BUILDING ON BINANCE CHAIN

JUNE 2019

TEAM CANYA

CONTACT
John-Paul Thorbjornsen
jp@canya.com
Telegram/Wechat: @jpthor
+61432390205

POTENTIAL PROJECTS

This document covers a number of modules that could be built on Binance Chain to benefit Binance Chain, the wider Cosmos Community and the CanYa project.

1) NON-INTERACTIVE MULTI-SIGNATURE MODULE

This module using Cosmos multi-key store to persist state about the wallets on-chain and is non-interactive since signers don't need to be online at the same time. Signatures and public keys are recorded on-chain, as opposed to off-chain. This functions similar to Ethereum's Gnosis Multi-sig contract, where the wallet is set up first before it can be used.

2) ESCROW MODULE

This module is a simple escrow module that allows members to register funds in an escrow that must be paid out in accordance with the set up. It is an adapted non-interactive multi-signature module that places enforceable restrictions on how the transaction is paid out.

3) HEDGED ESCROW MODULE

This module implements a price hedge into the escrow so that an external value can be specified in the payment and the payout correctly paid at all times. This allows escrows to pay out funds in an externally priced asset (such as paying BNB for a transaction that is priced in USD) and removes volatility risks to escrows. The hedged escrow has already been implemented successfully in the CanWork platform.

4) DAO MODULE

This module is a simple staking, election and voting module that allows members to stake assets that can only be unlocked after a period of time, start elections in communities and cast votes that represent on-chain governance.

5) CONTINUOUS LIQUIDITY POOL MODULE

This module allows users to stake assets and BNB in on-chain pools, and then perform trustless swaps across pools. The module features always-on liquidity and fair market-based prices that are resistant to manipulation. Stakers earn fees when staking assets, and users can instantly swap assets in a single transaction. This module is exciting because it will drive staking demand for BNB and BinanceChain assets, and solve liquidity problems for the ecosystem since it has the correct incentives to stake assets and earn fees. Developers can add a swap widget in their apps to instantly convert assets at market prices trustlessly with no counter-party.

1) NON-INTERACTIVE MULTI-SIGNATURE MODULE

OVERVIEW

Blockchains are very useful to transmitting value between participants that is not possible with legacy technologies. Often multiple parties are involved in transactions and must be included as part of the signing process. Bitcoin has native multi-signature support as part of Pay-to-Script-Hash, whilst Ethereum uses smart contracts to build the required functionality.

Binance Chain does not yet have support for multi-signatures. There are projects build interactive threshold signature libraries[1], but it is desirable to have a non-interactive option. This project will build a non-interactive multi-signature module that can be used on Binance Chain or other Cosmos blockchains.

This module using Cosmos multi-key store to persist state about the wallets on-chain (hence non-interactive since signers don't need to be online all at once). Signatures and public keys are recorded on-chain for audit-ability, as opposed to off-chain. It functions similar to Ethereum's Gnosis Multi-sig contract[2], where the wallet is set up first before it can be used.

WALLET GENERATION

The multi-signature wallet is generated by one of the intended parties calling the handler for the module which registers a new multi-signature wallet to their address public key. The wallet is given an identifier so that a lookup registrar can be used. Parameters specify the signature threshold. The state is stored in multi-key storage.

```
1.1 generateMultisig(m, n)     //add multi-sig parameters
     -> returns new walletID
```

The other participants call the register function which registers their public addresses and keys to the wallet. This state is stored in multi-key storage.

```
1.2 registerKey(walletID)      //register owner key in wallet
     -> registers signer public key
```

At this point, all three public keys of parties have been registered in the wallet. Any member can now call wallet generation handler which computes a compliant BECH32 BNB address from the parties' public keys and is stored in the multi-key storage.

```
1.3 generateAddress(walletID) //generate the multi-sig wallet
     -> returns walletAddress
```

[1] https://github.com/kzen-networks

[2] https://github.com/gnosis/MultiSigWallet

Any funds sent to this address will require signatures from parties with the correct threshold specified in the wallet generation.

