Maxima - A Fast Fearless Secure Sharded Block-Chain

Maximilian E. Chamberlin
MA (Oxford) M.Phil (Cambridge)
Maxima Organisation
mec@Maxima.org

Abstract

Recently, high demand and limited scalability have increased the average transaction times and fees in popular cryptocurrencies, yielding an unsatisfactory experience. Here we introduce Maxima a cryptocurrency with a sharded block-chain, one where the network is divided into partitions called shard which maintain their own blocks.

In Maxima, validator committees are sampled for each shard. Validators vote on the availability of blocks to the network, and are shuffled off shards very quickly. Once a committee has determined if a block is accessible to nodes on a shard, nodes on that shard execute the transactions and verify the blocks validity- producing a succint fraud proof if the block is invalid. Cross shard transactions are managed using tools from distributed systems, like locks and yanks.

Maxima is a globally distributed computer that is secure under an honest majority assumption, and puts in place mechanisms to prevent adaptive adversaries from corrupting the network. With 20 shards, Maxima is projected to process over 20,000 transactions per second.

Part I

Introduction

1 Motivation and Outline - Core Sharding

In Bitcoin, blocks are added to the block-chain approximately every 10 minutes, at a relatively fixed pace that results in a TPS of 5-7. Simple solutions to expanding the capacity of the block-chain have found natural limits to their efficacy: enlarging blocks runs up against user bandwidth limits; hastening the rate at which blocks are mined increases orphaning.

A more promising solution to the problem of scaling is to run separate chains that process transactions within partitions of the network, called shards. The key challenges are: (1) how to manage the communication between the different shards, a problem which is closely related to ensuring the atomicity of transactions in a distributed system and (2) how to run these many separate block-chains in a way that does not compromise on security, since each shard-chain will only have a fraction of the mining/validation power of the network.

In this paper, we propose mechanisms to ensure the atomicity of transactions and to ensure that the security of separate chains is not compromised. We adopt an honest majority assumption for the network, assuming fewer than a fraction f of 0.25 nodes are Byzantine. Note $f = 0.25$ is an

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

arbitrary constant bounded below 1/3 to ensure good constants. We also put in place safe-guards against adaptive adversaries who may bribe network participants.

With this in mind, our protocol can be described below. Validator committees are sampled for each shard. Sampling validators randomly means that with a sufficient number (400) we can ensure an honest majority of 2/3 of per shard almost surely.

Validators then vote on the availability of blocks to the network, and are shuffled off shards very quickly. This fast shuffling is to ensure that an adaptive adversary does not have the time to find and corrupt the validators of each shard. Because of this fast shuffling, the work that a validator can perform must be very minimal. Validators check that the data corresponding to the Merkle root of a block is accessible to a shard, so that the members of a shard can execute those transactions. If there is a dispute about transaction execution, validators also resolve disputes by considering succinct proofs of invalid execution. Cross shard transactions are managed using tools from distributed systems, like yanks, to ensure that transactions can be executed atomically.

Our aim is to reduce costs and improve scalability - this will be a constant theme throughout this work.

1.1 Protocol Structure

The construction of our protocol depends on the following, relatively independent components, which can be subdivided into three themes: voting schemes; data creation and concurrency controls.

1. **Motivation and Outline - Sharding**
   (a) Protocol Structure
2. Consensus:
   (a) Voting Scheme
   (b) Validator registration and Shuffling
3. Concurrency Controls
   (a) Locking State for Cross Shard Transactions
   (b) Stateless Clients
   (c)
4. UTXO model
5. Minimal Sharding - No Smart Contracts
   (a) Summary - a simple protocol to transfer currency.
6. **Motivation and Outline - Advanced Sharding**
7. Separating Validity Checking from Availability Checking
   (a) Erasure Coding
   (b) Fraud Proofs
   (c) Summary
8. Solving the State Problem
   (a) **Introducing rental fees to the block-chain**
   (b) **Building on top of IPFS**
9. Virtual Machine
   (a) EWASM VM
   (b) Transpilers and the future of block-chain

2 Achieving Fork-Free Consensus

2.1 Consensus Scheme - Modified Dfinity

In Tellabit, we adopt a proof of stake scheme. Here, we give a brief overview of the motivating factors. In a proof of stake system, validators place bets (stake) on whether a given block will be

included in the main chain. One argument against traditional proof of stake is that economies of scale eventually lead to a few centralised stakers**.** Aknowledging this aspect, delegated proof of stake emerged as a protocol - where nodes are chosen to stake on blocks on behalf of others, as used in EOS and NEO. Although this results in a simpler and faster algorithm, this has the undesirable effect that participation is less open. More recently, consensus protocols have emerged where a randomly selected group of validators are chosen from a global pool [Dfinity , Zilliqa, Algorand,]. These algorithms purport to be more open than delegated proof of stake- as participants are randomly sampled from a global pool- but offers faster finality by having large consensus groups (of the order 400) vote on each block. The protocols proposed by Dfinity and Zilliqa are inspired by the byzantine fault tolerant literature, with the key innovation being that the communication complexity has been reduced from $O(n^2)$ to $O(n)$ by using various cryptographic techniques, using threshold or Schnorr signatures to transmit information. Reducing communication complexity removes the bottleneck in BFT algorithms, and enables the formation of large , robust and open consensus groups which are formed through random sampling.

