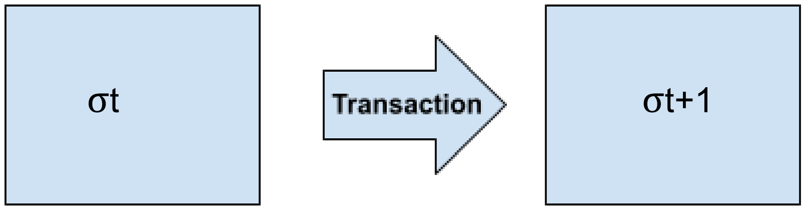


Figure-01

σt , $\sigma t+1$ is described as the previous state and current state, which is changed by the transaction (T) via the state transition function (Υ). According to the state machine concept, the general equation of state transformation can be written as follows (equation-1).

(1)[1] $$\sigma t+1 \equiv \Upsilon(\sigma t, T)$$

Transactions are packed into blocks and linked together to form a chain structure using a cryptographic hash. Using hash makes each block unique and unable to spoof. According to equation-2, a block (a series of transactions) can be applied to the equation. So, the equation is rewritten in terms of block-level as follows.

(2)[1] $$\sigma_{t+1} \equiv \Pi(\sigma_t, B)$$

(3)[1] $$B \equiv (...(T_0, T_1, ...), ...)$$

(4)[1] $$\Pi(\sigma, B) \equiv \Omega(B, \Upsilon(\Upsilon(\sigma, T_0), T_1)...)$$

From equation-3, a block(B) is a series of transactions and their components. After integrating with state-transition function (Υ) and block-finalization function (Ω), a block-level state-transition function (Π) is created. To summarize equation-2, 3, and 4, a new equation is formed as equation-5

(5) $$\sigma_{t+1} \equiv \Omega(B, \Upsilon(\Upsilon(\sigma_t, T_0), T_1)...)$$

2.1) Native currency units

Bitkub Chain has a native currency called **KUB** which is used to pay for transaction fees in order to reward node validators. The largest unit is KUB.

One KUB is defined as being 10^{18}. On the other hand, the smallest unit is called **Bit**. All sub denominations of KUB are listed in Table-01.

Power	Name
10^{0}	BIT
10^{9}	GBIT
10^{18}	KUB

Table-01

2.2) Bitkub Chain history

Bitkub Chain is a decentralized network; all validator nodes have the right to create a new block at every turn. However, in some cases, there is a possibility of producing a temporary forked block (chain is split) which occurs from coexisting with the states created by a different validator node. This situation can be shown in Figure-02.

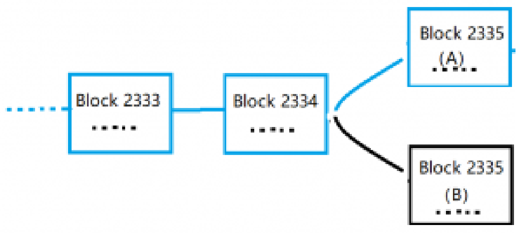


Figure-02 (source https://medium.com/@vrastromind)

The first chain created would be the main chain while the rest will be the uncle block as stated by the Ethereum GHOST protocol created by (Sompolinsky and Zohar ,2013). The uncle block will be discarded, but the canonical chain (the longest chain) as shown in Figure-03.

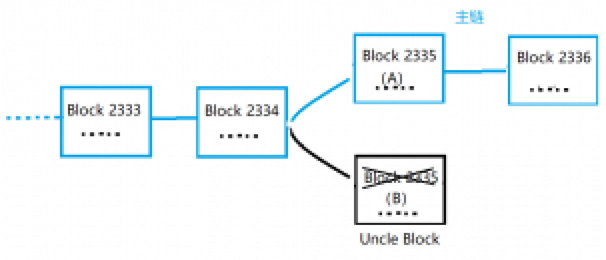


Figure-03 (source https://medium.com/@vrastromind)

Therefore, a new protocol must be implemented to upgrade the protocol occasionally. After implementation, it will modify an existing protocol and create a new one split from the original version, called "Hard Fork". As a result, the previous software using the old protocol version will be obsolete and have to be replaced with the latest software. Bitkub Chain has moved from Proof of Authority(PoA) to Proof of Stake Authority(PoSA) in the Erawan Hard Fork [28].

3. States & Tries

Like Ethereum, The trie is a core database in Bitkub Chain storing all significant states such as account, receipt, and transaction state.

3.1) Merkle Patricia Trie

The Merkle Patricia Trie [8] introduced by Ethereum [7] is a cryptographically authenticated data structure used to store key and value binding. It is a combination of the Merkle Tree and Patricia Tree (a binary radix tree) to improve the efficiency of the data's insertion, deletion, and selection. In addition, Merkle Patricia Trie is the primary data structure of Bitkub Chain (the same as Ethereum). There are three types of nodes: extension node, branch node, and leaf node [6].

3.1.1) Extension node

includes two items 1. encodedPath is a partial path encoded by RLP encoding function, and 2. its key is the next trie node to lookup.

3.1.2) Branch node

branch nodes are the path between one node to another node. For example, the node contains 17 array items. One will hold the possible value (hex character - nibble) in the path, and another one will hold the final target value after the path has been fully traversed.

3.1.3) Leaf node

is used to store the data, including two items inside a node encoded path and value. The encodedPath is the end path encoded by the RLP encoding function, and the value is the target value of the specific path.

3.2) States

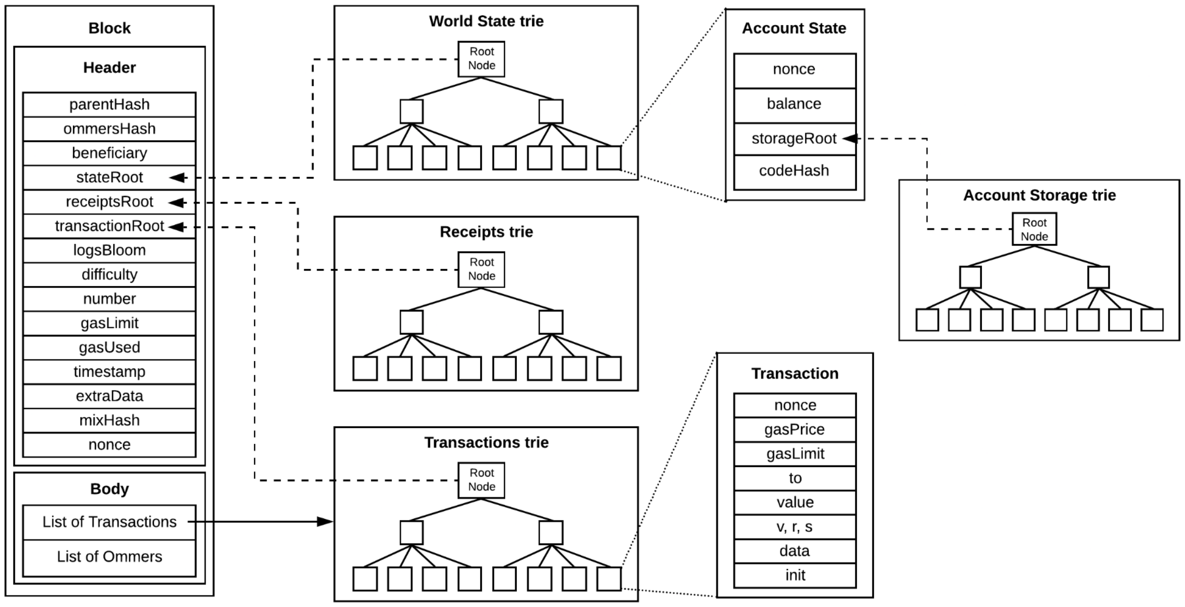


Figure-04: An overview of the tries [6]

3.2.1) State Root

Bitkub Chain stores the root of all the states in the block header. Every trie has a root, and each root is a 256-bit hash of the root node of each tree encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values.

(6) $$RootHash = KEC(RLP.encode(Trie))$$

Where KEC is a Keccak 256-bit hashing function, RLP.encode is an RLP encoding function, and the Trie is mapping the key-value set.

3.2.2) World State

The world state is one of the states that maps between addresses and account states. It is expressed that the world state and global state could be seen as one after it is constantly updated from transaction executions. Therefore, the balance of an account or the current state of a smart contract could be query from the world state.

a) World State Trie All the information about Bitkub Chain accounts is stored in the world state trie. The root node of the trie will be cryptographically hashed by using the Keccak-256 hashing function and included in the **stateRoot** field of the block header to represent the current state [14].

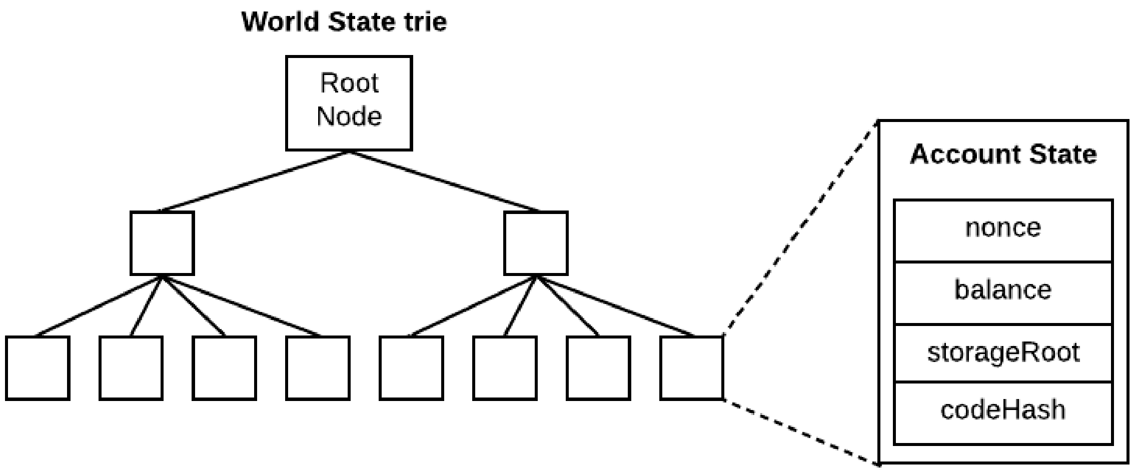


Figure-05: World State Trie [6]

b) Account State

i) Account state components

- **Nonce** is the number of transactions sent by an account.
- **Balance** is KUB balance of an account.
- **StorageRoot** consists of a 256-bit hash of the root node of the account storage trie.
- **codeHash** is the hash of the EVM code of an account.

ii) Account Storage Trie

The account storage trie is used to store the states of the contract accounts, as for External-Own Accounts (EOAs) the storageRoot will be empty. All smart contract data is persisted in the account storage trie as a mapping between 32-bytes integers.

3.2.3) Transaction State

a) Transaction state components

i) **Nonce** is the number of transactions sent by the account.

ii) gasPrice is the number of Bit that will be paid per unit of gas.
iii) gasLimit is the maximum amount of gas allowed to use for executing the transaction.
iv) To is the recipient address for a normal transaction, for a contract creation transaction will be zero address.
v) Value is the number of Bit that transferred to the recipient.
vi) R, S, V are the values used in the cryptographic signature to determine the sender's validity of the transaction.
vii) AccessList is a list of addresses and storage keys.
viii) Chain ID is the specific ID of the chain.
ix) Data is an input data in byte array of the message call (for transferring the value and sending a message call to a smart contract).
x) Init is the EVM-code utilized for initialization of the smart contract. (for contract creation only).

b) Transaction Trie

stores all of the transactions included in a block. The block header (in the transactionsRoot field) will include the hash of the root node of the transaction trie.

3.2.4) Transaction Receipt

a) Transaction Receipt Trie

stores all transaction receipts in the same block practice as Transaction Trie. The hash of the root node of the transaction receipts trie is also included in the receiptsRoot field of the block header.

4. GAS

All transactions sent on Bitkub Chain are subjected to fees called **Gas** to help prevent users from abusing their transactions and support the validator node's maintenance cost. The prices of gas units in each transaction varies based on the computational instructions of the Ethereum virtual machine (EVM), data length, memory usage, etc. (see Appendix A). The amount of gas collected is transferred to the in-turn validator node wallet.

4.1) Gas Limit and Gas Price

As stated above, every transaction spending gas has a limitation of gas usage known as **gasLimit**. The prices of gas units are implicitly purchased from the sender's account balance. The purchase happens at the **gasPrice** (Wood, 2022). The gasPrice pegs to 5 Gbit to promote the circulation of KUB Coin (Bitkub Chain's Native coin) on Bitkub Chain network and the company's gas tank. In addition, the remaining gas after the transaction had been executed would be refunded to the sender's address at the same gasPrice that why it was called gasLimit, and can express by the following equation:

(7) $$g_r = T_{gasL} - G_{used}$$

Where g_r is the amount of gas refund sent back to the sender's address. T_{gasL}, G_{used} means gasLimit set by the sender and actual gas used.

4.2 Mechanism of gas calculation

Bitkub Chain, forked Ethereum, uses the same method of gas calculation as Ethereum main network. The gas will be charged based on the gasLimit that the sender input and the remaining gas will be refunded after finalizing the process. Before any EVM code gets executed, some amount of gas was deducted from the sender's defined gasLimit. This amount is named **intrinsic gas**, g_0, and can be described from equation (8).

(8)[1] $$g_0 \equiv \sum_{i \in T_\mathrm{i}, T_\mathrm{d}} \begin{cases} G_{\mathrm{txdatazero}} & \text{if} \quad i = 0 \\ G_{\mathrm{txdatanonzero}} & \text{otherwise} \end{cases} + \begin{cases} G_{\mathrm{txcreate}} & \text{if} \quad T_\mathrm{t} = \varnothing \\ 0 & \text{otherwise} \end{cases}$$

$$+ G_{\mathrm{transaction}} + \sum_{j=0}^{\|T_{\mathbf{A}}\|-1} \left(G_{\mathrm{accesslistaddress}} + \|T_{\mathbf{A}}[j]_{\mathbf{s}}\| G_{\mathrm{accessliststorage}}\right)$$

T_i, T_d stands for the series of bytes. $G_{txcreate}$ gets value when a transaction is invoked in the contract creation process. In other words, if a contract creation transaction does not exist, $G_{txcreate}$ will be zero in value. $G_{transaction}$ is the initial fee that all transactions on the Bitkub Chain network have to pay. $G_{accesslistdaardss}$ and $G_{accessliststorage}$ are the cost of warm-up address and storage access. After intrinsic gas has been charged, the transaction will be executed. There are two types of interaction with Bitkub Chain.

4.2.1) Native coin transaction

This type of transaction is simple and does not invoke any smart contract or smart contract creation. Sending a KUB coin to a regular wallet address (not a smart contract address) is the only one for this type. According to equation (8), It does not have $G_{txcreate}$, $G_{accesslistdaardss}$, $G_{txdatanonzero}$, and $G_{txdatanonzero}$. Moreover, it does not interact with EVM (Only g_0 occurs to calculate the gas fee). To summarize, G_{used} can be re-written as follows**.**

$$G_{\mathrm{used}} \equiv G_{\mathrm{transaction}}$$

4.2.2) Smart contract creation and interaction

After intrinsic gas has been charged, EVM code will be executed, and gas is consumed by EVM instruction (also called OPCODE). Each instruction has a specific amount of gas consumption that can be clearly explained by equation $C(\sigma, \mu, A, I)$ (Appendix B). As a result, the actual gas used (G_{used}) is determined by equation (9).

$$G_{\mathrm{used}} = g_0 + C(\sigma, \mu, A, I) \tag{9}$$

This equation describes the actual gas consumed by the whole blockchain network. However, the gas calculation equation (9) is only writing action on smart contracts and a contraction creation transaction, but reading action from only wallet address does not pay the gas for calling the smart contract.

5. Execution Engine (EVM)

Bitkub Chain execution engine relies on Ethereum Virtual Machine (EVM). For processing the transaction and managing the state transition, the Ethereum Virtual Machine is a stack-based machine that executes the bytecode and performs the operation based on a set of instructions to transform the state from the previous state to the new state [18]. The EVM is a Turing-complete machine, the total number of computations limited by the amount of gas to prevent such an attack, e.g., denial-of-service (DDoS) attacks. The EVM also supports exception handling to avoid the invalid state transition.

The main components of EVM include:

- **World state *(σ)*** is a non-volatile state that stores a significant state, such as the account state.
- **Machine state *(μ)*** is a volatile state used during the execution, such as gas available ***g***, the program counter ***pc**,* the memory content ***m,*** and the stack content ***s.***
- **Stack** is a last-in-first-out (LIFO) storage with two abstract operations: PUSH and POP.
- **Memory** is a word-addressed byte array used to store all instructions. The memory is volatile because its state will be reset after the computation ends.
- **Gas *(g)*** is the remaining gas of the current execution.
- **Program Counter *(pc)*** is always set to zero at the beginning of execution and increased with instructions being read from memory.
- **Virtual ROM** is an immutable memory where the EVM bytecode or the program code is stored and accessible only through a particular instruction, e.g., CODECOPY instruction.
- **Storage** is persistent storage, a key-value store. The storage is non-volatile because the blockchain's system state is permanently persisted.

EVM architecture

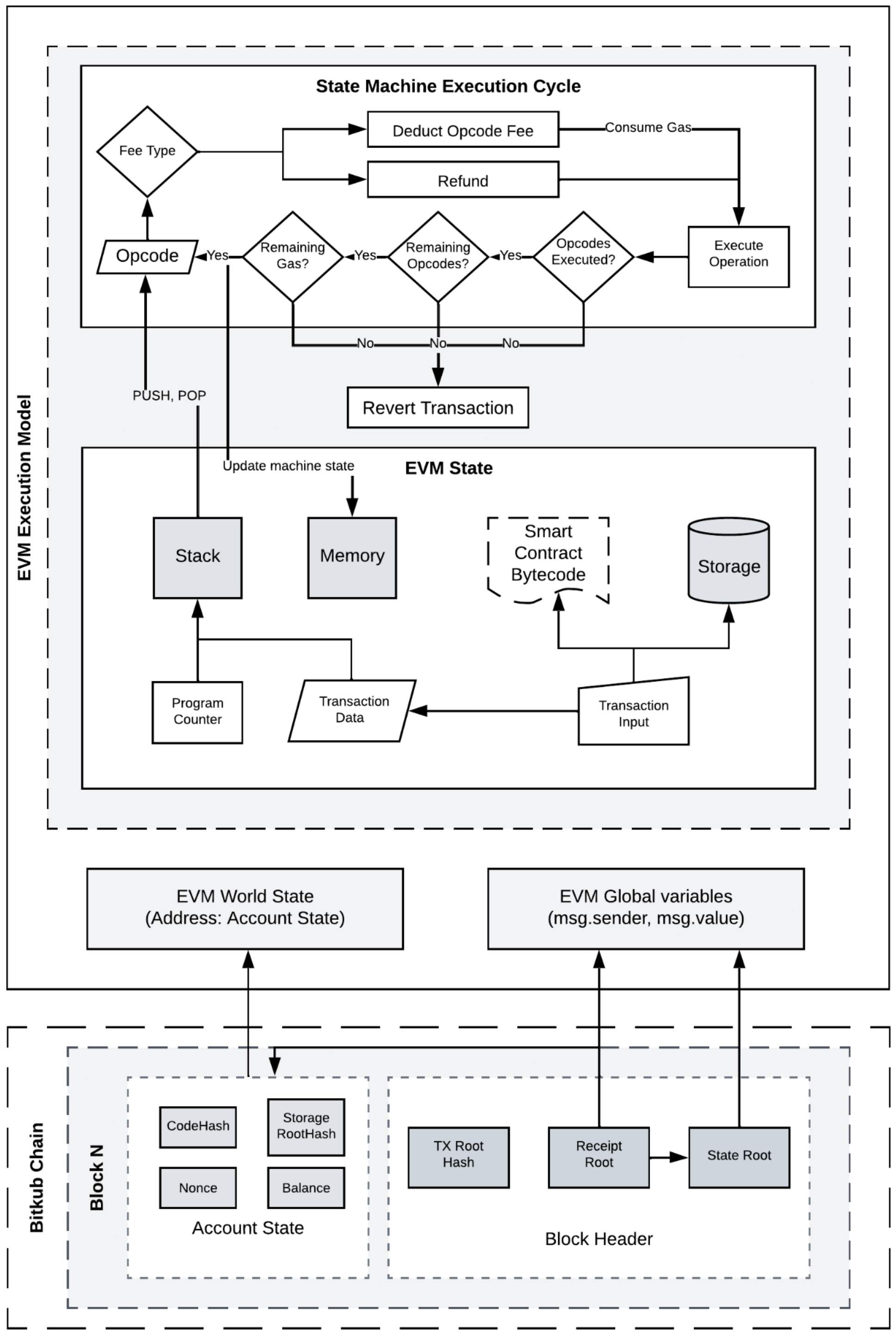


Figure-06: EVM architecture [18]

5.1) EVM Instructions Set

The smart contract code is written in a low-level, stack-based bytecode language called “EVM bytecode”. The bytecode consists of a series of bytes, where each byte represents an operation [13]. There are many types of operations:

5.1.1) Arithmetic operations, opcodes for arithmetic execution, e.g., ADD, MUL, SUB, DIV, MOD, SHA3, etc.

