Snark: Scaling Blockchain

06
2018
v0.194

Contents

Abstract

Data

The current design of blockchains, requires replaying all data from transaction index 0. This requires full chain data to be stored. This is how to achieve the current accurate UTXOs or State. Every transaction created needs to be stored, shared, and computed. The current distribution mechanism for this is blocks.

The new design creates a single zero knowledge proof and a merkle path to be stored for processing.

On-chain Data

The concept of blockchain as storage has become more prevalent. Blockchains are being used for on-chain storage. This further increases issue 1, but also compounds by adding execution overhead. On-chain data is specifically designed for proofs and not off-chain data.

Execution is decoupled from on-chain execution. Execution zero knowledge proofs are created and stored on-chain. Execution (such as the EVM) is scaled off-chain, secured on-chain.

Execution Scalability

Blockchain(s) are limited by throughput, each node has to replay execution to confirm state. Execution is only as fast as the slowest consensus joining node.

Execution decoupled from on-chain execution, allows for off-chain execution scalability. Verifiable computing (zero knowledge proofs) allow for asynchronous processing.

Network Scalability

With execution scalability and data bloat addressed, the next scalability vector is network propagation. Block size and p2p propagation increase fork frequency as either size or distance increase.

This led to the creation of a new type of Node. Network Nodes are deployed as a first point of contact for transactions and blocks. Network Nodes communicate via a separate protocol set using short codes for blocks and transactions to minimize network overhead, with a request on requirement principal.

Slow Consensus

Consensus was designed as a second class citizen. Gossip occurs for propagation. State transition occurs for state output. Then consensus occurs. Blocks are a scaling mechanism for Proof of Work based systems. Blocks are required to increase throughput and consensus is required to confirm blocks.

Snark incorporates consensus as a first class citizen, consensus is inherently part of gossip, and allows for real time finality of transactions

Economic Abstraction

ERC_{20} and other (E)VM based solutions create tokens as second class citizens. This limits the usage of tokens for economic fuel and reward.
State concurrency allows for concurrent state updates for first class citizen tokens.

1 Trustless Computing

The objective is trustless computing. We have a job provider j, and we have a worker w, with computation $c(a, b)$. j wishes to incentivize w to perform $c(a, b)$. w is assumed malicious. w could perform zero calculation and provide output ϕ. j has no way to verify the validity of ϕ. j rewards w.

Strategy One, j gives computation $c(a, b)$ to w_1, w_2, w_3, ... w_n. In order mentioned they provide ϕ, δ, δ, δ, j can assume δ is correct. j needs to reward w_2, w_3, till w_n. This is consensus based trustless computing. j rewarded for each successful result. *This is expensive.*

Strategy Two, *trusted hardware.* Trusted hardware ensures execution. Intel SGX is the most widely adopted. Requires pre-compilation of $c(a, b)$ into the enclave. Ensures $c(a, b)$ is trusted. Cloud providers do not currently have SGX mass adoption. *This excludes a large of a user base.*

Strategy Three, *zk-SNARKs (Strategy Three b, zk-STARKs, not currently production ready).* Trusted execution. Can be deployed to any system, no hardware limitation, one to one usage.

We will use libsnark and Zokrates to demonstrate how zk-SNARKs can solve trustless computing.

Given function $c(a, b)$ we can define it in as

```
def main(a,b):
  return a+b
```

Compiled via Zokrates, we receive output

```
Compiling add.code
Compiled program:
def main(a,b):
  return (a + b)
Compiled code written to 'out',
Human readable code to 'out.code'.
Number of constraints: 1
```

We compute the output for c given a:1 and b:1, with output 2 (easily verifiable visually)

```
Computing witness for:
def main(a,b):
       return (a + b)
Witness: {"a": 1, "b": 1, "out": 2}
```

Now we need to generate proof, given a witness of *"a": 1, "b": 1, "out": 2* and inputs of *1, 1, 1, 2*, we generate

Proof

$A = 0x541fe5245da55a46cecb6652384b5cab34d40$,
$0x15051b8fe37a5adf5b6df604134db9a696b4a5051$

$A_p = 0x2ba27b052527b48d516653332b4a19db63d$,
$0x1400f78265e0fa76828bdd96cfcafb59e6b8eae58$

$B = [0x415e735fc50f9f9f9ff1f50229b57e8ac6d6$,
$0x2f984d821dcaece59f63157aa273bbdad72d7913a]$,
x124595fc21a8dbefbfbcbd25e262f2146e01242d,
0x1bd8baae63531850c2c8508a637f5586972fb5055

$B_p = 0x1681530fdd46e99e4dd1aecfef0f6b5db8c$,
$0x1de7408fb56976238e1468f1c5243bf3dd6c2bfac$

$C = 0xda7717fc9a1de2df4217706f313f2d105dfd7$,
$0x1df8fc9e1c792fb723178fd59f4543e6c098d724d$

$C_p = 0x1366b4f96bc37dea3e80497101594f145f5$,
$0x2b4dcc139165568282c9fe48d6b38fb2d25c66928$

$H = 0x1830068bbe3a691e5e133d17450bb4abe0933$,
$0x3fa9b65402d42ed268af24396ecd10ed105d43a56$

$K = 0xd884128cb213e3d503cb207081dd1eb1db066$,
$0x2a34aa7bf099d72a7e12e43eb3f022fa94563472a$

The above proof will change based on inputs. If instead we execute $c(a, b)$ as $c(2, 2)$ we have witness

```
Computing witness for:
def main(a,b):
       return (a + b)
Witness: {"a": 2, "b": 2, "out": 4}
```

And we generate proof

Proof

$A = 0xdd0074a5eb8bef9f93d23e60614ae2023c012$,
$0x84f182531372f6b6b107bad7347a89565eb44d834$

$A_p = 0x21fefefe9fe6d5eb1aba42fd6dc4486329a$,
$0x3bba04f5fe4e292b53504ab5dd516c06bfb0c38476$

