# DOS NETWORK

A Decentralized Oracle Service boosting blockchain usability with off-chain data & verifiable computing power

info@dos.network
Last update: 2019.02
v1.5

# Table of Contents

# 1. Introduction

## 1.1 Motivation

Blockchain, or the distributed ledger technology, is gaining massive attention all over the world. Emerging from bitcoin[1], blockchain has features such as distributed global states, tamperproof in the byzantine environment, and no central authorities or single point of failure; thus blockchain is believed to be the building block of trust and value exchange. Moreover, the advent of Ethereum[2] pushes decentralized economy one step further, by introducing user-defined states and a Turing-complete virtual machine, enabling the booming of decentralized applications[3] as well as smart contracts.

Although that nowadays the killer apps of blockchains are probably decentralized trustless crowdfunding and some Game-Theory based financial games, smart contracts and Dapps are being developed by talents in all industries: content monetization, decentralized cloud storage, insurance, games, decentralized casino and prediction markets, etc. We strongly believe that blockchains and smart contracts are still in their infancy, with more and more unimaginable use cases coming.

However, due to the inherent limitation of existing blockchains' consensus mechanisms and their deterministic virtual machines, currently there are two biggest issues hindering the widespread adoption of smart contracts and the emergence of large-scale commercial Dapps:

- Smart contracts can neither directly pull in internet data nor invoke external web api by themselves, while it's inevitable for any commercial applications such as insurance to interact with the real world especially with the internet.
- On-chain computation resource and capacity are in fact very expensive and limited on existing smart contract platforms such as Ethereum. Together with execution gas cost, block gas limit and verifier's dilemma[4], it leads to execution scalability issues and makes it infeasible, if not impossible, to achieve commercial computation goals such as large-scale matrix multiplication, AI model training, 3D rendering, etc. by on-chain computing in the smart contract.

Overcoming these problems is as important as resolving current blockchains' transaction throughput issues. It is necessary to make same amount of effort to enrich smart contracts and decentralized economy with real world events and more computing powers before blockchain hits mass adoption.

## 1.2 Problem Definition and Existing Approaches

In complexity and computability theory, we define oracle[5] as an entity which is capable of resolving any decision problem and/or function problem. Oracle is basically treated as a “black box" that provides solutions for any given request. Specifically, in the context of blockchain and smart contract, we classify oracle machine into two categories:

- **Data feed oracle**: Feeding external data to smart contracts upon request, unlocking the potential of interactions between business-level smart contract and off-chain events.
- **Computation oracle**: Performing user-defined computation-intensive tasks off-chain, supplying unlimited computing power to existing blockchains, bringing decentralized token economy to the traditional computation market.

But as oracle is an off-chain third-party service that is not governed by blockchain consensus mechanism, the main concern then is how to trust those third-party services’ honesty.

There’re several existing centralized and decentralized oracle solutions: some of them are focused on feeding external data to smart contracts, while others are focused on fixing the on-chain computation problem, with majority of them are built for Ethereum blockchain:

- **Data feed oracle**:
    - Oraclize[6] provides a centralized data feed oracle service for Ethereum blockchain. With the help of amazon web services (AWS) and TLSNotary proof[7], Oraclize proves itself to have performed the calling contracts’ requests faithfully. However, Dapps do not favor a centralized solution in essence - not only that the trust dependency is shifted to Oraclize company and then to Amazon, but also since a system is only as decentralized as the most centralized component within it, Oraclize becomes the single point of failure (SPOF)[8] in the whole stack. Another dilemma is that TLSNotary proofs themselves are gigantic in size, passing proofs back on-chain costs tons of gas, which is paid by the end user - the calling contract. The proofs cannot be easily verified on-chain by the calling contract in real time. The extra cost of the bloated proof plus a premium charged by Oraclize as their profit make the total cost of using Oraclize much higher than what is claimed in their document. Oraclize has been running its business since 2016 and it turns out to be a quite popular oracle service for the time being, probably due to the lack of other friendly and usable oracle services.

- Town Crier[9] is another centralized data feed oracle built on Ethereum blockchain, mainly by utilizing Intel Software Guard Extension (SGX) to deliver authenticated data feeds to smart contracts. SGX manages a trusted execution environment (TEE)[10] named “enclave”, inside which core user program code is executed and is protected against other malicious programs, including the operating system itself. The secure execution of the core user program code inside SGX enclave could also be remotely attested by end users. SGX and many other commercial TEE are closed-source and/or undocumented, thus the trust dependency is shifted to the design and implementation of Intel company as well as to hardware manufacturers. This may sound better than Oraclize’s solution, but Town Crier also comes with its own problems:
    - As a centralized solution, it suffers from the same SPOF issue mentioned above.
    - SGX suffers from the most recent security vulnerabilities like Foreshadow[11] and Spectre[11] targeting Intel CPU and SGX, which is not easily patchable without severely hurting performance. Researchers have also revealed other security issues such as synchronization bugs[12] and other attacks[13] to hijack control flow and leaking private information from SGX enclaves.
    - Town Crier only supports limited types of apis and data feeds, and it is exclusively built for Ethereum blockchain.

  Town Crier has started serving Ethereum mainnet since May, 2017. But it is seldom used comparing to Oraclize.

- Chainlink[14] is the first proposed decentralized oracle on Ethereum blockchain. It aims to perform on-chain aggregation with governances to ensure data correctness in its current development plan. On-chain aggregation appears to be straight-forward, but it has several drawbacks, the biggest of which is its excessive gas consumptions - the number of transactions spamming the blockchain is proportional to the number of oracle clients participated in each consensus round. In its long-term roadmap, it would explore approaches to support off-chain aggregation. However, its protocol and signature scheme is interactive and involves multiple rounds of message communications, and in worst condition it requires majority of off-chain clients to participate - thus with poor performance and scalability. Moreover, reputation based oracle node selection easily leads to the Matthew Effect and is prone to collusion and targeted attacks. It also claims in the long-term plan to explore the use of Intel SGX, the pros and cons of which has been described in the previous section. Chainlink has finished its token sale in September, 2017. Until now, it is still developing on-chain aggregation implementation and no usable product yet.

- Augur[15] and Gnosis[16] are decentralized prediction markets that bring the predicted results of real world events into blockchain by means of collective intelligence. Ideally voting is distributed to different token holders and the predicted result is consented by majority votes. Augur and Gnosis are good at reporting low-frequent and future events such as presidential election result, sports bettings, etc. However, they are not appropriate for reporting the more-often real-time and on-demand events due to heavy user engagement and therefore long delay. Moreover, the credibility of predicted result may be compromised if the token distribution is not even - e.g. Gnosis team is holding 90% of all GNO tokens.

- **Computation oracle**:
    - Oraclize also supports simple computation or script running by outsourcing to a sandboxed AWS virtual machine. The computation task needs to follow specific format defined by Oraclize, and any large-scale use of Oraclize as a computation oracle hasn't been seen yet. Besides that, other features of Oraclize can be found in the data feed oracle section above.

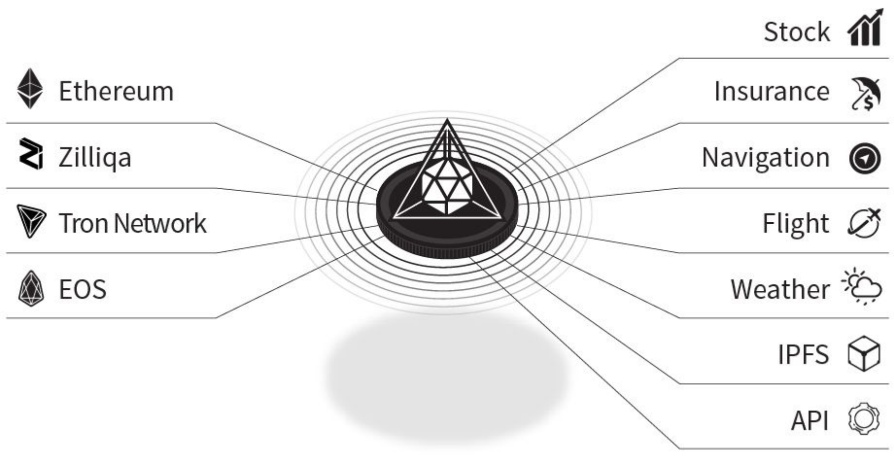
    - SGX based computation oracle: There are some projects and startups currently investigating the possibility of using SGX or other trusted hardware to perform off-chain computation. Although currently there is no existing workable product yet, this is an interesting topic attracting public attention. Below are some of our takeaways:
        - Regarding the technology of SGX itself, see the pros and cons discussed in the data feed oracle section above.
        - Regarding SGX's adoption rate and usability, SGX was first introduced in late 2015 along with the 6th generation Intel CPU based on Skylake microarchitecture. That means, a lot of personal computers and cloud servers are not yet equipped with SGX-enabled CPUs, thus SGX-based computation solution is limited in usability and it is actually highly centralized. Moreover, even with SGX-enabled processors, one also needs to turn on SGX in the BIOS settings[17][18] - this requires special knowledge to operate a personal computer, and many cloud server providers may not be willing to do so at all.

        - Regarding developer-friendly, except purchasing an SGX-enabled CPU and turning the BIOS settings on, developers need to request a commercial use license from Intel before using the SGX SDK, then to partition/refactor the user program code into untrusted and trusted parts, only the trusted part can run inside SGX enclave. This process is usually lengthy and bug-prone, and must be developed in low level programming languages such as C/C++, Rust, etc. For the trusted part running inside SGX enclave, there are also restrictions[18]: only limited numbers of library functions are provided by the SDK, many system calls (e.g. open a file) and CPU instructions are not allowed; the enclave memory size is also small, not originally designed for loading in all program code and user data directly.