However, these byzantine fault tolerant algorithms are not without weaknesses. In Zilliqa, there is a single leader orchestrating the vote, who can be corrupted to stall progress by proposing faulty blocks or by aggregating votes incorrectly. Dfinity is leaderless, a strong advantage, but has the disadvantages that (1): notary groups (the stakers) cannot be slashed for behaviour like going offline or voting for an incorrect block, and so there are no marginal incentives for good behaviour. Both of these cryptocurrencies have not solved the issue that consensus groups can be subjected to denial of service attacks (DOS), and so liveness may be affected. A further issue in Dfinity is that the signing committee is public , and so can be targeted well ahead of time.

A more secure consensus algorithm, whcih also makes use of random committees is Algorand. In Algorand, committees are formed on the fly after a block has been created, which prevents the DOS attack vectors and makes it harder for an adversary to corrupt a voting group. We use an algorand based consensus protocol. The one disadvantage of Algorand is that there is no way to punish a validator who has gone offline and reneged on the duty to validate blocks, which may be a liveness concern. It would also be useful to have a succinct proof that a validator has checked a block.

Formation of Committees

Suppose that each validator V_i has a private preimage CV_i, and there is a common randomness source R. For any given proposal P, we derive a randomness $RP = Hash[R, CP]$, where CP is the proposer's preimage. If there are in total N validators and we want a committee of size M , then anyone can make a signature by revealing CV_i and showing that $CV_i \leq 2256 * \frac{M}{N}$. On average, M of the N validators will be able to do so for any given proposal, with standard deviation $\sqrt{M}$, and so at least $M - \sqrt{M} \propto 99.85\%$ of the time, and at most $M + \sqrt{M} \propto 99.85\%$ of the time.

Suppose an attacker has fraction p of proposers, we can try to estimate the probability that a fraudulent proposal will get through for various values of p and M with privately selected committees (using Poisson distributions, so assuming N approaching infinity, values for N = 20000 are very close to the limit at $N \to \infty$).

In addition, it would be nice for shards to be acnhored to a common round value. Then blocks may only be added as quickly as the common reference block. This could be done in a sequential proof of work scheme, with a difficulty factor D, where D is adjusted via an on-chain game, targeting toward five seconds.

Leadership election

Leadership election would be done in a similar way, where we derive a randomness $RP = Hash[R, CP]$, where CP is the proposer's preimage.

Safety

The key safety properties are proved in section 55 of Algorand. We will add more details here.

Adding Incentives to Algorand

Incentivising liveness is relatively simple. We can award validation points to validators who attest to the validity or invalidity of blocks. We also passively drain a validators deposit ach round, which provides a second incentive for validators to stay online and check blocks. In a later section, we will discuss how we adapt the incentive scheme for advanced sharding.

3 Cross Shard Communication Through Message Passing

Message Passing is the facvoured approach for concurrency control in distributed systems.

We use a message passing model based on Erlang, which has a simple implementation of CSP (Communicating Sequential Processes) where soft-threads (threads) have each a single mailbox that can receive messages. The soft-thread can pop off and react to messages in its mailbox, also supporting some form of pattern matching for prioritization.

Contracts are like the threads of execution and we have a single mailbox per contract. Conceptually message passing is simple, it is no different from running the transaction, but with updated data, but with a special message type to distinguish it from ordinary data. Howeever, the only difference will be in the accumulation of messages. Validators bundle up the messages and feed them in as data to the contract, which can then pop messages off and react to them. Diagramatically, we could envidage something like this.

This introduces few overheads to the protocol, since popping off is just a data operation, which can be defined in the main-loop of any contract.

	S_1	S_2	S_3
t_0	M_1 to C_1 on S_3 on chain Sent to S_2 by val		
t_2		M_2 to C_1 on S_3 on chain Sent to S_2 by val	
t_3 t_4			Actor bundles M_1 and M_2 $[M_2, M_1]$ validated against Mroot C_1 reads $[M_2, M_1]$

We now talk about guarantees in our message passing model:

Availability:

A pertinent question arises: **what mechanism ensures that a message** on one shard makes its way over to a message on another shard? We charge the validators on Shard1 and Shard2, with disseminating the information to the third shard. Since there are approximately 400 validators per shard, this large number ensures that with very high likelihood $1 - (\frac{1}{3})^{400}$, that a message will get through. Certainly a message will get through eventually, if we take it that shards might be DOS'd. The message is sent with a transaction fee, and the transaction fee is split between block miners on both shards. In fact shards will periodically relay the message until it is included in a shard.

At Once:

Each message contains a signed nonce, which ensures that messages cannot be replayed.

Message Ordering:

Messages sent directly from one shard to another will not be received out-of-order.

Shard S1 sends messages M1, M2, M3 to S2

Actor S3 sends messages M4, M5, M6 to S2

However, we do not provide guarantees about the ordering of messages between shards.

Furthermore, specific concurrency control mechanisms may be built ontop of message passing systems. For instance locking schemes and other such things.

Other Concurrency Management

A question that emerges is what if the transaction fees are too low and it takes a long time for a message to be included on another shard? More broadly, should we be providing guarantees on message passing? We have provided a robust system, but we adopt the Erlang philosophy that we make the fallibility of communication explicit through message passing, and do not try to provide a leaky abstraction. Instead users can write implementations that provide guarantees on message delivery, and delegate these to higher level protocols. This is a model that has been used with great

success in Erlang and requires the users to design their applications around it. You can read more about this approach in the **Erlang documentation (section 10.9 and 10.10)** and Akka.