SIGNING GENERATION

When funds need to be sent from the wallet, one member generates the transaction data, and registers it to the wallet, calling the following function:

```
1.4 registerTx(accTo, bal, nonce)  //register the desired tx
     -> registers unsigned transaction data
```

Each party can query the intended transaction and generate a signed transaction. This is then registered in the wallet by calling the submit function. This cannot be undone, but serves as a record of who signs.

```
1.5 queryTx(walletID)
     -> returns intended unsigned transaction data
1.6 submitSignedTx(walletID, nonce, signedData)
     -> registers partially-signed transaction data from signer
```

Once the correct threshold of secrets have been registered in the wallet, anyone can call the spend transaction which combines the secrets to generate the final signed transaction, and then broadcasts that to a node.

```
1.7 spendTx(walletID, nonce)
     -> returns fully-signed transaction data for broadcast
```

Once the spend transaction is called, the wallet can be used again by incrementing the nonce, which invalidates all previous signatures.

OTHER ASPECTS

- **Fees.** The creation of a multi-sig should attract a 0.01-0.1 BNB fee per signer, since it adds cost to store details on the blockchain. Spending should also attract 0.02-0.2 BNB fee since it further adds state to the blockchain.
- **Data Use**. A 2 of 3 multi-signature will require storing three 64byte public keys on chain, with up to 32 bytes of extra data, with 32 bytes for the final wallet address — for a total of <300bytes. Spending from a multi-signature will require as a minimum one unsigned transaction (128 bytes), two partially-signed transactions (252 bytes) and one fully-signed transaction (128 bytes) — total of 512 bytes.
- **Off-chain Option.** Parties can elect to do signing of the transaction purely offline (to save fees). In this case if a nonce was not allocated by a registerTx call, then nothing further needs to be done. If a nonce was already allocated then it can be reused. If a nonce was already signed, then it cannot be reused.

2) ESCROW MODULE

OVERVIEW
Escrows are a type of multi-signature account, but specifically designed to enforce a pre-set release of funds with only two out of three members agreeing.

This project will build a simple escrow module that allows members to register funds in an escrow that must be paid out in accordance with the set up.

SET UP
Anyone can create an escrow instance, which itself is a modified multi-signature:

```
2.1 generateEscrow(m, n) // can by 1 of 1, or any m of n
     -> returns new escrowID, registering the owner
```

If the escrow is 1 of 1, the creator can immediately generate the escrow address (1.3 generateAddress). If it is m of n, the parties must go through registering their public keys as is the case for a normal multi-signature.

SENDING FUNDS
A member can now send funds to the escrow address, specifying parameters:

```
2.2 escrowFunds(escrowID, payoutAddress, balance, data)
     -> sends funds to the escrow address, returns nonce
```

Although the funds are stored on the multi-signature address, the multi-signature can only payout to either the payoutAddress or senderAddress specified against that nonce. The Binance transaction module will need to be modified to accompany this restriction.

RELEASE FUNDS
The sending member can request funds to be released by registering a transaction on-chain with the escrowID, and specifying the nonce. Once this is registered against the nonce, then the escrow can payout funds.

```
2.3 releaseFunds(escrowID, nonce)
     -> records a funds release against the nonce
```

PAYOUT FUNDS
The escrow owner can release funds once the release is requested, but includes a split parameter, which is a percentage split between the sender and the payout address (100 for full payout, 0 for full refund). The full amount is always paid.

```
2.4 releaseFunds(escrowID, nonce, split)
     -> registers intent to pay out the escrow
```

The payment can now be processed as a normal mult-signature transaction, but must be fully compliant with the registered transaction (2.4). If it is not, the transaction will fail.

QUERY
Any member can query the escrowID to determine the state of the transaction.

```
2.5 queryEscrow(escrowID, nonce)
     -> returns details regarding the transaction.
     -> Nonce can be blank
```

EXAMPLE
Alice escrows 100 BNB to pay to Bob for services in an escrow that Charlie made. After a week, Alice receives the services from Bob and authorises the escrow to release the funds. Charlie pays out the 100 BNB to Bob and collects a small owner fee.

OTHER ASPECTS

- **Fees.** The creation of an escrow should cost a fee since it requires on-chain storage and compute.
- **Owner Fee.** The owner of the escrow can charge a small fee that they deduct from the final payout.
- **Refund.** The payout member can record a transaction that automatically refunds the sender without requiring input from the escrow owner.