    - Truebit[19] is the first proposed scalable off-chain computation protocol for Ethereum blockchain by means of interactive proofs between off-chain solvers and challengers and creatively-designed incentive models. In case of dispute, solvers and challengers play an interactive off-chain verification game[20] by recursively checking on the remaining computation steps until at the first point where they disagree with the state change after applying this computation step. The final judge then happens on-chain in the smart contract to determine which state change is valid. So truebit also needs to implement an on-chain interpreter in Solidity in order to execute this computation step. Truebit introduces the concept of “forced error rate” and maintains a “jackpot repository” in order to incentivize challengers to perform their due diligence. This basically forms into the architecture of verifiable computation[22], where computation is outsourced off-chain but verification happens on-chain. Verifiable computation is the ideal solution to bring scalable computation to blockchains. However, Truebit is still in its early days with tons of development to do. Also the complex incentive model and interactive verification game add more security risks to itself[21], we expect to see more experimentation and progress on Truebit.

## 1.3 Introducing DOS Network

DOS Network is a scalable layer-2 protocol that offers decentralized data feed oracle and decentralized verifiable computation oracle to mainstream blockchains. It connects on-chain smart contracts to off-chain internet data, also supplies unlimited verifiable computation power to blockchains, enabling more commercial applications with real world use cases.



DOS Network is chain-agnostic, meaning that it could serve all existing smart contract platforms; it is decentralized, meaning that it has no single point of failure, no central trust in a single company or special hardware, the trust lives in math and code; it is horizontally scalable, meaning that with more nodes running DOS client software the whole network offers more capability and computation power to supported blockchains; it is designed with cryptoeconomic models, meaning that the protocol is resistant to sybil attacks and the network effect is expanded with provable credibility.

DOS Network is partitioned into two layers with several key components:

- **On-chain part**: A set of DOS system contracts deployed on supported blockchains, mainly including functionalities such as request handling and response/computation result verification, node registration and staking, stats monitoring, payment processing, etc. On-chain system contracts also provide a universal interface to all user contracts across supported chains.
- **Off-chain part**: A client software implementing the core protocol run by third party users aiming for economic rewards, constituting a distributed network. Client software includes several important modules: event monitoring and chain adaptor module, distributed randomness engine module, off-chain group consensus module, and request processing/computation task processing module depending on the type of oracle service the user node provides.

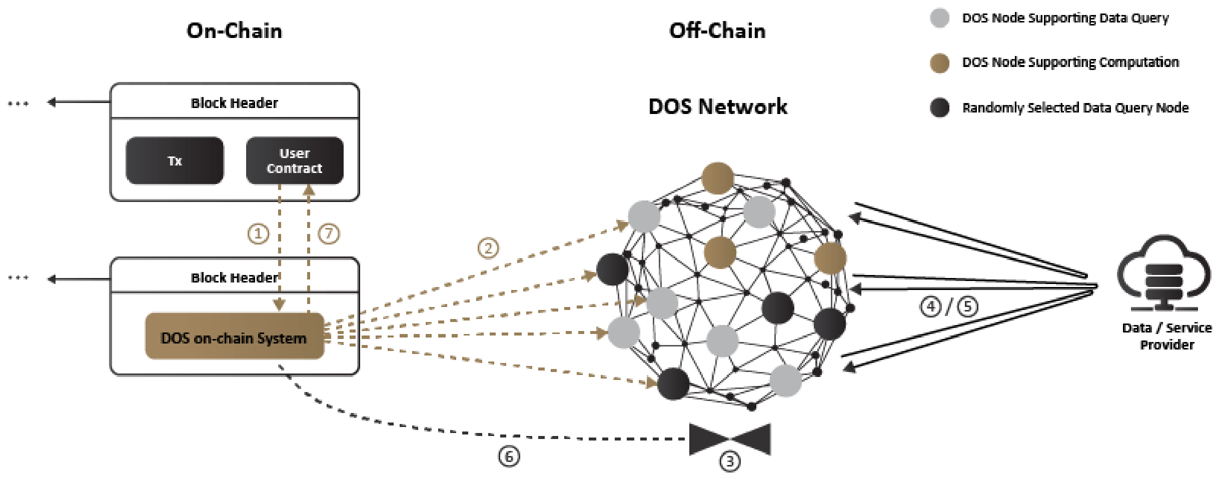
We will discuss aforementioned components of DOS Network in details in the following parts, analyzing DOS on-chain architecture and off-chain core protocols.

For data feed oracles we will demonstrate how the off-chain DOS clients reach consensus in the open and byzantine environment by means of unbiased verifiable randomness generation and non-interactive, deterministic threshold signatures. For computation oracles we will elaborate how we achieve the same verifiable computation architecture similar to that of Truebit but in a non-interactive way, powered by zkSNARK[23] and the state-of-art technique to generate zkSNARK public parameters (called the “setup phase”, or the “ceremony” by Zcash) in a scalable and trustless way[24] using the randomness engine we build for the data feed oracle.

We will also discuss DOS token economy which bootstraps and expands the network by incentivizing node operators to provide honest services in exchange for economic rewards. Based on the protocol and infrastructure we provide, a decentralized data feed marketplace could be built to onboard more data feeds for Dapps (demand side), and to allow premium data providers (supply side) to monetize from both blockchain traffic and traditional web traffic. A decentralized computation marketplace could also be built to bring decentralization and cryptoeconomics to commercial computation applications like video/audio transcoding, machine learning model training, 3D rendering and so on, which are currently monopolized by tech giants like Google, Amazon, Microsoft, etc.

## 2. Detailed Design

### 2.1 High-Level Architecture

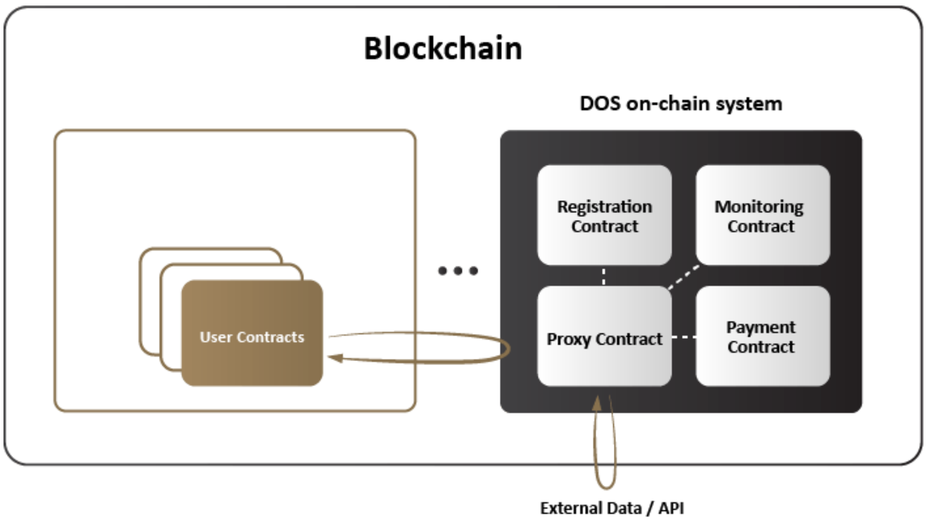


Life of an oracle request

We take Ethereum blockchain as an example to briefly discuss the overall process of an on-demand data query initiated by a user contract. It looks similar to the request

& response pattern, however, it is an asynchronous process from user contract's point of view:
①: User contract makes a data query request through a message call to DOS on-chain system (a bunch of smart contracts open sourced and published with well documentation provided to developers), specifically the DOS Proxy Contract;
②: DOS Proxy Contract triggers an event along with query parameters;
③: DOS clients (off-chain part of DOS Network running by users), which keep monitoring the blockchain for the defined event, are notified. Ideally there would be thousands of DOS nodes running, out of which a registered group will be randomly selected, by means of the distributed randomness engine built with verifiable random function (VRF).
④ & ⑤: Members in the selected group do the due diligence, calling a web api, performing a computation, or executing a configured script concurrently;
⑥: They will reach "in-group" consensus by the t-out-of-n threshold signature algorithm and report back the agreed result to DOS on-chain system, as long as more than t members in the randomly selected group are honest. The selected group members' identity and QoS (responsiveness/correctness, etc.) performance will be recorded on-chain, for monitoring and data analysis purposes.
⑦: DOS Proxy Contract notifies the user contract that the result is ready, by calling a callback function provided by user contract.
** The overall workflow of verifiable computation oracle looks similar and also make use of the distributed randomness engine, but with several differences for step 4~6. We'll discuss the details in later sections.

## 2.2 On-Chain Detailed Design



### 2.2.1 Proxy System

The proxy system provides standard on-chain interfaces to user contracts and will asynchronously callback to user contracts once the response is ready. The interface provided to user contracts is universal and simple as demonstrated below:

- Making a query request to DOS Proxy Contract;
- Consuming the result backfilled from __callback__ function to finish some post-processing work.

```
pragma solidity >= 0.4.24;
```

```
import "github.com/OpenZeppelin/zeppelin-solidity/contracts/ownership/Ownable.sol";
import "./DOSOnChainSDK.sol";

// An examnple user contract asks anything from off-chain world through a url.
contract Example is Ownable, DOSOnChainSDK {
    string public response;
    // query_id -> valid_status
    mapping(uint => bool) private _valid;
    bool public repeated_call = false;
    // Default timeout for Etheruem in seconds: Two blocks.
    uint public timeout = 14 * 2;
    string public last_queried_url;
    string public last_queried_selector;

    event SetTimeout(uint previousTimeout, uint newTimeout);
    event ResponseReady(uint requestId);

    function setQueryMode(bool new_mode) public onlyOwner {
        repeated_call = new_mode;
    }

    function setTimeout(uint new_timeout) public onlyOwner {
        emit SetTimeout(timeout, new_timeout);
        timeout = new_timeout;
    }

    function request(string memory url, string memory selector) public {
        last_queried_url = url;
        last_queried_selector = selector;
        uint id = DOSQuery(timeout, url, selector);
        if (id != 0x0) {
            _valid[id] = true;
        } else {
            revert("Invalid query id.");
        }
    }

    // User-defined callback function to take and process response.
    function __callback__(uint requestId, bytes memory result) external {
        require(msg.sender == fromDOSProxyContract(), "Unauthenticated response.");
        require(_valid[requestId], "Response with invalid query id!");

        emit ResponseReady(requestId);
        response = string(result);
        delete _valid[requestId];

        if (repeated_call) {
            request(last_queried_url, last_queried_selector);
        }
    }
}
```

Example of using on-chain sdk

### 2.2.2 On-chain Governance Systems

### Monitoring system

Monitoring system is proposed to keep on-chain records of the off-chain DOS nodes' QoS (Quality-of-Service) metrics and network stats, including:

- Random number generated by latest round's selected off-chain group, which could serve as a new kind of on-chain random source;
- Group size, number of registered groups, number of times each registered group has been selected, uptime and decomposition time to get rid of adaptive adversaries, etc.;
- Payment, weight percentage, and callback delay stats of processed and unhandled query requests;
- Quality score of registered off-chain DOS nodes including correctness and responsiveness of their reported results - nodes with extremely bad quality score will be excluded from off-chain consensus protocol and payment process;
- More...