5.1.2) Stack operations, opcodes for stack, memory, and storage management instructions, e.g., POP, PUSH, MLOAD, MSTORE, SLOAD, STORE, etc.

5.1.3) Process flow operations, opcodes for control the execution flow, e.g., STOP, JUMP, PC, JUMPDEST.

5.1.4) System operations, opcodes for the system executing the program, e.g., CREATE, CALL, RETURN, REVERT, DELEGATECALL, STATICCALL*, etc.*

5.1.5) Logic operations, opcodes for comparisons, and bitwise logic, e.g., LT, GT, XOR, OR, EQ, ISZERO, etc.

5.1.6) Environmental operations, opcodes for dealing with the execution environment information, e.g., GAS, ADDRESS, BALANCE, ORIGIN, EXTCODECOPY and etc.

5.1.7) Block operations, opcodes for accessing the information on the current block, e.g., BLOCKHASH, COINBASE, TIMESTAMP, DIFFICULTY, etc.

For more information about an EVM instructions cost [1]

5.2) Execution Cycle

Several essential information is required in the execution model inside the execution environment, including the system state $\boldsymbol{\sigma}$, the remaining gas $\boldsymbol{g}$, substate $\boldsymbol{A}$ (Appendix C), and tuple **I** (Appendix D). The primary function in the EVM is a state-progression function (**O**) which performs an iterator loop recursively to compute the resultant state $\boldsymbol{\sigma'}$, the remaining gas $\boldsymbol{g'}$, substate $\boldsymbol{A'}$, and the consequent output $\boldsymbol{I}$

(10)[1]

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \quad \equiv \quad (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I)$$

In the normal situation of the execution, the state-progression function (O) should be performed until reaching the normal halting and return the output result of execution.

(11)[1]

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) \text{ if } & w \in \{\text{RETURN}, \text{REVERT}\} \\ () \quad\quad \text{if} & w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing \quad\quad \text{otherwise} \end{cases}$$

The normal halting might be triggered by some opcodes such as RETURN, REVERT, STOP, SELFDESTRUCT. The other will be an empty set (∅). However, some unexpected cases could occur, and the execution would be halted and return to the exceptional halting state.

(12)[1]

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w = \text{JUMP} \ \wedge \ \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{JUMPI} \ \wedge \ \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \ \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{RETURNDATACOPY} \ \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \quad \vee \\ & (\neg I_{\mathrm{w}} \ \wedge \ W(w, \boldsymbol{\mu})) \quad \vee \\ & (w = \text{SSTORE} \ \wedge \ \boldsymbol{\mu}_g \leqslant G_{\text{callstipend}}) \end{aligned}$$

where

(13)[1]

$$\begin{aligned} W(w, \boldsymbol{\mu}) \equiv \quad & w \in \{\text{CREATE}, \text{CREATE2}, \text{SSTORE}, \\ & \text{SELFDESTRUCT}\} \ \vee \\ & \text{LOG0} \leq w \ \wedge \ w \leq \text{LOG4} \quad \vee \\ & w = \text{CALL} \ \wedge \ \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \end{aligned}$$

Many cases can cause the exceptional halting function (Z), such as insufficient gas, invalid instruction, insufficient stack items, JUMP/JUMPI invalid destination, the stack size larger than 1024, and state modification being attempted during a static call, etc.

6. Validation & Finalization

6.1) Finalization

Finalize means permanently added and can not be altered or reverted. In Bitkub Chain, a block will be considered to be a finalized block when the block components are valid such as block header, transactions, and uncle block are valid.

6.2) Validation

6.2.1) Block header validation

Block header validation is done by the consensus engine. In Bitkub Chain, the block header will be validated based on the rules set by Clique consensus as follows:

a) Block timestamp must not be more than the current UNIX time and is not too close to its parent block timestamp.

b) Nonce must be either **nonce to vote on adding a new validator** 0xffffffffffffffff or **nonce to vote on removing a validator** 0x0000000000000000, otherwise will be considered as invalid.

c) Extra data must be contained in both the vanity and signature, ensures that the block doesn't contain any uncles.

d) Difficulty must be either **difficulty for in-turn validator** two or **difficulty for out-of-turn validator** 1. Otherwise, it will be considered invalid.

f) The gas limit must be lower than the maximum gas limit $2^{63}-1$.

g) The gas used must be lower than the block gas limit.

6.2.2) Block body validation

a) Transaction validation

Check whether the transaction root hash in the block header T, is equal to the hash which return from the function C,

$$(T', R') = C(TXs, To) \tag{14}$$

where the TXs is a list of transactions and To is an empty tries, the function C will compute the output as a transaction root hash T', requires T = T'

b) The uncle

In Bitkub Chain is based on Clique, which require an empty uncle block.

6.2.3) State validation

a) The amount of used gas in the state execution process must be equal to the actual gas used in the block header, otherwise it will be considered to be invalid.

b) Validate the received block's bloom with the one derived from the generated receipts.

c) Check whether the state root in the block header S, is equal to the state root S', which returns from the function D (15),

$$S' = D(E) \tag{15}$$

Where E is a StateDB, and the function D will compute the output as a state root S', requires $S = S'$

7. Consensus

7.1) Consensus definition

In a centralized consensus system, it is easy to find an agreement because everything relies on the central authority, but there are some concerns, e.g., a single point of failure and unworthy of trust. In contrast with a centralized system, a distributed system such as Blockchain is reliable. However, there is a challenge to make everyone on the network agree on the same data without a centralized authority. Therefore, the consensus is used in a distributed system as a mechanism in order to find an agreement on a single version of the data between each node in the network.

7.2) Consensus Mechanism

The consensus mechanism is an essential component inside the Blockchain which both secures the network and plays on the crypto-economic game theory to incentivize the validators and prevent some kinds of attacks such as the 51% attack. There are many consensus mechanisms such as Proof-of-Work, Proof-of-Stake, Proof-of-Authority, etc. In Bitkub Chain, we use the Proof-of-Authority consensus until the Erawan Hardfork [28]. Then we move to use the Proof-of-Stake-Authority.

7.2.1) Proof-of-Stake-Authority (PoSA)

Bitkub Chain changes consensus to the Proof-of-Stake-Authority (PoSA) consensus because we want to lower the barrier of entry to become the validator node and encourage the decentralization of Bitkub Chain. Proof-of-Stake-Authority is a consensus mechanism that combines the techniques between Proof-of-Stake and Proof-of-Authority consensus. Proof-of-Stake-Authority uses the staking mechanism to find a set of authorized validators in an epoch. To become a validator, the candidates must deposit at least MINIMUM_STAKE_AMOUNT (Appendix E) of KUB coin (a native coin of Bitkub Chain) in our PoSA_Smart Contract (Appendix F), and once an epoch reaches a set of validators will be granted to have an authorized to validate and propose new block in a Bitkub Chain network. There are two main parts of the Bitkub Chain PoSA consensus includes

a) **Staking mechanism**
Bitkub Chain PoSA uses smart contracts to operate and manage logic such as staking logic, withdrawal logic, claim reward logic, etc. There are three main functions:

i) Deposit Once KUB coin is deposited to the PoSA smart contract and then wrapped KUB to KKUB (KAP-20 KUB). The stakeholder will receive a Bond (KAP-721) representing the staking information.

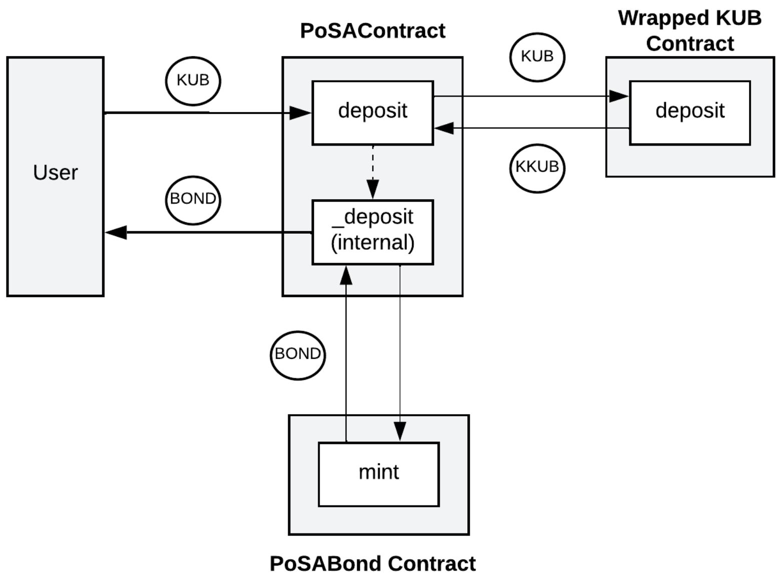


Figure-08: Deposit flow

ii) Withdraw The withdraw function will first claim the remaining reward and then update the Bond state by calling the update, withdrawing function, and distributing the KKUB to the stakeholders.

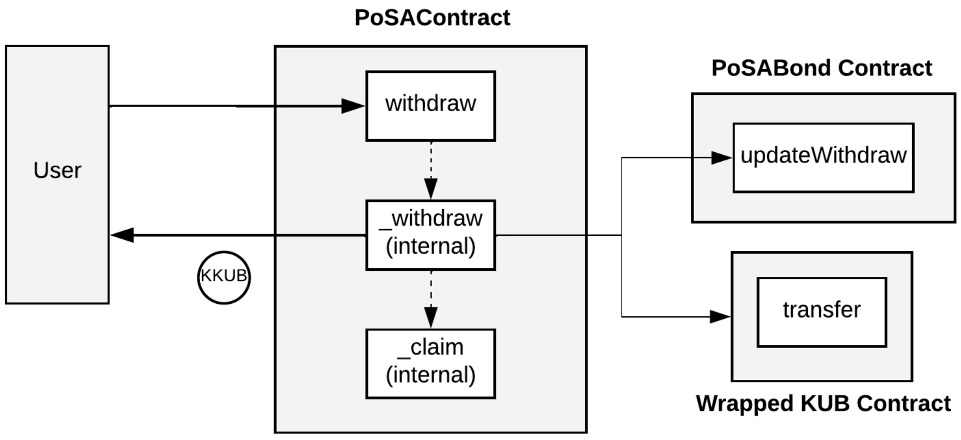


Figure-09: Withdrawal flow

iii) Claim To claim the reward, the claim function calculates the reward amount, updates the state of the Bond, and then transfers the reward as a KKUB to stakeholders.

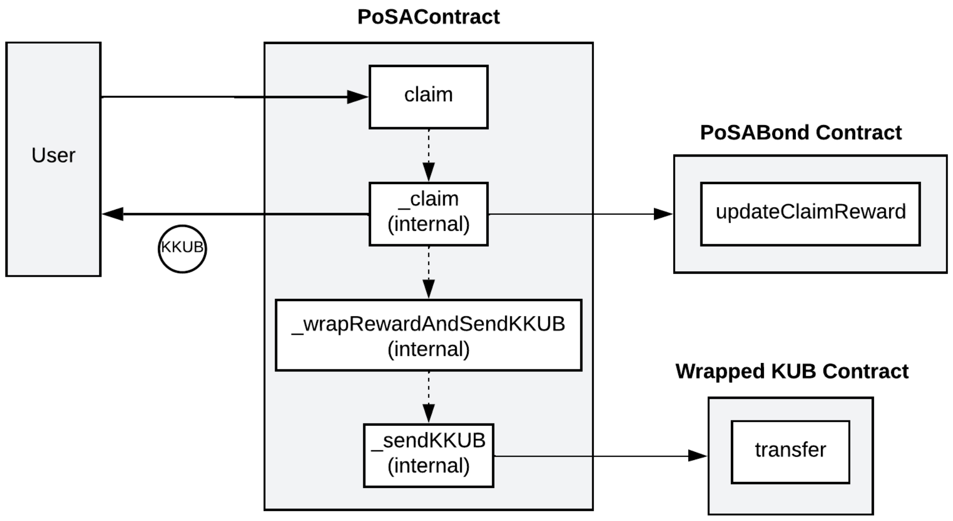


Figure-10: Claim reward flow

b) Authority mechanism

In the protocol layer, Bitkub Chain uses Clique to manage the authority mechanism to control the authorizes of each validator in an epoch and use the rotation schema for validating the block.

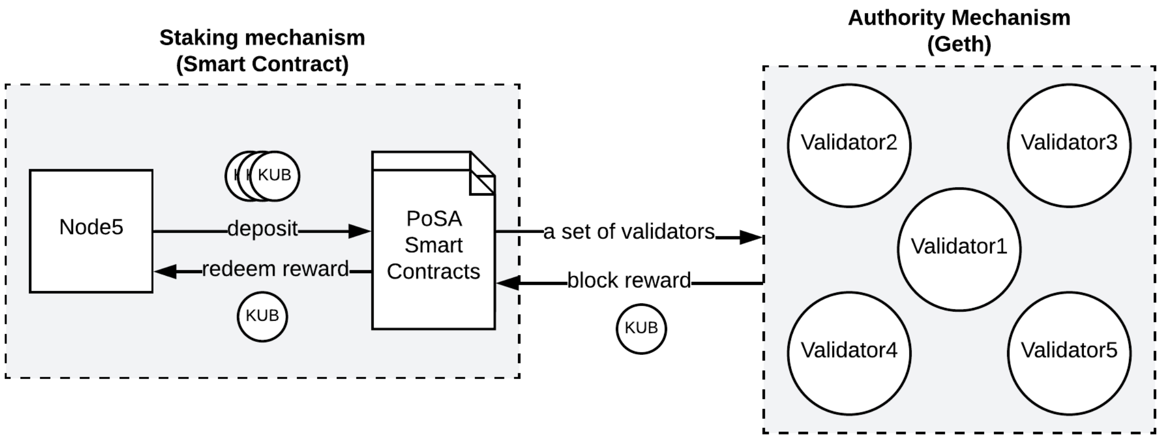


Figure-11: Bitkub Chain PoSA overview architecture

7.2.2) Difference between Clique PoA and Erawan PoSA

a) Block reward distribution

In the Clique consensus, the beneficiary or the Coinbase is a validator wallet address. However, in the case of PoSA, the beneficiary is the reward address (smart contract address).

b) Block header contents

In the Clique consensus, the Coinbase field of the block header keeps the proposed address, but In the PoSA, the Coinbase stores the reward address and uses the mixDigest field to keep the proposed address instead.

c) Block header validation

In the Clique consensus, whenever the network reaches an epoch block, the Coinbase field must be empty to be valid. But in the PoSA, the Coinbase field always has a value, so we change the validation logic from requiring Coinbase to mixDigest to be empty once it reaches an epoch block [28].

Components	**Clique (PoA)**	**Erawan (PoSA)**
Coinbase	Voted address	Reward address
MixDigest	NULL	Voted address
Sealer Address	NaN	Validator address
Config	CliqueConfig	ChainConfig

Table-02: Comparison of PoA and PoSA components

8. Mining and Reward distribution

8.1) Mining mechanism

Bitkub Chain constantly emits a new block every five seconds (see more in the block period section). Every validator will create a new block (an empty block) and take the pending transactions from the transaction pool into a new block. The pending transaction will be sequenced by the gas price (GBit), which means the transaction with more gas price will have more chances to be included in the new block.

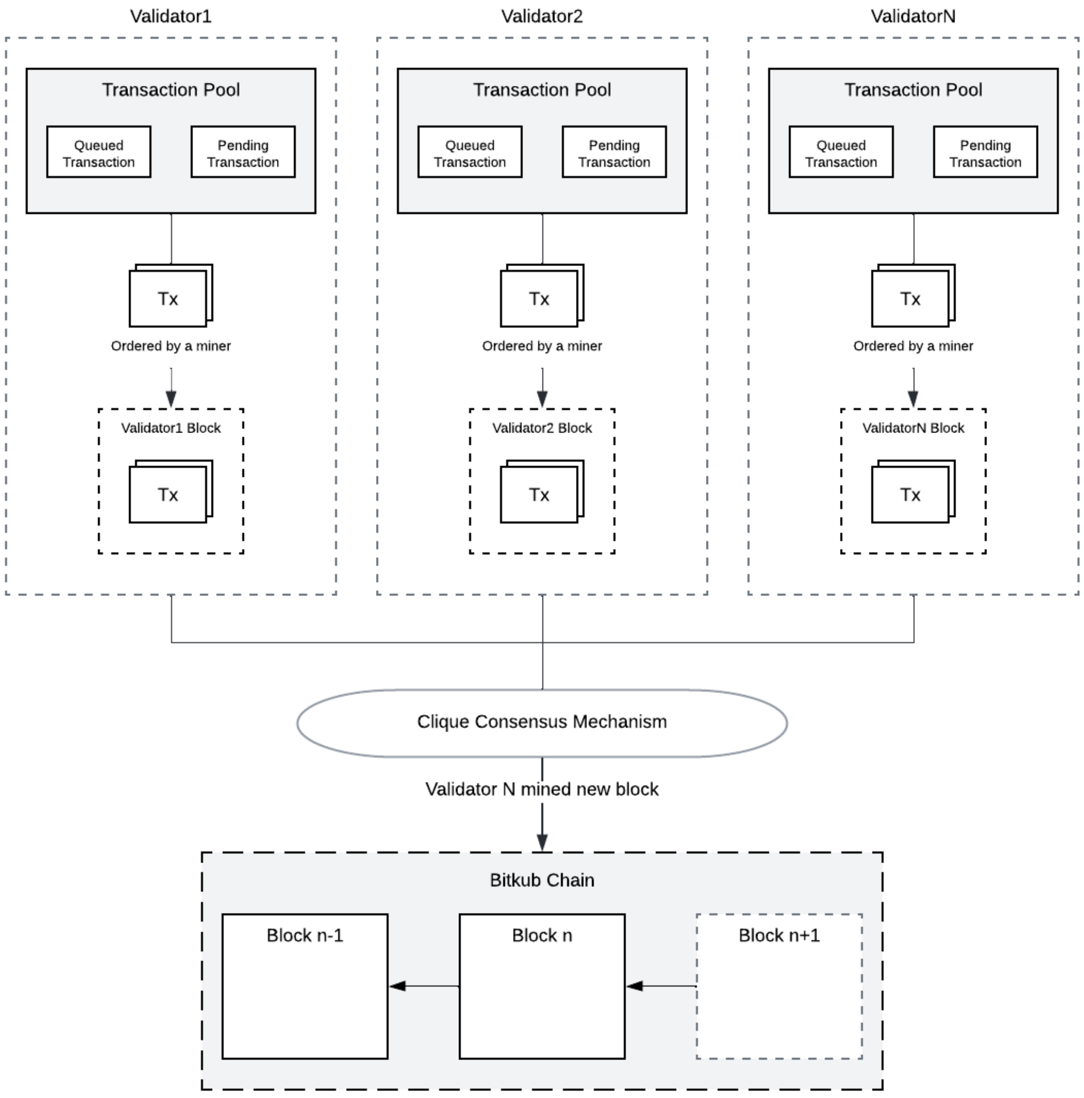


Figure-12: Mining a new block

Bitkub Chain uses the Clique mechanism to manage the consensus among the group of authorized validators to find a validator to mine a new block. The selected validator will commit, seal and append a new block to the canonical chain and then broadcast it to the other node in the network. The other nodes on the Bitkub Chain will receive a new block and perform a block validation algorithm including;

a) Check if the previous block referenced is valid and exists.
b) Check that the block timestamp is greater than the timestamp of the previous block, and not lower than the previous block timestamp plus the block period.
c) Check that the block number, transaction root, uncle root, difficulty, and block gas limit are valid.
d) Check if the signature of the validator is valid.
e) Check if the Merkle root of the state is equal to the final state root provided in the block header.

After all is executed and valid, each node will append a new block to the storage.

8.2) Block reward distribution mechanism

The system will incentivize the validator by rewarding the native coin (KUB coin) and the system will collect rewards from the fee of the transactions included in the newly mined block. The total block reward is a summation of the transaction fee in a block, following the equation of block reward.