Another angle on this issue is that by providing only basic guarantees those use cases which do not need stronger reliability do not pay the cost of their implementation; it is always possible to add stronger reliability on top of basic ones

On top of these other concurrency controls can be implemented in smart contracts, for instance various kinds of locking scheme.

The specific case of sending money?

	S_1	S_2	S_3
t_0	M_1 to C_1 on S_3 on chain Sent to S_2 by val; deduct		
t_3 t_4			Actor bundles M_1 and M_2 $[M_2, M_1]$ validated against Mroot C_1 reads $[M_2, M_1]$

4 UTXO model

Transaction ordering is a problem. The miners get to decide on the order of the transactions and this can have surprising consequences.

Also, how does one merge large blocks together.

In contrast a UTXO model has no such issues. the state consists of a set of objects. We use a slightly different mechanism, which has the same ethos as the UTXO model.

This is the hashgraph. Objects are re

- Objects are atoms that hold state in the Chainspace system. We usually refer to an object through the letter o, and a set of objects as o ∈ O. All objects have a cryptographically derived unique identifier used to unambiguously refer to the object, that we denote id(o). Objects also have a type, denoted as type(o), that determines the unique identifier of the smart contract that defines them, and a type name. In Chainspace object state is immutable. Objects may be in two meta-states, either active or inactive. Active objects are available to be operated on through smart contract procedures, while inactive ones are retained for the purposes of audit only.
- Contracts are special types of objects, that contain executable information on how other objects of types defined by the contract may be manipulated. They define a set of initial objects that are created when the contract is first created within Chainspace. A contract c defines a namespace within which types (denoted as types(c)) and a checker v for procedures (denoted as proc(c)) are defined
- A procedure, p, defines the logic by which a number of objects, that may be inputs or references, are processed by some logic and local parameters and local return values (denoted as lpar and lret), to generate a number of object outputs. Notionally, input objects, denoted as a vector ~w, represent state that is invalidated by the procedure; references, denoted as ~r represent state that is only read; and outputs are objects, or ~x are created by the procedure
- We denote the execution of such a procedure as:
-
- A key design goal is to achieve scalability in terms of high transaction throughput and low latency. To this end, nodes are organized into shards that manage the state of objects, keep track of their validity, and record transactions aborted or committed. Within each shard all honest nodes ensure they consistently agree whether to accept or reject a transaction: whether an object is active or inactive at any point, and whether traces from contracts they know check
- Consensus on committing (or aborting) transactions takes place in parallel across different shards. For transparency and auditability, nodes in each shard periodically publish a signed

hash chain of checkpoints: shards add a block (Merkle tree) of evidence including transactions processed in the current epoch, and signed promises from other nodes, to the hash chain.

- **Transparency**. Chainspace ensures that anyone in possession of the identity of a valid object may authenticate the full history of transactions and objects that led to the creation of the object. No transactions may be inserted, modified or deleted from that causal chain or tree. Objects may be used to self-authenticate its full history—this holds under both the HS and DS threat models. • Integrity. Subject to the HS threat model, when one or more transactions are submitted only a set of valid non-conflicting transactions will be executed within the system. This includes resolving conflicts—in terms of multiple transactions using the same objects— ensuring the validity of the transactions, and also making sure that all new objects are registered as active. Ultimately, Chainspace transactions are accepted, and the set of active objects changes, as if executed sequentially—however, unlike other systems such as Ethereum [Woo14], this is merely an abstraction and high levels of concurrency are supported. • Encapsulation. The smart contract checking system of Chainspace enforces strict isolation between smart contracts and their state—thus prohibiting one smart contract from directly interfering with objects from other contracts. Under both the HS and DS threat models. However, cross-contract calls are supported but mediated by well defined interfaces providing encapsulation. • Non-repudiation. In case conflicting or otherwise invalid transactions were to be accepted in honest shards (in the case of the DS threat model), then evidence exists to pinpoint the parties or shards in the system that allowed the inconsistency to occur. Thus, failures outside the HS threat model, are detectable; the guildy parties may be banned; and appropriate off-line recovery mechanisms could be deployed
- Smart Contract developers in Chainspace register a smart contract c into the distributed system managing Chainspace, by defining a checker for the contract and some initial objects. Users may then submit transactions to operate on those objects in ways allowed by the checkers.
- Transactions are atomic: either all their procedures run, and produce outputs, or none of them do. Transactions are also consistent: in case two transactions are submitted to the system using the same active object inputs, at most one of them will eventually be executed to produce outputs. Other transactions, called conflicting, will be aborted.
- To generate a set of traces composing the transaction, a user executes on the client side all the smart contract procedures required on the input objects, references and local parameters, and generates the output objects and local returns for every procedure—potentially also using secret parameters and returns. Thus the actual computation behind the transactions is performed by the user, and the traces forming the transaction already contain the output objects and return parameters, and sufficient information to check their validity through smart contract checkers. This design pattern is related to traditional optimistic concurrency control
- Smart contract composition. A contract procedure may call a transaction of another smart contract, with specific parameters and rely upon returned values. This is achieved through passing the dep variable to a smart contract checker, a validated list of traces of all the sub-calls performed. The checker can ensure that the parameters and return values are as expected, and those dependencies are checked for validity by Chainspace. Composition of smart contracts is a key feature of a transparent and auditable computation platform. It allows the creation of a library of smart contracts that act as utilities for other higher-level contracts: for example, a simple contract can implement a cryptographic currency, and other contracts— for e-commerce for example—can use this currency as part of their logic. Furthermore, we compose smart contracts, in order to build some of the functionality of Chainspace itself as a set of 'system' smart contracts, including management of shards mapping to nodes, key management of shard nodes, and governance. Chainspace also supports the atomic batch execution of multiple procedures for efficiency, that are not dependent on each other
- Privacy by design. Defining smart contract logic as checkers allows Chainspace to support privacy friendly-contracts by design. In such contracts some information in objects is not in the clear, but instead either encrypted using a public key, 4 or committed using a secure commitment scheme as [P +91]. The transaction only contains a valid proof that the logic