3) HEDGED ESCROW MODULE

OVERVIEW
Escrows of digital assets are problematic for payments since they fluctuate in value wildly. Thus holding an escrow for a period of time exposes parties to price changes. This is an issue for payment platforms where prices are specified in USD, but assets are used for payments instead (such as BNB or CAN).

This project seeks to implement a price hedge into the escrow so that a price can be specified in the payment and the payout correctly paid at all times. The hedged escrow has already been implemented successfully in the CanWork platform.

SET UP
The hedged escrow instance is a modified escrow module that specifies which asset to peg transaction values to:

```
3.1 generateEscrow(m, n, asset)     // can by 1 of 1, or any m of n
     -> returns new escrowID, specifying the asset to peg
```

Going forward, all prices will be referenced to the price of the external asset.

SENDING FUNDS
A member can now send funds to the escrow address, specifying parameters which includes an input of the value of the transaction, priced in the external asset:

```
3.2 escrowFunds(escrowID, escrowAddress, payoutAddress, balance,
data, value)
     -> sends funds to the escrow address, returns nonce
```

DEPOSIT FUNDS
The escrow owner can deposit a float into the escrow address in order to underwrite future transactions:

```
3.3 depositFunds(escrowID, escrowAddress, balance)
```

This float is used to cover future transactions where the value of the underwritten asset drops below the agreed price. It is up to the escrow owner to maintain this float.

PAYOUT FUNDS
The escrow owner can release funds once the release is requested, but the module requests the latest closed price of the asset in order determine the correct payout.

```
3.4 releaseFunds(escrowID, nonce, split)
     -> queries latest market price of asset and pays out enough
assets to preserve the original agreed transaction value
```

If the price of the asset is less than what is originally agreed, more is paid out. If it is more, then less is paid out.

EXAMPLE
Alice escrows $USD100 of BNB (4 BNB) for a delayed payment to Bob. After 2 weeks, the price of BNB has dropped by 10%. On paying out, Charlie, the escrow owner signs a transaction that ensures $USD 100 of BNB (4.4 BNB) is paid out to Bob, calculated from the last closed price of the BNB:USD (Stablecoin) pair. The extra 0.4 BNB comes from the escrow float that Charlie maintains.

OTHER ASPECTS
- **Fees.** The creation of a hedged escrow should cost a fee since it requires on-chain storage and compute.
- **Insufficient Funds.** The hedged escrow does not guarantee transactions can always be covered. In the case of insufficient funds, the maximum is paid but does not error out. Subsequent users of the escrow should inspect the float to determine ability of the escrow to underwrite transactions.

4) DAO MODULE

OVERVIEW

Decentralised autonomous organisations (DAOs) allow a distributed group of people to work together in the pursuit of a venture and share in the value of an ecosystem knowing that the system is resistant to the tragedy of the commons.

DAOs principally reside around the concept of members economically staking assets that have value and signalling via on-chain voting actions. This allows the group members to gather intent from a distributed group of people in a manner that is sybil-resistant.

This project will build a simple DAO module that allows members to stake assets that can only be unlocked after a period of time, and cast votes that represent on-chain governance in elections.

STAKING

The staking aspect is differentiated from simple freezing of assets in the sense that there is an unbonding period. Adding an unbonding period causes an opportunity cost for the member which dissuades malicious activity whilst staking.

This can be implemented by adding a time component to the existing freeze command. The time period can be based on block number ahead (1000 blocks) or seconds ahead.

```
4.1 stake(transactionData, timeToUnlockData)
     -> freezes asset and records the time to unlock data
```

The member can unlock their assets by calling the unstake handler:

```
4.2 unstake(transactionData)
     -> registers the unstake transaction. Assets are unlocked
after the time period.
```

After the time period is passed (or blockheight), the member's assets are movable.

VOTING

Members can elect to create elections by calling a registration handler. This registers an election in the registrar with a unique ID, with the type (simple, veto), minimum stake amount to vote and asset type, expiry (blockheight or time) and a minimum deposit amount.

```
4.3 createElection(type, asset, stakeAmount, expiry)
     -> creates an election with initial parameters
     -> returns electionID
```

As an example an election is specified that allows voting from members who are already staking at least 1000 TokenA, and who deposit at least 10 TokenA into the contract.