$B = [0x2102870c6e06616b3a36bc577ff65558575f$,
$0x2ea6d955cdf85efed75b31c0115b0a0427801bc08]$,
x11f96b0d84fa60c28298592fc452c82ecfd136fc4,
0x2737ba58389f11476259b6e60b7f69dccf497294c

$B_p = 0x27d5315a1bd60c20f66b9309901480cffaf6$,
$0x2c58cdf509223714a00a6501c8c41285666bc6023$

$C = 0x1727552a09eeef24b2c6de27e9a477649dd646$,
$0xc480d98902bc11b1aa58bdf7b77f0cf1317ec508d7$

$C_p = 0x2429d278d28b2239bd62567eca8c3946f2b4$,
$0x94cfba292c22119c0e7695a39d7fc98c1194ce4348$

$H = 0x2ec0e277bb480166f69c7802db72d7f4368009$,
$0xa2ef09658b53c6bdffaf60f281f110eb70fc599680$

$K = 0xb982887647e7233c7a0e596732dc49a11379ac$,
$0x2ce92003ae12da46f0f61a3027ad76842171a474a4$

As part of compiling $c(a, b)$ we generated a verification key

Verification Key

$A = [0x541d36c3829ea9aac29a2998ade42d4605706$,
$0x78807cf5ba700c48778a28ad853034775814996d2]$,
x28dccfd45dc98460507fe1e2115380b637dd6aa31,
0x6a5a4cce6ab4bb87556ab1bf0c1b02ee7a4e179c0
)

$B = 0x102c07154f29573b3bd81fa12341dec3f0326b$,
$0x713a8567660c7614330568439b91f5c8107434a999$

$C = [0x82a168432a424803dc52e79436c6fe725804e$,
$0x26a800643ef7c2afad9184465d837e4f9f2e21186]$,
x9835f892538a70adbf5a654224a85714e586ee70a,
0x1826d31e0e821e7a4e8661e5dd1111c6960b643535

$gamma = [0x2d9a228dc20c28e8c748458882435c65c$,
$0x2f5a7c2b081852b4183b1c3be8874343552c81805cf]$,
x187b417e65cade35542c881d83043594926157fdd5,
0x111f855c55be54781545df00b4351c03740cafe07e

$gammaBeta1 = x7b0b5dcc51a534b36681405841b22993$,
$0x2aecea0cbbb5e0c21fc4e135520a29d1345cda20c99$

$gammaBeta2 = 0x23271135bd4f510bb68462b21a84e1$,
$0x1e5ef5da96e2f93a96628038db68fd2534d09673c]$,
x87b354ba532413585ffe076cfa780fd54353d17b3,
0x2ac5269aed51e34532366325bc1997bd286413a90

$Z = [0x152db951f108453468fab572c2663945afc6d$,
$0x23f6b1ece5ecd1221fc2334537aff6aa56f9212076]$,
x1dee23db1f5ab7234532b36baa161e56cc597535bb,
0x32127bee8a254ee65bbc345b6213ff856d3b741624

$IC[0] = 0x48c5ba00654348782985999 6539a6036556$,
$0x2b99882e215915c0a4556543452e09655fcc61ca033$

$IC[1] = 0x2c4404345443aaa83128a6850bde150c635$,
$0x1e1c067398244b87f90f4534332991116384235 7c8$;

$IC[2] = 0x1b80766deb05430912893f56685a20a16a6$,
$0x8d7e990328c1036cd7e5964345435634cfc63e6bddc$

$IC[3] = 0x23594fe3ebb6544bdc6238bf47e68f09f4$,
$0x11f45ed83caf38374b7f654483292a184685c82683$

Given Proof π, and Inputs i we can use Verification Key vk, to validate Inputs i (public inputs a, b and outputs o) for computation $c(a, b)$

To understand the above, let us discuss interactive proofs in a practical example. We want to buy item x from you for 1 token (1_{tk}). You want to make sure we have 1_{tk}, we show you our wallet address $0xA$, you can confirm it contains 1_{tk}. You are unsure that we are the owner of wallet $0xA$, so you use the public key (or wallet address) $0xA$ to sign message m, that creates hash h. You give us hash h. We use our private key pk to decrypt hash h and read message m, we respond to you with message m (or perhaps instead answer a to question phrased in message m).

Now you have proof that we own $0xA$. This was an interactive proof, specifically you provided m and h.

Now, if instead we wanted to preempt your request for proof. We created message m (With the time and your name) which outputs hash h. If we then give you m, h and $0xA$, you can confirm that h was created from m by the pk for $0xA$. This was a non interactive (or zero knowledge) proof.

Consider this same concept for zk-SNARKs, we can confirm that Verification Key vk is generated by compiling $c(a, b)$, and given inputs i and outputs o the worker w can create a Proving Key pk that can be used along with vk to confirm the output.

This allows us to achieve verifiable computing in a trustless space.

2 On-chain proof

We compartmentalize Ethereum into two sections. One, we describe as the native state. Ethereum addresses and balances. Two, we describe as the EVM. An ERC_{20} token exists within the EVM, we can describe it as EVM addresses and balances.

EVM execution is confirmed by every node. Every node executes the smart contract to arrive at the same output. This is a form of verifiable computing.

We have computation *c(a,b)* to execute. We give *c(a,b)* to untrusted parties. We assume standard $^2/_3$ fault tolerance, so we send *c(a,b)* to 3 parties. 2 parties return with the same results for *c(a,b)*. We assume the result for *c(a,b)* is correct. This is verifiable computing.

There are a few forms of verifiable computing, our two focus areas are;

- Intel SGX (Also known as trusted hardware)
- zk-SNARKs (zero-knowledge succinct non-interactive argument of knowledge)

zk-SNARKs prove with zero knowledge that something is true, and it is provable by providing zero knowledge.

EVM execution occurs on chain because we use multi party consensus to verify computing. We provide proof with zero knowledge that execution occurred in a trusted manner. Execution no longer needs to occur on-chain.