Based on these rich on-chain metrics a monitoring Dapp could be built, showcasing the live status of DOS Network.

**Registration system**

For DOS off-chain nodes to join the network, they need to stake and lock some DOS tokens as security deposit, and to register their deposit address as well as payment address in the registration contract. They will be registered within at least one threshold group $G_i$ as groups may overlap with each other. The registration process of threshold groups are described in the off-chain architecture section below.

The security deposit makes the system resistant to sybil attacks and enhances network security. It also serves as a kind of commitment that the node operator would contribute bandwidth and computation power to strengthen DOS Network, and they will be compensated for "mining" rewards as well as earning processing fees. The lockup period helps stabilize the network to get rid of too frequent registration and deregistration flips. Any instance with out-of-bound offtime would also be penalized by forfeiting part of its deposit. Groups with no responses up to certain limit would be removed from registration system.

**Payment system**

The payment of a query request goes to the selected threshold group that handles it, and is distributed among honest members. Payments will be stored in the payment contract first, as the transfer to node runners doesn't necessarily happen in real time - a withdrawal pattern is preferable and node operators are able to check and withdraw their earnings through a frontend UI or interacting with payment contract directly.

DOS token is used in the form of natively supported payment token, as well as the staking token. However, for blockchains with widely-accepted stablecoins (e.g. Ethereum), the stablecoin would be a preferable payment token since node runners won’t be risking for the volatility of crypto prices; the pricing model for the fees will also be easier to make. We’ll support DOS as payment token first, in the long run, node runners and token holders will have governance rights to vote for which stablecoins (DAI/USDC/TUSD/etc.) to be accepted as extra payment tokens.

Different payment schemes will also be supported: pay-per-use will be widely adopted and suitable for personal developers and light-use Dapps, while discounted subscription model will be more favorable to heavily dependent applications such as stablecoins and other decentralized open finance platforms.

The on-chain system will take modular design pattern into mind that all on-chain contracts will be upgradable. Since it’s an open and distributed network environment with different parties have different economic appeals, there’s no simple perfect model to rule them all. More governance experiments and economic models will be researched and explored in future.

## 2.3 Off-chain Detailed Design

### 2.3.1 Part I: Decentralized Data Feed Oracle

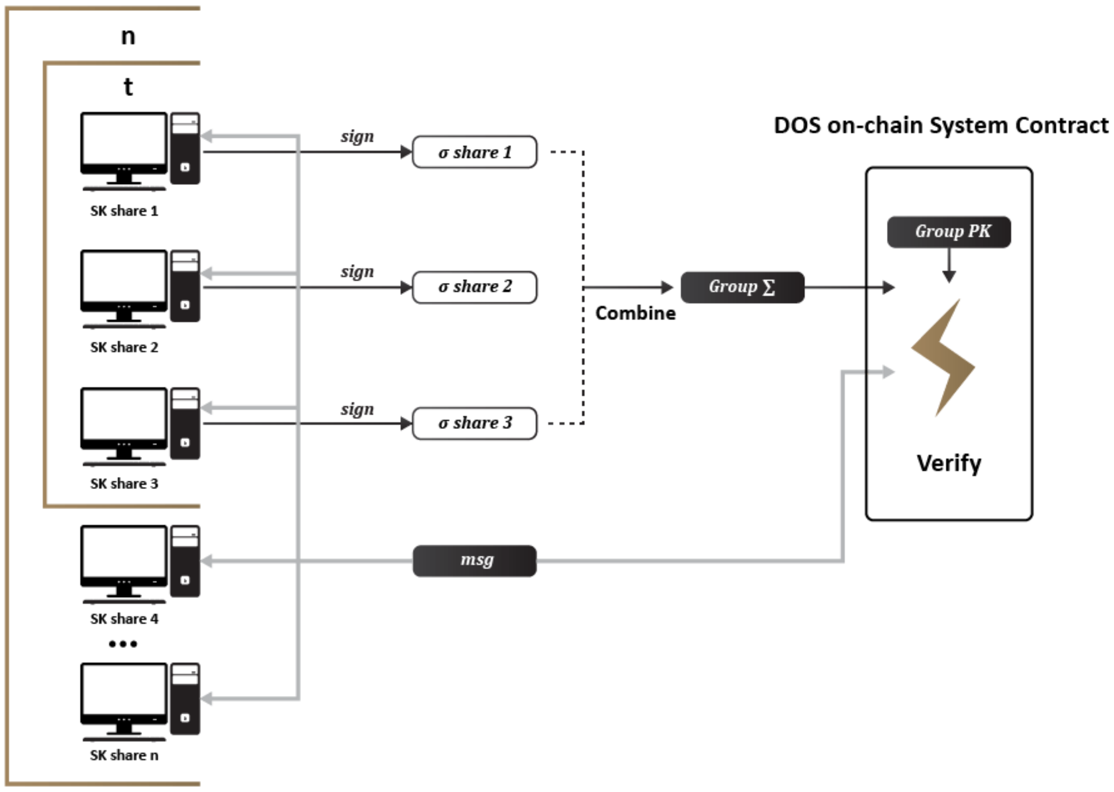
It looks very similar to the problem that blockchains need to resolve in order for off-chain DOS nodes to agree on the result of the same api call in open and byzantine network environment. Taking a look back of how bitcoin and other Proof-of-X system works, in essence they’re trying to achieve random leader election in each round of protocol, by means of certain resource that hopefully no single party can easily monopolize; as long as the majority of the network is honest, the blockchain reaches consensus in the sense of probability. Generating unbiased randomness is specifically important in reaching byzantine fault tolerant consensus. We demonstrate the off-chain consensus protocol below, mainly by leveraging verifiable random function (VRF) and threshold signature scheme[25]. To put it simple, instead of doing leader election among all nodes, we perform a random group selection among all registered groups for each protocol run; by utilizing the threshold signature cryptography, the selected group could reach off-chain consensus as long as more than $t$ (the threshold) members are honest.

We define the random number generated by last round $i-1$ is published on-chain as $r_{i-1}$; each registered group is of the same group size $M$; and there’re $T$ registered groups, then for current round $i$:

- **Group registration and re-grouping:** Newly registered nodes come to pending state first, once the pending pool contains enough pending nodes and they have been waited for sufficient time, they will be randomly selected to form new threshold groups. To handle the condition of “burst registration” while still generating honest majority group, $k$ working groups would be randomly selected and dissolved, along with $M$ pending nodes there would be $(k+1)\times M$ nodes in total. Fisher-Yates algorithm is applied to shuffle them into $k+1$ new working groups and get them registered. To eliminate adaptive adversaries, each group also carries a maturity period, say, several days, after which group members are dissolved into pending nodes. An one-time non-interactive DKG (Distributed Key Generation) protocol[26] with threshold $t$ will then be kicked off among members of newly registered group $G_i$ so that:
  - No centralized party holds the group secret key $G_{i,sk}$. $G_{i,sk}$ only logically exists but it will never be computed or revealed by any member during the lifetime of group $G_i$ unless a malicious party controls majority members in the group.
  - Each member $j$ will be allocated with a group secret key share $G^{j}_{i,sk}$ in a verifiable and trustless way.
  - The group public key $G_{i,pk}$ is collectively generated and published to the registration contract on-chain, updating number of registered groups $T = T + 1$.

- **Random group selection**: the randomly selected group for current round $i$ would be: $G_i = G[r_{i-1}\ mod\ T]$.

- **Group consensus via $(t, n)$ threshold signature**: Each off-chain member $j$ processes the query request and gets its corresponding response $D_j$. Member $j$ signes $D_j$ with its own group secret key share $G^{j}_{i,sk}$ by $\sigma^{j}_{i,D_j} = sign(D_j, G^{j}_{i,sk})$ and broadcasts the signature share $\sigma^{j}_{i,D_j}$ to its peers in the threshold group $G_i$, waiting for other members’ signature shares at the same time. Applying $(t, n)$ threshold signature scheme here means that at least $t$ members’ signature shares of the same response $D$ are required to generate a valid group signature $\sigma_{i,D}$, and any combination of the $t$ valid group signature shares could be combined into the same and deterministic $\sigma_{i,D}$, but there’s no way for less than $t$ members to generate a valid group signature.



(t,n) threshold signature scheme

- **Verification and payout**: The first honest member who successfully combines the group signature delivers $\sigma_{i,D}$ to its peers, $\{D, \sigma_{i,D}\}$ to the blockchain, and gets accepted after on-chain verification by $verify(D, G_{i,pk}, \sigma_{i,D})$. Other honest members stop processing upon receiving and verifying the group signature $\sigma_{i,D}$. Ideally there'll be only 1 valid response to the blockchain, but in cases multiple honest nodes report simultaneously, only the first verified response will be honored by the on-chain proxy system and others will be omitted. As threshold group is randomly selected and one DOS instance could belong to multiple groups, as long as DOS nodes are geographically distributed, the chance of each honest instance being reporter and getting accepted will be amortized during a period of time. Processing fee of an oracle request to the handled threshold group is locked in the on-chain payment system contract, and can be withdrawn by node runners at anytime. Ten percent of processing fees goes to foundation token pool, all honest members in the threshold group get even split of the remaining part.