or invariants of the smart contract procedure were applied correctly or hold respectively, and can take the form of a zero-knowledge proof, or a Succinct Argument of Knowledge (SNARK). Then, generalizing the approach of [MGGR13], the checker runs the verifier part of the proof or SNARK that validates the invariants of the transactions, without revealing the secrets within the objects to the verifiers.

- **Reads**. Besides executing transactions, Chainspace clients, need to read the state of objects, if anything, to correctly form transactions. Reads, by themselves, cannot lead to inconsistent state being accepted into the system, even if they are used as inputs or references to transactions. This is a result of the system checking the validity rules before accepting a transaction, which will reject any stale state.

Data Structures

- Hash-DAG structure. Objects and transactions naturally form a directed acyclic graph (DAG): given an initial state of active objects a number of transactions render their inputs invalid, and create a new set of outputs as active objects. These may be represented as a directed graph between objects, transactions and new objects and so on. Each object may only be created by a single transaction trace, thus cycles between future transactions and previous objects never occur. We prove that output object identifiers resulting from valid transactions are fresh (see Security Theorem 1). Hence, the graph of objects inputs, transactions and objects outputs form a DAG, that may be indexed by their identifiers.
- **Specifically, we define a function id(Trace) as the identifier of a trace contained in transaction T. The identifier of a trace is a cryptographic hash function over the name of contract and the procedure producing the trace; as well as serialization of the input object identifiers, the reference object identifiers, and all local state of the transaction (but not the secret state of the procedures); the identifiers of the trace's dependencies are also included. Thus all information contributing to defining the Trace is included in the identifier, except the output object identifiers.**
- We also define the id(o) as the identifier of an object o. We derive this identifier through the application of a cryptographic hash function, to the identifier of the trace that created the object o, as well as a unique name assigned by the procedures creating the trace, to this output object. (Unique in the context of the outputs of this procedure call, not globally, such as a local counter.)
- An object identifier id(o) is a high-integrity handle that may be used to authenticate the full history that led to the existence of the object o. Due to the collision resistance properties of secure cryptographic hash functions an adversary is not able to forge a past set of objects or transactions that leads to an object with the same identifier. Thus, given id(o) anyone can verify the authenticity of a trace that led to the existence of o. A very important property of object identifiers is that future transactions cannot re-create an object that has already become inactive. Thus checking object validity only requires maintaining a list of active objects, and not a list of past inactive objects.
- Why we use a HASH-DAG - it is the default for storing stuff.
-

5 Minimal Sharding - No Smart Contracts

For each shard, the threshold relay assigns a different committee group to the shard. Each committee votes of the current block,checking for the validitiy of the transactions. Initially, we will only support money being sent from one shard to another. However, we will later add virtual machine functionality. A question arises: what if a beacon group stalls because it is DOS'd. After a few minutes, if no relay group has been selected, two relay groups can vote to correct a shard.

6 Advanced Sharding - Motivation and Introduction

We have outlined a simple protocol to achieve sharded message calls. However, we want to position ourselves at the forefront of technology. Key to this is understanding the direction

Virtual machines and Interoperability.

A sharded chain allows one to run the consensus logic of many chains. For instance you could imagine one chain running bitcoin transactionson one chain and the transactions of another coin on another. Key to this would be cross compilation. So, if we had a WASM virtual machine for our transactions, all that would be needed would be to find a way to transpile code from one virtual machine to another. Then one could have all the code written on EThereum . Transpilation is a well studied topic, and has been used to convert C++ to Javascript etc.

The language of the WEB

Having a wasm virtual machine has another benefit in that it is becoming the de facto standard for the web. This means that any code that could be executed in the browser, something that is key to hosting websites could be run on our block-chain. This would need to be coupled with efficient storage mechanisms. Currently on Ethereum the main cost, is in storing data. This is because Ethereum is getting to be at capacity in terms of the blocks that each node must sync on when toring data. In sharding, this removes the storage problem by a huge constant factor.

Efficient Storage

Apart from sharding, one idea to improve storage is to move to a model where there is an efficient market for storing data. This will be crucial, the greatest expense to ethereum is the cost of storage, so the better we can bring this down the more competitive our sharding solution will be.

7 Separating Validity checking from State Execution

In the advanced sharding section, we propose a second variant of Maxima, which utilises data availability proofs. Under such a scheme validators vote on the availability of a block to the network at large. The actual correctness of execution is left to the network itself, in a similar manner to Truebit style interactive games. This has the advantage that:

1. Validators do not need to store all the witness data required to execute transactions, which is ideal as we imagine them moving between shards rapidly.
2. We can request for proofs that validators have checked the availability of a transaction.