A zero knowledge proof EVM, can ensure verified computing.

We have secured execution, but we still have outputs, for example ERC_{20} addresses and balances. We have a zk-proof, that proves that a state transition occurred.

We knew state s_1, and we can prove that transition $\sum$ occurred, we can prove s_2. We need s_1 and transactions t_n to prove s_2. Given Merkle m and transition proof π, we can prove that participant p has balance b.

To have verified data, all the chain needs to save is merkle m and proof π.

$$t = \sum(s_1, t_n) = s_2 \rightarrow m$$

$$\sigma \leftarrow setup(t)$$

t with witness w

$\pi \leftarrow prove(\sigma, t_n, w)$

$verify(\sigma, t_n, \pi)$

3 On-chain data scaling

On-chain data is expanding. Given the deterministic nature of a blockchain, this will keep growing. One strategy is checkpoints. Points in time agreed to, ignoring historic data. We trust a specific point in time.

We spoke towards achieving verified computing results which give us a merkle path m and a zero knowledge proof π. Applying this same logic on-chain, would mean that at any given point, we only needed to store the m, and π.

Let's consider a block, a block is our proof of transactions in it. A block is proof of a state transition. State s_1 when applied with block b_n gives us state s_2. So b_n is a state transition. We showed how a state transition can be represented as a zero knowledge proof π. So instead of a block, we could represent this state transition with π.

Ethereum block size is currently 25k bytes, π, 288 bytes. At time of writing Ethereum is 1 099 511 627 776 bytes. By simply replacing the blocks with zero knowledge blocks, you would decrease the system to 12 195 622 907 bytes, 1% of the current size.

The above also considers you want to keep each state transition and merkle from the beginning of time, which isn't required. This is a different form of verified computing, a full deterministic transaction history isn't required. All that is required is the most recent merkle and most recent π.

4 Trustless stateless (E)VM

We have established that verifiable computing is possible in a trustless space. Next, we look at the Virtual Machine.

A Virtual Machine (VM) is an encapsulated execution environment. Following our previous example of Worker w, Job Provider j, and computation $c(a, b)$, w does not wish to execute arbitrary function $c(a, b)$ provided by j since j might be malicious and attempt to damage $w'ssystems.Soinstead$w executes $c(a, b)$ in a virtual machine. Here, even if damage is caused, it will not effect w directly.

The VM does not execute $c(a, b)$, $c(a, b)$ is compiled to bytecode b and then b lives inside the VM. This then allows $b.c(a, b)$ to be called.

The above separation is important for understanding the (Ethereum) Virtual Machine.

At the bottom layer we have the VM (EVM), as the second layer, living inside of the EVM we have the binary (compiled contracts and Application Binary Interface (ABI)), and at the top layer we have solidity contracts.

For those familiar with Java, this is the same as the JVM (Java VM), ByteCode (class files) and Java source code. We would have HelloWorld.java, compile it via the Java Compiler (javac) to HelloWorld.class, and then we can execute via the JVM.

For the EVM, we have HelloWorld.sol, we compile and deploy to address $0xC$. We can execute functions inside of $0xC$.

Consider $0xC$ as the address pointer.

For this article, we consider a stateless EVM. No data will be saved. Consider the stateless function;

```
def f(a,b):
 return a+b
```

Two important parts here. One, did the EVM compile the code correctly, and Two, if we execute $f(1,1)$, does it execute correctly.

For both of these, we provide a zk-SNARK enabled compiler, and a zk-SNARK enabled EVM.

Execution of $f(a,b)$ will provide witness $\backslash b\text{"} : 2, \backslash a\text{"} : 1, \backslash out_0\text{"} : 3$ as well as the Proving Key pk. The return is embedded and validated within the EVM as well as via Web3 compliant interfaces.

Execution in the EVM (or other compatible VMs) is ensured in a trustless environment. Consensus is no longer required for execution, EVM processing can be asynchronously parallelized, allowing for trusted off-chain execution.

5 Consensus first protocol

The block in blockchain, exists as a result of Proof of Work. If Proof of Work had to occur on a transaction level, we would have 1 transaction every 10 minutes (in Bitcoin). Instead we pack transactions into blocks, and apply Proof of Work to the block. This is a scaling mechanism for Proof of Work systems.

Proof of Work blockchains work off of economic detriment. You take a financial expense (electricity spent on Proof of Work) which assumes the incentive to be malicious is reduced.

Proof of Stake solutions replace Proof of Work solutions by assuming the cost incurred to stake offsets the benefit of being malicious. This statement is not true for systems that lack a punishment mechanism, slashing for Ethereum.

Current designs replace Proof of Work with Proof of Stake. Instead, we consider a new design. The current flow can be described as; transactions received, transactions executed, transactions packed in blocks, blocks propagated into the network.

In Proof of Stake, as soon as a block has been voted on by 51% staked majority it becomes finalized.

Consensus first, stipulates that the consensus is the primary foundation.

Current p2p systems keep track of *Node Info*. We can describe this as a set of *meta data, IP, name, other*. Consensus first adds *weight, reputation, stake, and transaction short codes* to this *Node Info*.

This allows the system to be aware of two metrics. One, the total stake of all connected nodes. Two, the total stake attributed to a specific transaction. When the total stake for a specific transaction reaches 51% majority, it can be removed from the pool.

This allows finalization of transactions in a Consensus first approach, and this is the cornerstone behind Eventual Consistent Consensus.

Consensus as a first class citizen, not consensus as an arbitrator.

6 Eventually Consistent

Eventual consistency states that eventually all items will be consistent across a network. We have so far established that using zk-SNARKs we can prove how something happened, but not necessarily what happened.