- **Membership eligibility / Malicious member punishment**: Malicious members could send arbitrary response to the blockchain, however, they wouldn't pass the on-chain verification as long as they don't fetch and report the real response data in the first place, and they'll be flagged as ineligible and be excluded from future protocol run and the payment process. In this case their security deposit will also be forfeited, 50% of which will be burned forever, 25% will go to foundation token pool, and the remaining 25% will be deposited into the transaction fee reimbursement pool (see below section)

- **Freeloading protection**: Since all honest members get even split of the payout, the algorithm implicitly overcomes the "freeloading" issue, i.e. malicious nodes simply monitor pending transactions to the system contract and frontrun the pending ones with a much higher gas, without actually handling the request. Also in order to incentivize participation in the protocol and provide good bandwidth/computation resources with short delay, each group carries a quality score: if after certain timeout period there's no response for the selected group, its negative score is incremented and a subsequent group is selected. A group with abnormal negative/positive ratio or too high negative scores will be kicked out of protocol run and all members inside the group are losing if nobody participated.

- **Gas sustainability and tx fee reimbursement pool**: Ethereum's gas model requires the gas costs to be paid by node who initiates a transaction, including the gas costs of any subsequent message calls derived from that. The gas costs of delivering back the response data, group signature and executing the on-chain verification function need to be compensated to honest members to ensure the sustainability of the unstoppable, unbiased, and secure random number generation process, so a transaction fee reimbursement pool will be reserved. The initial funding comes from foundation donation and ecosystem building token pool, with 25% of malicious nodes' security deposit to be forfeited and deposited into the pool to cover future reimbursement. Noting that execution of calling contracts' customized callback function also costs gas, which should not be charged or reimbursed to node runners though, as the complexity and gas consumption of callback function is purely at the hands of calling contracts, thus should only be covered by themselves. Calling contracts are able to set the gas price and they need to make sure they have enough balance left to cover the gas consumption of their callback functions. The amount is escrowed in the payment contract, in case of response verification failure it would be returned to calling contracts, otherwise it'll be paid to the node successfully delivers response and passes verification.

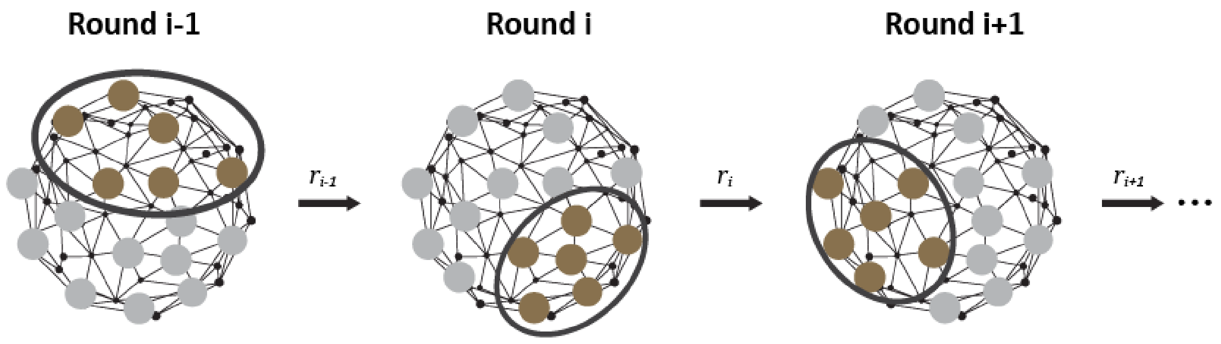
- **Non-interactiveness and immediate finality**: Unlike pBFT or other communication-based consensus algorithms, there's no multiple rounds of message transmissions. That is to say, the algorithm is scalable, even with group size of hundreds the estimated size of message transmitted is capped to several KBs in total. Also, unlike PoW/PoS or other resource-based consensus algorithms, the finality here is immediate. This is due to the magic of $(t, n)$ threshold signature that any $t$ valid signature share could recover the same original signature.

BLS threshold signature scheme is a good candidate because of its unique, deterministic, non-interactive, shorter signature and other features mentioned in [27].

- **Randomness generation for next round**: Member $j$ of $G_i$ signes $r_{i-1}$ using their group secret key share $G^j_{i,sk}$ by $\sigma^j_{i,r} = sign(last\ blockhash \parallel r_{i-1}, G^j_{i,sk})$, broadcasting group signature share $\sigma^j_{i,r}$ to peers and applying the same threshold signature scheme as above:

  - As long as one member receives any $t$ valid signature shares it combines them into the group signature $\sigma_{i,r}$. Thus random number generated for next round could be the hash of this group signature: $r_i = H(\sigma_{i,r})$, which is unpredictable until it's generated and it is also verifiable on-chain.

  - $r_i$ is published on-chain and is used to select threshold group and generate random number for the next round of protocol run - the iteration continues forever and this is called distributed randomness engine.



**Safety analysis and protocol enhancements**:

Assuming DOS Network has a total number of $N$ nodes, the number of byzantine nodes is $B$ where $N \geq 3B + 1$[28]. Threshold is set to $t$ meaning that at least $t$ members are required for a group to reach consensus. Define $P(k \mid M, B, N)$ to be the probability of randomly sampling $M$ nodes out of $N$ nodes, $k$ out of $M$ are byzantine nodes. Thus:

$$P(k \mid M, B, N) = \frac{C_B^k \cdot C_{N-B}^{M-k}}{C_N^M};$$

which conforms to the hypergeometric probability distribution[29]. Then the probability of selected group reporting byzantine result is:

$$P_{byzantine\ result} = P(k \geq t \mid M, B, N);$$

and the probability of selected group couldn't reach consensus for honest members is:

$$P_{no\ result} = P(M - k < t \mid M, B, N) = P(k \geq M - t + 1 \mid M, B, N);$$

Thus the confidence we have on the result is:

$$\begin{aligned} P_{confident} &= min \left\{ 1 - P_{byzantine\ result},\ 1 - P_{no\ result} \right\} \\ &= min\ \{P(k \leq t - 1 \mid M, B, N),\ P(k \leq M - t \mid M, B, N)\}; \end{aligned}$$

The probability of confidently reaching consensus on the correct result comes to maximum when $t = \lfloor \frac{M+1}{2} \rfloor$ ($M = 2t - 1$):

$$P_{confident} = P(k \leq t - 1 \mid M, B, N) = P(k < \lfloor \tfrac{M+1}{2} \rfloor \mid M, B, N);$$
$$P_{attack} = 1 - P_{confident};$$

using online calculator[30][31] it's obvious to get the result below:

| $N$ | $B$ | Byzantine ratio | $M$ | $t$ | $P_{confident}$ | $P_{attack}$ |
|---|---|---|---|---|---|---|
| 1000 | 100 | 10.0% | 21 | 11 | 99.999911% | $8.9000 \times 10^{-7}$ |
| 1000 | 150 | 15.0% | 33 | 17 | 99.999943% | $5.7200 \times 10^{-7}$ |
| 1000 | 200 | 20.0% | 49 | 25 | 99.999939% | $6.1300 \times 10^{-7}$ |
| 1000 | 250 | 25.0% | 73 | 37 | 99.999923% | $7.7400 \times 10^{-7}$ |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 1000 | 300 | 30.0% | 115 | 58 | 99.999920% | $8.0100 \times 10^{-7}$ |
| 1000 | 333 | 33.3% | 159 | 80 | 99.999905% | $9.5000 \times 10^{-7}$ |

Based on above observation we propose several protocol updates below:

- **Network size**: 1000 is a very reasonable short to mid term estimate: ZenCash has a network of ~7000[32] secure nodes as of 01/31/2018 after debut on Oct 2017; smartcash has a network of ~8000[33] masternodes as of 01/31/2018 only after 30 days launch announcement. We expect DOS Network grows into several thousand nodes in the long run.

- **Minimizing byzantine ratio**: The byzantine ratio of the whole DOS Network significantly affects system safety - even with a small decrease of byzantine ratio we see a huge confidence increase. Taking advantage of verifiable threshold signature scheme and the punishment of malicious nodes we could significantly decrease the byzantine ratio. Each group also carries a maturity period to eliminate adaptive adversaries.

- **Maximizing parallelism with two types of groups**: It is obvious that the larger the group size the safer DOS Network would be. However, there's a dilemma that the larger the group size, the unfriendlier it is to the data provider - we should get rid of someone abusing DOS Network to perform DDoS attack on the data source. With an aim to provide at least six nines (99.9999%, with 1 millionth attack rate) confidence as well as overcoming the dilemma, there'll be two types of threshold groups:
    - **Randomness engine group** - with group size large and safe enough e.g. 159, driving the whole system solely by generating unbiased and unpredictable random numbers.
    - **Worker group** - groups processing the real query, with the group size bounded to 11~21 to prevent exploiting data providers' resources. Worker groups are selected by the hash of random number $r$ generated by randomness engine group along with $query\_id_j$:

    $$G_{worker_{query_j}} = G_{workers}[H(r_{i-1} \,||\, query\_id_j) \ mod \ T_{workers}\ ];$$

    Thus different worker groups will be selected for different queries and they can be processed parallely to achieve maximum scalability.

### 2.3.2 Part II: Computation Oracle

Verifiable computation means a client could outsource the computing of some functions to untrusted third parties with enough computing power. The result, along with a proof demonstrating the validity and soundness of the result, would be returned to the client, the client could then execute a verification step instead of performing the original computation.

We could already provide a consensus-based computation oracle solution once part I is done, as the idea is quite straightforward: For each computation request, randomly select a group of computing nodes, execute the deterministic computation task by each group member and send back the result agreed by the majority. This could serve as our short term plan for the computation oracle service we'd like to offer to supported blockchains, however, there is a better option we'd like to offer in the long-term roadmap.