Under such a scheme: a proof that they have checked a block, together with a sequential proof of work on that block, for which they voted yes to the availability of. This serves to purposes (1): offline validators will not be able to respond to the challenge and so will be slashed. (2): validators will have to demonstrate that they have checked the availability of a transaction, which can only be just in time.

Weighing up the benefits and disadvantages of our approach

There may be criticisms in that marginal incentives are not properly aligned in these schemes, compared with slashing proof of stake schemes like casper. However, as both rely on an honest majority assumption we deem them both as safe as each other. Slashing may also be introduced probabilistically to the Dfinity scheme as we will later show, to introduce the correct marginal incentives. It is important to note that: we ultimately make 2 assumptions which safe-guard the security.

We try and make realistic security assumptions rather than theoretical mathematical guarantees:

(2) **Rational Miner** - We assume a large mining pool which controls some 5% of the stake, may introduce its own software on top of our protocol to increase its profit - however they will not intentionally do so to destroy their steak and the system as a whole unless bribed by an amount proportional to their stake. Only once was the Bitcoin consensus reverted, through a mining pool mining on invalid blocks, which introduced a bug in its software. We are not immune to majority attacks - and would use a social consensus mechanism to fork the coin if ever presented with a catastrophic failure, similar to .

(3) **Bribing attacker** - We build in mechanisms that an attacker would need to revert consensus on each shard. Bribing before the event - they'd need a budget of at least 25%. Bribing after a shard has been formed. We assume that due to the speed with which blocks are agreed upon, roughly 1 minutes, this becomes infeasible.

Algorithm in depth

In Dfinity, the network is grouped into threshold relay groups. These groups are created through random sampling and have a size of about 400. A random number is generated which selects the first relay group. These relay groups then sign on the random number to produce another random number and select the next relay group. The random number also selects a number of block proposers, and the blocks if valid may be signed by the current threshold relay group. This algorithm (with block proposers) and notaries who sign on blocks have similarities with PBFT schemes. Whereas PBFT commits only one block and forges consensus after a round fo prepare and commits, the Dfinity scheme may commit more than one block. Dfinity achieves its high speed and short block times exactly because notarization is not full consensus. However, notarization can be seen as optimistic consensus because it will frequently be the case that only one block gets notarized. Hence, whenever the broadcast network functions normally a transaction is final in the Dfinity consensus after two notarized confirmations plus a network traversal time. The notarization step makes it impossible for the adversary to build and sustain a chain of linked, notarized blocks in secret. For this reason, Dfinity does not suffer from the selfish mining attack [4] or the nothing-at-stake problem.

Advantages

Fork-free protocols typically use BFT style algorithms. These are systems where a vote for a block consists of a prepare followed by a commit. They are fork-free, which is essential for cross-shard communication. Traditionally, PBFT couldn't scale to the size of a network. So instead, we:

Subsample from the entire population so many validators are voting but not all (in Zilliqa, they use X). If say 400 participants are voting and we assume that f<25% are byzantine. We can show that the probability more than 2/3 of those voting are also byzantine is less than the number of atoms in the universe.

What are nodes voting on?

However, a key question here is: what precisely are the nodes voting on? A simple scheme would be for nodes to run the transactions in a block and verify that they are valid. If more than half the nodes agree that a block is valid it is finalised during notarisation. However, this presents problems in the context of sharding: the validators do not have the storage capacity or bandwidth to maintain the state of all shards, and yet they must be shuffled quickly between shards. One solution to this problem is to store the witness data with the transaction in a block, entailing that a block is "self-authenticating", and the witness data need not be downloaded.

Another solution is for validators to just check that data is available, rather than the correctness of a block. If a block is unavailable some honest actor in the network may respond with a fraud proof and take the faulty block creator's deposit. Ultimately, we offload the actual checking of whether a block is correct or not to the network. Just checking data availability has a second advantage: the committee can share the workload of the task. Below we give an outline of how this may be made possible:

Blocks are expanded with an erasure code to a size of 400, say. The erasure code is a redundant encoding scheme which ensures that if 1/6 of the code is available, the original block can be reconstructed by honest nodes within the network. Then each of the 400 committee members checks for 3 or 4 pieces of data. By the probabilistic sampling, we can ensure that 2/3 of the committee is honest. We also require a threshold signature of 1/2 for a block to be notarised for its validitiy.

If a block is available, then 2/3 of the committee will vote and so we have no issues.

If a block is unavailable, then fewer than 1/6 of honest members can have pieces of data. And so even with these votes, and 1/3 of the dishonest majority, they cannot reach the 50% threshold required. We devote a further section to data availability proofs.

We can reduce the amount of checking that a validator does, which will enable smart contracts to run very heavy processing algorithms.

Blocks are created within each shard, and the merkle root is signed on and shared with other shards. The question we now have is this?

(1) How can we ensure that all of the data contained within a shard is available to all members of the shard?

(2) Given that data isavailable, this means anyone can report flaws.

The question is how can we prove that a block is available?

One could require a majority of half the sampled validators perform availability checks and then sign off on the availability of a root.

Question is how can we feed in the right marginal incentives?