Let's consider Node n_1 with transaction tx_1, tx_2, tx_4 and Node n_2 with tx_1, tx_2, tx_3. If Node n_1 applies tx_1, tx_2, and tx_4 to state s_{1a} it arrives at zk $zk124$, and Node n_2 applies tx_1, tx_2, and tx_3 to state s_{1b} it arrives at zk $zk123$. The systems can be considered out of sync, however, as long as there is not a conflicting state (detailed later), then s_{1a} will eventually receive tx_3 and s_{1b} will eventually receive tx_4, and they will both be in $zk1234$. This is eventual consistency.

In a honest environment the above is great

to ensure parallel processing, since each node can simply process it's own transactions with the assumption of eventually reaching mutual consistency.

So what happens when bad actor a sends tx_{10a} with balance transfer from a to b to Node n_1 and tx_{10b} with balance transfer from a to c to Node n_2? We arrive in a conflicted state. Eventually tx_{10a} arrives at n_2 and tx_{10b} at n_1, both feel their result is the most correct, so how is this resolved?

A few solutions are available, standard consensus protocol, Proof-of-Work, Byzantine ($^2/_3$+1), or a random consensus protocol where a judge is elected.

Option 1: PoW, every time we have a new state transition zk we attempt to mine it, if we successfully mine it, we broadcast it out to the rest, this will then need to contain the same transactions which were used to achieve the state transition, and essentially we are back at a block, not a block for a chain, but simply a block to agree on the zk. This also has the overhead that it needs to happen on every event. We want to design a solution where the overhead of consensus is only called when needed. If the system is performing correctly, there is no need to disrupt the flow.

Option 2: Nodes vote on the respective zk's, as soon as one achieve $^2/_3 + 1$ majority it becomes the primary. Elegant solution, but a slow process.

Option 3: Random node is elected to decide which block is more accurate, could also end up having different results and lead to a new conflicting state, calling in a new judge, until an arbitrator is found.

The problem with eventual consistency, is that at any given point in time, the nodes might not be in agreed consensus, they will eventually get there, but how do you arbitrate at any given point in time, without having the specific knowledge of that time?

At first design, we consider time. If we could incorporate time we could select the first of the two events. This seems like an easy solution, we incorporate time in the signing process and we take the event that was first processed by whichever Node. While this might resolve the issue 99.99% of events, a pure conflict possibility does still exist.

We could simply ignore it, wait for tx_{10a} or tx_{10b} to propagate through the entire system and see which one gains majority, but this increases Time To Finality.

We could discard both.

The question we are trying to solve, is which one to apply first. Should we hold up the whole system while we decide? Ideally, we could remove the conflicting transactions (and any connected transactions), continue processing in an eventually consistent manner, and then start conflict resolution on the malicious event.

Consider 4 nodes, described as n_1, n_2, n_3, n_4, having staked 1, 2, 3, 4 TK respectively.

Total system stake is 10, with weightings 10%, 20%, 30%, and 40%. Soft majority at 51%, finality assumed at 67%.

At the start of a round the weightings are finalized.

- n_1 processes Transaction 1 denoted as tx_1. tx_1 has 10% weight.
- n_1 uses gossip to propagate tx_1 to n_2 and n_3. tx_1 has 60% weight (soft majority achieved).
- n_2 propagates tx_1 to n_4, 100% weight. tx_1 is finalized.

By virtue of having been processed by 67% of stake of nodes, we can consider a transaction finalized. All nodes become eventually consistent. This further allows for asynchronous processing of events.

A round will include the starting stake of all participating nodes. And all transactions finalized in

the round.

This round is replayed on the input state with state transition transactions to create the zero knowledge proof. The proof along with the current merkle root output is the current state.

No blocks, and no single state processor. Nodes participate in an ecosystem to benefit financially. Each round will have a fixed reward as well as accumulate all fees. Reward and fees are distributed proportionate to all participating nodes of the round.

7 Finalization

Blockchains are comprised of;

- p2p
- accounts
- transactions
- transaction pool or queue
- state processor
- consensus
- blocks
- blockchain
- rpc

A blockchain is an event driven system, we define the following events

- A user unlocks their wallet (*accounts*)
- A user creates a transaction (*accounts, transaction*)
- A user sends the transaction to a node's transaction pool(*transaction, rpc, transaction pool*)
- Transactions are received from nearby nodes (*transaction, p2p*)
- The transaction is proxied to known nodes (*transaction, p2p*)
- The transaction is validated and applied to the state (*transaction, transaction pool, state processor*)
- Successful transactions are put into a block (*transaction, block*)
- Blocks are mined (*block, consensus*)
- Blocks are received from nearby nodes (*block, p2p*)
- Received blocks are replayed for consensus (*block, consensus, transaction, state processor*)
- Successfully mined blocks are added to the chain (*block, blockchain*)
- Blocks added to the chain are propagated (*block, p2p*)

Snark's key changes

Transactions are grouped in *queued* — *pending* — *accepted* — *finalized*

- Transaction is received and set to queued
- Transaction nonce is checked to see if execution can occur, if false, stay queued, else move to pending
- Pending Transactions are applied to the state, if successful move to accepted, else discard
- Accepted transactions are monitored for 51% consensus. If achieved, move to finalized

Architectural changes

- Transaction pool includes accepted and finalized types

- Node provides stake information (for liveliness)
- Stake confirmed against precompiled smart contract
- Stake not matched ignored
- Node provides short codes for accepted but not finalized transactions
- Known Nodes transaction accepted stake / Known Nodes total stake $> 51\%$, finalize transaction
- Known Nodes transaction accepted stake / Known Nodes total stake $> r\%$ where $r > 51\%$ and < 100

Challenge, perpetual accepted transactions. Transactions that achieve $> 51\%$ but never reach $> r\%$

Challenge, deterministic replay. Node stake is mutable across time.

Both challenges are address with fixed voting epochs. Stake is locked in for a fixed epoch. Release requests only trigger after epoch end. Transactions not finalized in an epoch are reset.