With the successful Byzantium hard fork[34] on October, 2017, Ethereum is now capable of verifying zkSNARK proofs in smart contracts (see Appendix II). This enables us to build a zkSNARK-based verifiable computation oracle, where the execution happens off-chain and verification happens on-chain. The whole process could be roughly broken down into three phases:

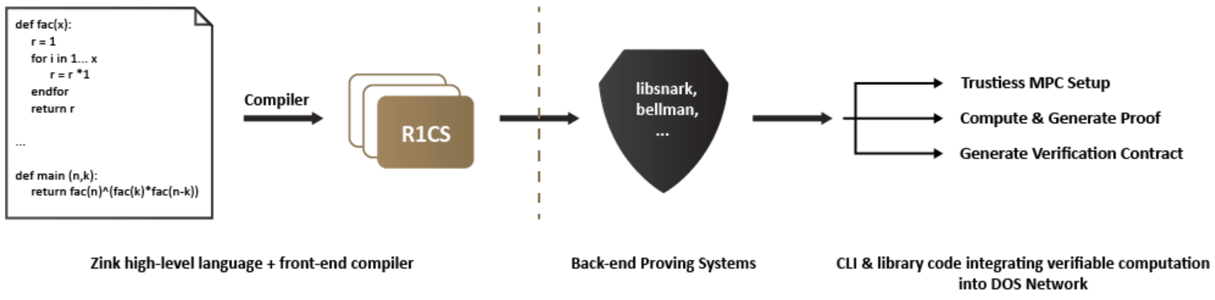
- **Setup phase**: For each specific computation, define $C$ is its equivalent arithmetic circuit[35], $\lambda$ is secret random numbers ("toxic-waste" by Zcash) that must be destroyed after the setup phase. Two public available keys would be generated by the setup phase: $(P_k, V_k) = Setup(\lambda, C)$, where $P_k$ is called proving key to be used in the off-chain computing phase and $V_k$ is called verification key to be used in on-chain verification phase. This setup phase only needs to run once for each type of computation task $C$, as long as the computation logic/step doesn't change, $P_k$ and $V_k$ could be reused for different inputs. A verification contract with hardcoded $V_k$ could be deployed on-chain by now to verify future computation results for different inputs (see an example contract in Appendix II).
- **Off-chain computation phase**: The computation could then be carried on off-chain with any valid input $i$. This could further be broken down into two steps: (i) Performing computation and come up with the result: $o = Compute(C, i)$. (ii) Generating a proof for $o$ using proving key $P_k$: $\pi_{proofs} = GenerateProof(C, P_k, i, o)$. The proof along with the computation result is then sent back on-chain to the verification contract.
- **On-chain verification phase**: One final step happening on-chain is to check the validity of computation result $o$ given corresponding input $i$: $verifier.at(contract\ address).Verify([\pi_{proofs}],\ [i,\ \ldots,\ o])$.

Comparing with consensus-based computation oracle or interactive verification game between solver and challenger (Truebit), zkSNARK based off-chain computation only needs to be executed once. There's no interaction between the prover and verifier, meaning it is non-interactive. The proof is succinct, meaning its size is small and independent of the complexity of the computation; the verification algorithm verifies the validity quickly and it’s also independent of the complexity of the computation, it only depends on the size of the input. These features make it an ideal solution to bring unbounded computation power and execution scalability to blockchains.

There’re several tricky points though. First is the toxic waste that’s used in the setup phase, it must be destroyed and must not be leaked otherwise fake proofs would be generated to attest computation. Second, for the arithmetic circuit generation, a computation task cannot apply zkSNARK directly before being converted to the right form called “Rank-1 Constraint System” (R1CS) then finally to “Quadratic Arithmetic Program” (QAP). Working directly with R1CS or QAP sounds like writing assembly code by hand, which is error-prone and involves non-trivial work, not friendly for most developers.

These problems need to be resolved in order to bring practical zkSNARK-based verifiable computation to blockchains. A new multiparty computation (MPC) protocol[24] that could be scaled to hundreds or even thousands of participants has recently been proposed by Zcash researchers to fix the trusted setup issue. The protocol has the property that all of the participants have to be compromised or dishonest in order to compromise the final parameters. Moreover, now the trustless setup has been separated into two stages: one big, single “system-wide” trustless setup called “Powers of Tau”[36] has produced partial zkSNARK public parameters to be reused for all zkSNARK circuits within a given size bound. An application-specific trustless MPC setup is still needed for each type of computation task, but because of the partial parameters produced by Powers of Tau, the application specific MPC is much cheaper and achievable through the randomly selected working group.

To address the second issue, we will define and formalize a domain specific language (DSL) called “Zinc” for verifiable computation, the grammar is similar to python or javascript, with high-level abstractions such as variables, conditions and flow control statements, loops, functions, module/file imports, etc. So developers are able to write off-chain computation code in high-level programming language without the need of understanding the crazy math under zkSNARK or to deal with low-level details like R1CS. We’ll also develop a toolchain including an SDK and a front-end compiler compiling high-level Zinc code into low-level R1CS, and connecting to existing proving system like SCIPR lab’s libsnark[37] as back-end. The toolchain will also provide a command line tool as well as library code to be integrated into DOS client software to enable verifiable computations and make them adaptable to supported chains automatically.

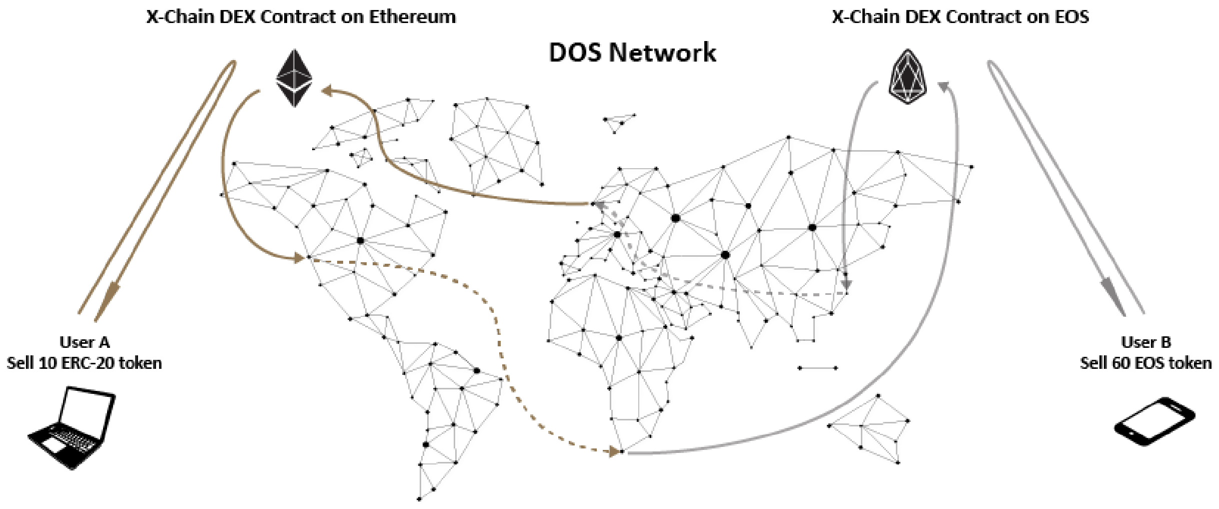


def fac(x):
r = 1
for i in 1... x
r = r *1
endfor
return r
...
def main (n,k):
return fac(n)^(fac(k)*fac(n-k))
Compiler
R1CS
libsnark, bellman, ...
Trustless MPC Setup
Compute & Generate Proof
Generate Verification Contract
Zink high-level language + front-end compiler
Back-end Proving Systems
CLI & library code integrating verifiable computation into DOS Network

## 2.4 Cross-Chain Interoperability

DOS Network opens a door to perform cross-chain interaction between heterogeneous blockchains. Assuming DOS Network supports data feed oracle services to both Ethereum blockchain and EOS blockchain, then theoretically, smart contract on Ethereum is able to trigger cross-chain state changes, flowing through DOS client nodes, calling into smart contract on EOS. DOS Network thus is acting as connectors or bridges between supported heterogeneous chains.

A simple application is like exchanging heterogenous crypto asset atomically. Decentralized exchanges nowadays can only trade homogeneous crypto assets in the same blockchain, decentralized exchanges e.g. EtherDelta or 0x relayers are unable to trade against EOS tokens directly. However, with the help of DOS Network, it’s achievable by deploying two DEX contracts on Ethereum and EOS blockchain respectively and defining two cooperative functions to trigger orderbook and account balance state changes upon calling from the other DEX contract address through DOS connectors. This example showcases the potential of DOS Network in cross-chain interoperability.