- Have rewards for attesting yes, but with penalties if they don't have proof they checked for data availability. This means nodes are incentivised to say yes if they do have the data, but won't declare yes if they don't and know others won't check them. small reward for saying no.
- randomly check availables for a reveal, with sufficinelty high penalty that makes lying unprofitable. Also allow challenges for those who know ing wrong. For those who don't require a random reveal with sufficiently high penalties. => say that a person has 10 blocks to do so before being slashed.
- unavailable|available are losing their rewards

If they are aware many other people do not have access to the data, they may vote yes anyway.

However, we don't want validators to have to actualy execute the state. This would take too long.

1. How can we resolve this issue? Ethereum uses Truebit, which takes the form of an interactive verification game. The main issue with truebit is that it requires multiple rounds of verification and thus can result in low latency in cases where there is a dispute.
2. The approach that we take is to erasure encode the trace of the execution. We use a 3d erasure encoding to ensure that the fraud proofs are of a size cube-root(n).
3. Again using sampling techniques, we can assign validators to each transaction.

Self-Authenticating Erasure Codes

We want validators to be able to guarantee that honest nodes can access data on the network, or stated more succinctly that **data is available**. A simple way for validators to check that data is available is to download a whole block. So, in the current scheme we would sample a set of validators - say 400, which is enough to guarantee an honest majority of participants. Then all of these shards would attempt to download the data from the network, and finaly take a vote between themselves on whether the data is available.

However, to increase efficiency, we would like to use a scheme where each validator need not download the entire block. If blocks were really large, say GB sized, which would be ideal for scalability, it may be impractical to have validators enter a shard and download multiple blocks to verify availability.

We can avoid this, if we take a sampling approach. However, validators will not sample directly from the block itself, but instead will sample from an erasure code.

Why do we want validators to check that data is available for a block? If the data for a block is available to the network, this means that honest executors on a shard can check to see if there are any faults with it and produce succint fraud proofs, O(1) sized proofs that a block contains invalid data.

What this ultimately means is that executors on a shord can check to see if

We don't actually want the validators to perform the executions themselves because doing so would take too long for large blocks of for transactions that take a long tiem to execute.

- N - the number of verifiers
- P - the number of checks made by each client
- Denot

Collectively N verifiers want to check a block of size M is available. They wish to do so without

8 Construction

The below is inefficient. If we have a row of root(m) values , we can recompute the entire row from that. To do a fraud proof, we still need to authenticate the column. What if we don't have a column value? So this still requires availability of some values to prove consistency, but we may not have those values.

8.1 Glossary

K - The expansion factor of each read solomon code

Let D_{ij}, F_{ij}, G_{ij} represent the data int he first, second and third squares respectively

x,y,z are respectively the bilinear accumulator constants for the committed values

First Square:

Let d_{ij} represent chunk i, j of the original data

- x_{ij} is the bilinear accumulator witness for the tuple (d_{ij}, i, j), that is $e[(d_{ij}, i, j), x_{ij}] = x$
- D_{ij} represent $(d_{ij}, i, j), x_{ij}$

Second Square:

- Let f_{ij} represent the j th evaluation point in the erasure code for row D_{i*}
- y_{ij} is the bilinear accumulator witness for the tuple (f_{ij}, i, j), that is $e[(f_{ij}, i, j), y_{ij}] = y$
- F_{ij} represent $(f_{ij}, i, j), y_{ij}$

Third Square:

- Let g_{ij} represent the i th evaluation point in the erasure code for column F_{*j}
- z_{ij} is the bilinear accumulator witness for the tuple (g_{ij}, i, j), that is $e[(g_{ij}, i, j), z_{ij}] = z$
- G_{ij} represent $(g_{ij}, i, j), z_{ij}$

An example is given below with K =2/3:

Definition: for an element $G_{ij} = (g_{ij}, i, j), z_{ij}$ to be **available**, an honest participant of the network must have access to it and it must be authenticated, so the following constraint holds: $e[(g_{ij}, i, j), z_{ij}] = z$.

$(d_{00}, 0, 0), x_{00}$	$(d_{01}, 0, 1), x_{01}$	$(f_{00}, 0, 0), y_{00}$	$(f_{01}, 0, 1), y_{01}$	$(f_{02}, 0, 2), y_{02}$
$(d_{10}, 1, 0), x_{10}$	$(d_{11}, 1, 1), x_{11}$	$(f_{10}, 1, 0), y_{10}$	$(f_{11}, 1, 1), y_{11}$	$(f_{12}, 1, 2), y_{12}$
		$(g_{00}, 0, 0), z_{00}$	$(g_{01}, 0, 1), z_{01}$	$(g_{02}, 0, 2), z_{02}$
		$(g_{10}, 1, 0), z_{10}$	$(g_{11}, 1, 1), z_{11}$	$(g_{12}, 1, 2), z_{12}$
		$(g_{20}, 2, 0), z_{20}$	$(g_{21}, 2, 1), z_{21}$	$(g_{22}, 2, 2), z_{22}$

8.2 Authentication/Availability Transitivity

If a set of values X in an erasure code are available, and can reproduce some data Y, then if $y \in$Y is not authenticated this can be proven in $O(\sqrt{m})$ steps. The authentication transitivity assumption is this : unless an $O(\sqrt{(m)})$ fraud proof is given by the network: then if X is available and produces Y, Y must be authenticated and thus available.