Example

- A sends transaction tx_1 to Node n_1
- n_1 has (for each live node) knowledge of Node Info, Node accepted transactions, Node *Stake*.
- n_1 calculates tx_1 *known stake* 0% and the network *total stake* 100%.
- n_1 propagates tx_1 to all known Nodes.
- Known Nodes transmit Node Info, accepted transactions, *stake*.
- n_1 receives tx_1 in accepted transaction lists from nodes.
- n_1 calculates tx_1 known stake $x\%$ and the network total stake $y\%$
- $x/y > 51\%$, finalized transaction
- $x/y > r\%$, stop broadcasting transaction

Challenge, known nodes stake $< 51\%$ of total system stake. Transaction finalized and removed.

Liveliness is important, this issue is resolved with the Network Nodes and transmitting to Network Nodes for up to date data.

Challenge, all Network Nodes DDOS, and known nodes stake $< 51\%$ of total system stake.

Introduction of finalization confirmation.

Nodes transmit accepted transactions and current observed stake. When node has realized 51% stake, move transaction to finalized state. $r\%$ of known nodes set to finalized, hard finalize.

Challenge, $r - 51\%$ total stake owned by malicious node. Creating transaction DDOS. Slasher protocol for malicious behavior.

8 First Class State

Ethereum is a state consensus engine. It keep track of accounts and balances in the Ethereum state. The Ethereum State can be denoted as s_1.

Each ERC token can be categorized as a Sub State. ERC tokens are not first class citizens of the Ethereum State. Snark's design allows for state concurrency as first class citizens. Sub States are promoted to full States.

We use ERC contract representing token TK as our base. TK could fork Ethereum and launch their own mainnet. They would be incentivized to do this to allow TK to be used for fees and block rewards.

Snark allows for state concurrency. TK can launch a new State denoted as s_{TK}. State transitions will occur on s_{TK} instead of s_1. Transactions will be

denoted as tx_{TK}. This concurrency allows stakers to be rewarded with multi currency for each state they include.

This approaches *economic abstraction*. A key problem with non first class tokens are their inability to be used natively as fuel or gas. Current proposed solutions create DEX abstractions for swopping the underlying asset.

Snarks multi state concurrency allows for concurrent native tokens within one eco system.

The problem this solution often creates is hash overpowering. Since a sub state has significantly less hashing power, it is easy to redirect the attack. This is already addresses via the Consensus First, Eventually Consistent Consensus mechanism.

9 Economic Abstraction

Consensus free finalization allows for state concurrency updates.

A transaction can be described as

```
transaction {
 data //data struct below
 sender //who is sending the
     transaction
}
data {
 nonce //incrementing numeric value
     for transaction ordering
 price //price of doing the
     transaction
 gas //upper bound of gas the sender
      is willing to spend
 receiver //receiving entity
 value //value being transferred
 payload //mutable data
 v //signatures* (oversimplified)
 r //signatures* (oversimplified)
 s //signatures* (oversimplified)
}
```

The life cycle of this transaction can be described as

- Account creates and signs transaction (creation)
- Transactions is sent to Node via RPC (transport)
- Transaction is received and validated by node (queued)
- Transaction nonce is validated as possible transaction (pending)
- Transaction is unwound and applied to state (accepted)
- Transaction has 51% known acceptance (finalized)

Detail interest here is accepted.

State transition in Ethereum can be described as;

```
func (st *StateTransition)
    TransitionDb() (ret []byte,
    usedGas uint64, failed bool, err
     error) {
 sender := st.from() // err checked
     in preCheck
// Pay intrinsic gas
 gas, err := IntrinsicGas(st.data)
 if err = st.useGas(gas); err != nil
      {
  return nil, 0, false, err
 }
var (
  evm = st.evm
  vmerr error
 )
 // Increment the nonce for the next
      transaction
```

```
  st.state.SetNonce(sender.Address()
      , st.state.GetNonce(sender.
      Address())+1)
  ret, st.gas, vmerr = evm.Call(
      sender, st.to().Address(), st.
      data, st.gas, st.value)
 if vmerr != nil {
  if vmerr == vm.
      ErrInsufficientBalance {
   return nil, 0, false, vmerr
  }
 }
 st.refundGas()
 st.state.AddBalance(st.evm.Coinbase
     , new(big.Int).Mul(new(big.Int).
     SetUint64(st.gasUsed()), st.
     gasPrice))
return ret, st.gasUsed(), vmerr !=
    nil, err
}
sender = st.from().Address()
nonce = st.state.GetNonce(st.from().
    Address())
to = st.to().Address()
value = st.value
gas = st.gas
data = st.data
```

EVM transition can be described as;

```
func (evm *EVM) Call(caller
    ContractRef, addr common.Address
    , input []byte, gas uint64,
    value *big.Int) (ret []byte,
    leftOverGas uint64, err error) {
var (
  to = AccountRef(addr)
  snapshot = evm.StateDB.Snapshot()
 )
 evm.Transfer(evm.StateDB, caller.
     Address(), to.Address(), value)
 if err != nil {
  evm.StateDB.RevertToSnapshot(
      snapshot)
  if err != errExecutionReverted {
   contract.UseGas(contract.Gas)
  }
 }
 return ret, contract.Gas, err
}
func Transfer(db vm.StateDB, sender,
     recipient common.Address,
    amount *big.Int) {
 db.SubBalance(sender, amount)
 db.AddBalance(recipient, amount)
}
```

State DB

```
StateDB {
 db Database
 trie Trie
// This map holds 'live' objects,
    which will get modified while
    processing a state transition.
 stateObjects map[common.Address]*
     stateObject
 stateObjectsDirty map[common.
     Address]struct{}
// DB error.
 // State objects are used by the
     consensus core and VM which are
 // unable to deal with database-
     level errors. Any error that
     occurs
 // during a database read is
     memoized here and will
     eventually be returned
 // by StateDB.Commit.
 dbErr error
// The refund counter, also used by
    state transitioning.
 refund uint64
thash, bhash common.Hash
 txIndex int
 logs map[common.Hash][]*types.Log
 logSize uint
preimages map[common.Hash][]byte
```

```
// Journal of state modifications.
    This is the backbone of
 // Snapshot and RevertToSnapshot.
 journal journal
 validRevisions []revision
 nextRevisionId int
lock sync.Mutex
}
```

Simply put, a transaction informs the state to subtract value from the sender, and add value to the recipient. Fees for this transaction are added to the block creator. Block reward is provided to the block miner.