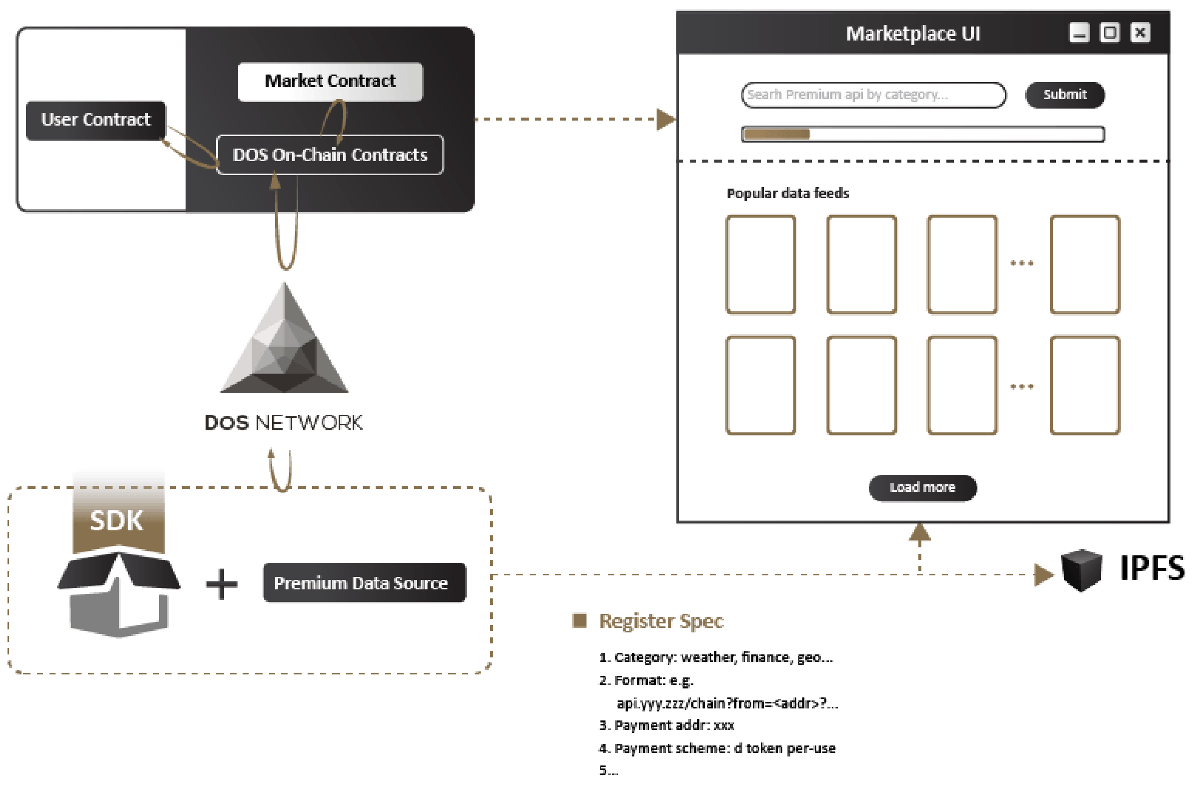


The operation and maintenance of any newly supported blockchain's nodes (full node, light node, or utilizing remote full node services like infura[38]) is up to node operators' own economic interests and capability, while DOS Network team is responsible for porting and deploying on-chain system contracts to newly supported chains and releasing off-chain core client software including protocol update and new adapter support.

To start up oracle services to newly supported chains we need to go through similar bootstrapping process as mentioned in section 2.3.1, mainly the one-time group registration and non-interactive DKG process. Noting that for various supported blockchains, chain-wide system variables like group size $M$ and number of registered groups $T$ could be different; the random number $r$ generated and published on different chains are also different in general.

## 3. Premium Data Feeds & Marketplace

Data and APIs are powering web apps, mobile apps, IoT devices, and are now growing into a multi-billion-dollar business[39]. In addition to bringing in open and free apis to blockchain through decentralized data feed oracles, we're also planning to build a data feed marketplace, specifically targeting premium data feeds and making them available on-chain. We keep the KISS principle[40] in mind for premium data providers and Dapp developers so that they only need to undergo minimum or even no change from their sides.



A market contract would be developed and deployed on supported public chains. Its main purposes are 1. For premium data providers to register and publish their specs and offers; and 2. A place to group and demonstrate premium data feeds for Dapp developers to choose and purchase from. We would host a server and marketplace UI frontend for better user experiences and support structured data discovery like searching and ranking premium data feeds by category and keyword. But actually all marketplace information is stored in market contract on-chain and in IPFS, our hosted server only acts as a caching layer and any direct interaction with market contract and IPFS is totally feasible.

Premium data providers register first to publish their specs and offers, including but not limited to api category, format, addresses receiving payments from the on-chain calling contracts, payment scheme e.g. pay-per-use, subscription-based, etc. An SDK is going to be released to premium data providers including functionalities of light clients with network connection to full nodes, so data providers are expected to make minimum changes from their side.

User contracts calling for a specific category of off-chain premium data would simply search for existing ones from marketplace contract and pick the most suitable one. If there's no such premium data feed available on-chain yet, developers could start a bounty using DOS tokens and the community would reach out to corresponding data providers to help integrate their service on-chain.

As all the on-chain data is public for now, no secret information like private key or api token should be stored on-chain in smart contracts. We're addressing the problem in other approaches: queries for premium off-chain data would use a different interface from querying open and free data, but the calling sequence is similar to normal ones and goes into proxy contract, which talks to market contract to make sure the requested premium data source has been registered and checks whether the calling contract has enough DOS tokens to pay to premium data providers. The payment is then held in on-chain payment contracts and wouldn't be delivered to premium data providers until the response is backfilled to the calling contract. The off-chain DOS Network talks to the premium data provider, with the help of light client SDK, data providers make sure the pending payment coming from a calling contract with matched query id is escrowed in the payment contract, before giving back data. In this way premium data providers are able to monetize from blockchain traffic without being abused by unpaid user contracts or normal unauthorized internet traffic.

# 4. Future R&D Work

Some related ongoing and future research work to improve DOS Network are listed below.

**Threshold cryptography, Distributed Key Generation, and VRF**

We are actively exploring other signature schemes besides threshold-BLS signature, with properties including verifiability, uniqueness/determinism, with non-interactive threshold version and short in size. Other advanced DKG protocols besides Feldman's and Pedersen's and verifiable secret sharing schemes are also in our eyes.

We're seeing emerging research and applications of verifiable random function in the blockchain space, notably the consensus engines of Algorand, Dfinity and Cardano are highly dependent on VRF. We'd like to explore more applications of VRF in byzantine fault-tolerant consensus algorithms and in non-interactive zero knowledge systems.

**Verifiable off-chain computaion**

We're actively exploring and working on zk-SNARK related topics especially like general purpose front-end compilers from high-level languages to R1CS, low-level libraries and implementations of proving systems e.g. libsnark, bellman, etc.

We're also evaluating and keeping eyes on other advanced verifiable computation techniques that are probably still more in theoretic stage than to be production ready, like fully homomorphic encryption[41], program obfuscation, and the latest scalable, toxic-waste-free, post-quantum-secure zk-STARK[42] technology.

# 5. Token Economy

We'll issue DOS token which is a utility token based on Ethereum ERC20 standard as Ethereum is the first blockchain we support. DOS token will be utilized to incentivize various participants in the system and ensure growth and development of the ecosystem. In future, we may swap tokens to other supported primary smart contract platforms like EOS, Thunder, Tron, etc.

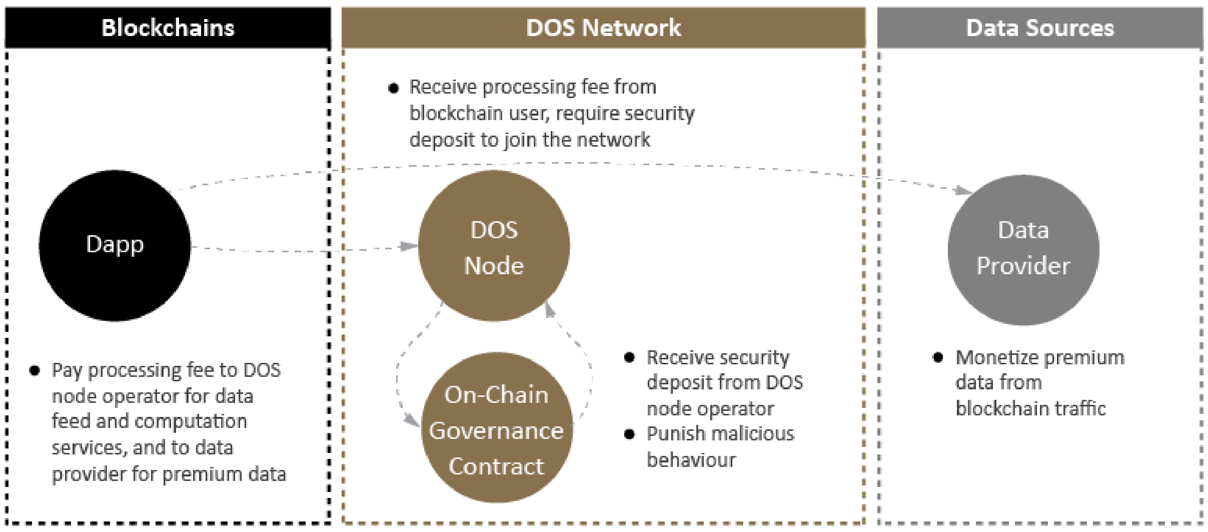
## 5.1 Participants in DOS System

There are mainly three types of participants in DOS Network. They are tightly connected by DOS token and together constitute the DOS ecosystem:

- **Dapp developers**: Dapp developers and/or calling contracts need to pay processing fees to DOS nodes for each fulfilled request on pay-per-use or subscription-based scheme in order to request external data or perform off-chain verifiable computation tasks. DOS token will be the first accepted fee token used to pay processing fees to node runners, however token holders have the governance right to vote for what else to be accepted as fee tokens, like for example, stablecoins. Developers also pay to premium data feeds per their specs and they could set bounties using DOS tokens on the marketplace contract to request for missing premium data feeds.

- **Node runners**: The off-chain DOS Network is consisted of third party user-engaged node runners. There are two types of nodes: data carrier node and computation node. Node operator could maintain either or both types depending on their economic interests and capabilities. Each operator locks certain amount of DOS tokens as security deposit, say 10000 DOS token, and they will be awarded with processing fees for the oracle services they provide. Malicious nodes will be detected by protocols and their security deposit will be forfeited.

- **Premium data providers**: By registering on the open marketplace with customized payment schemes and specs, premium data providers could have an entirely new path to monetize their valuable data feeds from blockchain traffic in addition to the traditional internet traffic.


Blockchains
DOS Network
Data Sources
Receive processing fee from blockchain user, require security deposit to join the network
Dapp
DOS Node
Data Provider
Pay processing fee to DOS node operator for data feed and computation services, and to data provider for premium data
On-Chain Governance Contract
Receive security deposit from DOS node operator
Punish malicious behaviour
Monetize premium data from blockchain traffic

## 5.2 Network Bootstrap

Several proposals are made for a cold startup of the DOS Network:

- **Incentive program for Dapp developers**: A developer program would start to encourage more Dapps and developers utilizing the DOS service. Developers who submit development proposals and proof of usages will be granted with DOS tokens as free trials for reimbursement of the processing fees they pay to node operators.