(18)

$$\sum_{g = G1}^{g = Gn} g$$

g = Gas fee of a transaction
G1 = Gas fee of the first transaction in a block
Gn = Gas fee of the n^{th} transaction in a block

After the reward has been collected, the system will be distributed to the beneficiary by default Clique consensus [10]. Hence, it will set the beneficiary to be the validator or signer address. However, after the Erawan Hard Fork [28], the beneficiary address will be set to a reward address (contract address).

9. Block period

In general, Ethereum's main network uses Proof of Work (PoW) as their consensus, which the block time depends on the number of computational power (blockchain miner) and difficulty. This consensus consumes more time and energy than other means in order to create a single block. Nowadays, carbon emission is an international concern, so PoW should not be the candidate for Bitkub Chain. As a result, Proof of Stake Authority (PoSA) was chosen to be a Bitkub Chain's consensus (see session-10). PoSA is faster, and the block time (T_{block}) is fixed to five seconds. Even though the block time was set to five seconds, the actual time (T_{total}) still fluctuates, affected by many factors such as network, storage, CPU, memory, number of online validators, etc. The actual can be described as the equation-19 below.

(19)[1] $$T_{total} = T_{block} + y$$

Let Y follow an ordinary half-normal distribution function. Then, according to the central limit theorem (see Appendix G), The function came from the collection of block time aggregated together to create a normal distribution function (see the equation-20) that estimates the possible variance of block time.

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where $x \geq 5$ Sec.

(20)[1]

$$\begin{aligned} \mu &= \text{ Mean} \\ \sigma &= \text{ Standard Deviation} \\ \pi &\approx 3.14159\cdots \\ e &\approx 2.71828\cdots \end{aligned}$$

From equation [20], it is a continuous function, but the collected data is a discrete function because we need to observe the collected information for comparison and estimation. Therefore, the simplest way to estimate is to fit it into a normal distribution function.

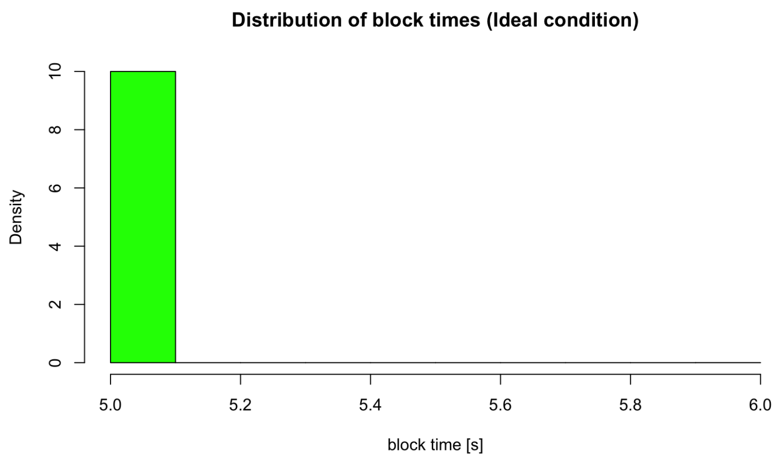
9.1) Performance comparison

full-online and partial-online validators can cause different block times. The block time will increase when the whole system condition is not running in an ideal situation (see Table-03 for comparison).

Description	Min	Max	Mean	Median
Ethereum Main network	1.00	174.00	14.35	10.00
Ideal Condition (Bitkub chain)	5.00	5.00	5.00	5.00
Bad Condition (Bitkub chain)	5.00	15.00	5.228	5.00

Table-03 (value in seconds)

The degradation of performance is from the bad condition of the chain. From the experiment, the distribution of block times can be different as Figure-14.



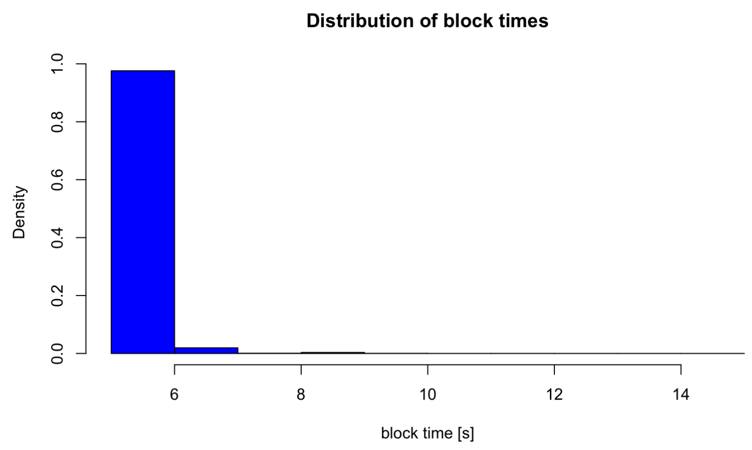
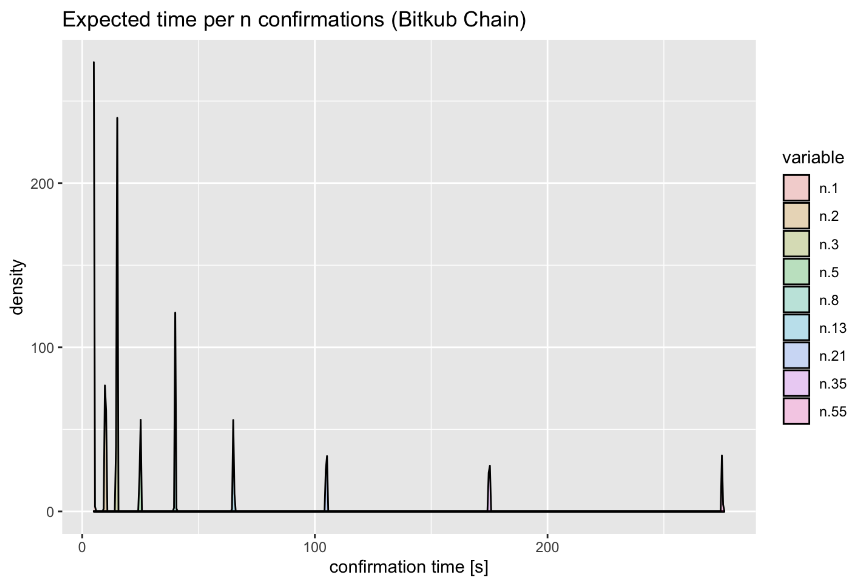


Figure-13 compares ideal and bad conditions.

When many blocks are produced, the time per block will increase due to system inconsistency. In addition, the overall delay time is converted to the normal distribution function (Equation-20). When the delay from each block was aggregated together, the expected time per block can be plotted as shown in the figure below.



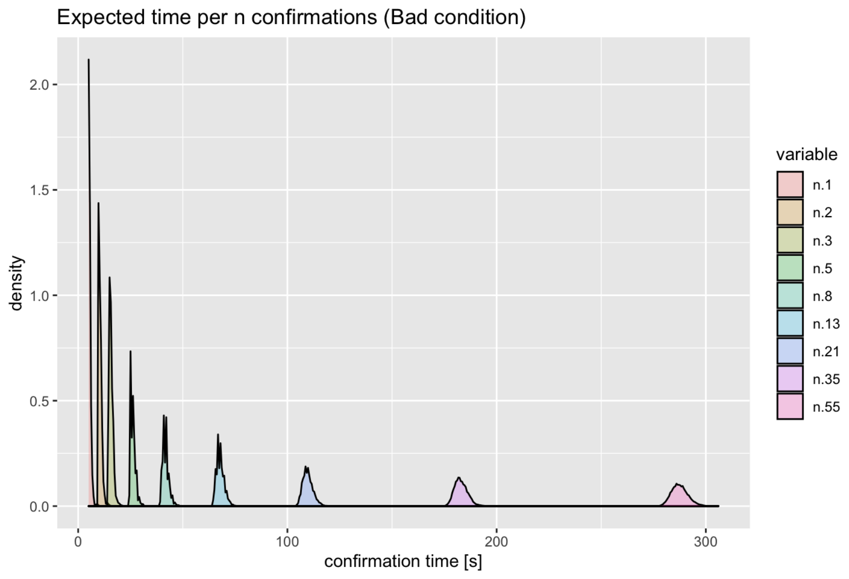


Figure-14

According to CLT, as we can see from the figures, the delay is spreading out and converging to a normal distribution function in unsuitable conditions.

10. Architecture

10.1) Blockchain layer

Bitkub Chain relies on the internet protocol (IP), which is widely used nowadays. The blockchain protocol builds on top of TCP/IP. It transcends IP and becomes the Internet which can store or send the value called the Internet of value. The Architecture of the Bitkub Chain can explain as a layer (see Figure-15).

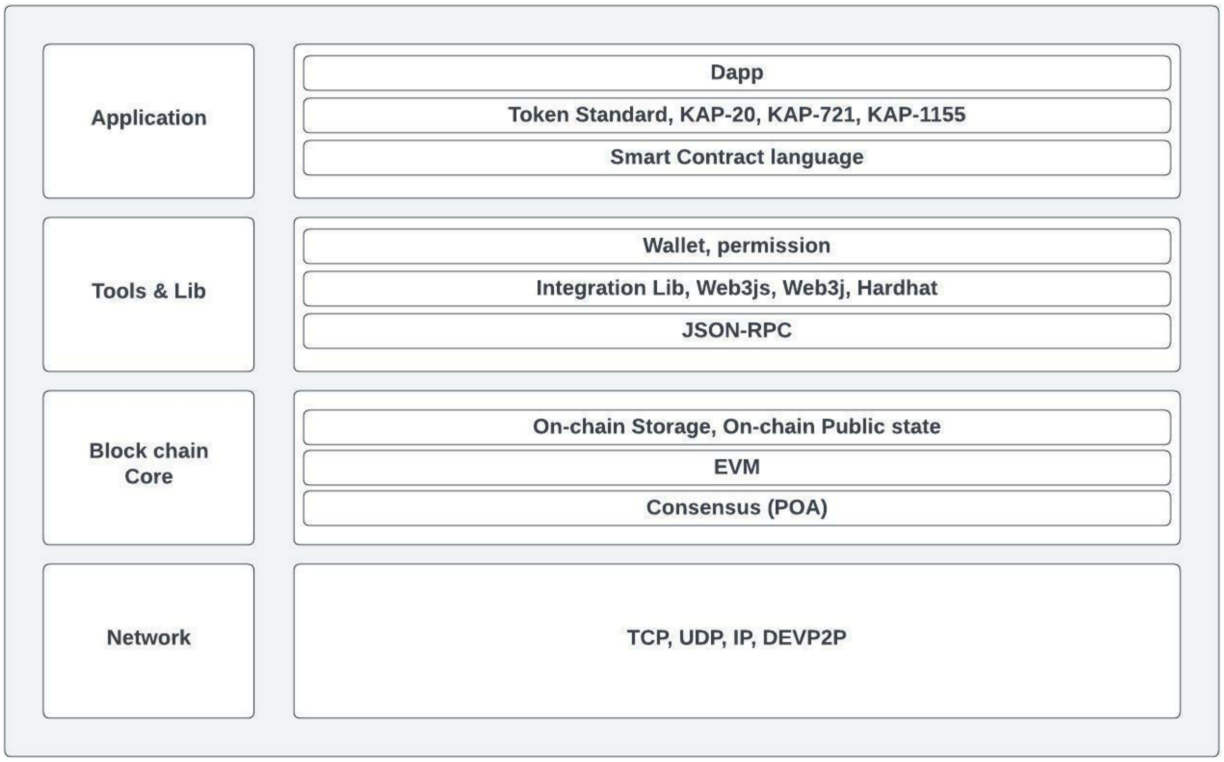


Figure-15: the layer of the chain

10.1.1) Network layer

is the foundation of blockchain because it consists of all components required to build other higher layers. Generally, DEVP2P [27] uses UDP to discover another node and TCP to make peer-to-peer connections.

10.1.2) Blockchain core

is the core component of blockchain. It is the source of consensus. After combining with EVM (Execution module) and on-chain data storage, it will become the concept of the blockchain paradigm (see section 2)

10.1.3) Tools & library

facilitate developers to develop applications on the chain more easily. For example, Web3 can provide ease of programming and connection. This tool connects web applications and the blockchain.

10.1.4) Application

can refer to a programming language, token standards, and decentralized application. The contract is nothing but byte code running on every node verified by the validator nodes.

10.2) Network Architecture

is the network structure of the blockchain connecting peer-to-peer. However, there is more than one type of node to create a blockchain service. Therefore, we need four kinds of nodes (see Figure-17).

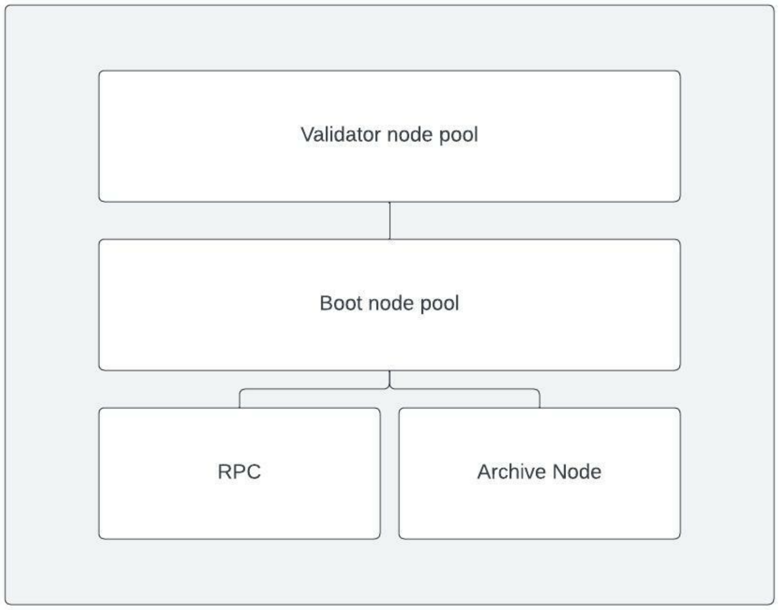


Figure-16: high-level architecture

10.2.1) Validator node pool

is a series of validator nodes. The primary function is to verify transactions and then create the block. This type of node is an essential part of the blockchain system. The chain will stop working if the number of running validator nodes is less than 50% of all validator nodes. Voting from the existing validator node is required to add a new validator node to the pool.

10.2.2) Boot node pool
is a bunch of boot nodes that support the discovered protocol to search other nodes. Boot nodes are like a relayer helping each node to find other nodes on the same network via UDP protocol.

10.2.3) Remote procedure call (RPC) Node
is to communicate between blockchain networks and applications. Everyone in the public domain can run the RPC node independently without permission from authorized parties.

10.2.4) Archive Node
collects all blockchain headers and contents. In a normal RPC node, it keeps only the last 128 blocks. However, the archive node collects the whole on-chain data.

11. The Advantages of Bitkub Chain

11.1) Blockchain Layer

11.1.1) Block size

The block size of Bitkub Chain is currently set to 60,000,000 gas. However, it could be increased in the future, which is more than some other chains, such as the Ethereum blockchain, which sets the limit to 30,000,000 gas. The higher limit of gas per block means higher number of transactions can be included in a block. However, the larger block size will also increase the storage cost and require a higher hardware specification to run a blockchain node.

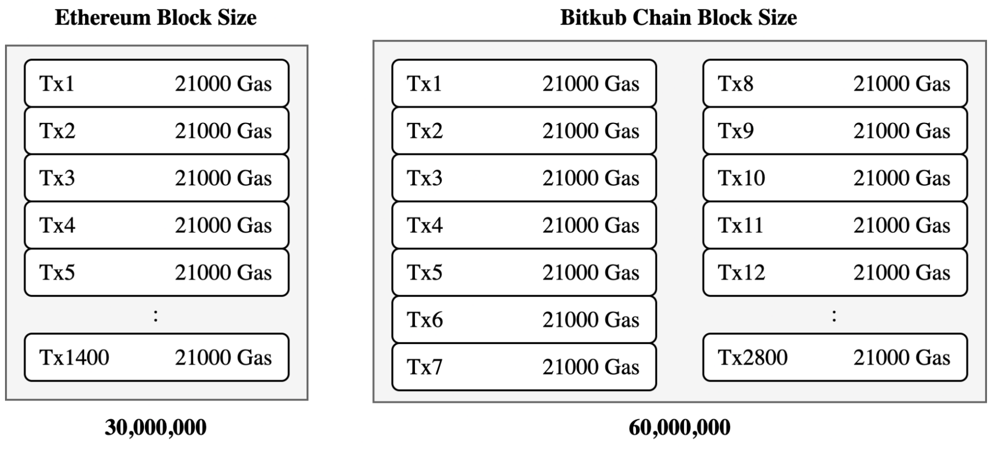


Figure-17: Bitkub Chain block size

11.1.2) Block time

According to chapter 9 (Block period), Bitkub Chain's system is fixed to 5 seconds which is faster than the Ethereum network (on average). Furthermore, the time variation is extremely low due to the low latency of Bitkub Chain's network. The speed of a blockchain can be calculated from block size and block time in terms of TPB (transactions per block).

11.1.3) Transaction fees

The high transaction fees are one of the pain points on the big primitive blockchains such as Bitcoin or Ethereum. The sender might have to pay fees worth over a dollar to send a token worth a dollar. There are many reasons that cause high transaction fees like this, such as coin price, gas price per unit, and the amount of gas used by a transaction. The coin price is out of control because it depends on the market. But the gas price depends on the crypto-economic model. Ethereum uses the dynamic gas price mechanism based on network congestion. If the network is congested, the gas price and the transaction fees will be higher. However, Bitkub Chain uses a different mechanism called a fixed gas price mechanism that will stabilize the transaction fees. Bitkub Chain sets the gas price to 5 GBit (see chapter 9), which is suitable for many business use cases.

11.1.4) Infrastructure design

All validator nodes on the Bitkub Chain agree to place their node in the corresponding standard. The environment has to meet our minimum criteria. It was designed to cover all of the regions to maximize the load or network connection speed, including CPU speed and memory. Each node connects to the other nodes via an internal-link network shown in Figure-18.

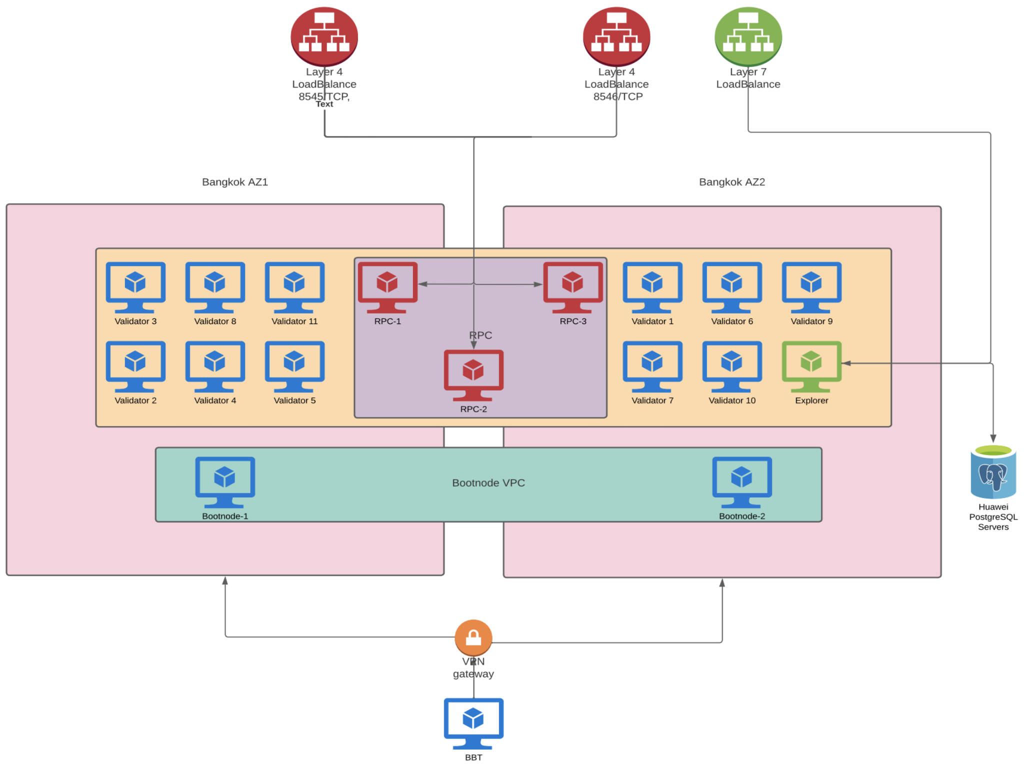


Figure-18: Network architecture

When the validators are in the corresponding environment, the bandwidth between nodes is almost unlimited because of the optical fiber network. It connects point-to-point to form a high-speed network. In the future, this standard will support the scalability of the Bitkub chain.