A state transitions is therefor tx_n applied to s_1 to create s_2.

s_1 in the current example would be ETH, so we can say s_{1ETH}. Snark allows for s_{1ETH} to co-exist along with s_{1BTC}.

The following changes are implemented

```
transaction {
 state //ID of the state transaction
      should be applied to

 data //data struct below
 sender //who is sending the
     transaction
}
data {
 nonce //incrementing numeric value
     for transaction ordering
 price //price of doing the
     transaction
 gas //upper bound of gas the sender
      is willing to spend
 receiver //receiving entity
 value //value being transferred
 payload //mutable data
 v //signatures* (oversimplified)
 r //signatures* (oversimplified)
 s //signatures* (oversimplified)
}
```

```
func (env *Work) commitTransaction(
    tx *types.Transaction) (error,
    []*types.Log) {
 state := env.GetState(tx.getState()
     )
 snap := state.Snapshot()
receipt, _, err := core.
    ApplyTransaction(state.config,
    state.gp, state, state.header,
    tx, &state.header.GasUsed, state
    .Config{})
 if err != nil {
  state.RevertToSnapshot(snap)
  return err, nil
 }
 env.txs = append(env.txs, tx)
 env.receipts = append(env.receipts,
      receipt)
 env.setState(state, tx.getState())
return nil, receipt.Logs
}
```

A transaction is applied to a specific state. This allows for the creation of new states, as well as configurable rules per state. (*OpCodes* remain fixed for now)

10 Network Node

After storage and execution are addressed, we are left with networking. Forks increase as block size increases. Forks increase as block time decreases.

We extend Boot Nodes to provide further functionality as first point of contact Network Nodes. Boot Nodes are the cornerstone of the ecosystem. They are assumed at least to some extent, as always online. They are the entry point for new nodes into the system. p2p propagation using gossip, is a slow process that spreads across a network. With Consensus First Protocol, we no longer needs blocks.

So instead our focus is on transactions. Transactions are sent to Network Nodes as a point of first contact, and then P2P propagation occurs as required or as a fallback should Network Nodes be forced offline. Transactions are first pulled from Network Nodes, before neighbor requests occur.

Network Nodes are configured on an infrastructure layer. Using data center infrastructure to connect Network Nodes and provide minimum distance routing. Network Nodes are designed to propagate amongst each other based on the shortest configured route. As a secondary mechanism they begin their gossip protocol.

By reducing time to transfer processing can occur faster and potential forks can be decreased.

With the design of Consensus First Protocol, we also see the Node Info struct as well as the Node Info transaction list. Transactions are encoded with short references to avoid full transfer. Network Nodes are also a first point of contact for the full Node Info struct.