Members then vote by casting their choice with a deposit amount. The vote is registered in the chain state against the electionID. If a member who isn't staking the required amount votes, the vote fails.

```
4.4 vote(electionID, vote)
     -> registers a vote
```

At any time, or the after the election anyone can query the results of the election:

```
4.5 query(electionID)
     -> returns election details
```

EXAMPLE
Alice, Bob and Charlie are part of a DAO that requires 100 TKNs to be staked. They all stake their assets, with a minimum of 30 days unbending period. The assets stay on wallets they control.

Charlie starts an election that looks for support for an idea he has for the community, with an expiry time of 2 weeks. All three cast a vote to the election and the final result can be queried on chain after the period has ended.

OTHER ASPECTS
- **Election type.** More exotic election types can be created, such as elections with vetos, deposits and slashing.
- **Data Use**. A simple election entry will hold roughly 32 bytes per vote, with about 128 bytes of data to set up.
- **Funds.** More advanced DAO modules can be built, such as a DAO that combines a non-interactive multi-signature with voting to allow automated disbursement of assets.

5) CONTINUOUS LIQUIDITY POOL MODULE

OVERVIEW

Continuous Liquidity Pools (CLPs) are an exciting evolution of trustless asset swaps. Instead of a limit order-based exchange, assets are pooled together and a continuous market-maker algorithm employed to ensure that assets are always fairly priced. As a result, assets are always available and prices are always market-based. This model has been used successfully in Uniswap on Ethereum.

This project will build a module that allows users to stake assets and BNB, and then perform trustless swaps across pools. A widget can then be widely distributed across dApps, Wallets and webApps with always-on liquidity and fair market-based prices. This module is exciting because it will drive staking demand for BNB and BinanceChain assets, and solve liquidity problems for the ecosystem since it has the correct incentives to stake assets and earn fees. Developers can add a swap widget in their apps to instantly convert assets at market prices trustlessly with no counter-party.

THEORY

Assets on one side of the pool are bound to the base asset (BNB) on the other. The output can be determined given an input and pool depth:

$$X * Y = K$$

$$\frac{y}{Y} = \frac{x}{x+X} \Rightarrow y = \frac{xY}{x+X}$$

DEFINITIONS

X	*Balance of TKN in the input side of the pool*	x	*Input*
Y	*Balance of BNB in the output side of the pool*	y	*Output*
K	*Constant*		

PRICES AND SLIP

The expected slip trade and the pool, based only on the input and the depth of the input side of the pool can be determined:

$$P_0 = \frac{X}{Y}, \quad P_1 = \frac{X+x}{Y-y}$$

$$outputSlip = \frac{x/P_0 - t}{x/P_0} = 1 - \frac{Xy}{xY} = \frac{x}{x+X}$$

$$poolSlip = \frac{P_1 - P_0}{P_0} = \frac{xY+Xy}{XY-Xy} = \frac{x(2X+x)}{X^2}$$

DEFINITIONS

P0	*Starting Price*	*outputSlip*	*Slip of the output compared to input*
P1	*Final Price*	*poolSlip*	*Slip of the pool after the output is removed*

LIQUIDITY FEE
Stakers stake symmetrically and earn liquidity fees, which is proportional to slip. Slip is proportional to trade size and liquidity depth. Thus staking is incentivised in pools with out-sized trades.

$$liqFee = tradeSlip * tokensOutputted$$

$$liqFee = \frac{x}{x+X} * \frac{xY}{x+X} = \frac{x^2Y}{(x+X)^2}$$

$$tokensEmitted = tokensOutputted - liqFee$$

$$tokensEmitted = \frac{xY}{x+X} - \frac{x^2Y}{(x+X)^2}$$

$$tokensEmitted = \frac{xYX}{(x+X)^2}$$

$$swapSlip = \frac{xY/X - tokensEmitted}{xY/X} = \frac{x(2X+x)}{(x+X)^2}$$

$$poolSlip = \frac{x(2X+x)}{X^2}$$

DEFINITIONS

tokensOutputted	*Assets outputted from the formula before the fee is applied.*	*outputSlip*	*The slip of price between input and output*
tokensEmitted	*Assets emitted from the pool after the fee is applied.*	*swapSlip*	*The slip of price between input and emission*
		poolSlip	*The slip of price in the pool after the swap*

Thus the final price that the user receives, as well as the final price of the pool, are:

$$Price_{user} = Price_0 * (1 - swapSlip)$$

$$Price_{pool} = Price_0 * (1 - poolSlip)$$

ATOMIC SWAP CALCULATIONS

A single pool, TKN1, is paired to BNB. The user wishes to swap TKN1 to BNB.