11.1.5) The comparison of Bitkub Chain

Chain	**Bitkub Chain**	**Ethereum**	**Huobi ECO Chain (HECO)**
Consensus	Proof-of-Stake-Authority	Proof-of-Work	Proof-of-Stake-Authority
Consensus Algorithm	Clique	Ethash	Congress
Execution Engine	EVM	EVM	EVM
Smart Contract Language	Solidity, Vyper	Solidity, Vyper	Solidity, Vyper
Transaction Per Block (TPB)	2800	1400	1900
Transaction Fees model	Fixed-rate	Variable-rate	Variable-rate
Block Time (second)	5	~15	3
Native Currency	KUB	ETH	HT

Table-04: The comparison of Bitkub Chain

11.2) Application Layer

11.2.1) Registration

Register Smart Contract (Appendix I) is used on Bitkub Chain for registering user's addresses. Moreover, it controls the access to the Bitkub NEXT.

a) Registration levels

i) Level 0 represents the unregistered addresses.

ii) Level 2 represents the primary registered address. This level will unlock the Unwrap KKUB feature.

iii) Level 4 represents the secondary registered address. This level will unlock additional features such as transfer Token or NFT.

iv) Level 16 represents the advanced registered address. This level will unlock the specific smart contract feature.

b) Register procedure

i) Registration

- Register via https://kkub-otp.bitkubchain.com by using a phone number, and their wallet address will be assigned to level 2
- Register via https://accounts.bitkubnext.com/register by using phone number, email address, and password will be assigned to level 4

Once registered successfully, the backend service will call the **batchSetKycCompleted** function on the smart contract to record the level on the blockchain.

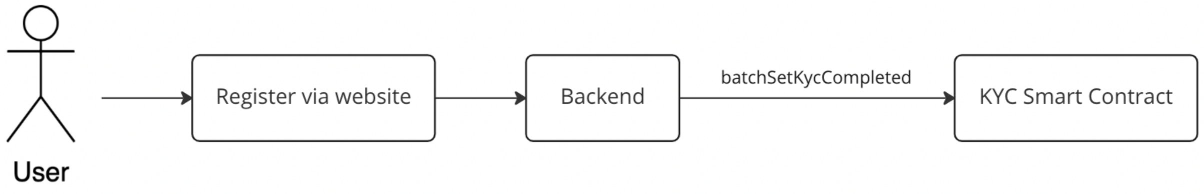


Figure-19: Registration flow

ii) Verify: All official and verified projects must verify the level of user's registration on Bitkub NEXT by calling **kycLevels** function in the smart contract.

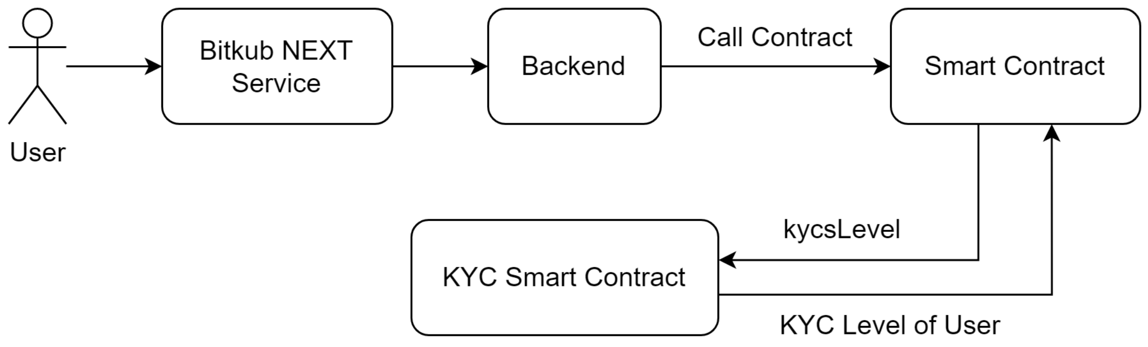


Figure-20: Verification flow

11.2.2) KAP token standard

Bitkub Chain has its token standards called KAP, which is inspired by the Ethereum Request for Comments (ERC) standard. The standard for fungible tokens is KAP-20 (see Appendix J), and the standard for non-fungible tokens is KAP-721 (see Appendix J). The main difference between ERC-20 and KAP-20 is the adminTransfer function. This standard is useful when the rug pull occurs because an admin of the token can transfer the token back from the attacker's address. Table-05 is the illustration of the difference between ERC-20 and KAP-20.

	ERC-20	KAP-20
Events	Transfer(from, to, value)	Transfer(from, to, tokens)
	Approval(owner, spender, value)	Approval(tokenOwner, spender, tokens)
Functions	totalSupply()	totalSupply()
	balanceOf(account)	balanceOf(account)
	allowance(owner, spender)	allowance(owner, spender)
	approve(spender, amount)	approve(spender, amount)
	transfer(to, amount)	transfer(to, amount)
	transferFrom(from, to, amount)	transferFrom(from, to, amount)
		getOwner()
		batchTransfer(from, to[], value[])
		adminTransfer(from, to, value)

Table-05: The comparison of ERC-20 and KAP-20

a) **adminTransfer**

adminTransfer is a KAP-specific function that can help the victims by recovering their tokens from the suspicious wallet in the case of a fraudulent. The function shall be used by a relevant issuer of tokens developed on KAP-20 standard.

The improvement of adminTransfer function

The latest upgrade on the adminTransfer function was done to make the technical process more secure and trustworthy. The significant difference between the previous version and the new version is the committee's ability to transfer tokens. Instead of having the full control of the tokens movement, the committee under the new version of adminTransfer has restricted ability to transfer tokens from the suspicious wallet to only one address, i.e. a recovery wallet which shall be specifically created for such alleged fraudulent transaction and managed by the Committee Smart Contract. This ensures that the suspicious tokens are frozen during the verification process. To clearly illustrate, the distinction between the previous version and the new version is shown in Table-06.

Actions	adminTransfer (Old)	adminTransfer (New)
The ability to move tokens **from** the suspicious wallet	Yes	Yes
The ability to move tokens **to** any wallet	Yes (According to the relevant policy, the committee will move the tokens to the designated address instructed by the proven victims or the government authorities (as the case may be).)	No (Only to a recovery wallet or related KYC wallet)

Table-06: The distinction between the previous version and the new version

How does adminTransfer works

1. Wallet B is suspected of the token fraud.
2. The victim sends an adminTransfer usage request to the Execution Committee. Alongside the request, the following evidence should be attached: (a) a police report and/or written request from the government authority, (b) on-chain sequences of transaction hashes that correspond to the fraud between Wallet A and Wallet B.
3. The execution committee executes a function on the Committee Smart Contract and passes the related data of the wallets of the suspicious wallet (Wallet B) and the victim's wallet.
   3.1 The system stores the related data (Txid, wallet address, amount) of the wallets of the suspicious wallet (Wallet B) and the victim's wallet.
   3.2 The system automatically creates a new wallet (Recovery Wallet).
4. The Committee Smart Contract executes an adminTransfer function to force transferring the tokens in question from a suspicious wallet (Wallet B).

5. The tokens have been transferred from Wallet B to be locked in the Recovery Wallet.
6. In order to recover the locked tokens, the victim has to create a new wallet (Wallet C) and re-perform a KYC process.
   6.1 The system generates a cryptographic hash from the KYC data using a keccak256 hashing function (this hash will represent the identity of a victim).
   6.2 The system stores a hash on-chain.
7. The victim sends a writ, a court or tribunal order or judgment, and/or an order or request of any government or supervisory authorities (as the case may be) alongside new wallet (Wallet C) address to the Bitkub Chain operation team (in case of KKUB) or the relevant operation team of the issuer of any other KAP-20 based token.
8. The execution committee executes a function to transfer the locked tokens to a new wallet (Wallet C).
   8.1 The system verifies the previous hash and the current hash of a victim (which must be matched).
   8.2 The Committee Smart Contract executes a transfer function.
9. The locked tokens has been transferred from the Recovery Wallet to the new wallet (Wallet C) or transferred back to Wallet B in the case that the alleged transaction has been proven not a fraud pursuant to a writ, a court or tribunal order or judgment, and/or an order or request of any government or supervisory authorities (as the case may be).

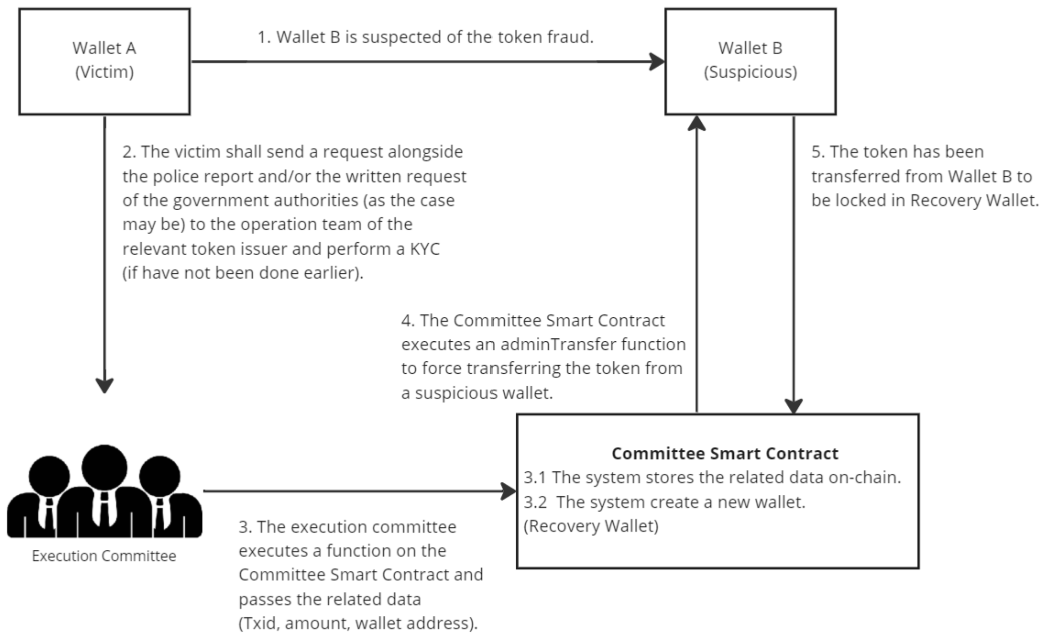

Wallet A
(Victim)
1. Wallet B is suspected of the token fraud.
Wallet B
(Suspicious)
2. The victim shall send a request alongside the police report and/or the written request of the government authorities (as the case may be) to the operation team of the relevant token issuer and perform a KYC (if have not been done earlier).
5. The token has been transferred from Wallet B to be locked in Recovery Wallet.
4. The Committee Smart Contract executes an adminTransfer function to force transferring the token from a suspicious wallet.
Committee Smart Contract
3.1 The system stores the related data on-chain.
3.2 The system create a new wallet. (Recovery Wallet)
Execution Committee
3. The execution committee executes a function on the Committee Smart Contract and passes the related data (Txid, amount, wallet address).

Figure-21: adminTransfer Process (1)

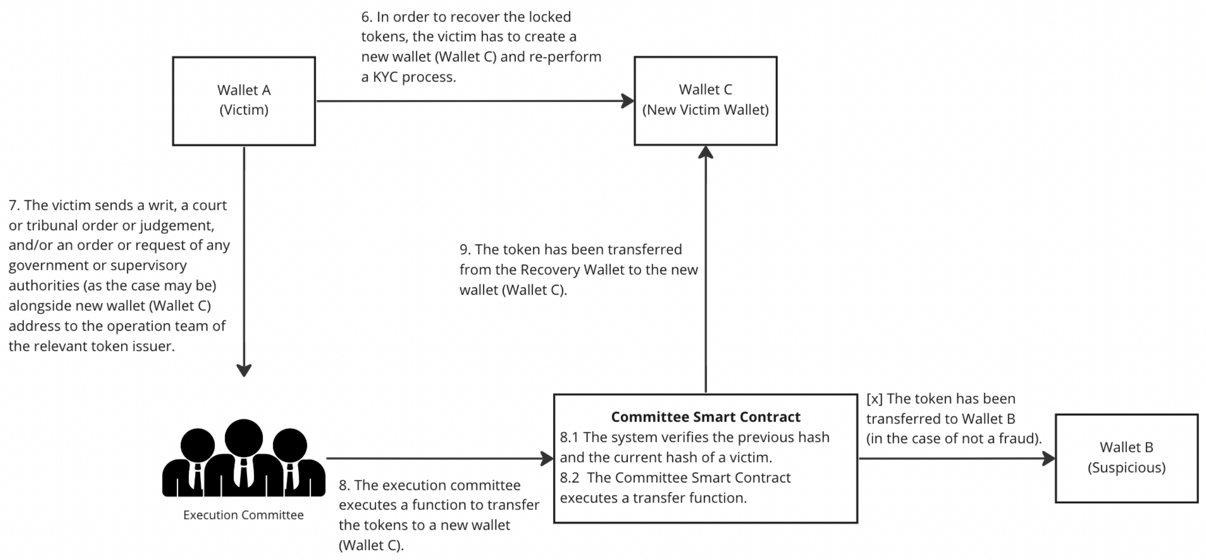

6. In order to recover the locked tokens, the victim has to create a new wallet (Wallet C) and re-perform a KYC process.
Wallet A
(Victim)
Wallet C
(New Victim Wallet)
7. The victim sends a writ, a court or tribunal order or judgement, and/or an order or request of any government or supervisory authorities (as the case may be) alongside new wallet (Wallet C) address to the operation team of the relevant token issuer.
9. The token has been transferred from the Recovery Wallet to the new wallet (Wallet C).
Committee Smart Contract
8.1 The system verifies the previous hash and the current hash of a victim.
8.2 The Committee Smart Contract executes a transfer function.
[x] The token has been transferred to Wallet B (in the case of not a fraud).
Wallet B
(Suspicious)
Execution Committee
8. The execution committee executes a function to transfer the tokens to a new wallet (Wallet C).

Figure-22: adminTransfer Process (2)

Note: If any person entitled to the locked tokens pursuant to a writ, a court or tribunal order or judgement, and/or an order or request of any government or supervisory authorities is deceased or unable to perform the KYC by himself/herself, e.g. a disabled person or incompetent person etc, the locked tokens shall be released to or for the benefit of the entitled person or, in case of the dead, his/her administrator only when (i) the aforementioned evidence of such person's entitlement and a court order appointing an administrator, a curator or a guardian (as the case may be) of such entitled person are present to the Bitkub Chain team (in case of KKUB) or the relevant issuer of any other KAP-20 based token; and (ii) such administrator, curator or guardian (as the case may be) conducts KYC. In such case, the operation team of the Bitkub Chain (in case of KKUB) or the relevant issuer of such other KAP-20 based token will revise the hash of the entitled person's original wallet (with traceable changes) and requests such administrator, curator or guardian (as the case may be) of such entitled person to create a new wallet and conduct KYC under his/her name such that the locked tokens of the entitled person can be transferred to the new wallet created by such administrator, curator or guardian (as the case may be) of the entitled person.

b) **batchTransfer**

The batch transfer is a KAP-specific function for transferring tokens to multiple wallets at the same time.

How does batchTransfer works

1. Collect multiple token transfer transaction data
2. Batch them together into a transaction
3. Send a batchTransfer transaction
4. Token transferred to the multiple wallets

The advantages of batchTransfer

1. Cost efficiency: Paying gas only once instead of multiple times.
2. Time reduction: Send and wait only a transaction instead of wait multiple transactions

   For example: sending a token to the 100 wallets without the batchTransfer function you have to send a token 100 times, make 100 transactions, spend enormously gas. On the other hand, utilizing a batchTransfer function can dramatically save your gas by sending just a single transaction and paying gas only once.

c) **getOwner**
   Get a KYC wallet address of an owner of the smart contract

11.2.3) KKUB

is a token pegged to the KUB coin and conforms to the KAP-20 token standard. This process can easily convert KUB into KKUB through a process known as "wrapping" by depositing KUB coins to the smart contract. Then KKUB will be minted in the same amount as the deposit amount. or convert KKUB to KUB through a process known as "unwrapping" by withdrawing the KKUB; it will burn the KKUB and send the KUB back to the user. The advantage of KKUB (the KAP-20 version of the KUB coin) is the ability to integrate with Decentralized-Finance (De-Fi) or Decentralized Application (DApp), which is programmable through the smart contract. The smart contract's code was written in solidity language (Appendix H).

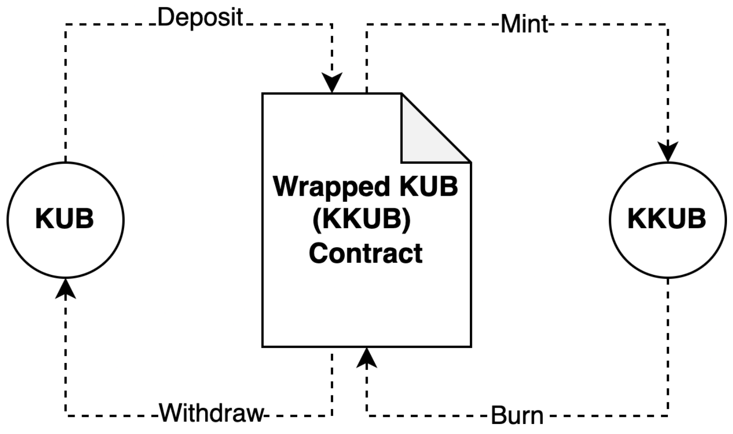


Figure-23: Wrap & Unwrap process of KKUB

11.2.4) HyperBlock

is an algorithm to aggregate transactions from many wallet addresses on the Bitkub Chain network by making a call to a smart contract (must contain a batchTransfer function). For example, if A wants to send a token to B, C, D, and E, A will send only one transaction as a batch transfer (function on smart contract) instant of sending four separate transactions. This process can increase the speed of transactions by over **1,000** transactions per second (TPS). HyperBlock consists of four parts:

a) **Backend Service**
is responsible for creating a backed transaction pool where an available thread runs on a background process and matches them with the admin pool addresses. The transactions matched with a specific admin address will send out to the smart contract address. The process can be explained below (Figure-24).

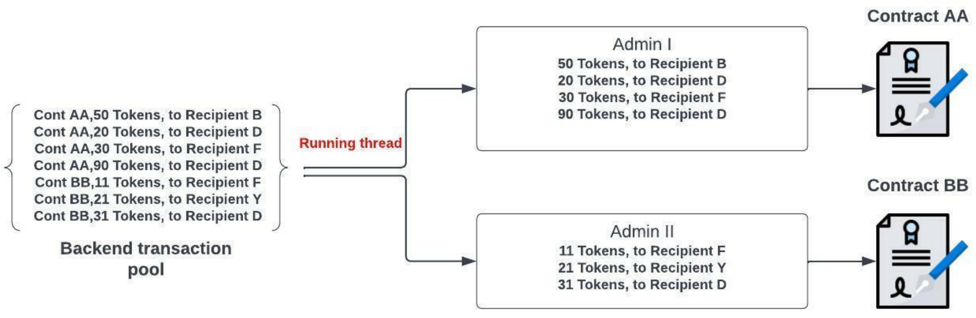

Cont AA,50 Tokens, to Recipient B
Cont AA,20 Tokens, to Recipient D
Cont AA,30 Tokens, to Recipient F
Cont AA,90 Tokens, to Recipient D
Cont BB,11 Tokens, to Recipient F
Cont BB,21 Tokens, to Recipient Y
Cont BB,31 Tokens, to Recipient D
Backend transaction pool
Running thread
Admin I
50 Tokens, to Recipient B
20 Tokens, to Recipient D
30 Tokens, to Recipient F
90 Tokens, to Recipient D
Admin II
11 Tokens, to Recipient F
21 Tokens, to Recipient Y
31 Tokens, to Recipient D
Contract AA
Contract BB

Figure-24 HyperBlock (backend service flow)

b) **Gas Tank**
is wallet addresses. Those addresses will pay the transaction fees (gas) for the users in Bitkub's official project. Therefore, the users do not need to store KUB coins in their wallets as long as they are under the Bitkub ecosystem (see Appendix J) such as Bitkub NEXT. In addition, the users will not be affected by the delay caused by changing the gasPrice because the Gas tank will always guarantee a reasonable gasPrice rate to maintain constant transaction speed.

c) **Smart Contract**
According to the KAP token standard (see section 11.2.2), batchTransfer function (Appendix H) is called by the admin from the pool to send a batch transaction. Without this function, Hyperblock will not work. All developers working on the Bitkub Chain's official projects must implement the KAP Token standard.

d) Fail-Safe mechanism

Normally, when a transaction fails to execute on the blockchain, the sender needs to manually re-sent it by themselves. However, Bitkub Chain has provided the mechanism to automatically re-sent the failed transaction to maximize the user experience. The Fail-Safe mechanism works by using an off-chain engine to detect the failed transaction in the transaction pool and trying to diagnose the root cause of the failure, and then trying to re-sent it. For example, the users might send a transaction with a low gas price relative to the determined gas price by the blockchain, which will cause the failure. In this case, the Fail-Safe mechanism will automatically increase the gas price and re-sent.