The above implementation allows for standard propagation with a first point of contact mechanism to achieve faster transactional volume.

```
// This file is MIT Licensed.
//
// Copyright 2017 Christian Reitwiessner
// Permission is hereby granted, free of charge, to any person obtaining a copy of this
    software and associated documentation files (the "Software"), to deal in the Software
     without restriction, including without limitation the rights to use, copy, modify,
    merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
    permit persons to whom the Software is furnished to do so, subject to the following
    conditions:
// The above copyright notice and this permission notice shall be included in all copies
    or substantial portions of the Software.
// THE SOFTWARE IS PROVIDED "AS␣IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
    INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
    PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
    HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
    CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
    OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

pragma solidity ^0.4.14;
library Pairing {
    struct G1Point {
        uint X;
        uint Y;
    }
    // Encoding of field elements is: X[0] * z + X[1]
    struct G2Point {
        uint[2] X;
        uint[2] Y;
    }
    /// @return the generator of G1
    function P1() pure internal returns (G1Point) {
        return G1Point(1, 2);
    }
    /// @return the generator of G2
    function P2() pure internal returns (G2Point) {
        return G2Point(
            [11559732032986387107991004021392285783925812861821192530917403151452391805634,

             10857046999023057135944570762232829481370756359578518086990519993285655852781],

            [4082367875863433681332203403145435568316851327593401208105741076214120093531,
             8495653923123431417604973247489272438418190587263600148770280649306958101930]
        );
    }
```

```
    /// @return the negation of p, i.e. p.addition(p.negate()) should be zero.
    function negate(G1Point p) pure internal returns (G1Point) {
        // The prime q in the base field F_q for G1
        uint q =
            21888242871839275222246405745257275088696311157297823662689037894645226208583;
        if (p.X == 0 && p.Y == 0)
            return G1Point(0, 0);
        return G1Point(p.X, q - (p.Y % q));
    }
    /// @return the sum of two points of G1
    function addition(G1Point p1, G1Point p2) internal returns (G1Point r) {
        uint[4] memory input;
        input[0] = p1.X;
        input[1] = p1.Y;
        input[2] = p2.X;
        input[3] = p2.Y;
        bool success;
        assembly {
            success := call(sub(gas, 2000), 6, 0, input, 0xc0, r, 0x60)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require(success);
    }
    /// @return the product of a point on G1 and a scalar, i.e.
    /// p == p.scalar_mul(1) and p.addition(p) == p.scalar_mul(2) for all points p.
    function scalar_mul(G1Point p, uint s) internal returns (G1Point r) {
        uint[3] memory input;
        input[0] = p.X;
        input[1] = p.Y;
        input[2] = s;
        bool success;
        assembly {
            success := call(sub(gas, 2000), 7, 0, input, 0x80, r, 0x60)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require (success);
    }
    /// @return the result of computing the pairing check
    /// e(p1[0], p2[0]) * .... * e(p1[n], p2[n]) == 1
    /// For example pairing([P1(), P1().negate()], [P2(), P2()]) should
    /// return true.
    function pairing(G1Point[] p1, G2Point[] p2) internal returns (bool) {
        require(p1.length == p2.length);
        uint elements = p1.length;
```

```
        uint inputSize = elements * 6;
        uint[] memory input = new uint[](inputSize);
        for (uint i = 0; i < elements; i++)
        {
            input[i * 6 + 0] = p1[i].X;
            input[i * 6 + 1] = p1[i].Y;
            input[i * 6 + 2] = p2[i].X[0];
            input[i * 6 + 3] = p2[i].X[1];
            input[i * 6 + 4] = p2[i].Y[0];
            input[i * 6 + 5] = p2[i].Y[1];
        }
        uint[1] memory out;
        bool success;
        assembly {
            success := call(sub(gas, 2000), 8, 0, add(input, 0x20), mul(inputSize, 0x20),
                out, 0x20)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require(success);
        return out[0] != 0;
    }
    /// Convenience method for a pairing check for two pairs.
    function pairingProd2(G1Point a1, G2Point a2, G1Point b1, G2Point b2) internal
        returns (bool) {
        G1Point[] memory p1 = new G1Point[](2);
        G2Point[] memory p2 = new G2Point[](2);
        p1[0] = a1;
        p1[1] = b1;
        p2[0] = a2;
        p2[1] = b2;
        return pairing(p1, p2);
    }
    /// Convenience method for a pairing check for three pairs.
    function pairingProd3(
            G1Point a1, G2Point a2,
            G1Point b1, G2Point b2,
            G1Point c1, G2Point c2
    ) internal returns (bool) {
        G1Point[] memory p1 = new G1Point[](3);
        G2Point[] memory p2 = new G2Point[](3);
        p1[0] = a1;
        p1[1] = b1;
        p1[2] = c1;
        p2[0] = a2;
        p2[1] = b2;
```

```
        p2[2] = c2;
        return pairing(p1, p2);
    }
    /// Convenience method for a pairing check for four pairs.
    function pairingProd4(
            G1Point a1, G2Point a2,
            G1Point b1, G2Point b2,
            G1Point c1, G2Point c2,
            G1Point d1, G2Point d2
    ) internal returns (bool) {
        G1Point[] memory p1 = new G1Point[](4);
        G2Point[] memory p2 = new G2Point[](4);
        p1[0] = a1;
        p1[1] = b1;
        p1[2] = c1;
        p1[3] = d1;
        p2[0] = a2;
        p2[1] = b2;
        p2[2] = c2;
        p2[3] = d2;
        return pairing(p1, p2);
    }
}
contract Verifier {
    using Pairing for *;
    struct VerifyingKey {
        Pairing.G2Point A;
        Pairing.G1Point B;
        Pairing.G2Point C;
        Pairing.G2Point gamma;
        Pairing.G1Point gammaBeta1;
        Pairing.G2Point gammaBeta2;
        Pairing.G2Point Z;
        Pairing.G1Point[] IC;
    }
    struct Proof {
        Pairing.G1Point A;
        Pairing.G1Point A_p;
        Pairing.G2Point B;
        Pairing.G1Point B_p;
        Pairing.G1Point C;
        Pairing.G1Point C_p;
        Pairing.G1Point K;
        Pairing.G1Point H;
    }
    function verifyingKey() pure internal returns (VerifyingKey vk) {
```

```
        vk.A = Pairing.G2Point([0
            x541d382d36c3829ea9aac29a2942d4605706920676a6f033f28049d682ae888, 0
            x78807cf5baf993700c48778a853034775814996d8f362224b17249b7c368182], [0
            x28dccfd45b25dc98460507fe1e80b637dd6aa31a21723b13c74f6195188150dd, 0
            x6a5a4cce6a00d2ab4bbab1bf0c1b02ee7a4e179c081f08d749229b1b854878e]);
        vk.B = Pairing.G1Point(0
            x102c0714f154f29573b3bd81fadec3f0326b3a98d3d6644431738abcc0a4c7e, 0
            x713a60c7619a84330568439b91f5c8107434a9997b51b08855a5d00b9a95a2);
        vk.C = Pairing.G2Point([0
            x82a1aefe424803dc52e79436c6fe725804e3c3dbf015a96443b98c310287c22, 0
            x26a800643ec7ff7c2afad918d837e4f9f2e21186d4ff5bc7ea1e4a6b9c83863c], [0
            x9835f892572738a70adbf5a4a85714e586ee70aeeb0da31e8e807ba8b55ac72, 0
            x1826d31e0e8d3e82e7a4e8661e5dd1111c6960b63771742baa74bfa413ec77a7]);
        vk.gamma = Pairing.G2Point([0
            x25c4d9a8dc20c28e8c748458882c65cbebc192baa2d65f1cde2419a397cb0591, 0
            x2f5a7c2b0833d1852b4183b1c3be8872c81805cf2b493b1ecb042b0d722d45a8], [0
            x187b417e665b5cade42c881d83094926157fdd5e09638cd510fc1d2654425568, 0
            x111f855c551bf7be54781df00b1c03740cafe07e235e5a12f789791af7d3463c]);
        vk.gammaBeta1 = Pairing.G1Point(0
            x7b0b5dccab3668140841b229939aef6954fb694a971f8fecdca20be198930bd, 0
            x2aecea0cbb65b5e0c21fc4e20a29d15cda20c9996bb9c1d3afaaf7c4fae184c8);
        vk.gammaBeta2 = Pairing.G2Point([0
            x23271bd4f0bb68462b21a84e13194bff408dc6065e11c1a10c1aa08ff6863d25, 0
            x1e5ef5da9639e2f93a96628038dbfd2d09673c896cda269dd1c870725be76934], [0
            x87b354ba5d732413585ffe076cfa0fd53d17b3e95e3ab64ced138206a66123e, 0
            x2ac5269aed757351e2366325bc1997bd2413a90cec7068863c50495ff8335948]);
        vk.Z = Pairing.G2Point([0
            x152db872951f10868fab572c26639afc6dac34cb8da9d5408a5eaccf46a0ab37, 0
            x23f6b1ece50ed4ecd1221fc237aff6aaf920765b56d64dc327bb3a3f91394f65], [0
            x1dee23db18fcf5ab722b36baa161ecc5975bb221e54adbb6926ba360d78d4e86, 0
            x32127bee8abe9254eebbcb6213ff8d3b741624df450f6f32ff607c9eba48e4b]);
        vk.IC = new Pairing.G1Point[](4);
        vk.IC[0] = Pairing.G1Point(0
            x48abfc5ba0068298599939a60365562bba6278d5b038adf5fb98c0ce3d3bfc0, 0
            x2b99882e2174a5915c0a4552e095fcc61ca033abe355f905fe0591b8dfa4611f);
        vk.IC[1] = Pairing.G1Point(0
            x2c2cd4404aaa8312a680bde150c635cb2802f300563cc3a2424d2ab33d43dfc6, 0
            x1e1c067398ca0244b87f90f9111638423457c84e74dff868a5aa0cb66020f94c);
        vk.IC[2] = Pairing.G1Point(0
            x1bb1e80766deb009f56685a20a16a6ccca4c15a638a85e23487bec040b3bbc5b, 0
            x8d7e9903285bfc1036cd7e59334cfc63e6bddca326d03796d4dcd9217d87727);
        vk.IC[3] = Pairing.G1Point(0
            x23c9f594fe3ebbbdc6bf47e68f09f4239c471f2b0ce488a4ec81e177be61620b, 0
            x11f45ed8aaea23caf38374b7f8292a15c82683e9e2bd41ae89164abccbb97c45);
    }
    function verify(uint[] input, Proof proof) internal returns (uint) {
```

```
        VerifyingKey memory vk = verifyingKey();
        require(input.length + 1 == vk.IC.length);
        // Compute the linear combination vk_x
        Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
        for (uint i = 0; i < input.length; i++)
            vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
        vk_x = Pairing.addition(vk_x, vk.IC[0]);
        if (!Pairing.pairingProd2(proof.A, vk.A, Pairing.negate(proof.A_p), Pairing.P2()))
             return 1;
        if (!Pairing.pairingProd2(vk.B, proof.B, Pairing.negate(proof.B_p), Pairing.P2()))
             return 2;
        if (!Pairing.pairingProd2(proof.C, vk.C, Pairing.negate(proof.C_p), Pairing.P2()))
             return 3;
        if (!Pairing.pairingProd3(
            proof.K, vk.gamma,
            Pairing.negate(Pairing.addition(vk_x, Pairing.addition(proof.A, proof.C))), vk.
                gammaBeta2,
            Pairing.negate(vk.gammaBeta1), proof.B
        )) return 4;
        if (!Pairing.pairingProd3(
                Pairing.addition(vk_x, proof.A), proof.B,
                Pairing.negate(proof.H), vk.Z,
                Pairing.negate(proof.C), Pairing.P2()
        )) return 5;
        return 0;
    }
    event Verified(string);
    function verifyTx(
            uint[2] a,
            uint[2] a_p,
            uint[2][2] b,
            uint[2] b_p,
            uint[2] c,
            uint[2] c_p,
            uint[2] h,
            uint[2] k,
            uint[3] input
        ) public returns (bool r) {
        Proof memory proof;
        proof.A = Pairing.G1Point(a[0], a[1]);
        proof.A_p = Pairing.G1Point(a_p[0], a_p[1]);
        proof.B = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
        proof.B_p = Pairing.G1Point(b_p[0], b_p[1]);
        proof.C = Pairing.G1Point(c[0], c[1]);
        proof.C_p = Pairing.G1Point(c_p[0], c_p[1]);
        proof.H = Pairing.G1Point(h[0], h[1]);
```

```
        proof.K = Pairing.G1Point(k[0], k[1]);
        uint[] memory inputValues = new uint[](input.length);
        for(uint i = 0; i < input.length; i++){
            inputValues[i] = input[i];
        }
        if (verify(inputValues, proof) == 0) {
            emit Verified("Transaction successfully verified.");
            return true;
        } else {
            return false;
        }
    }
}
```