- **“Mining” incentives for node runners**: A node incentive program called “mining” would start to incentivize more node runners to join to bootstrap the off-chain P2P Network. Three hundred and fifty million (35% out of total) DOS tokens are reserved as mining incentive. In addition to the normal processing fees they earn, node operators will also “mine” DOS token from mining reserve at a determined return rate. With more nodes joining the network, the return rate decreases over time, as demonstrated by the table below:

| Number of nodes at time of joining | Year 1 ROI | Year 2 ROI | Year 3 ROI | Year 4 ROI | Year 5+ ROI |
|---|---|---|---|---|---|
| [1, 1000) | 100% | 66% | 43.6% | 28.7% | 18.9% |
| [1000, 2000) | 50% | 33% | 21.8% | 14.3% | 9.48% |
| [2000, 4000) | 25% | 16.5% | 10.9% | 7.18% | 4.74% |
| [4000, 8000) | 12.5% | 8.25% | 5.45% | 3.59% | 2.37% |
| [8000,20000] | 6.25% | 4.12% | 2.72% | 1.8% | 1.18% |

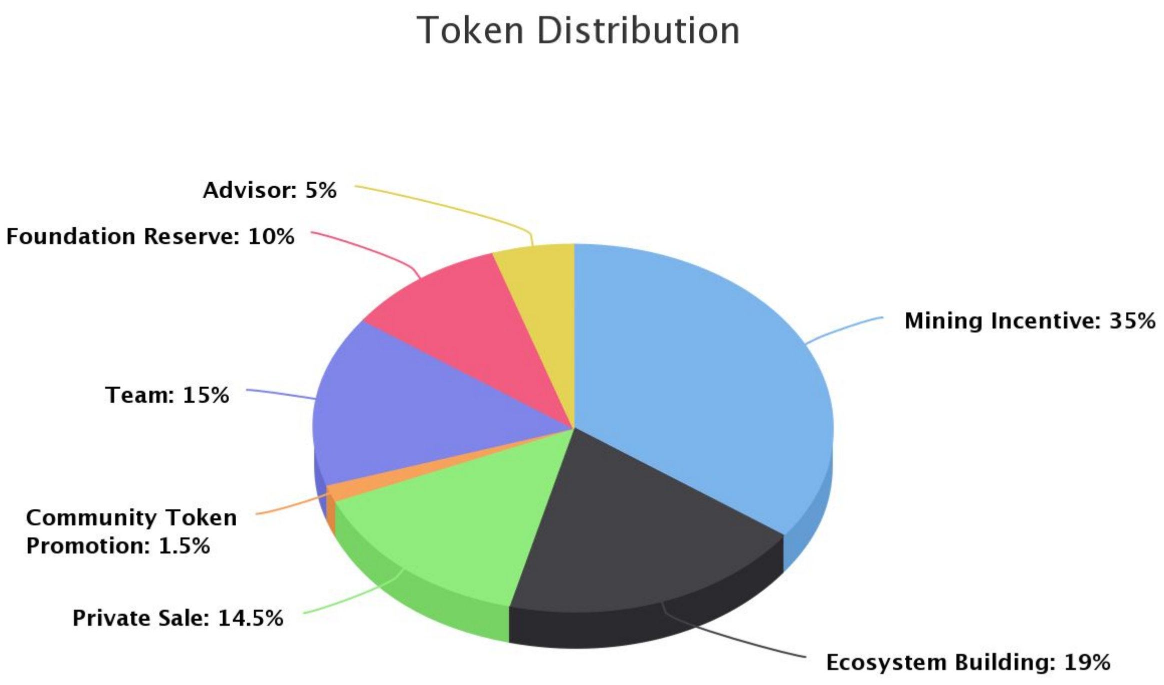
Mining incentive stops after the mining reserve is exhausted, which will roughly sustain ten years.

- **Incentive for premium data providers**: First of all premium data providers would collect corresponding bounties started by developers from the marketplace contract. Besides that, in order to incentivize premium data provider to integrate with DOS Network and onboard more premium events, a bonus program is designed to award data providers for each milestone they achieve, e.g. for the first 1000, 5000 and 10000 paid calls from calling contracts data providers will be awarded with 25% bonus DOS tokens from the ecosystem reserve.

Other details of incentive program and bootstrap plan are to be determined, including but not limited to lock drop, developer bounty programs, hackathons, etc. Combining with mining incentive 55% of all DOS tokens are reserved for network effects and ecosystem usage.

## 5.3 Token Distribution

**Symbol**: DOS
**Total Supply**: 1 Billion (1,000,000,000)
**Allocation**:
Mining Incentive: 35%
Ecosystem Building: 19% (For node lock drop, exchange listing fees, network bootstrap incentives, bounty tasks, strategic partnership etc.)
Community Token Promotion: 1.5% (50% no lock up, vesting 50% after 3 months)
Private Sale: 14.5% (25% no lock up, vesting quarterly over 9 months)
Team: 15% (Lock up in first 9 months and vesting monthly in next 24 months)
Foundation Reserve: 10% (For marketing, legal, PR, business development, etc.)
Advisor: 5% (Lock up in first 9 months)



## 6. Use Cases

DOS Network provides the ability for on-chain smart contracts reaching and reacting to external events in a decentralized and trustless way. This unlocks a lot of use cases that smart contracts are now capable of with the help of DOS Network.

### 6.1 Decentralized Derivatives

Derivatives are financial contracts between two or more parties whose values are based on the underlying assets. Derivatives allow people to put different viewpoints (long or short) on the underlying assets and in essence promote the financial stability. Public smart contract platforms make it possible to create and trade financial derivatives including blockchain based assets; for example, Market Protocol[43], Decentralized Derivatives Association[44], and DyDx Protocol[45] are all working hard to push the boundary forward. DOS Network could take a significant role in decentralized derivatives by providing price feeds, settlement values and contract expirations to determine gain or loss for participating parties.

### 6.2 Stablecoins

Stablecoins are cryptocurrencies with stable fiat value, reducing volatility and making them more appealing as a store of value and medium of exchange in many ways, so they are honored as the holy grail in digital currency. When referring to stablecoins we're not talking about IOUs issued by a centralized company, such as tether[46] or digix[47], but more about decentralized and algorithmically controlled cryptocurrency, such as collateral backed stablecoins like bitUSD[48] and Dai[49], and Seigniorage Shares[50] based stablecoins like Basecoin[51] and kUSD[52]. All stablecoins need the help of oracle system like DOS Network to get external data about exchange rate between stablecoins and the asset they're pegged to.

## 6.3 Decentralized Lending Platform

Decentralized peer-to-peer lending platforms like SALT Lending[53] and ETHLend[54] allow anonymous users to pledge crypto assets on blockchain in exchange for fiat or crypto loans. DOS Network could be applied to introduce market rates during loan creation and to monitor the ratio of crypto collaterals to the loaned amounts, triggering liquidation events if loan terms are met. Furthermore, ETHLend also uses oracles to import borrowers' social media data and other identity info to differentiate interest rates between different borrowers.

## 6.4 Decentralized Insurance

Etherisc[55] is building a platform for decentralized insurance applications like flight delay insurance and crop insurance by bringing in efficiency and transparency with lower operational costs. Users purchase insurance policy and pay the premium in ether and they'll get corresponding payout in ether back according to the policy in case of agreed-on conditions are met. By introducing external data and events into smart contracts, DOS Network helps those decentralized insurance products in policy underwriting and payout decisions in case of claims as well as schedule future checks by the time of policy expiration to achieve automatic payout.

## 6.5 Decentralized Casino

Decentralized casinos like Dice2Win[56], Etheroll[57] and Edgeless[58] benefit a lot from blockchains in terms of transparency, near-instant secure fund transfer, and provably fair with 0% house edge comparing to traditional online casinos with 1%~15% house edge. Unpredictable and verifiable random number generation is the core of any casino game, but random number generation in a pure deterministic environment (on-chain) is difficult or even impossible in theory. DOS Network is able to generate provably secure, verifiable, unbiased and unstoppable random entropies for those Dapps to use.

## 6.6 Decentralized Prediction Markets

Decentralized prediction markets like Augur and Gnosis utilize wisdom of the crowds to predict real world outcomes such as presidential election and sports betting result. DOS Network could be used for fast and secure resolution in case of the voted results being challenged by users.

## 6.7 Decentralized Computation Markets and Execution Scalability

Bypassing the block gas limit and expensive on-chain computation cost, DOS Network connects redundant third party computing power with business computation intensive tasks such as machine learning model training, 3D rendering, scientific computing like DNA sequencing. In our long-term roadmap, zkSNARK based computation oracle would also offer privacy to computing tasks as private input is supported. Also for the current blockchain scalability debate, it would bring unlimited execution scalability to supported chains.

# References

[1] Satoshi Nakomoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, https://bitcoin.org/bitcoin.pdf

[2] Vitalik Buterin, “A Next-Generation Smart Contract and Decentralized Application Platform”, https://github.com/ethereum/wiki/wiki/White-Paper

[3] State of the Dapps, https://www.stateofthedapps.com

[4] Verifier’s dilemma, Loi Luu, Jason Teutsch, etc. “Demystifying Incentives in the Consensus Computer”, https://eprint.iacr.org/2015/702.pdf

[5] Oracle machine, https://en.wikipedia.org/wiki/Oracle_machine

[6] Oraclize, http://www.oraclize.it

[7] TLSNotary - cryptographic proof of online accounts, https://tlsnotary.org

[8] SPOF, https://en.wikipedia.org/wiki/Single_point_of_failure

[9] Town Crier, http://www.town-crier.org

[10] TEE, https://en.wikipedia.org/wiki/Trusted_execution_environment

[11] Foreshadow attack, https://foreshadowattack.eu. Meltdown and Spectre attack, https://meltdownattack.com

[12] AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves, https://www.ibr.cs.tu-bs.de/users/weichbr/papers/esorics2016.pdf

[13] Malware Guard Extension: Using SGX to Conceal Cache Attacks, https://arxiv.org/abs/1702.08719

[14] Chainlink, https://www.smartcontract.com

[15] Augur, http://www.augur.net

[16] Gnosis, https://gnosis.pm

[17] SGX enabled processors, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/606636

[18] SGX security and dev review, https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review.pdf