To summarize, HyperBlock is an engine that combines different modules and various techniques to maximize the ability to send the transaction to the Bitkub Chain. The **backend module** (build-up off-chain) is responsible for orchestrating the list of transactions requested by the users and then serving them to the **smart contract module** (build-up on-chain) as a batch of transactions. This technique practically improves the number of transactions (token transfers) to more than one thousand per second. Moreover, the **gas tank module** (build-up off-chain) will facilitate user interaction with the Bitkub Chain without paying the transaction fees by themselves (only Bitkub Chain official projects) to maximize the user experience and eliminate the barrier to using the Bitkub Chain. The overview of system flow can be shown in Figure-25.

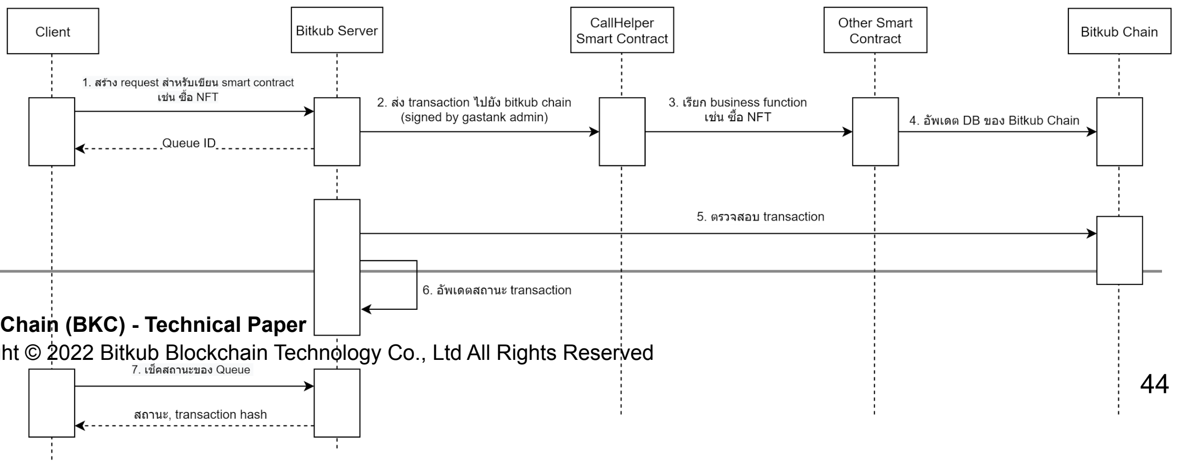

Client
Bitkub Server
CallHelper Smart Contract
Other Smart Contract
Bitkub Chain
1. สร้าง request สำหรับเขียน smart contract เช่น ซื้อ NFT
Queue ID
2. ส่ง transaction ไปยัง bitkub chain (signed by gastank admin)
3. เรียก business function เช่น ซื้อ NFT
4. อัพเดต DB ของ Bitkub Chain
5. ตรวจสอบ transaction
6. อัพเดตสถานะ transaction
7. เช็คสถานะของ Queue
สถานะ, transaction hash

Figure-25: HyperBlock Overview

12. Future Technical Goals

12.1) Scalability

12.1.1) Layer1 Scaling

Scalability is a major challenge in blockchain systems when the number of nodes and transactions increases. Thus, a Layer1 scaling technique becomes the primary step to solve this challenge and increase the block size.

a) Increasing block size

To scale up the limit of transaction Bitkub Chain plans to increase the block gas limit to more than 60,000,000 gas in the future. However, increasing block size is one of the Layer1 scaling techniques which helps enlarger block sizes and accommodate more transactions per block. However, this technique is simple, but it may come with many risks.

i) Increased block size will lead to the need to expand the storage size of the node as well as to upgrade the disk type to be more efficient. As a result, the cost of running the node increases.

ii) Increased block synchronization time As the number of data increases, it takes longer to download and verify the historical data.

iii) Increased the number of uncle block or chain reorganization as the increase in block size results in longer block execution time and longer broadcast block time to other nodes.

iv) Increased possibility to attack, such as an attack in the form of Denial-of-Service (DDoS). The larger the attackers could send the block size, the more malicious transactions. The attack may cause the network to halt.

b) Sharding

Sharding is one of the layer1 scaling solutions developed and introduced by Vitalik Buterin [15] Sharding is a technique that

upgrades to improve the scalability and capacity of a blockchain network by splitting a blockchain into smaller partitions to increase the capability to spread the load and process more transactions. This solution will help the system improve efficiency and be able to handle more than a thousand transactions per second.
This sharding concept divides a chain into many sub-chains known as "shard chains" each shard comprises its data and consists of a unique set of validators called "committees". Each committee will be assigned to verify each block broadcasted on a different set, which it could run altogether in parallel. Once all blocks are executed and validated, validators will attest to the fact and sign a signature to verify that the block is valid. Hence, it added speed and accuracy to the verification process.
There are two types of transaction validation on Sharding; a) the validity of the processing (computation) and b) the availability of the data (data availability).

i) Validating computation
There are two methods of validating computation: Fraud proofs and ZK-SNARKs.

- **Fraud-proofs**
  By default, a system will accept the result of computation but still leaves open the opportunity for someone else with a staked deposit to make a challenge when they find or want to argue that the processing is invalid. However, they can challenge it by depositing tokens as collateral for proof.

- **ZK-SNARKs**
  use the principle of cryptographic proof to perform computation and prove that all processing results are correct, but this method is considerably more complex and complicated to develop compared to fraud-proof.

ii) Validating Data availability
Verifying data's existence is more difficult when compared to verifying processing validity. It is impossible to distinguish who was right and wrong makes it impossible to have a

working fraud-proof scheme for data availability. Thus, It cannot use the same techniques such as fraud-proof or ZK-SNARKs but instead have to use the data availability sampling technique. Hence, instead of extracting all the information that we gathered for verification. We will use a random method to check some chucks of the data instead to improve slow performance. [23]

12.1.2) Layer2 Scaling

Layer2 Scaling is a solution designed to help increase scalability for handling more transactions off-chain. By executing it outside (off-chain) of the main chain (Layer1), it can be done by many techniques such as Roll-up, Validium, or Plasma.

a) Rollups

i) Optimistic Rollups: Optimistic Rollups is a technique in which transactions are processed at Layer2, then bundled into a series of transactions (batch), and then the data is posted (roll-up) to Layer 1. Optimistic Rollup uses a technique called fraud-proof to verify the authenticity and correctness of transactions. Fraud-proof will have the disadvantage of having a window of time to allow anyone to prove and challenge the frauds. The current standard delay was seven days.

ii) ZK-Rollups: ZK-Rollup: Zero Knowledge-Rollup is a similar concept to "Optimistic Rollups" but uses a different cryptographic technique called "Zero-Knowledge Proof" to validate transactions. Before it gets posted to Layer1, this technique can prove transactions without the need for a challenging period (7-day delay) like "Optimistic Rollups". However, this solution may increase the level of difficulty and create complexity in the development process compared to the "Optimistic Roll Ups" technique.

b) Plasma

Plasma is a layer2 scaling solution that is built on top of the root chain (parent chain). The transaction from the parent chain will be off-loaded and executed by the Plasma chain (child chain) while maintaining the validity of the transaction state by using the fraud proof mechanism [24]. The child chain has to commit the child chain state periodically by submitting the Merkle roots of the state of the child chain to the Plasma contract deployed on-chain. The users can enter the plasma chain by bridging their token from the parent chain to the child chain via bridge smart contract and can exit the child chain by withdrawing the token back to the parent chain via bridge. However, withdrawing the token to Layer1 needs a delay time for the fraud proof which is normally set to 7 days.

c) Validium

Validiums is a Layer2 scaling solution that processes and stores the state off-chain only the proof that is stored on-chain (Layer1). Validium is like ZK-Rollups but the data availability of the Validium is stored off-chain rather than on-chain. Validiums can process ~9,000 transactions (or more) per second [24] higher than any other Layer2 solutions because of the off-chain operation. However, storing and executing data off-chain has a high risk and requires trust of the operator. To mitigate the risk validium uses the “Validity proofs” that can be computed and generated by using both ZK-SNARKs and ZK-STARKs to prove that the off-chain state is valid and submit it to the on-chain (Layer1) as a state commitment and finalization.

12.2) Consensus Improvement

11.2.1) Proof-of-Stake + 3

Proof-of-Stake plus three (PoS+3) is a consensus mechanism that will reward three validators instead of only a validator compared with normal Proof-of-Stake. The reward will be split into 3 levels, the in-round (actual) validator will receive the most reward. The PoS+3 system will select the top three validators based on the amount of KUB and the last recent time of reward. This mechanism will help reduce the capitalism problems and increase the incentive to validators to stay active on secure Bitkub Chain. While we still keep the core feature of the standard Proof-of-Stake. In the PoS to achieve distributed consensus. It requires users to stake their KUB coins to become validators in the network. The validators are responsible for the same tasks as validators in Proof-of-Authority (PoA) and the Proof-of-Stake-Authority (PoSA), which include ordering transactions, validating the validity of transactions, and creating new blocks. So that all nodes can agree on the same state of the Bitkub Chain network. PoS+3 have many of advantages includes:

a) **Better incentivized (reward) system** than normal Proof-of-Stake, the validators have more chances to get the block reward.

b) **Lower barriers** are required to promote a new validator, it doesn't require submitting the request to existing committees to allow a new joining before entering the network as a validator. Instead, it needed a stake of at least a minimum number of KUB to become a validator, which means in the Proof-of-Stake consensus, anyone can be a validator of the Bitkub Chain.

c) **Stronger immunity** to centralization – in Proof-of-Stake consensus doesn't require hardware devices like Proof-of-Work and doesn't require authorization like proof-of-authority. Becoming a validator requires only a chuck of KUB coins to stake. It would make it more accessible to be a validator, and it's easier to decentralize the network.
Proof-of-Stake is like any other consensus; it can have a malicious node. In this case, proof-of-stake introduced a mechanism called "Slashing." It helps prevent the deliberately misbehaving of validators [11]. The concept of slashing is to remove some part of the validator stake or the whole stake in the worst case.

12.3) Bitkub Chain SDK (SDK)

The developer is an essential actor in the decentralized network. However, creating or developing a decentralized application requires specific technical knowledge and tools. Bitkub Chain wants to encourage the developer to build on our ecosystem by introducing the Bitkub Chain software development kit (SDK), a development tool. The SDK supports developers in developing a project. Currently, we already publish the Bitkub Chain Javascript SDK [25] any blockchain developer can easily install by just using the npm or yarn. However, our team is developing the SDK in the different programming languages to maximize the development tools to develop the DApp on Bitkub Chain.

12.4) On-chain static page

Hosting static pages on the blockchain combines the encoding technique and the on-chain state together. Blockchain is used as a data layer to store the static page detail and a specific smart contract is used as an encoder and decoder module to encode the static page source code to the specific format of data and then pack the data and store on-chain. The on-chain static page will be distributed among the blockchain node making the static page available for everyone with high availability (HA) and publicly accessible by everyone in the world without censorship from the centralized authority (censorship-resistance).

12.5) Verifying standard bytecode

However, many smart contracts had been attacked and the asset in the contract because there is a bug in the code even though they are audited by the auditors. One of the reasons that causes the bug is the additional logic and functions that implemented add-on from the standard function. To prevent or reduce this kind of vulnerability in the Bitkub Chain ecosystem, we introduced the "Verifying Standard Bytecode" feature to verify the inheritance of the standard from the smart contract by comparing the bytecode of the contract with the bytecode of the standard contract.

13. Conclusion

Bitkub Chain was created to democratize the opportunity for everyone to access a new economy (digital asset economy) that gives the right to innovate, design, and develop under a decentralized infrastructure, and encourages the decentralization and ownership of digital assets. In addition, Bitkub Chain technically derived some of the technology from Ethereum, resulting in the same standards, such as EVM compatibility, making it easier for developers to create and develop applications on top of Bitkub Chain. Moreover, the faster block times, lower fees, and a modern consensus system makes users send faster transactions and have a better user experience.

References

[1] Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain
[2] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. Fast money grows on trees, not chains, 2013.
[3] Ethereum Whitepaper https://ethereum.org/en/whitepaper/
[4] Ethereum Development Documentation https://ethereum.org/en/developers/docs/
[5] How does Ethereum work, anyway
https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway
[6] Merkle tree and Ethereum objects
https://medium.com/coinmonks/detailed-explanation-of-ethereum-yellow-paper-merkle-tree-and-ethereum-objects-d85edf5051b8
[7] Merkling in Ethereum https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/
[8] Modified Merkle Patricia Trie — How Ethereum saves a state
https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd
[9] Proof of Authority Explained
https://academy.binance.com/en/articles/proof-of-authority-explained
[10] EIP-225: Clique proof-of-authority consensus protocol
https://eips.ethereum.org/EIPS/eip-225
[11] Proof-of-stake (PoS) definition
https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/
[12] Layer2 Scaling https://ethereum.org/en/developers/docs/scaling/
[13] Diving Into The Ethereum Virtual Machine
https://blog.qtum.org/diving-into-the-ethereum-vm-6e8d5d2f3c30
[14] Ethereum State Trie Architecture Explained
https://medium.com/@eiki1212/ethereum-state-trie-architecture-explained-a30237009d4e
[15] Why sharding is great: demystifying the technical properties
https://vitalik.ca/general/2021/04/07/sharding.html
[16] The Limits to Blockchain Scalability https://vitalik.ca/general/2021/05/23/scaling.html
[17] Why Proof-of-Work Is Not Viable in the Long-Term
https://medium.com/logos-network/why-proof-of-work-is-not-viable-in-the-long-term-dd96d2775e99
[18] The Ethereum Virtual Machine https://faun.pub/the-ethereum-virtual-machine-d70dfa5f045b
[19] Mastering Ethereum https://cypherpunks-core.github.io/ethereumbook/
[20] Ethereum EVM illustrated
https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf
[21] Go-Ethereum https://github.com/ethereum/go-ethereum

[22] The Central Limit Theorem and Means https://www.statisticshowto.com/probability-and-statistics/normal-distributions/central-limit-theorem-definition-examples/
[23] An explanation of the sharding + DAS proposal https://hackmd.io/@vbuterin/sharding_proposal#ELI5-data-availability-sampling
[24] Scaling https://ethereum.org/en/developers/docs/scaling/
[25] Bitkub Chain JS SDK https://www.npmjs.com/package/@bitkub-blockchain/sdk
[26] Ethash https://ethereum.org/th/developers/docs/consensus-mechanisms/pow/mining-algorithms/ethash
[27] Devp2p https://github.com/ethereum/devp2p
[28] BKC Erawan Hardfork https://github.com/bitkub-blockchain/bkc/releases

Appendix A. Fee Schedule

The fee schedule G is a tuple of scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may affect.

Name	**Value**	**Description**
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
$G_{jumpdest}$	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{warmaccess}$	100	Cost of a warm account or storage access.
$G_{accesslistaddress}$	2400	Cost of warming up an account with the access list.
$G_{accessliststorage}$	1900	Cost of warming up a storage with the access list.
$G_{coldaccountaccess}$	2600	Cost of a cold account access.
$G_{coldsload}$	2100	Cost of a cold storage access.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the Homestead transition.

$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
$G_{keccak256}$	30	Paid for each KECCAK256 operation.
$G_{keccak256word}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for each BLOCKHASH operation.

Appendix B: Virtual Machine Specification

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv C_{\text{mem}}(\boldsymbol{\mu}'_{\text{i}}) - C_{\text{mem}}(\boldsymbol{\mu}_{\text{i}}) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SSTORE} \\ G_{\text{exp}} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{\text{exp}} + G_{\text{expbyte}} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{\text{verylow}} + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w \in W_{\text{copy}} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) & \text{if} \quad w \in W_{\text{extaccount}} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{\text{logtopic}} & \text{if} \quad w = \text{LOG1} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{\text{logtopic}} & \text{if} \quad w = \text{LOG2} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{\text{logtopic}} & \text{if} \quad w = \text{LOG3} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{\text{logtopic}} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{if} \quad w \in W_{\text{call}} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ G_{\text{create}} & \text{if} \quad w = \text{CREATE} \\ G_{\text{create}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CREATE2} \\ G_{\text{keccak256}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[1] \div 32 \rceil & \text{if} \quad w = \text{KECCAK256} \\ G_{\text{jumpdest}} & \text{if} \quad w = \text{JUMPDEST} \\ C_{\text{SLOAD}}(\boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SLOAD} \\ G_{\text{zero}} & \text{if} \quad w \in W_{\text{zero}} \\ G_{\text{base}} & \text{if} \quad w \in W_{\text{base}} \\ G_{\text{verylow}} & \text{if} \quad w \in W_{\text{verylow}} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{\text{mid}} & \text{if} \quad w \in W_{\text{mid}} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \\ G_{\text{blockhash}} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\text{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\text{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

$$C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

$$C_{\text{aaccess}}(x, A) \equiv \begin{cases} G_{\text{warmaccess}} & \text{if} \quad x \in A_{\mathbf{a}} \\ G_{\text{coldaccountaccess}} & \text{otherwise} \end{cases}$$

with C_{CALL}, $C_{\text{SELFDESTRUCT}}$, C_{SLOAD} and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:
W_{zero} = {STOP, RETURN, REVERT}

W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE,TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS}
$W_{verylow}$ = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR,CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}
W_{low} = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND, SELFBALANCE}
W_{mid} = {ADDMOD, MULMOD, JUMP}
W_{high} = {JUMPI}
W_{copy} = {CALLDATACOPY, CODECOPY, RETURNDATACOPY}
W_{call} = {CALL, CALLCODE, DELEGATECALL, STATICCALL}
$W_{extaccount}$ = {BALANCE, EXTCODESIZE, EXTCODEHASH}

Note the memory cost component, given as the product of G_{memory} and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, μi in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes. Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory
to be extended to the beginning of the range. μ0i is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

Note also that C_{mem} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

Appendix C: Substrate

Substrate A, is a set of information which accumulated during the transaction execution process. The information is a tuple that include self-destruct set (S), the log series (L), the set of touched accounts (T), the refund balance (R), the set of accessed account addresses (C) and the set of accessed storage keys (K).

$$\boxed{A = \text{Tuple}(S, L, T, R, C, K)}$$

Appendix D: The execution environment information

List of the execution environment information refer as the tuple I, include:

- Address of the account that owns the code that is executing (A)
- Address of the sender of the transaction that originated this execution (O)
- Address of the account that caused the code to execute which could be different from the original sender (S)
- Gas price of the transaction that originated this execution (P)
- Input data for the execution (D)
- Value (in Wei) passed to this account as part of the current execution (V)
- Machine code to be executed (B)
- Block header of the current block (H)
- Depth of the present message call or contract creation stack (E)
- The permission to modify the state. (W)

$$\boxed{I = \text{Tuple}(A, O, S, P, D, V, B, H. E. W)}$$

Appendix E: Staking Constant Variables

MINIMUM_STAKE = 250,000 KUB

Appendix F: PoSA Smart Contracts

1. PoSAContract

```
pragma solidity 0.8.0;

contract POSAContract is IPOSAContract, IKYC, Authorization, Committee, KYCHandler,
ReentrancyGuard {
 using EnumerableSetAddress for EnumerableSetAddress.AddressSet;

 event AdminKAP20RouterSet(
   address indexed oldAdminKAP20Router,
   address indexed newAdminKAP20Router,
   address indexed caller
 );
 event KKUBSet(address indexed oldKKUB, address indexed newKKUB, address indexed
caller);
 event POSABondSet(address indexed oldPOSABond, address indexed newPOSABond, address
indexed caller);
 event POSABondInfoSet(address indexed oldPOSABondInfo, address indexed
newPOSABondInfo, address indexed caller);
 event POSARewardCalculatorSet(
   address indexed oldPOSARewardCalculator,
   address indexed newPOSARewardCalculator,
   address indexed caller
 );
 event KYCSet(address indexed oldKYC, address indexed newKYC, address indexed
caller);
 event AcceptedKYCLevelSet(
   uint256 indexed oldAcceptedKYCLevel,
   uint256 indexed newAcceptedKYCLevel,
   address indexed caller
 );

 uint256 public hardcap; // expected to be X KUB
 uint256 public totalDeposited;
 uint256 public totalClaimed;
 uint256 public maximumDepositPerUser; // X KUB / user
 uint256 public endDepositDate;
 uint256 public withdrawDate;

 Flag public paused;
 IPOSABond public POSABond;
 IPOSABondInfo public POSABondInfo;
 IPOSARewardCalculator public POSARewardCalculator;
 IAdminKAP20Router public adminKAP20Router;
 IKKUB public immutable KKUB;

 bool public metaMaskEnabled;
 bool public bitkubNextEnabled;

 mapping(address => uint256) public depositBalance;

EnumerableSetAddress.AddressSet private _whitelistLimit;
 uint256[] private _tokenIDs;

 modifier whenNotPausedDeposit() {
   require(!paused.depositPaused, "Pause: deposit paused");
   _;
 }

 modifier whenNotPausedWithdraw() {
   require(!paused.withdrawPaused, "Pause: withdraw paused");
   _;
 }

 modifier whenNotPausedClaim() {
   require(!paused.claimPaused, "Pause: claim paused");
   _;
 }

 modifier whenNotPausedEmergencyWithdraw() {
   require(!paused.emergencyWithdrawPaused, "Pause: emergency withdraw paused");
   _;
 }

 modifier checkBitkubNext(address _bitkubNext) {
   require(bitkubNextEnabled, "BitkubNext disabled");
   require(kyc.kycsLevel(_bitkubNext) >= acceptedKYCLevel, "only BitkubNext user");
   _;
 }

 modifier checkMetaMask(address _sender) {
   require(metaMaskEnabled, "Metamask disabled");
   require(_whitelistLimit.contains(_sender), "not in whitelist");
   _;
 }

 modifier checkDepositRequirement(uint256 _amount, address _sender) {
   require(_amount > 0, "require amount > 0");
   require(totalDeposited + _amount <= hardcap, "amount exceeds hardcap");
```

```
    require(block.timestamp < endDepositDate, "not in deposit period");
    _;
  }

 constructor(ConstructorInput memory _input) {
    require(_input.maximumDepositPerUser > 0, "require maximumDepositPerUser>0");
    require(_input.endDepositDate > block.timestamp, "require endDepositDate > now");
    require(_input.withdrawDate > _input.endDepositDate, "need
withdrawDate>endDepositDate");
    adminProjectRouter = IAdminProjectRouter(_input.adminInput.adminProjectRouter);
    adminKAP20Router = IAdminKAP20Router(_input.adminInput.adminKAP20Router);
    committee = _input.adminInput.committee;

    POSABond = IPOSABond(_input.bondInput.POSABond);
    POSABondInfo = IPOSABondInfo(_input.bondInput.POSABondInfo);
    POSARewardCalculator =
IPOSARewardCalculator(_input.bondInput.POSARewardCalculator);

    kyc = IKYCBitkubChain(_input.kycInput.kyc);
    acceptedKYCLevel = _input.kycInput.acceptedKYCLevel;

    KKUB = IKKUB(_input.KKUB);
    hardcap = _input.hardcap;
    maximumDepositPerUser = _input.maximumDepositPerUser;
    endDepositDate = _input.endDepositDate;
    withdrawDate = _input.withdrawDate;
    metaMaskEnabled = _input.enabled[0];
    bitkubNextEnabled = _input.enabled[1];

    paused.depositPaused = false;
    paused.withdrawPaused = false;
    paused.claimPaused = false;
    paused.emergencyWithdrawPaused = true;
  }

  ///////////////////////// read //////////////////////////

  function whitelistLimit() external view override returns (address[] memory) {
    return _whitelistLimit.getAll();
  }

  function whitelistLimitLength() external view override returns (uint256) {
    return _whitelistLimit.length();
  }

  function whitelistLimitByPage(uint256 _page, uint256 _limit) external view override
returns (address[] memory) {
    return _whitelistLimit.get(_page, _limit);
  }

  function tokenLength() external view returns (uint256) {
    return _tokenIDs.length;
  }

 function tokenIDByPage(uint256 _page, uint256 _limit) external view returns
(uint256[] memory) {
    require(_page > 0 && _limit > 0);
    uint256 tempLength = _limit;
    uint256 cursor = (_page - 1) * _limit;
    uint256 _uintLength = _tokenIDs.length;
    if (cursor >= _uintLength) {
      return new uint256[](0);
    }
    if (tempLength > _uintLength - cursor) {
      tempLength = _uintLength - cursor;
    }
    uint256[] memory uintList = new uint256[](tempLength);
    for (uint256 i = 0; i < tempLength; i++) {
      uintList[i] = _tokenIDs[cursor + i];
    }
    return uintList;
  }

  function pendingReward(uint256 _tokenID) public view override returns (uint256) {
    (uint256 amount, , , uint256 rewardClaimed, , address poolAddress, bool
activated) = POSABondInfo.bondInfo(
      _tokenID
    );
    require(poolAddress == address(this), "not bond of this pool");
    if (activated) {
      return _pendingReward(amount, rewardClaimed);
    } else {
      return 0;
    }
  }

  function _pendingReward(uint256 _amount, uint256 _rewardClaimed) internal view
returns (uint256) {
    return
      POSARewardCalculator.calculateReward(
        address(this).balance,
        _amount,
        _rewardClaimed,
        totalDeposited,
      totalClaimed
    );
  }

  function cumulativeReward() external view override returns (uint256) {
    return address(this).balance + totalClaimed;
 }

  ///////////////////////////////////////////////////////////

  //////////////////////// setter ////////////////////////

  function setAdminProjectRouter(address _adminProjectRouter) public override
onlyCommittee {
    require(_adminProjectRouter != address(0), "Authorization: new admin project
router is the zero address");
    emit AdminProjectRouterSet(address(adminProjectRouter), _adminProjectRouter,
msg.sender);
    adminProjectRouter = IAdminProjectRouter(_adminProjectRouter);
  }

  function setAdminKAP20Router(address _adminKAP20Router) external override
onlyCommittee {
    emit AdminKAP20RouterSet(address(adminKAP20Router), _adminKAP20Router,
msg.sender);
    adminKAP20Router = IAdminKAP20Router(_adminKAP20Router);
  }

  function setPOSABond(address _POSABond) external onlyCommittee {
    emit POSABondSet(address(POSABond), _POSABond, msg.sender);
    POSABond = IPOSABond(_POSABond);
  }

  function setPOSABondInfo(address _POSABondInfo) external onlyCommittee {
    emit POSABondInfoSet(address(POSABondInfo), _POSABondInfo, msg.sender);
    POSABondInfo = IPOSABondInfo(_POSABondInfo);
  }

  function setPOSARewardCalculator(address _POSARewardCalculator) external
onlyCommittee {
    emit POSARewardCalculatorSet(address(POSARewardCalculator),
_POSARewardCalculator, msg.sender);
    POSARewardCalculator = IPOSARewardCalculator(_POSARewardCalculator);
  }

  function setKYC(address _kyc) external override onlyCommittee {
    emit KYCSet(address(kyc), _kyc, msg.sender);
```

```
   kyc = IKYCBitkubChain(_kyc);
  }

  function setAcceptedKYCLevel(uint256 _acceptedKYCLevel) external override
onlyCommittee {
    emit AcceptedKYCLevelSet(acceptedKYCLevel, _acceptedKYCLevel, msg.sender);
    acceptedKYCLevel = _acceptedKYCLevel;
  }

  /////////////////////////////////////////////////////////

  function setHardcap(uint256 _hardcap) external override onlyCommittee {
    require(_hardcap > 0, "require hardcap > 0");
    require(_hardcap >= totalDeposited, "require hardcap>=totalDeposited");
    hardcap = _hardcap;
  }

  function setMaximumDepositPerUser(uint256 _maximumDepositPerUser) external override
onlyCommittee {
    require(_maximumDepositPerUser > 0, "require maximumDepositPerUser>0");
    maximumDepositPerUser = _maximumDepositPerUser;
  }

  function setEndDepositDate(uint256 _endDepositDate) external override onlyCommittee
{
    require(block.timestamp < endDepositDate, "require now < endDepositDate");
   require(_endDepositDate > 0, "require endDepositDate > 0");
    require(_endDepositDate < withdrawDate, "require endDeposit < withdraw");
    endDepositDate = _endDepositDate;
  }

  function setWithdrawDate(uint256 _withdrawDate) external override onlyCommittee {
    require(block.timestamp < withdrawDate, "require now < withdrawDate");
    require(_withdrawDate > endDepositDate, "require withdraw > endDeposit");
    withdrawDate = _withdrawDate;
  }

  /////////////////////////////////////////////////////////

  function setEnabled(bool[2] memory _enabled) external override onlyCommittee {
    metaMaskEnabled = _enabled[0];
    bitkubNextEnabled = _enabled[1];
  }

  function addWhitelistLimit(address _addr) external override onlySuperAdmin {
    require(_whitelistLimit.add(_addr), "can't add this address");
  }
```

```
function removeWhitelistLimit(address _addr) external override onlySuperAdmin {
  require(_whitelistLimit.remove(_addr), "can't remove this address");
}

//////////////////////////////////////////////////////////

function pause(Flag memory _paused) external override onlyCommittee {
  paused = _paused;
}

function emergencyWithdrawKUB(address _to, uint256 _amount)
  external
  override
  onlyCommittee
  whenNotPausedEmergencyWithdraw
{
  (bool success, ) = _to.call{ value: _amount }("");
  require(success, "unable to send value");
}

//////////////////////////////////////////////////////////

/////////////////////// deposit //////////////////////////

function deposit()
  external
  payable
  override
  nonReentrant
  checkMetaMask(msg.sender)
  checkDepositRequirement(msg.value, msg.sender)
{
  KKUB.deposit{ value: msg.value }();
  _deposit(msg.sender, msg.value);
}

function deposit(uint256 _amount)
  external
  override
  nonReentrant
  checkMetaMask(msg.sender)
  checkDepositRequirement(_amount, msg.sender)
{
  bool success = KKUB.transferFrom(msg.sender, address(this), _amount);
  require(success, "failed to transfer KKUB");
  _deposit(msg.sender, _amount);
```

```
 }

function deposit(uint256 _amount, address _bitkubNext)
   external
   override
   nonReentrant
   onlySuperAdmin
   checkBitkubNext(_bitkubNext)
   checkDepositRequirement(_amount, _bitkubNext)
 {
   if (!_whitelistLimit.contains(_bitkubNext)) {
     require(depositBalance[_bitkubNext] + _amount <= maximumDepositPerUser, "amount
exceeds maximum deposit");
   }
   adminKAP20Router.externalTransferKKUB(address(0), _bitkubNext, address(this),
_amount, 0);
   _deposit(_bitkubNext, _amount);
 }

 function _deposit(address _sender, uint256 _amount) internal whenNotPausedDeposit {
   uint256 tokenID = POSABond.mint(_sender, _amount);
   _tokenIDs.push(tokenID);
   depositBalance[_sender] += _amount;
   totalDeposited += _amount;
   emit Deposit(_sender, tokenID, _amount);
 }

 /////////////////////////////////////////////////////////

 /////////////////////// claim //////////////////////////

 function claim(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
   (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);

   _claim(msg.sender, _tokenID, amount, rewardClaimed);
 }

 function claim(uint256 _tokenID, address _bitkubNext)
   external
   override
   nonReentrant
   onlySuperAdmin
   checkBitkubNext(_bitkubNext)
 {
   (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);
```

```
    _claim(_bitkubNext, _tokenID, amount, rewardClaimed);
  }

  function _claim(
    address _sender,
    uint256 _tokenID,
   uint256 _amount,
    uint256 _rewardClaimed
  ) internal whenNotPausedClaim {
    require(block.timestamp >= endDepositDate, "require now >= endDepositDate");
    require(_sender == POSABond.ownerOf(_tokenID), "not owner");

    uint256 reward = _pendingReward(_amount, _rewardClaimed);
    POSABondInfo.updateClaimReward(_tokenID, reward);

    if (reward > 0) {
      totalClaimed += reward;
      _wrapRewardAndSendKKUB(_sender, reward);
    }
    emit ClaimReward(_sender, _tokenID, reward);
  }

  //////////////////////////////////////////////////////////

  /////////////////////// withdraw ////////////////////////

  function withdraw(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
    _withdraw(_tokenID, msg.sender);
  }

  function withdraw(uint256 _tokenID, address _bitkubNext)
    external
    override
    nonReentrant
    onlySuperAdmin
    checkBitkubNext(_bitkubNext)
  {
    _withdraw(_tokenID, _bitkubNext);
  }

  function _withdraw(uint256 _tokenID, address _sender) internal
whenNotPausedWithdraw {
    require(block.timestamp >= withdrawDate, "require now >= withdrawDate");

    (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);
    _claim(_sender, _tokenID, amount, rewardClaimed);

    POSABondInfo.updateWithdraw(_tokenID);
    _sendKKUB(_sender, amount);
    emit Withdraw(_sender, _tokenID, amount);
  }

  function emergencyWithdraw(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
    _emergencyWithdraw(_tokenID, msg.sender);
  }

  function emergencyWithdraw(uint256 _tokenID, address _bitkubNext)
    external
    override
    nonReentrant
    onlySuperAdmin
    checkBitkubNext(_bitkubNext)
  {
    _emergencyWithdraw(_tokenID, _bitkubNext);
  }

  function _emergencyWithdraw(uint256 _tokenID, address _sender) internal
whenNotPausedEmergencyWithdraw {
    require(_sender == POSABond.ownerOf(_tokenID), "not owner");

    (uint256 amount, ) = _loadBondInfo(_tokenID);

    POSABondInfo.updateWithdraw(_tokenID);
    _sendKKUB(_sender, amount);
    emit EmergencyWithdraw(_sender, _tokenID, amount);
  }

  //////////////////////////////////////////////////////////

  function _loadBondInfo(uint256 _tokenID) internal view returns (uint256, uint256) {
    (uint256 amount, , , uint256 rewardClaimed, , address poolAddress, ) =
POSABondInfo.bondInfo(_tokenID);
    require(poolAddress == address(this), "not bond of this pool");
    return (amount, rewardClaimed);
  }

  function _wrapRewardAndSendKKUB(address _addr, uint256 _amount) internal {
    KKUB.deposit{ value: _amount }();
    _sendKKUB(_addr, _amount);
  }
```

```
 function _sendKKUB(address _addr, uint256 _amount) internal {
   bool success = KKUB.transfer(_addr, _amount);
   require(success, "failed to transfer KKUB");
 }
}
a solidity 0.8.0;

contract POSAContract is IPOSAContract, IKYC, Authorization, Committee, KYCHandler,
ReentrancyGuard {
 using EnumerableSetAddress for EnumerableSetAddress.AddressSet;

 event AdminKAP20RouterSet(
   address indexed oldAdminKAP20Router,
   address indexed newAdminKAP20Router,
   address indexed caller
 );
 event KKUBSet(address indexed oldKKUB, address indexed newKKUB, address indexed
caller);
 event POSABondSet(address indexed oldPOSABond, address indexed newPOSABond, address
indexed caller);
 event POSABondInfoSet(address indexed oldPOSABondInfo, address indexed
newPOSABondInfo, address indexed caller);
 event POSARewardCalculatorSet(
   address indexed oldPOSARewardCalculator,
   address indexed newPOSARewardCalculator,
   address indexed caller
 );
 event KYCSet(address indexed oldKYC, address indexed newKYC, address indexed
caller);
 event AcceptedKYCLevelSet(
   uint256 indexed oldAcceptedKYCLevel,
   uint256 indexed newAcceptedKYCLevel,
   address indexed caller
 );

 uint256 public hardcap; // expected to be X KUB
 uint256 public totalDeposited;
 uint256 public totalClaimed;
 uint256 public maximumDepositPerUser; // X KUB / user
 uint256 public endDepositDate;
 uint256 public withdrawDate;

 Flag public paused;
 IPOSABond public POSABond;
 IPOSABondInfo public POSABondInfo;
 IPOSARewardCalculator public POSARewardCalculator;
```

```
IAdminKAP20Router public adminKAP20Router;
IKKUB public immutable KKUB;

bool public metaMaskEnabled;
bool public bitkubNextEnabled;

mapping(address => uint256) public depositBalance;

EnumerableSetAddress.AddressSet private _whitelistLimit;
uint256[] private _tokenIDs;

modifier whenNotPausedDeposit() {
  require(!paused.depositPaused, "Pause: deposit paused");
  _;
}

modifier whenNotPausedWithdraw() {
  require(!paused.withdrawPaused, "Pause: withdraw paused");
  _;
}

modifier whenNotPausedClaim() {
  require(!paused.claimPaused, "Pause: claim paused");
  _;
}

modifier whenNotPausedEmergencyWithdraw() {
  require(!paused.emergencyWithdrawPaused, "Pause: emergency withdraw paused");
  _;
}

modifier checkBitkubNext(address _bitkubNext) {
  require(bitkubNextEnabled, "BitkubNext disabled");
  require(kyc.kycsLevel(_bitkubNext) >= acceptedKYCLevel, "only BitkubNext user");
  _;
}

modifier checkMetaMask(address _sender) {
  require(metaMaskEnabled, "Metamask disabled");
  require(_whitelistLimit.contains(_sender), "not in whitelist");
  _;
}

modifier checkDepositRequirement(uint256 _amount, address _sender) {
  require(_amount > 0, "require amount > 0");
  require(totalDeposited + _amount <= hardcap, "amount exceeds hardcap");
  require(block.timestamp < endDepositDate, "not in deposit period");
```

```
    _;
  }

  constructor(ConstructorInput memory _input) {
   require(_input.maximumDepositPerUser > 0, "require maximumDepositPerUser>0");
    require(_input.endDepositDate > block.timestamp, "require endDepositDate > now");
    require(_input.withdrawDate > _input.endDepositDate, "need
withdrawDate>endDepositDate");
    adminProjectRouter = IAdminProjectRouter(_input.adminInput.adminProjectRouter);
    adminKAP20Router = IAdminKAP20Router(_input.adminInput.adminKAP20Router);
    committee = _input.adminInput.committee;

    POSABond = IPOSABond(_input.bondInput.POSABond);
    POSABondInfo = IPOSABondInfo(_input.bondInput.POSABondInfo);
    POSARewardCalculator =
IPOSARewardCalculator(_input.bondInput.POSARewardCalculator);

    kyc = IKYCBitkubChain(_input.kycInput.kyc);
    acceptedKYCLevel = _input.kycInput.acceptedKYCLevel;

    KKUB = IKKUB(_input.KKUB);
    hardcap = _input.hardcap;
    maximumDepositPerUser = _input.maximumDepositPerUser;
    endDepositDate = _input.endDepositDate;
    withdrawDate = _input.withdrawDate;
    metaMaskEnabled = _input.enabled[0];
    bitkubNextEnabled = _input.enabled[1];

    paused.depositPaused = false;
    paused.withdrawPaused = false;
    paused.claimPaused = false;
    paused.emergencyWithdrawPaused = true;
  }

  ///////////////////////// read /////////////////////////

  function whitelistLimit() external view override returns (address[] memory) {
    return _whitelistLimit.getAll();
  }

  function whitelistLimitLength() external view override returns (uint256) {
    return _whitelistLimit.length();
  }

  function whitelistLimitByPage(uint256 _page, uint256 _limit) external view override
returns (address[] memory) {
    return _whitelistLimit.get(_page, _limit);
```

```
 }

 function tokenLength() external view returns (uint256) {
   return _tokenIDs.length;
}

 function tokenIDByPage(uint256 _page, uint256 _limit) external view returns
(uint256[] memory) {
   require(_page > 0 && _limit > 0);
   uint256 tempLength = _limit;
   uint256 cursor = (_page - 1) * _limit;
   uint256 _uintLength = _tokenIDs.length;
   if (cursor >= _uintLength) {
     return new uint256[](0);
   }
   if (tempLength > _uintLength - cursor) {
     tempLength = _uintLength - cursor;
   }
   uint256[] memory uintList = new uint256[](tempLength);
   for (uint256 i = 0; i < tempLength; i++) {
     uintList[i] = _tokenIDs[cursor + i];
   }
   return uintList;
 }

 function pendingReward(uint256 _tokenID) public view override returns (uint256) {
   (uint256 amount, , , uint256 rewardClaimed, , address poolAddress, bool
activated) = POSABondInfo.bondInfo(
     _tokenID
   );
   require(poolAddress == address(this), "not bond of this pool");
   if (activated) {
     return _pendingReward(amount, rewardClaimed);
   } else {
     return 0;
   }
 }

 function _pendingReward(uint256 _amount, uint256 _rewardClaimed) internal view
returns (uint256) {
   return
     POSARewardCalculator.calculateReward(
       address(this).balance,
       _amount,
       _rewardClaimed,
       totalDeposited,
       totalClaimed
```

```
    );
  }

  function cumulativeReward() external view override returns (uint256) {
    return address(this).balance + totalClaimed;
  }

  //////////////////////////////////////////////////////////

  //////////////////////// setter ////////////////////////

  function setAdminProjectRouter(address _adminProjectRouter) public override
onlyCommittee {
    require(_adminProjectRouter != address(0), "Authorization: new admin project
router is the zero address");
    emit AdminProjectRouterSet(address(adminProjectRouter), _adminProjectRouter,
msg.sender);
    adminProjectRouter = IAdminProjectRouter(_adminProjectRouter);
  }

  function setAdminKAP20Router(address _adminKAP20Router) external override
onlyCommittee {
    emit AdminKAP20RouterSet(address(adminKAP20Router), _adminKAP20Router,
msg.sender);
    adminKAP20Router = IAdminKAP20Router(_adminKAP20Router);
  }

  function setPOSABond(address _POSABond) external onlyCommittee {
    emit POSABondSet(address(POSABond), _POSABond, msg.sender);
    POSABond = IPOSABond(_POSABond);
  }

  function setPOSABondInfo(address _POSABondInfo) external onlyCommittee {
    emit POSABondInfoSet(address(POSABondInfo), _POSABondInfo, msg.sender);
    POSABondInfo = IPOSABondInfo(_POSABondInfo);
  }

  function setPOSARewardCalculator(address _POSARewardCalculator) external
onlyCommittee {
    emit POSARewardCalculatorSet(address(POSARewardCalculator),
_POSARewardCalculator, msg.sender);
    POSARewardCalculator = IPOSARewardCalculator(_POSARewardCalculator);
  }

  function setKYC(address _kyc) external override onlyCommittee {
    emit KYCSet(address(kyc), _kyc, msg.sender);
    kyc = IKYCBitkubChain(_kyc);
  }

  function setAcceptedKYCLevel(uint256 _acceptedKYCLevel) external override
onlyCommittee {
    emit AcceptedKYCLevelSet(acceptedKYCLevel, _acceptedKYCLevel, msg.sender);
    acceptedKYCLevel = _acceptedKYCLevel;
  }

  //////////////////////////////////////////////////////////

 function setHardcap(uint256 _hardcap) external override onlyCommittee {
    require(_hardcap > 0, "require hardcap > 0");
    require(_hardcap >= totalDeposited, "require hardcap>=totalDeposited");
    hardcap = _hardcap;
  }

  function setMaximumDepositPerUser(uint256 _maximumDepositPerUser) external override
onlyCommittee {
    require(_maximumDepositPerUser > 0, "require maximumDepositPerUser>0");
    maximumDepositPerUser = _maximumDepositPerUser;
  }

  function setEndDepositDate(uint256 _endDepositDate) external override onlyCommittee
{
    require(block.timestamp < endDepositDate, "require now < endDepositDate");
    require(_endDepositDate > 0, "require endDepositDate > 0");
    require(_endDepositDate < withdrawDate, "require endDeposit < withdraw");
    endDepositDate = _endDepositDate;
  }

  function setWithdrawDate(uint256 _withdrawDate) external override onlyCommittee {
    require(block.timestamp < withdrawDate, "require now < withdrawDate");
    require(_withdrawDate > endDepositDate, "require withdraw > endDeposit");
    withdrawDate = _withdrawDate;
  }

  //////////////////////////////////////////////////////////

  function setEnabled(bool[2] memory _enabled) external override onlyCommittee {
    metaMaskEnabled = _enabled[0];
    bitkubNextEnabled = _enabled[1];
  }

  function addWhitelistLimit(address _addr) external override onlySuperAdmin {
    require(_whitelistLimit.add(_addr), "can't add this address");
  }
```

```
function removeWhitelistLimit(address _addr) external override onlySuperAdmin {
  require(_whitelistLimit.remove(_addr), "can't remove this address");
}

/////////////////////////////////////////////////////////

function pause(Flag memory _paused) external override onlyCommittee {
  paused = _paused;
}

function emergencyWithdrawKUB(address _to, uint256 _amount)
  external
  override
  onlyCommittee
  whenNotPausedEmergencyWithdraw
{
  (bool success, ) = _to.call{ value: _amount }("");
  require(success, "unable to send value");
}

//////////////////////////////////////////////////////////

/////////////////////// deposit /////////////////////////

function deposit()
  external
  payable
  override
  nonReentrant
  checkMetaMask(msg.sender)
  checkDepositRequirement(msg.value, msg.sender)
{
  KKUB.deposit{ value: msg.value }();
  _deposit(msg.sender, msg.value);
}

function deposit(uint256 _amount)
  external
  override
  nonReentrant
  checkMetaMask(msg.sender)
  checkDepositRequirement(_amount, msg.sender)
{
  bool success = KKUB.transferFrom(msg.sender, address(this), _amount);
  require(success, "failed to transfer KKUB");
  _deposit(msg.sender, _amount);
}
```

```
 function deposit(uint256 _amount, address _bitkubNext)
   external
   override
   nonReentrant
   onlySuperAdmin
   checkBitkubNext(_bitkubNext)
   checkDepositRequirement(_amount, _bitkubNext)
 {
   if (!_whitelistLimit.contains(_bitkubNext)) {
     require(depositBalance[_bitkubNext] + _amount <= maximumDepositPerUser, "amount
exceeds maximum deposit");
   }
   adminKAP20Router.externalTransferKKUB(address(0), _bitkubNext, address(this),
_amount, 0);
   _deposit(_bitkubNext, _amount);
 }

 function _deposit(address _sender, uint256 _amount) internal whenNotPausedDeposit {
   uint256 tokenID = POSABond.mint(_sender, _amount);
   _tokenIDs.push(tokenID);
   depositBalance[_sender] += _amount;
   totalDeposited += _amount;
   emit Deposit(_sender, tokenID, _amount);
 }

 //////////////////////////////////////////////////////////

 //////////////////////// claim //////////////////////////

 function claim(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
   (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);

   _claim(msg.sender, _tokenID, amount, rewardClaimed);
 }

 function claim(uint256 _tokenID, address _bitkubNext)
   external
   override
   nonReentrant
   onlySuperAdmin
   checkBitkubNext(_bitkubNext)
 {
   (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);

   _claim(_bitkubNext, _tokenID, amount, rewardClaimed);
 }

 function _claim(
   address _sender,
   uint256 _tokenID,
   uint256 _amount,
   uint256 _rewardClaimed
 ) internal whenNotPausedClaim {
   require(block.timestamp >= endDepositDate, "require now >= endDepositDate");
   require(_sender == POSABond.ownerOf(_tokenID), "not owner");

   uint256 reward = _pendingReward(_amount, _rewardClaimed);
   POSABondInfo.updateClaimReward(_tokenID, reward);

   if (reward > 0) {
     totalClaimed += reward;
     _wrapRewardAndSendKKUB(_sender, reward);
   }
   emit ClaimReward(_sender, _tokenID, reward);
 }

 ///////////////////////////////////////////////////////////

 /////////////////////// withdraw /////////////////////////

 function withdraw(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
   _withdraw(_tokenID, msg.sender);
 }

 function withdraw(uint256 _tokenID, address _bitkubNext)
   external
   override
   nonReentrant
   onlySuperAdmin
   checkBitkubNext(_bitkubNext)
 {
   _withdraw(_tokenID, _bitkubNext);
 }

 function _withdraw(uint256 _tokenID, address _sender) internal
whenNotPausedWithdraw {
   require(block.timestamp >= withdrawDate, "require now >= withdrawDate");

   (uint256 amount, uint256 rewardClaimed) = _loadBondInfo(_tokenID);

   _claim(_sender, _tokenID, amount, rewardClaimed);
```

```
    POSABondInfo.updateWithdraw(_tokenID);
    _sendKKUB(_sender, amount);
    emit Withdraw(_sender, _tokenID, amount);
  }

 function emergencyWithdraw(uint256 _tokenID) external override nonReentrant
checkMetaMask(msg.sender) {
    _emergencyWithdraw(_tokenID, msg.sender);
  }

  function emergencyWithdraw(uint256 _tokenID, address _bitkubNext)
    external
    override
    nonReentrant
    onlySuperAdmin
    checkBitkubNext(_bitkubNext)
  {
    _emergencyWithdraw(_tokenID, _bitkubNext);
  }

  function _emergencyWithdraw(uint256 _tokenID, address _sender) internal
whenNotPausedEmergencyWithdraw {
    require(_sender == POSABond.ownerOf(_tokenID), "not owner");

    (uint256 amount, ) = _loadBondInfo(_tokenID);

    POSABondInfo.updateWithdraw(_tokenID);
    _sendKKUB(_sender, amount);
    emit EmergencyWithdraw(_sender, _tokenID, amount);
  }

  ///////////////////////////////////////////////////////////

  function _loadBondInfo(uint256 _tokenID) internal view returns (uint256, uint256) {
    (uint256 amount, , , uint256 rewardClaimed, , address poolAddress, ) =
POSABondInfo.bondInfo(_tokenID);
    require(poolAddress == address(this), "not bond of this pool");
    return (amount, rewardClaimed);
  }

  function _wrapRewardAndSendKKUB(address _addr, uint256 _amount) internal {
    KKUB.deposit{ value: _amount }();
    _sendKKUB(_addr, _amount);
  }

  function _sendKKUB(address _addr, uint256 _amount) internal {
    bool success = KKUB.transfer(_addr, _amount);
    require(success, "failed to transfer KKUB");
  }
}
```