8.3 Proof of Uniqueness

Since all available elements are authenticated. If it is possible to construct more than one element in the $ijth$ index of any of D, F, G, then an O(1) proof is given by presenting something like:

$(d_{00}, 0, 0), x_{00}$, $(d'_{00}, 0, 0), x'_{00}$, where $d_{00} \neq d'_{00}$

This means if two distinct items are unavailable a succinct fraud proof can be given.

8.4 Proof of Liveness

Given points 8.1, we can prove liveness.

Suppose no authentication fraud proof is given by the network, then:

If the data for a row D_{i*} is unavailable then: (by authentication transitivity)

#available elements in $F_{i*} < \sqrt{m}$ or equivalently:

#unavailable elements in $F_{i*} > (k-1)\sqrt{m}$. This implies:

#unavailable elements in $G > (k-1)^2 m$ or equivalently: (by authentication transitivity)

#available elements in $G < k^2 m - (k-1)^2 m = (2k-1)m$

Therefore a liveness check needs to ensure that more than $(2k-1)m$ elements of G are available with a high likelihood. If this be the case, we can reconstruct some dataset D.

Collaboratively Generated Erasure Codes

Collaborative generation, check two merkle roots.

$(d_{00},0,0),x_{00}$ $(d_{10},1,0),x_{10}$	$(d_{01},0,1),x_{01}$ $(d_{11},1,1),x_{11}$	$(f_{00},0,0),y_{00}$ $(f_{10},1,0),y_{10}$	$(f_{01},0,1),y_{01}$ $(f_{11},1,1),y_{11}$	$(f_{02},0,2),y_{02}$ $(f_{12},1,2),y_{12}$	
$x_{0*},\bar{x}_{0*}$	$x_{1*},\bar{x}_{1*}$	f	f	f	
		$(g_{00},0,0),z_{00}$	$(g_{01},0,1),z_{01}$	$(g_{02},0,2),z_{02}$	$z_{0*},\bar{z_{0*}}$
		$(g_{10},1,0),z_{10}$	$(g_{11},1,1),z_{11}$	$(g_{12},1,2),z_{12}$	$z_{*1},\bar{z_{*1}}$
		$(g_{20},2,0),z_{20}$	$(g_{21},2,1),z_{21}$	$(g_{22},2,2),z_{22}$	$z_{*2},\bar{z_{*2}}$

What are the implications to censoring of codes created by trust scores? Competition amongst many sellers with high trust rating, think amazon. So should not degenerate to a one proposer model. Anyone can build up trust. **So then partly depends on fee, could also require to rotate proposers.**

9 Solving The State Problem

Suppose that we use the state transition lingo, STF(S, B) -> S', where S and S' are states, B is a block (or it could be a transaction T), and STF is the state transition function. Then, we can transform: S -> the state root of S (ie. the 32 byte root hash of the Merkle Patricia tree containing S) B -> (B, W), where W is a "witness" - a set of Merkle branches proving the values of all data that the execution of B accesses STF -> STF', which takes as input a state root and a block-plus-witness, uses the witness as a "database" any time the execution of the block needs to read any accounts, storage keys or other state data [exiting with an error if the witness does not contain some piece of data that is being asked for], and outputs the new state root. That is, full nodes would only store state roots, and it would be miners' responsibility to package Merkle branches ("witnesses") along with the blocks, and full nodes would download and verify these expanded blocks. It's entirely possible for stateless full nodes and regular full nodes to exist alongside each other in a network; you could have translator nodes that take a block B, attach the required witness, and broadcast (B, W) on a different network protocol that stateless nodes live on; if a miner mines a block on this stateless network, then the witness can simply be stripped off, and the block rebroadcasted on the regular network.

The simplest way to conceive the witness in a real protocol is to view it as an RLP-encoded list of objects, which could then be parsed by the client into a {sha3(x): x} key-value map; this map can then simply be plugged into an existing ethereum implementation as a "database".

One limitation of the above idea being applied to Ethereum as it exists today is that it would still require miners to be state-storing full nodes. To solve this, we put the witness outside the signed data in the transaction, and allow the miner that includes the transaction to adjust the witness as needed before including the transaction. If miners maintain a policy of holding onto all new state

tree nodes that were created in, say, the last 24 hours, then they will necessarily have all the needed info to update the Merkle branches for any transactions published in the last 24 hours.

Miners and full nodes in general no longer need to store any state. This makes "fast syncing" much much faster (potentially a few seconds). All of the thorny questions about state storage economics that lead to the need for designs like rent (eg. https://github.com/ethereum/EIPs/issues/35 18 http://github.com/ethereum/EIPs/issues/87 14 http://github.com/ethereum/EIPs/issues/88 11) and even the current complex SSTORE cost/refund scheme disappear, and blockchain economics can focus purely on pricing bandwidth and computation, a much simpler problem) Even for state-storing clients, the account lists allow clients to pre-fetch storage data from disk, possibly in parallel, greatly reducing their vulnerability to DoS attacks. In a sharded blockchain, security is increased by reshuffling clients between shards frequently; the more quickly clients are reshuffled, the more adaptive the adversaries that the scheme is secure against in a BFT model. In a stateless client, this cost drops to zero, allowing clients to be reshuffled between every single block that they create. One problem that this introduces is: who does store state?