[19] Truebit, https://truebit.io

[20] Truebit whitepaper p12, verification game, https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf

[21] Truebit whitepaper p40, security patches, https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf

[22] Verifiable computation, https://en.wikipedia.org/wiki/Verifiable_computing

[23] zkSNARK in a nutshell, https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell

[24] Sean Bowe, Ariel Gabizon, Ian Miers, “Scalable MPC for zkSNARK Parameters in the Random Beacon Model”, https://eprint.iacr.org/2017/1050.pdf

[25] Timo Hanke, Mahnush Movahedi, Dominic Williams, “Dfinity Consensus System”, https://dfinity.org/pdf-viewer/pdfs/viewer?file=../library/dfinity-consensus.pdf

[26] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, Tal Rabin, “Secure Distributed Key Generation for Discrete-Log Based Cryptosystems”, https://link.springer.com/article/10.1007/s00145-006-0347-3

[27] Dan Bone, Ben Lynn, Hovav Shacham, “Short Signatures from the Weil Pairing”, https://link.springer.com/article/10.1007/s00145-004-0314-9

[28] Leslie Lamport, Robert Shostak, Marshall Pease, “The Byzantine Generals Problem”, http://lamport.azurewebsites.net/pubs/byz.pdf
[29] Hypergeometric distribution, https://en.wikipedia.org/wiki/Hypergeometric_distribution
[30] Hypergeometric Calculator, http://stattrek.com/online-calculator/hypergeometric.aspx
[31] Hypergeometric distribution calculator, http://keisan.casio.com/exec/system/1180573201
[32] ZenCash secure node network stats, https://securenodes2.eu.zensystem.io
[33] SmartCash masternode network stats, https://smartcash.bitcoiner.me/smartnodes/worldmap
[34] Byzantium hard fork, https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement
[35] Arithmetic circuit, https://en.wikipedia.org/wiki/Arithmetic_circuit_complexity
[36] Powers of Tau Ceremony, https://z.cash.foundation//blog/conclusion-of-powers-of-tau
[37] libsnark, https://github.com/scipr-lab/libsnark
[38] Infura, https://infura.io
[39] Internet api economy, http://www-03.ibm.com/press/us/en/pressrelease/48026.wss
[40] KISS principle, https://en.wikipedia.org/wiki/KISS_principle
[41] Homomorphic encryption, https://en.wikipedia.org/wiki/Homomorphic_encryption
[42] zk-STARK, https://eprint.iacr.org/2018/046.pdf
[43] Market Protocol, https://www.marketprotocol.io
[44] Decentralized Derivatives Association, https://www.decentralizedderivatives.org
[45] DyDx Protocol, https://dydx.exchange
[46] Tether, https://tether.to
[47] Digix, https://digix.global
[48] bitUSD, https://coinmarketcap.com/currencies/bitusd
[49] Dai, a decentralized stablecoin, https://makerdao.com
[50] Robert Sams, “A Note on Cryptocurrency Stabilisation: Seigniorage Shares”, https://bravenewcoin.com/assets/Whitepapers/A-Note-on-Cryptocurrency-Stabilisation-Seigniorage-Shares.pdf
[51] Basecoin, http://www.getbasecoin.com
[52] kUSD, https://www.kowala.tech
[53] SALT Lending, https://www.saltlending.com
[54] ETHLend, http://ethlend.io
[55] Etherisc, https://etherisc.com
[56] Dice2Win casino, https://dice2.win
[57] Etheroll casino, https://etheroll.com
[58] Edgeless casino, https://edgeless.io

# Appendixes

## **Appendix I**: Supported query types in DOS network.

| Query Type | Example |
|---|---|
| Web API | DOSQuery(timeout, "https://api.coinbase.com/v2/prices/BTC-USD/buy", “$.data.amount”); |
| Premium Web API* | DOSQueryPremium(timeout,“https://us-zipcode.api.smartystreets.com/lookup?state=CA&zipcode=94085”, “$.result”, [“@from_uc_addr”, “@other_param”]); |
| Cross-Chain Contract Call* | DOSQueryXChain(timeout, “EOS”, [“@ddex_contract_addr”, “@EOS_token”, “Buy”, 60]); |
| Consensus-based off-chain Computation* | DOSCompute(timeout, “@ipfs_code_hash”, [@arg1, @arg2, ...]); |
| zkSNARK-based off-chain computation** | DOSZKCompute(timeout,  “@ipfs_code_hash”, [@arg1, @arg2, ...]) |

*  mid-term goal.
** long-term goal.

## **Appendix II:** Example verificaion code of zkSNARK proofs in solidity.

```
pragma solidity ^0.4.24;

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;
    }

    // Generator of G1
    function P1() internal returns (G1Point) {
        return G1Point(1, 2);
    }

    // Generator of G2
    function P2() internal returns (G2Point) {
        return G2Point(
[11559732032986387107991004021392285783925812861821192530917403151452391805634,
10857046999023057135944570762232829481370756359578518086990519993285655852781],
[4082367875863433681332203403145435568316851327593401208105741076214120093531,
8495653923123431417604973247489272438418190587263600148770280649306958101930]
        );
    }

    // @return the negation of p, i.e. p.add(p.negate()) should be zero.
    function negate(G1Point p) 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), 0x6, 0, input, 0x80, r, 0x40)
        // 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), 0x7, 0, input, 0x60, r, 0x40)
        // Use "invalid" to make gas estimation work
        switch success case 0 { invalid }
    }
    require (success);
}

// Return the result of pairing check:
// e(p1[0], p2[0]) *  .... * e(p1[n], p2[n]) == 1
// E.g. 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), 0x8, 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;
}
```

```
    // Helper funciton for pairing check of 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);
    }

    // Helper funciton for pairing check of 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);
    }

    // Helper funciton for pairing check of 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([0x8fc6f88aebb07ff31599a9f795e6b054e200a2fd8891da79ead9bd20993c2fe,
0x29e8d591cbeb1058b1d96adf8da925dc9a4eda7c6512822a6ef1e2b53d980a3c],
[0x1c82f39b66c99016f09b6b2b71363f7c95f1d44b28cd0457fb7c02a0e8c25686,
0x1a9d5545d5a9521ac5cd1a839a231f3e93eca713a76b4658c9df4ebdcee9f79e]);
        vk.B = Pairing.G1Point(0x1d03d0c73ca77f67dad316abfbcb816d9b87fb68d8690de5a60f8663db4d984b,
0x29f592fbadc5002b3d712d56825baefb2218888da0db14e331149f440e8a5c3);
        vk.C = Pairing.G2Point([0x2083bec38ffd4907d5529f264bc4d2e4706b5b416c2b053ae0e20616f2938bfc,
0x23bb65bc2d210605d19cb5b18149210c781d09a604c85257110070606b673688],
[0x13f16ebf23012bfac5105660dd580dc291a2f71ee9d74aaf951bf9b412e67382,
0x281ee8184f1b9134fb5c557c73f6dd800f7e8493818ace57a821f5aa5df46ec]);
        vk.gamma =
Pairing.G2Point([0xcb7185daaa7975449cda20cbe15b4d05522922046073e225301c7e910a58c0d,
0x1671513e95e7ca110717c5a97535edfc11c290178eecf1d5edca9b3e969e444e],
[0x11ebae83ea23087468fc5a091579766be5fd9f16c3f3bcfd2be984372f08d10e,
0x20bafbfd1d9c28421e11745e42079ab0eb6672ea1457d1bcea9b9f50e8ba852b]);
        vk.gammaBeta1 =
Pairing.G1Point(0x2630f9382fd2e3fdb4366451e53fff5311cfbbbb153cda6bc1a04f12cea0121d,
0x305b72a49f5eca1bb7072721ae92f767264e8442d409a04cd1ad6433f36eb96e);
        vk.gammaBeta2 =
Pairing.G2Point([0x17f41c70bfed7b867f6127a8834cd0d782051c0fa0cccfc1f7b671b5c0a1617a,
0x19887db9ad194fa3356f8bae18fde1c7d48471e6a455c946046453f2252cb77c],
[0x1552950341b23118fede032878d5f07ecda6c6f585a625112c7ddcd364e846a8,
0x2085f64c8b3de07adc2cc3c44896f38b45ee19ad6ed0a3cacc08da827705bea1]);
        vk.Z = Pairing.G2Point([0x124af821d93fdf6bb81af237bdca7d8c208d506b6544a1d1841c96c9e5a8146e,
0x23e951d40d1bbdc29e9e0d6b94296c8f048467c28da8eef1084a62b42e0ca26d],
[0x1b6eadbbc87f60aab45c02bbb13872678f9f6aba54427ee08bb39aa647ff9dae,
0x120427f29479ef0dd35f71247a5e23990dea5528251741990a61ca657f99e67c]);
        vk.IC = new Pairing.G1Point[](4);
        vk.IC[0] =
Pairing.G1Point(0x274df1d03d8f810f2d9be82af2c22b11d8b78f30bed1ea88b7de0c232d2ad0a8,
0x1db29ef17911a50b5fbfc0c4868bb1ed12f9fe25fb2c53f1d8c3e9048ede51e6);
        vk.IC[1] =
Pairing.G1Point(0x14b89067062304bd7fa8be61aeb9f7d8cb9efdf2507beb9224fdc0121749f39b,
0x22b34646c8d65b4476e7d6ffeb37d4093be4a77c41bf94c65d9893b32aa0386c);
        vk.IC[2] =
Pairing.G1Point(0x1ab8958aca16d7aec7930545dc8034931c1ac10d9a9f8871d5b9304a13cfeafc,
0x3013c2db7149a2f7ac1197a279bb112802dd18415211be13aac6e5fe1886e184);
        vk.IC[3] =
Pairing.G1Point(0x353f350f29cbcb57dd2455bc95c07f69208c2ce23f4b3f77111b32c429c7c09,
0x1a16cd7a7d387051086dd3e26e5333ce7a54eed6f9964cf4ac8e202be45d440c);
    }

    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 s);

    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("Successfully verified simple computation function addition(a,b).");
            return true;
        } else {
            return false;
        }
    }
}
```