2. PoSABond

```
contract POSABond is KAP721 {
  using EnumerableSetUint for EnumerableSetUint.UintSet;

  event POSAPoolVerifierSet(
    address indexed oldPOSAPoolVerifier,
    address indexed newPOSAPoolVerifier,
    address indexed caller
  );
  event POSABondInfoSet(address indexed oldPOSABondInfo, address indexed
newPOSABondInfo, address indexed caller);
  event TransferProposalSet(
    address indexed oldTransferProposal,
    address indexed newTransferProposal,
    address indexed caller
  );
  event TokenURISet(uint256 indexed tokenId, string tokenURI, address indexed
caller);
  event BaseURISet(string oldBaseURI, string newBaseURI, address indexed caller);

  modifier onlySuperAdminOrHolder(uint256 _tokenId) {
    require(
      adminProjectRouter.isSuperAdmin(msg.sender, PROJECT) || msg.sender ==
ownerOf(_tokenId),
      "BitkubKAP721: restricted only super admin or holder"
    );
    _;
  }

  modifier onlyPOSAPool() {
    require(POSAPoolVerifier.isPool(msg.sender), "POSABond: sender is not a pool");
    _;
  }

 mapping(address => mapping(address => EnumerableSetUint.UintSet)) private
_holderTokensByPool;
```

```
 IPOSAPoolVerifier public POSAPoolVerifier;
 IPOSABondInfo public POSABondInfo;
 ITransferProposal public transferProposal;

 uint256 public tokenIdCounter = 1;

 struct ConstructorInput {
   address POSAPoolVerifier;
   address POSABondInfo;
   address transferProposal;
 }

 constructor(
   string memory _name,
   string memory _symbol,
   string memory _baseURI,
   address _kyc,
   address _adminProjectRouter,
   address _committee,
   address _transferRouter,
   uint256 _acceptedKYCLevel,
   ConstructorInput memory _constructorInput
 ) KAP721(_name, _symbol, _baseURI, _kyc, _adminProjectRouter, _committee,
_transferRouter, _acceptedKYCLevel) {
   POSAPoolVerifier = IPOSAPoolVerifier(_constructorInput.POSAPoolVerifier);
   POSABondInfo = IPOSABondInfo(_constructorInput.POSABondInfo);
   transferProposal = ITransferProposal(_constructorInput.transferProposal);
 }

 function exists(uint256 _tokenId) external view returns (bool) {
   return _exists(_tokenId);
 }

 function tokenOfOwnerByPoolByPage(
   address _owner,
   address _pool,
   uint256 _page,
   uint256 _limit
 ) external view returns (uint256[] memory) {
   return _holderTokensByPool[_owner][_pool].get(_page, _limit);
 }

 function tokenOfOwnerByPool(address _owner, address _pool) external view returns
(uint256[] memory) {
   return _holderTokensByPool[_owner][_pool].getAll();
 }
 function balanceOfByPool(address _owner, address _pool) external view returns
(uint256) {
   return _holderTokensByPool[_owner][_pool].length();
 }

 function getTokenInfoAddress(uint256 _tokenId) external view returns (address) {
   return address(POSABondInfo);
 }


///////////////////////////////////////////////////////////////////////////////////
///

 function setPOSAPoolVerifier(address _POSAPoolVerifier) external onlyCommittee {
   emit POSAPoolVerifierSet(address(POSAPoolVerifier), _POSAPoolVerifier, msg.sender);
   POSAPoolVerifier = IPOSAPoolVerifier(_POSAPoolVerifier);
 }

 function setPOSABondInfo(address _POSABondInfo) external onlyCommittee {
   emit POSABondInfoSet(address(POSABondInfo), _POSABondInfo, msg.sender);
   POSABondInfo = IPOSABondInfo(_POSABondInfo);
 }

 function setTransferProposal(address _transferProposal) external onlyCommittee {
   emit TransferProposalSet(address(transferProposal), _transferProposal,
msg.sender);
   transferProposal = ITransferProposal(_transferProposal);
 }

 function setTokenURI(uint256 _tokenId, string calldata _tokenURI) external
onlyCommittee {
   emit TokenURISet(_tokenId, _tokenURI, msg.sender);
   _setTokenURI(_tokenId, _tokenURI);
 }

 function setBaseURI(string calldata _baseURI) external onlyCommittee {
   emit BaseURISet(baseURI, _baseURI, msg.sender);
   _setBaseURI(_baseURI);
 }

 function pause() external onlyCommittee {
   _pause();
 }

 function unpause() external onlyCommittee {
   _unpause();
 }


////////////////////////////////////////////////////////////////////////////////////
///

 function mint(address _to, uint256 _amount) external onlyPOSAPool whenNotPaused
returns (uint256 tokenId) {
   tokenId = tokenIdCounter;
  tokenIdCounter = tokenIdCounter + 1;

   POSABondInfo.setBondInfoOnMint(tokenId, _amount, msg.sender);

   _mint(_to, tokenId);
 }

 function burn(uint256 _tokenId) external onlySuperAdminOrHolder(_tokenId)
whenNotPaused {
   _burn(_tokenId);
 }

 // add _holderTokensByPool
 function adminTransfer(
   address _from,
   address _to,
   uint256 _tokenId
 ) public override {
   (, , , , , address poolAddress, ) = POSABondInfo.bondInfo(_tokenId);
   _holderTokensByPool[_from][poolAddress].remove(_tokenId);
   _holderTokensByPool[_to][poolAddress].add(_tokenId);

   KAP721.adminTransfer(_from, _to, _tokenId);
 }

 function _beforeTokenTransfer(
   address from,
   address to,
   uint256 tokenId
 ) internal override {
   if (from != address(0) && transferProposal != ITransferProposal(address(0))) {
     transferProposal.useTransferProposal(from, to, tokenId);
   }

   (, , , , , address poolAddress, ) = POSABondInfo.bondInfo(tokenId);
   _holderTokensByPool[from][poolAddress].remove(tokenId);
   _holderTokensByPool[to][poolAddress].add(tokenId);
 }
```

```
}
```

3. POSABondInfo

```
contract POSABondInfo is IPOSABondInfo, Committee {
  modifier onlyExactPoolOfToken(uint256 _tokenID) {
    require(msg.sender == bondInfo[_tokenID].bondPool, "The token does not belong to
the calling pool");
    _;
  }

  event POSABondSet(address indexed oldPOSABond, address indexed newPOSABond,
address indexed caller);
  event ActivateBond(
    address indexed sender,
    uint256 indexed tokenID,
    bool oldValue,
    bool newValue,
    bool indexed manual
  );
  event EditBondInfoOldValue(
    address indexed sender,
    uint256 indexed tokenID,
    uint256 amount,
    uint256 depositTime,
    uint256 lastRewardTime,
    uint256 rewardClaimed,
    uint256 principleClaimed,
    address bondPool,
    bool activated
  );
  event EditBondInfoNewValue(
    address indexed sender,
    uint256 indexed tokenID,
    uint256 amount,
    uint256 depositTime,
    uint256 lastRewardTime,
    uint256 rewardClaimed,
    uint256 principleClaimed,
    address bondPool,
    bool activated
  );
  event SetBondInfoOnMint(
    address indexed sender,
    uint256 indexed tokenID,
    uint256 amount,
    uint256 depositTime,
    uint256 lastRewardTime,
    uint256 rewardClaimed,
  uint256 principleClaimed,
    address bondPool,
    bool activated
  );
  event UpdateClaimReward(uint256 indexed tokenID, uint256 lastRewardTime, uint256
rewardClaimed);

  struct BondInfo {
    uint256 amount;
    uint256 depositTime;
    uint256 lastRewardTime;
    uint256 rewardClaimed;
    uint256 principleClaimed;
    address bondPool;
    bool activated;
  }

  mapping(uint256 => BondInfo) public override bondInfo;

  address public POSABond;
  uint256[] public tokenIDs;

  constructor(address _committee, address _POSABond) {
    committee = _committee;
    POSABond = _POSABond;
  }

  function tokenIDsLength() external view returns (uint256) {
    return tokenIDs.length;
  }

  function setPOSABond(address _POSABond) external onlyCommittee {
    emit POSABondSet(address(POSABond), _POSABond, msg.sender);
    POSABond = _POSABond;
  }

  function setBondInfoOnMint(
    uint256 _tokenID,
    uint256 _amount,
    address _poolAddress
  ) external override {
    require(msg.sender == POSABond, "Sender is not POSABond");

    bondInfo[_tokenID] = BondInfo({
      amount: _amount,
      depositTime: block.timestamp,
    lastRewardTime: block.timestamp,
      rewardClaimed: 0,
      principleClaimed: 0,
      bondPool: _poolAddress,
      activated: true
    });
    tokenIDs.push(_tokenID);

    emit SetBondInfoOnMint(msg.sender, _tokenID, _amount, block.timestamp,
block.timestamp, 0, 0, _poolAddress, true);
  }

  function updateClaimReward(uint256 _tokenID, uint256 _amountClaimed)
    external
    override
    onlyExactPoolOfToken(_tokenID)
  {
    require(bondInfo[_tokenID].activated, "Bond is deactivated");

    if (_amountClaimed > 0) {
      bondInfo[_tokenID].lastRewardTime = block.timestamp;
      bondInfo[_tokenID].rewardClaimed += _amountClaimed;
      emit UpdateClaimReward(_tokenID, bondInfo[_tokenID].lastRewardTime,
bondInfo[_tokenID].rewardClaimed);
    }
  }

  function updateWithdraw(uint256 _tokenID) external override
onlyExactPoolOfToken(_tokenID) {
    bondInfo[_tokenID].principleClaimed = bondInfo[_tokenID].amount;
    _activateBond(_tokenID, false, false);
  }

  function activateBond(uint256 _tokenID, bool _activated) external onlyCommittee {
    _activateBond(_tokenID, _activated, true);
  }

  function _activateBond(
    uint256 _tokenID,
    bool _activated,
    bool _manual
  ) internal {
    require(bondInfo[_tokenID].activated != _activated, "The current activated state
must be different from input");
    bondInfo[_tokenID].activated = _activated;
    emit ActivateBond(msg.sender, _tokenID, !_activated, _activated, _manual);
  }

  // This function can only be called under a notice from the court
  function editBondInfo(
    uint256 _tokenID,
    uint256 _amount,
    uint256 _depositTime,
    uint256 _lastRewardTime,
    uint256 _rewardClaimed,
    uint256 _principleClaimed,
    address _bondPool,
    bool _activated
  ) external onlyCommittee {
    BondInfo storage info = bondInfo[_tokenID];

    emit EditBondInfoOldValue(
      msg.sender,
      _tokenID,
      info.amount,
      info.depositTime,
      info.lastRewardTime,
      info.rewardClaimed,
      info.principleClaimed,
      info.bondPool,
      info.activated
    );

    emit EditBondInfoNewValue(
      msg.sender,
    _tokenID,
      _amount,
      _depositTime,
      _lastRewardTime,
      _rewardClaimed,
      _principleClaimed,
      _bondPool,
      _activated
    );

    info.amount = _amount;
    info.depositTime = _depositTime;
    info.lastRewardTime = _lastRewardTime;
    info.rewardClaimed = _rewardClaimed;
    info.principleClaimed = _principleClaimed;
    info.bondPool = _bondPool;
    info.activated = _activated;
  }
}
```

4. POSARewardCalculator

```
contract POSARewardCalculator is IPOSARewardCalculator {
  function calculateReward(
    uint256 _poolBalance,
    uint256 _depositAmount,
    uint256 _rewardClaimed,
    uint256 _totalDeposited,
    uint256 _totalClaimed
  ) external pure override returns (uint256) {
    require(
      _depositAmount <= _totalDeposited && _rewardClaimed <= _totalClaimed,
      "POSARewardCalculator: invalid inputs"
    );
    if (_totalDeposited > 0) {
      // totalReward should never decrease
      uint256 totalReward = _poolBalance + _totalClaimed;
      // totalDeposited must never change after first reward claim
      uint256 reward = (_depositAmount * totalReward) / _totalDeposited;
      if (reward > _rewardClaimed) {
        return reward - _rewardClaimed;
      }
    }
    return 0;
  }
}
```

Appendix G: Central Limit Theorem

The Central Limit Theorem states that the sampling distribution of the sample means approaches a normal distribution (i.e., a Bell curve) as the sample size gets larger. In other words, CLT is a statistical premise that, given a sufficiently large sample size from a population with a finite level of variance, the mean of all sampled variables from the same population will be approximately equal to the mean of the whole population. Furthermore, these samples approximate a normal distribution, with their variances being approximately equal to the variance of the population as the sample size gets larger. CLT can be described more precisely using the definition of a limit. The CDF of the standardized sample mean ($\bar{X} - \mu)/\sigma$ converges pointwise to the Cumulative distribution function (CDF) (Φ) of the standard normal distribution. This is shown with the integral:

$$\lim_{n \to \infty} \mathbb{P} \frac{\left(\overline{X}_n - \mu\right)}{\sigma} \leq z = \Phi(z)$$

CLT equation

Where: X_n is a sequence of sample data

P is a probability function which states the following equation.