Index

References

[1] Manuel Blum, Paul Feldman, and Silvio Micali. Non-Interactive Zero-Knowledge and Its Applications. Proceedings of the twentieth annual ACM symposium on Theory of computing (STOC 1988). 103–112. 1988

[2] Jens Groth, Rafail Ostrovsky, Amit Sahai: Perfect Non-interactive Zero Knowledge for NP. EUROCRYPT 2006: 339–358

[3] Jens Groth, Rafail Ostrovsky, Amit Sahai: Non-interactive Zaps and New Techniques for NIZK. CRYPTO 2006: 97–111

[4] Jens Groth, Amit Sahai: Efficient Non-interactive Proof Systems for Bilinear Groups. EUROCRYPT 2008: 415–432

[5] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. ASIACRYPT 2010: 321–340

[6] Uriel Feige, Dror Lapidot, Adi Shamir: Multiple Non-Interactive Zero Knowledge Proofs Under General Assumptions. SIAM J. Comput. 29(1): 1–28 (1999)

[7] Uriel Feige, Dror Lapidot, Adi Shamir: Multiple Non-Interactive Zero Knowledge Proofs Under General Assumptions. SIAM J. Comput. 29(1): 1–28 (1999)

[8] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. An adaptively-secure, semi-synchronous proof-of-stake protocol. (2017)

[9] Jean-Luc Beuchat, Jorge Enrique González Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, Tadanori Teruya, Pairing. High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves. (2010)

[10] Laura Fuentes-Castañeda, Edward Knapp, Francisco Rodríguez-Henríquez, SAC. Faster hashing to G2. (2011)