DEFINITIONS

X	*Balance of TKN1 in the input side of the pool*	x	*Input of TKN1*
Y	*Balance of BNB in the output side of the pool*	y	*Output of BNB*

$$tokensOutputted = \frac{xY}{x+X} \qquad outputSlip = \frac{x}{x+X}$$

$$liqFee = \frac{x^2Y}{(x+X)^2} \qquad tokensEmitted = \frac{xYX}{(x+X)^2}$$

$$swapSlip = \frac{x(2X+x)}{(x+X)^2} \qquad poolSlip = \frac{x(2X+x)}{X^2}$$

ATOMIC SWAPS OVER TWO POOLS:

There are two pools, TKN1 & TKN2, both paired to BNB. The user wishes to swap TKN1 to TKN2.

DEFINITIONS

X	*Balance of TKN1 in the input side of the pool*	x	*Input of TKN1*
Y	*Balance of BNB in the output side of the pool*	y	*Output of BNB*
B	*Intermediary Input Balance (BNB)*		
Z	*Final Output Balance (TKN2)*	z	*Final Output*

$$swapSlip_1 = \frac{x(2X+x)}{(x+X)^2} \quad liqFee_1 = \frac{x^2Y}{(x+X)^2} \quad int_{emission} = y = \frac{xYX}{(x+X)^2}$$

$$swapSlip_2 = \frac{y(2B+y)}{(y+B)^2} \quad liqFee_2 = \frac{y^2Z}{(y+B)^2} \quad tokensEmitted_2 = \frac{yZB}{(y+B)^2}$$

Using just the pool depths, and the input, the final output and slip can be determined:

$$tokensEmitted_2 = z = \frac{xXYBZ(x+X)^2}{(xXY+Bx^2+2BxX+BX^2)^2}$$

$$P_{X0} = \frac{X}{Y}, \quad P_{Z0} = \frac{C}{Z}, \quad P_0 = P_{X0} * P_{Z0} = \frac{XC}{YZ}$$

$$finalSlip = \frac{x/P_0 - z}{x/P_0} = \frac{\frac{xYZ}{XC} - z}{\frac{xYZ}{XC}} = 1 - \frac{C^2X^2(x+X)^2}{(C(x+X)^2 + xXY)^2}$$

POOL SHARE

Stakers stake assets to earn a share of the pool. Stake average is the average of their two stakes (of BNB and TKN) at the time they staked.

DEFINITIONS

B	*Balance of BNB in the pool*	*T*	*Balance of TKN in the pool*
stakeB	*Stake of BNB*	$stake_T$	*Stake of TKN*
poolFeesBNB	*Total accumulated fees in BNB*	*poolFeesTKN*	*Total accumulated fees in TKN*
$stakeAve_{Xi}$	*Averaged stake for staker X*	$stakeAve_X$	*Sum of averaged stakes for staker X*
$poolStake_X$	*Sum of all stakes for staker X*	*poolTotal*	*Sum of pool stakes for all stakers*
$poolShare_X$	*Share of the pool for staker X*	*poolShareX*	*Share of the pool for staker X*
BNBFeesX	*Share of the BNB fees for staker X*	*TKNFeesX*	*Share of the TKN fees for staker X*
BNBStakeX	*Share of the BNB fees for staker X*	*TKNStakeX*	*Share of the TKN fees for staker X*

$$stakeAve_{Xi} = (\frac{stake_B}{B + stake_B} + \frac{stake_T}{T + stake_T}) * \frac{1}{2}$$

$$poolStake_{Xi} = stakeAve_{Xi} * (T + stake_T)$$

Users can only withdraw their stake partially or fully, or add more. This is tracked as the same all all stakes for that user:

$$poolStake_X = [poolStake_{X0} + poolStake_{X1} + \dots n]$$

A further number tracks the sum of every averaged stake from every user that has been made into the pool, including the first:

$$poolTotal = \sum_{i=0}^{n} poolStake_i \quad \text{or } poolTotal = [poolStake_X + poolStake_Y + \dots n]$$