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{\frac{x^2}{2}} dz \, .$$

Equation of P

Appendix H: Wrap KUB contract (KKUB)

```
pragma solidity 0.6.6;

interface IAdminAsset {
    function isSuperAdmin(address _addr, string calldata _token) external view returns (bool);
}

interface IKYC {
    function kycsLevel(address _addr) external view returns (uint256);
}

interface IKAP20 {
    event Transfer(address indexed from, address indexed to, uint256 tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint256 tokens);

    function totalSupply() external view returns (uint256);

    function balanceOf(address tokenOwner) external view returns (uint256 balance);

    function allowance(address tokenOwner, address spender) external view returns (uint256 remaining);

    function transfer(address to, uint256 tokens) external returns (bool success);

    function approve(address spender, uint256 tokens) external returns (bool success);

    function transferFrom(address from, address to, uint256 tokens) external returns (bool success);

    function getOwner() external view returns (address);

    function batchTransfer(address[] calldata _from, address[] calldata _to, uint256[] calldata _value) external
returns (bool success);

    function adminTransfer(address _from, address _to, uint256 _value) external returns (bool success);
}

contract KKUB is IKAP20 {
    string public name     = "Wrapped KUB";
    string public symbol   = "KKUB";
    uint8  public decimals = 18;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed tokenOwner, address indexed spender, uint256 value);
    event Deposit(address indexed dst, uint256 value);
    event Withdrawal(address indexed src, uint256 value);
    event Paused(address account);
    event Unpaused(address account);


     mapping (address => uint256) balances;
```

```
 mapping (address => mapping (address => uint256)) allowed;
mapping (address => bool) public blacklist;

IAdminAsset public admin;
IKYC public kyc;
bool public paused;

uint256 public kycsLevel;

modifier onlySuperAdmin() {
    require(admin.isSuperAdmin(msg.sender, symbol), "Restricted only super admin");
    _;
}

modifier whenNotPaused() {
    require(!paused, "Pausable: paused");
    _;
}

modifier whenPaused() {
    require(paused, "Pausable: not paused");
    _;
}

constructor(address _admin, address _kyc) public {
    admin = IAdminAsset(_admin);
    kyc = IKYC(_kyc);
    kycsLevel = 1;
}

function setKYC(address _kyc) external onlySuperAdmin {
    kyc = IKYC(_kyc);
}

function setKYCsLevel(uint256 _kycsLevel) external onlySuperAdmin {
    require(_kycsLevel > 0);
    kycsLevel = _kycsLevel;
}

function getOwner() external view override returns (address) {
    return address(admin);
}

fallback() external payable {
    deposit();
}

 receive() external payable {
    deposit();
}
```

```
 function deposit() public whenNotPaused payable {
    balances[msg.sender] += msg.value;
      emit Deposit(msg.sender, msg.value);
      emit Transfer(address(0), msg.sender, msg.value);
}

function withdraw(uint256 _value) public whenNotPaused  {
    require(!blacklist[msg.sender], "Address is in the blacklist");
    _withdraw(_value, msg.sender);
}

function withdrawAdmin(uint256 _value, address _addr) public onlySuperAdmin {
    _withdraw(_value, _addr);
}

function _withdraw(uint256 _value, address _addr) internal {
    require(balances[_addr] >= _value);
    require(kyc.kycsLevel(_addr) > kycsLevel, "only kyc address registered with phone number can withdraw");

    balances[_addr] -= _value;
    payable(_addr).transfer(_value);
    emit Withdrawal(_addr, _value);
    emit Transfer(_addr, address(0), _value);
}

function totalSupply() public view override returns (uint256) {
    return address(this).balance;
}

function balanceOf(address _addr) public view override returns (uint256) {
    return balances[_addr];
}

function allowance(address _owner, address _spender) public view override returns (uint256) {
    return allowed[_owner][_spender];
}

function approve(address _spender, uint256 _value) public override whenNotPaused returns (bool) {
    require(!blacklist[msg.sender], "Address is in the blacklist");
    _approve(msg.sender, _spender, _value);
    return true;
}

function _approve(address owner, address spender, uint256 amount) internal {
    require(owner != address(0), "KAP20: approve from the zero address");
    require(spender != address(0), "KAP20: approve to the zero address");

    allowed[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

function transfer(address _to, uint256 _value) public override whenNotPaused returns (bool) {
    require(_value <= balances[msg.sender], "Insufficient Balance");
    require(blacklist[msg.sender] == false && blacklist[_to] == false, "Address is in the blacklist");

    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);

    return true;
}

 function transferFrom(
    address _from,
    address _to,
    uint256 _value
) public override whenNotPaused returns (bool) {
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);
    require(blacklist[_from] == false && blacklist[_to] == false, "Address is in the blacklist");

    balances[_from] -= _value;
    balances[_to] += _value;
    allowed[_from][msg.sender] -= _value;
    emit Transfer(_from, _to, _value);
    return true;
}

function batchTransfer(
    address[] calldata _from,
    address[] calldata _to,
    uint256[] calldata _value
) external override onlySuperAdmin returns (bool) {
    require(_from.length == _to.length && _to.length == _value.length, "Need all input in same length");

    for (uint256 i = 0; i < _from.length; i++) {
        if(blacklist[_from[i]] == true || blacklist[_to[i]] == true){
              continue;
        }

        if (balances[_from[i]] >= _value[i]) {
            balances[_from[i]] -= _value[i];
            balances[_to[i]] += _value[i];
            emit Transfer(_from[i], _to[i], _value[i]);
        }
    }

    return true;
}
```

```
    function adminTransfer(
        address _from,
        address _to,
        uint256 _value
    ) external override onlySuperAdmin returns (bool) {
        require(balances[_from] >= _value);
        balances[_from] -= _value;
        balances[_to] += _value;
        emit Transfer(_from, _to, _value);

        return true;
    }

    function pause() external onlySuperAdmin whenNotPaused {
        paused = true;
        emit Paused(msg.sender);
    }

    function unpause() external onlySuperAdmin whenPaused {
        paused = false;
        emit Unpaused(msg.sender);
    }

    function addBlacklist(address _addr) external onlySuperAdmin {
        blacklist[_addr] = true;
    }

    function revokeBlacklist(address _addr) external onlySuperAdmin {
        blacklist[_addr] = false;
    }
}
```

Appendix I: Register smart contract

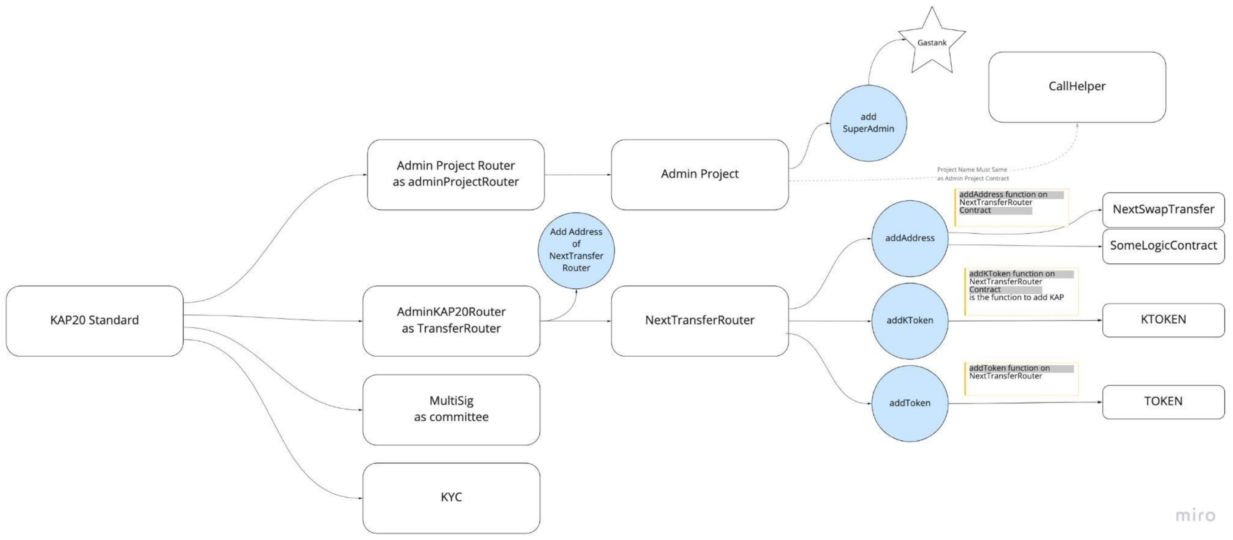
```
// Sources flattened with hardhat v2.4.0 https://hardhat.org

// File contracts/interfaces/IAdmin.sol

pragma solidity 0.6.6;

interface IAdmin {
    function isSuperAdmin(address _addr) external view returns (bool);

    function isAdmin(address _addr) external view returns (bool);
}


// File contracts/KYCBitkubChainV2.sol

pragma solidity 0.6.6;

contract KYCBitkubChainV2 {
    IAdmin public admin;

    mapping(address => uint256) public kycsLevel;
    mapping(address => bool) public isAddressKyc;
    address[] public kycAddresses;

    // projectName => functionName => KYC Levels
    mapping(string => mapping(string => uint256)) public kycsProjectLevel;

    mapping(string => uint256) public kycTitleToLevel;
    mapping(uint256 => string) public kycLevelToTitle;

    uint256 public version = 2;

    event KycCompleted(address indexed addr, address indexed caller, uint256 previousLevel, uint256 level);
    event KycRevoked(address indexed addr, address indexed caller, uint256 previousLevel, uint256 level);

    event KycProject(address indexed _caller, string projectName, string functionName, uint256 level);
    event KycTitle(address indexed _caller, string title, uint256 level);

    modifier onlySuperAdmin() {
        require(admin.isSuperAdmin(msg.sender), "Restrict only super admin");
        _;
    }

    modifier onlyAdmin() {
        require(
            admin.isSuperAdmin(msg.sender) || admin.isAdmin(msg.sender),
            "Restrict only address is admin smart contract"
        );
        _;
    }

    constructor(address _admin) public {
        admin = IAdmin(_admin);
    }
```

```
function kycAddressesLength() external view returns (uint256) {
    return kycAddresses.length;
}

function setAdmin(address _admin) external onlySuperAdmin {
    admin = IAdmin(_admin);
}

function _isPowerOfTwo(uint256 n) private pure returns (bool) {
    return n > 0 ? (n & (n - 1)) == 0 : false;
}

function setKycTitle(string calldata _title, uint256 _level) external onlySuperAdmin {
    require(_isPowerOfTwo(_level), "Level must be power of 2");

    kycTitleToLevel[_title] = _level;
    kycLevelToTitle[_level] = _title;

    emit KycTitle(msg.sender, _title, _level);
}

function setKycProjectLevel(
    string calldata _projectName,
    string calldata _functionName,
    uint256 kycsLevel_
) external onlySuperAdmin {
    kycsProjectLevel[_projectName][_functionName] = kycsLevel_;
    emit KycProject(msg.sender, _projectName, _functionName, kycsLevel_);
}

function setKycCompleted(address _addr, uint256 _level) public onlyAdmin {
    _setKycCompleted(_addr, _level);
}

function batchSetKycCompleted(address[] calldata _addrs, uint256 level) external onlyAdmin {
    for (uint256 i = 0; i < _addrs.length; i++) {
        _setKycCompleted(_addrs[i], level);
    }
}

function _setKycCompleted(address _addr, uint256 _level) internal {
    if (_level > 1) {
        uint256 previousLevel = kycsLevel[_addr];
        kycsLevel[_addr] = _level;

        if (!isAddressKyc[_addr]) {
            kycAddresses.push(_addr);
            isAddressKyc[_addr] = true;
        }

        emit KycCompleted(_addr, msg.sender, previousLevel, _level);
    }
}

// No kyc level set to no kyc
function setKycRevoked(address _addr) external onlyAdmin {
    _setKycRevoked(_addr);
}
```

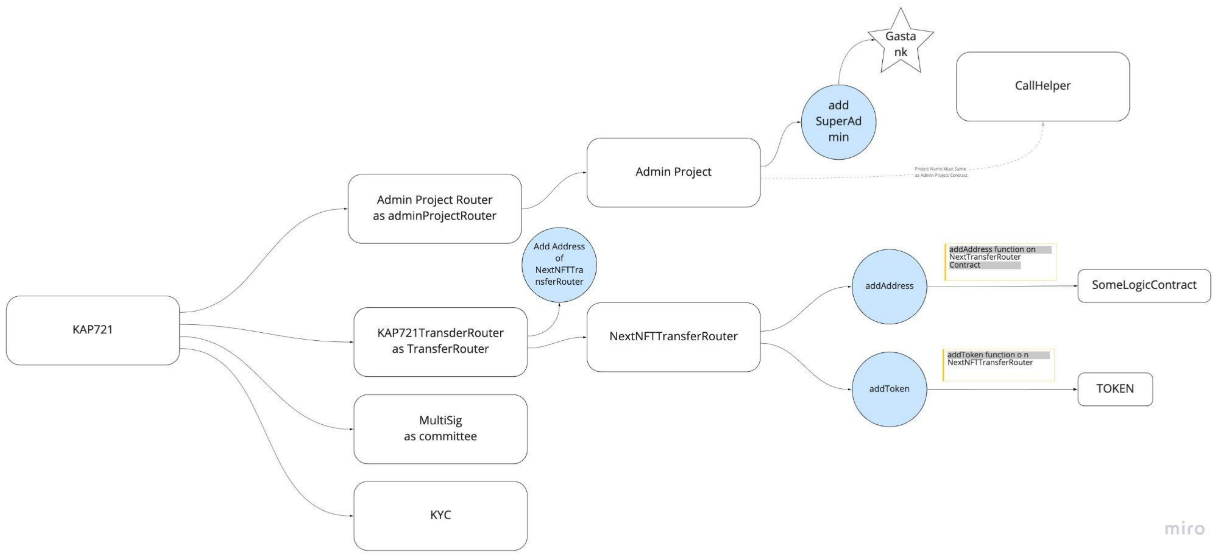
```
    function batchSetKycRevoked(address[] calldata _addrs) external onlyAdmin {
        for (uint256 i = 0; i < _addrs.length; i++) {
            _setKycRevoked(_addrs[i]);
        }
    }

    function _setKycRevoked(address _addr) internal {
        uint256 previousLevel = kycsLevel[_addr];
        kycsLevel[_addr] = 0;
        emit KycRevoked(_addr, msg.sender, previousLevel, 0);
    }
}
```

Appendix J: HyperBlock Smart contract & Gas tank flow

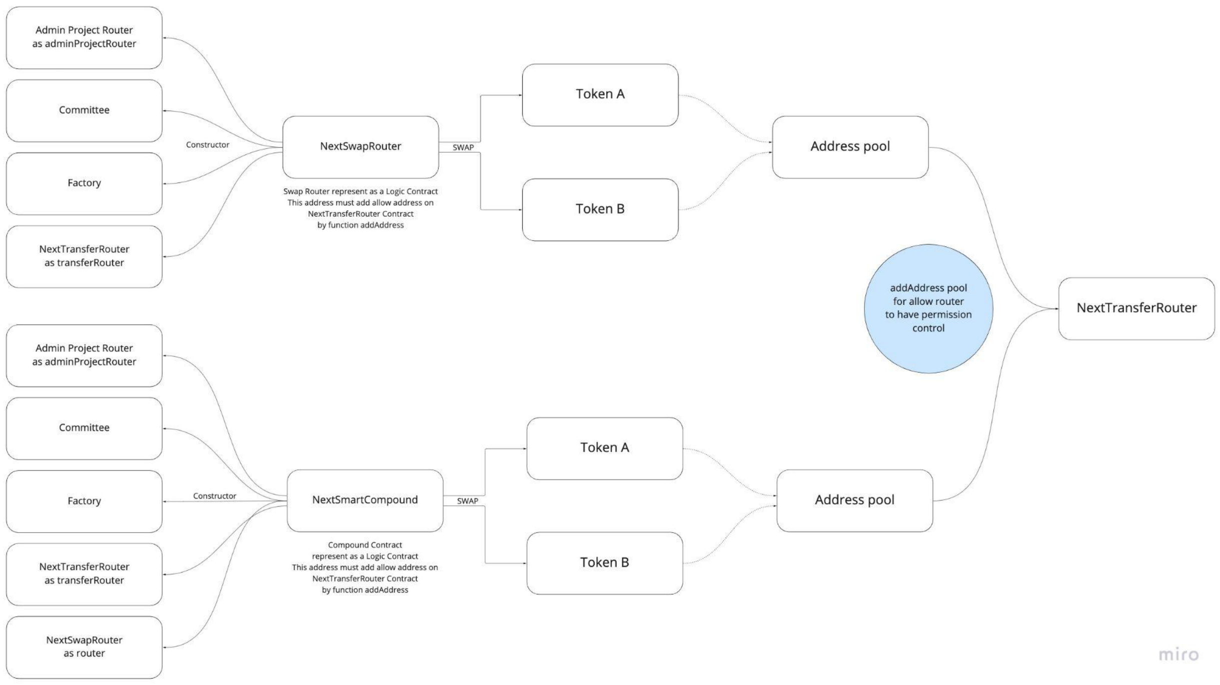
1. **KAP-20 technical flow**


Gastank
CallHelper
add SuperAdmin
Admin Project Router as adminProjectRouter
Admin Project
Project Name Must Same as Admin Project Contract
addAddress function on NextTransferRouter Contract
NextSwapTransfer
SomeLogicContract
Add Address of NextTransfer Router
addAddress
addKToken function on NextTransferRouter Contract is the function to add KAP
KAP20 Standard
AdminKAP20Router as TransferRouter
NextTransferRouter
addKToken
KTOKEN
addToken function on NextTransferRouter
MultiSig as committee
addToken
TOKEN
KYC
miro

2. **KAP-721 technical flow**


Gasta nk
CallHelper
add SuperAd min
Admin Project
Project Name Must Same as Admin Project Contract
Admin Project Router as adminProjectRouter
Add Address of NextNFTTra nsferRouter
addAddress function on NextTransferRouter Contract
addAddress
SomeLogicContract
KAP721
KAP721TransderRouter as TransferRouter
NextNFTTransferRouter
addToken function o n NextNFTTransferRouter
addToken
TOKEN
MultiSig as committee
KYC
miro

3. Swap KAP-20 technical flow


Admin Project Router as adminProjectRouter
Committee
Factory
NextTransferRouter as transferRouter
Constructor
NextSwapRouter
SWAP
Swap Router represent as a Logic Contract This address must add allow address on NextTransferRouter Contract by function addAddress
Token A
Token B
Address pool
addAddress pool for allow router to have permission control
NextTransferRouter
Admin Project Router as adminProjectRouter
Committee
Factory
Constructor
NextTransferRouter as transferRouter
NextSwapRouter as router
NextSmartCompound
SWAP
Compound Contract represent as a Logic Contract This address must add allow address on NextTransferRouter Contract by function addAddress
Token A
Token B
Address pool
miro