Any new state trie object that gets created or touched gets by default stored by all full nodes for 3 months. This will likely be around 2.5 GB, and this is like "welfare storage" that is provided by the network on a voluntary basis. We know that this level of service definitely can be provided on a volunteer basis, as the current light client infrastructure already depends on altruism. After 3 months, clients can forget randomly, so that for example a state trie object that was last touched 12 months ago would still be stored by 25% of nodes, and an object last touched 60 months ago would still be stored by 5% of nodes. Clients can try to ask for these objects using the regular light client protocol.

Clients that wish to ensure availability of specific pieces of data much longer can do so with payments in state channels, similar to file coing. A client can set up channels with paid archival nodes, and make a conditional payment in the channel of the form "I give up $0.0001, and by default this payment is gone forever. However, if you later provide an object with hash H, and I sign off on it, then that $0.0001 instead goes to you". This would signal a credible commitment to being possibly willing to unlock those funds for that object in the future, and archival nodes could enter many millions of such arrangements and wait for data requests to appear and become an income stream. We expect dapp developers to get their users to randomly store some portion of storage keys specifically related to their dapp in browser localstorage. This could even deliberately be made easy to do in the web3 API. In practice, we expect the number of "archival nodes" that simply store everything forever to continue to be high enough to serve the network until the total state size exceeds ~1-10 terabytes after the introduction of sharding, so the above may not even be needed.

Links discussing related ideas:

Further data availability schemes may be found here... **IPFS for storage. File coin. They augment IPFS with a proof of storage over time. However the key is do you also have th up to dat weitnesses, which is not covered in their protocol. Whenever witnesses come int hey update the state. Doing this would greatly reduce costs and allow others to become miners.**

10 Virtual Machine - Web Assembly?

Fast & Efficient: To truly distinguish Ethereum as the World Computer we need to have a very performant VM. The current architecture of the VM is one of the greatest blockers to raw performance. WebAssembly aims to execute at near native speed by taking advantage of common hardware capabilities available on a wide range of platforms. This will open the door to a wide array of uses that require performance/throughput.

Toolchain Compatibility: A LLVM front-end for WASM is part of the MVP. This will Allow developers to write contracts and reuse applications written in common languages such as C/C++, go and rust.

Portability: WASM is targeted to be deployed in all the major web browsers which will result in it being one of the most widely deployed VM architecture. Contracts compiled to eWASM will share compatibility with any standard WASM environment. Which will make running a program either directly on Ethereum, on a cloud hosting environment, or on one's local machine - a frictionless process.

Optional And Flexible Metering: Metering the VM adds overhead but is essential for running untrusted code. If code is trusted then metering maybe optional. eWASM defines metering as an optional layer to accommodate for these use cases.

Metering VMs is the same concept as electrical power companies have when charging you for the amount of electricity that you used. With VM's we attempt to get a measurement of computation time of some code and instead of electricity used, you are charged for the CPU's time used. We call this metering.

Not off the bat, a transpiler will have to be created to compile existing EVM code into eWASM. As far as other High level languages you should be able to use an language that can be compiled by LLVM (c/c++/rust/go)

Part of the project goal is to eliminate nasal-demons. It's in the MVP. There are still a couple of edge case like sign values on NaNs but they can be canonicalized by AST transforms.

Most languages with a virtual machine can be transpiled, meaning you could run the contracts of some other coin in here. If we use same base as Zilliqa or Dfinity, then we can re-use any transpilers they build. 'Ultimately, the battle will not be won on language features, but on cost. Since in the future, you will be able to deploy any contract on any system.

11 Comparison With Other Coins

(1) Transpilers ; (2)

The first question we should ask of the architecture is the nature of forks. Should a sharded-blockchain be fork-free?

Dfinity overview:

- Random number generation to select block maker and notarisation group
- Notarization proves that a block has been published at some time, based on a group vote.
- Notarisation is more secure than a single vote. => Finality after two rounds.
- **Kind of taking a hybrid approach pBft and block based consensus.**
- Good: **notarization in Dfinity is not primarily a validity guarantee but rather a timestamp plus a proof of publication'**
- **Good: almost fork-free; Good: still uses probabilistic consensus; Prevents long-range stake attacks**

Neo:

- Delegated Proof of Stake.
- Finality after a single round. Why based on voting.

Zilliqa:

- Sampling into shards.
- Honest majority assumotio + PBFT, much more robust.
- **Good: Uses random sampling so robust vopting groups;**
- **Bad: no slashable conditions.**

Ethereum:

- Splitting problems of availability from validity?
- Availability can be decided quickly.
- Validity decided by a kind of Trubit systle mechanism, which means validity can then be checked quickly.

`https://cmt.research.microsoft.com/NIPS2016/`

References

References follow the acknowledgments. Use unnumbered first-level heading for the references. Any choice of citation style is acceptable as long as you are consistent. It is permissible to reduce the font size to `small` (9 point) when listing the references. **Remember that you can use a ninth page as long as it contains *only* cited references.**

[1] Alexander, J.A. & Mozer, M.C. (1995) Template-based algorithms for connectionist rule extraction. In G. Tesauro, D.S. Touretzky and T.K. Leen (eds.), *Advances in Neural Information Processing Systems 7*, pp. 609–616. Cambridge, MA: MIT Press.

[2] Bower, J.M. & Beeman, D. (1995) *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* New York: TELOS/Springer–Verlag.

[3] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region CA3. *Journal of Neuroscience* **15**(7):5249-5262.