At any stage, any user's share of the pool and its fees is thus given by the proportion between their averaged stake and the pool total:

$$poolShare_X = \frac{poolStake_X}{pool_{total}}$$

Thus when a user *X* withdraws either the fees or their share of the pool, the following are the equations:

$$TKNFees_X = poolShare_X * poolFees_{TKN}$$
$$BNBFees_X = poolShare_X * poolFees_{BNB}$$
$$TKNStake_X = poolShare_X * bal_{TKN}$$
$$BNBStake_X = poolShare_X * bal_{BNB}$$

TRACKING POOL METRICS

Multi-key store holds all information regarding all pools.

DEFINITIONS

bal_{BNB}	*Total balance of BNB in the contract*	*BNBpooli*	*Balance of BNB reserved for pool i*
balTKN1	*Total balance of TKN1 in the contract*	*TKNpooli*	*Balance of TKN reserved for pool i*

The total balance of BNB represents the total balance of BNB held for each pool which matches the total balance held in the contract.

The balance for each pool is held separately, and is tracked across a static variable. It should equal the balance of the contract. The balance of every pool is the sum of all stakes and inputs, minus all emissions.

$$bal_{BNB} == self.balance(BNB)$$

$$bal_{BNB} = [BNB_{pool_1} + BNB_{pool_2} + \ldots n],$$

$$BNB_{pool1} = \sum_{i=0}^{n} stake_i + \sum_{i=0}^{n} inputs_i - \sum_{i=0}^{n} emissions_i$$

The balance for each pool is held separately, and is tracked across a static variable. The balance of every pool is the sum of all stakes and inputs, minus all emissions.

$$bal_{TKN1} == TKN_{pool1}$$
$$TKN_{pool1} = \sum_{i=0}^{n} stake_i + \sum_{i=0}^{n} inputs_i - \sum_{i=0}^{n} emissions_i$$

Thus when a trade across a pool occurs (BNB -> TKN), BNB and TKN balances are changed atomically:

$BNB_{pool1_1} = BNB_{pool1_0} + x,$
and $bal_{BNB_1} == bal_{BNB_0} + x$

$TKN_{pool1_1} = TKN_{pool1_0} - y,$
and $bal_{TKN_1} == bal_{TKN_0} - y$

Across two pools, (TKNA -> TKNB via BNB), all balances are changed atomically:

$TKN_{pool1_1} = TKN_{pool1_0} + x,$
and $bal_{TKN1_1} == bal_{TKN1_0} + x$

$BNB_{pool1_1} = BNB_{pool1_0} - y,$
$BNB_{pool1_1} = BNB_{pool1_0} + y,$
and $bal_{BNB_1} == bal_{BNB_0}$

$TKN_{pool1_1} = TKN_{pool1_0} - z,$
and $bal_{TKN1_1} == bal_{TKN1_0} - z$

The fees are recorded in a global variable, which sums up all transaction fees for that pool, minus every time they were distributed by stakers. A global variable tracks the total fees for that pool every accumulated.

$poolFees_{TKN1} = \sum_{i=0}^{n} liqFee_{TKN1_i} - \sum_{i=0}^{n} FeesDistributed_{TKN1_i}$, where

$poolFees_{TKN1} = \sum_{i=0}^{n} liqFee_{TKN1_i}$

Since BNB is held in a global balance, an array of fees for each pool must also be tracked:

$poolFees_{BNB_{1_i}} = \sum_{i=0}^{n} liqFee_{BNB_{1_i}} - \sum_{i=0}^{n} FeesDistributed_{BNB_{1_i}}$, where

$poolFees_{BNB_1} = [liqFee_{BNB_{1_0}} + liqFee_{BNB_{1_1}} + \ldots n]$, for n transactions, and

$poolFees_{BNB} = [liqFee_{BNB_1} + liqFee_{BNB_2} + \ldots n]$, for n pools.

EXAMPLE
Alice and Bob stake assets in the TKNA and TKNB pools, with BNB as the base pair. They earn fees based on liquidity depth and volume of each pool. Bob can now swap at market prices between BNB, TKNA and TKNB in any direction from anywhere.